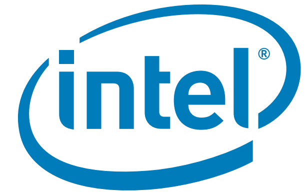


Intel® 64 and IA-32 Architectures Optimization Reference Manual

Order Number: 248966-032
January 2016

Intel technologies features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting http://www.intel.com/design/literature.htm.

Intel, the Intel logo, Intel Atom, Intel Core, Intel SpeedStep, MMX, Pentium, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 1997-2016, Intel Corporation. All Rights Reserved.

CONTENTS

PAGE

CHAPTER 1 INTRODUCTION

1.1 TUNING YOUR APPLICATION. 1-1
1.2 ABOUT THIS MANUAL. 1-1
1.3 RELATED INFORMATION 1-3

CHAPTER 2 INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

2.1 THE SKYLAKE MICROARCHITECTURE. 2-2
2.1.1 The Front End 2-3
2.1.2 The Out-of-Order Execution Engine 2-3
2.1.3 Cache and Memory Subsystem 2-5
2.2 THE HASWELL MICROARCHITECTURE 2-6
2.2.1 The Front End 2-7
2.2.2 The Out-of-Order Engine 2-8
2.2.3 Execution Engine 2-8
2.2.4 Cache and Memory Subsystem 2-10
2.2.4.1 Load and Store Operation Enhancements 2-11
2.2.5 The Haswell-E Microarchitecture. 2-11
2.2.6 The Broadwell Microarchitecture 2-12
2.3 INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE 2-12
2.3.1 Intel® Microarchitecture Code Name Sandy Bridge Pipeline Overview 2-13
2.3.2 The Front End 2-14
2.3.2.1 Legacy Decode Pipeline 2-15
2.3.2.2 Decoded ICache. 2-17
2.3.2.3 Branch Prediction. 2-18
2.3.2.4 Micro-op Queue and the Loop Stream Detector (LSD) 2-18
2.3.3 The Out-of-Order Engine 2-19
2.3.3.1 Renamer 2-19
2.3.3.2 Scheduler 2-20
2.3.4 The Execution Core 2-20
2.3.5 Cache Hierarchy 2-22
2.3.5.1 Load and Store Operation Overview 2-22
2.3.5.2 L1 DCache. 2-23
2.3.5.3 Ring Interconnect and Last Level Cache 2-27
2.3.5.4 Data Prefetching 2-28
2.3.6 System Agent 2-29
2.3.7 Intel® Microarchitecture Code Name Ivy Bridge 2-30
2.4 INTEL® CORE™ MICROARCHITECTURE AND ENHANCED INTEL® CORE™ MICROARCHITECTURE 2-30
2.4.1 Intel® Core™ Microarchitecture Pipeline Overview 2-31
2.4.2 Front End 2-32
2.4.2.1 Branch Prediction Unit 2-33
2.4.2.2 Instruction Fetch Unit. 2-33
2.4.2.3 Instruction Queue (IQ). 2-34
2.4.2.4 Instruction Decode. 2-35
2.4.2.5 Stack Pointer Tracker 2-35
2.4.2.6 Micro-fusion 2-35
2.4.3 Execution Core 2-36
2.4.3.1 Issue Ports and Execution Units 2-36
2.4.4 Intel® Advanced Memory Access 2-38
2.4.4.1 Loads and Stores 2-39
2.4.4.2 Data Prefetch to L1 caches. 2-40
2.4.4.3 Data Prefetch Logic. 2-40

PAGE

2.4.4.4 Store Forwarding . . . 2-41
2.4.4.5 Memory Disambiguation. . . . 2-42
2.4.5 Intel® Advanced Smart Cache . . . 2-42
2.4.5.1 Loads . . . 2-43
2.4.5.2 Stores. . . . 2-44
2.5 INTEL® MICROARCHITECTURE CODE NAME NEHALEM . . . 2-44
2.5.1 Microarchitecture Pipeline . . . 2-45
2.5.2 Front End Overview. . . . 2-46
2.5.3 Execution Engine . . . 2-47
2.5.3.1 Issue Ports and Execution Units . . . 2-48
2.5.4 Cache and Memory Subsystem . . . 2-48
2.5.5 Load and Store Operation Enhancements . . . 2-49
2.5.5.1 Efficient Handling of Alignment Hazards . . . 2-49
2.5.5.2 Store Forwarding Enhancement . . . 2-50
2.5.6 REP String Enhancement . . . 2-52
2.5.7 Enhancements for System Software . . . 2-53
2.5.8 Efficiency Enhancements for Power Consumption . . . 2-53
2.5.9 Hyper-Threading Technology Support in Intel® Microarchitecture Code Name Nehalem . . . 2-53
2.6 INTEL® HYPER-THREADING TECHNOLOGY . . . 2-53
2.6.1 Processor Resources and HT Technology . . . 2-54
2.6.1.1 Replicated Resources . . . 2-55
2.6.1.2 Partitioned Resources . . . 2-55
2.6.1.3 Shared Resources . . . 2-55
2.6.2 Microarchitecture Pipeline and HT Technology . . . 2-55
2.6.3 Front End Pipeline . . . 2-56
2.6.4 Execution Core . . . 2-56
2.6.5 Retirement . . . 2-56
2.7 INTEL® 64 ARCHITECTURE . . . 2-56
2.8 SIMD TECHNOLOGY . . . 2-57
2.9 SUMMARY OF SIMD TECHNOLOGIES AND APPLICATION LEVEL EXTENSIONS . . . 2-58
2.9.1 MMX™ Technology . . . 2-59
2.9.2 Streaming SIMD Extensions . . . 2-59
2.9.3 Streaming SIMD Extensions 2. . . . 2-59
2.9.4 Streaming SIMD Extensions 3. . . . 2-59
2.9.5 Supplemental Streaming SIMD Extensions 3 . . . 2-60
2.9.6 SSE4.1 . . . 2-60
2.9.7 SSE4.2 . . . 2-60
2.9.8 AESNI and PCLMULQDQ . . . 2-61
2.9.9 Intel® Advanced Vector Extensions . . . 2-61
2.9.10 Half-Precision Floating-Point Conversion (F16C) . . . 2-61
2.9.11 RDRAND . . . 2-62
2.9.12 Fused-Multiply-ADD (FMA) Extensions . . . 2-62
2.9.13 Intel AVX2 . . . 2-62
2.9.14 General-Purpose Bit-Processing Instructions . . . 2-62
2.9.15 Intel® Transactional Synchronization Extensions . . . 2-62
2.9.16 RDSEED . . . 2-62
2.9.17 ADCX and ADOX Instructions . . . 2-62

CHAPTER 3
GENERAL OPTIMIZATION GUIDELINES

3.1 PERFORMANCE TOOLS . . . 3-1
3.1.1 Intel® C++ and Fortran Compilers . . . 3-1
3.1.2 General Compiler Recommendations . . . 3-2
3.1.3 VTune™ Performance Analyzer . . . 3-2
3.2 PROCESSOR PERSPECTIVES . . . 3-2
3.2.1 CPUID Dispatch Strategy and Compatible Code Strategy . . . 3-3
3.2.2 Transparent Cache-Parameter Strategy. . . . 3-3
3.2.3 Threading Strategy and Hardware Multithreading Support . . . 3-3
3.3 CODING RULES, SUGGESTIONS AND TUNING HINTS . . . 3-3
3.4 OPTIMIZING THE FRONT END. . . . 3-4
3.4.1 Branch Prediction Optimization . . . 3-4
3.4.1.1 Eliminating Branches. . . . 3-5
3.4.1.2 Spin-Wait and Idle Loops . . . 3-6
3.4.1.3 Static Prediction . . . 3-6

PAGE

3.4.1.4 Inlining, Calls and Returns . . . 3-8
3.4.1.5 Code Alignment. . . 3-8
3.4.1.6 Branch Type Selection . . . 3-9
3.4.1.7 Loop Unrolling . . . 3-10
3.4.1.8 Compiler Support for Branch Prediction . . . 3-11
3.4.2 Fetch and Decode Optimization. . . 3-12
3.4.2.1 Optimizing for Micro-fusion. . . 3-12
3.4.2.2 Optimizing for Macro-fusion . . . 3-12
3.4.2.3 Length-Changing Prefixes (LCP) . . . 3-16
3.4.2.4 Optimizing the Loop Stream Detector (LSD) . . . 3-17
3.4.2.5 Exploit LSD Micro-op Emission Bandwidth in Intel® Microarchitecture Code Name Sandy Bridge. . . 3-18
3.4.2.6 Optimization for Decoded ICache . . . 3-19
3.4.2.7 Other Decoding Guidelines . . . 3-20
3.5 OPTIMIZING THE EXECUTION CORE . . . 3-20
3.5.1 Instruction Selection . . . 3-20
3.5.1.1 Use of the INC and DEC Instructions . . . 3-21
3.5.1.2 Integer Divide . . . 3-21
3.5.1.3 Using LEA . . . 3-22
3.5.1.4 ADC and SBB in Intel® Microarchitecture Code Name Sandy Bridge . . . 3-23
3.5.1.5 Bitwise Rotation. . . 3-24
3.5.1.6 Variable Bit Count Rotation and Shift . . . 3-24
3.5.1.7 Address Calculations . . . 3-25
3.5.1.8 Clearing Registers and Dependency Breaking Idioms . . . 3-25
3.5.1.9 Compares . . . 3-27
3.5.1.10 Using NOPs. . . 3-27
3.5.1.11 Mixing SIMD Data Types. . . 3-28
3.5.1.12 Spill Scheduling . . . 3-28
3.5.1.13 Zero-Latency MOV Instructions. . . 3-28
3.5.2 Avoiding Stalls in Execution Core . . . 3-30
3.5.2.1 ROB Read Port Stalls. . . 3-30
3.5.2.2 Writeback Bus Conflicts . . . 3-31
3.5.2.3 Bypass between Execution Domains. . . 3-31
3.5.2.4 Partial Register Stalls . . . 3-32
3.5.2.5 Partial XMM Register Stalls. . . 3-33
3.5.2.6 Partial Flag Register Stalls . . . 3-33
3.5.2.7 Floating-Point/SIMD Operands. . . 3-34
3.5.3 Vectorization. . . 3-35
3.5.4 Optimization of Partially Vectorizable Code. . . 3-36
3.5.4.1 Alternate Packing Techniques . . . 3-37
3.5.4.2 Simplifying Result Passing. . . 3-38
3.5.4.3 Stack Optimization. . . 3-38
3.5.4.4 Tuning Considerations . . . 3-39
3.6 OPTIMIZING MEMORY ACCESSES. . . 3-40
3.6.1 Load and Store Execution Bandwidth . . . 3-41
3.6.1.1 Make Use of Load Bandwidth in Intel® Microarchitecture Code Name Sandy Bridge . . . 3-41
3.6.1.2 L1D Cache Latency in Intel® Microarchitecture Code Name Sandy Bridge . . . 3-42
3.6.1.3 Handling L1D Cache Bank Conflict . . . 3-43
3.6.2 Minimize Register Spills . . . 3-44
3.6.3 Enhance Speculative Execution and Memory Disambiguation. . . 3-44
3.6.4 Alignment . . . 3-45
3.6.5 Store Forwarding . . . 3-47
3.6.5.1 Store-to-Load-Forwarding Restriction on Size and Alignment . . . 3-47
3.6.5.2 Store-forwarding Restriction on Data Availability. . . 3-51
3.6.6 Data Layout Optimizations . . . 3-52
3.6.7 Stack Alignment . . . 3-54
3.6.8 Capacity Limits and Aliasing in Caches. . . 3-54
3.6.8.1 Capacity Limits in Set-Associative Caches . . . 3-55
3.6.8.2 Aliasing Cases in the Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo and Intel® Core™ 2 Duo Processors 3-55
3.6.9 Mixing Code and Data . . . 3-56
3.6.9.1 Self-modifying Code . . . 3-57
3.6.9.2 Position Independent Code . . . 3-57
3.6.10 Write Combining . . . 3-57
3.6.11 Locality Enhancement. . . 3-58
3.6.12 Minimizing Bus Latency . . . 3-59
3.6.13 Non-Temporal Store Bus Traffic . . . 3-60

PAGE

3.7 PREFETCHING . . . 3-60
3.7.1 Hardware Instruction Fetching and Software Prefetching . . . 3-61
3.7.2 Hardware Prefetching for First-Level Data Cache . . . 3-61
3.7.3 Hardware Prefetching for Second-Level Cache . . . 3-63
3.7.4 Cacheability Instructions . . . 3-63
3.7.5 REP Prefix and Data Movement . . . 3-64
3.7.6 Enhanced REP MOVSB and STOSB operation (ERMSB) . . . 3-66
3.7.6.1 Memcpy Considerations . . . 3-66
3.7.6.2 Memmove Considerations . . . 3-67
3.7.6.3 Memset Considerations . . . 3-68
3.8 FLOATING-POINT CONSIDERATIONS . . . 3-68
3.8.1 Guidelines for Optimizing Floating-point Code . . . 3-68
3.8.2 Microarchitecture Specific Considerations . . . 3-69
3.8.2.1 Long-Latency FP Instructions . . . 3-69
3.8.2.2 Miscellaneous Instructions . . . 3-69
3.8.3 Floating-point Modes and Exceptions . . . 3-69
3.8.3.1 Floating-point Exceptions . . . 3-69
3.8.3.2 Dealing with floating-point exceptions in x87 FPU code . . . 3-70
3.8.3.3 Floating-point Exceptions in SSE/SSE2/SSE3 Code . . . 3-70
3.8.4 Floating-point Modes . . . 3-70
3.8.4.1 Rounding Mode . . . 3-71
3.8.4.2 Precision . . . 3-72
3.8.5 x87 vs. Scalar SIMD Floating-point Trade-offs . . . 3-73
3.8.5.1 Scalar SSE/SSE2 . . . 3-73
3.8.5.2 Transcendental Functions . . . 3-73
3.9 MAXIMIZING PCIE PERFORMANCE . . . 3-74

CHAPTER 4
CODING FOR SIMD ARCHITECTURES

4.1 CHECKING FOR PROCESSOR SUPPORT OF SIMD TECHNOLOGIES . . . 4-1
4.1.1 Checking for MMX Technology Support . . . 4-2
4.1.2 Checking for Streaming SIMD Extensions Support . . . 4-2
4.1.3 Checking for Streaming SIMD Extensions 2 Support . . . 4-2
4.1.4 Checking for Streaming SIMD Extensions 3 Support . . . 4-3
4.1.5 Checking for Supplemental Streaming SIMD Extensions 3 Support . . . 4-3
4.1.6 Checking for SSE4.1 Support . . . 4-4
4.1.7 Checking for SSE4.2 Support . . . 4-4
4.1.8 DetectiON of PCLMULQDQ and AESNI Instructions . . . 4-4
4.1.9 Detection of AVX Instructions . . . 4-5
4.1.10 Detection of VEX-Encoded AES and VPCLMULQDQ . . . 4-7
4.1.11 Detection of F16C Instructions . . . 4-7
4.1.12 Detection of FMA . . . 4-8
4.1.13 Detection of AVX2 . . . 4-9
4.2 CONSIDERATIONS FOR CODE CONVERSION TO SIMD PROGRAMMING . . . 4-10
4.2.1 Identifying Hot Spots . . . 4-12
4.2.2 Determine If Code Benefits by Conversion to SIMD Execution . . . 4-12
4.3 CODING TECHNIQUES . . . 4-12
4.3.1 Coding Methodologies . . . 4-13
4.3.1.1 Assembly . . . 4-14
4.3.1.2 Intrinsics . . . 4-14
4.3.1.3 Classes . . . 4-15
4.3.1.4 Automatic Vectorization . . . 4-16
4.4 STACK AND DATA ALIGNMENT . . . 4-17
4.4.1 Alignment and Contiguity of Data Access Patterns . . . 4-17
4.4.1.1 Using Padding to Align Data . . . 4-17
4.4.1.2 Using Arrays to Make Data Contiguous . . . 4-17
4.4.2 Stack Alignment For 128-bit SIMD Technologies . . . 4-18
4.4.3 Data Alignment for MMX Technology . . . 4-18
4.4.4 Data Alignment for 128-bit data . . . 4-19
4.4.4.1 Compiler-Supported Alignment . . . 4-19
4.5 IMPROVING MEMORY UTILIZATION . . . 4-20
4.5.1 Data Structure Layout . . . 4-20
4.5.2 Strip-Mining . . . 4-23
4.5.3 Loop Blocking . . . 4-24

PAGE

4.6 INSTRUCTION SELECTION 4-26
4.6.1 SIMD Optimizations and Microarchitectures. 4-27
4.7 TUNING THE FINAL APPLICATION. 4-28

CHAPTER 5
OPTIMIZING FOR SIMD INTEGER APPLICATIONS

5.1 GENERAL RULES ON SIMD INTEGER CODE 5-1
5.2 USING SIMD INTEGER WITH X87 FLOATING-POINT 5-2
5.2.1 Using the EMMS Instruction 5-2
5.2.2 Guidelines for Using EMMS Instruction 5-2
5.3 DATA ALIGNMENT 5-3
5.4 DATA MOVEMENT CODING TECHNIQUES 5-5
5.4.1 Unsigned Unpack 5-5
5.4.2 Signed Unpack. 5-5
5.4.3 Interleaved Pack with Saturation 5-6
5.4.4 Interleaved Pack without Saturation 5-7
5.4.5 Non-Interleaved Unpack. 5-8
5.4.6 Extract Data Element 5-9
5.4.7 Insert Data Element. 5-10
5.4.8 Non-Unit Stride Data Movement 5-11
5.4.9 Move Byte Mask to Integer 5-12
5.4.10 Packed Shuffle Word for 64-bit Registers 5-12
5.4.11 Packed Shuffle Word for 128-bit Registers. 5-13
5.4.12 Shuffle Bytes. 5-13
5.4.13 Conditional Data Movement 5-14
5.4.14 Unpacking/interleaving 64-bit Data in 128-bit Registers 5-14
5.4.15 Data Movement 5-14
5.4.16 Conversion Instructions 5-14
5.5 GENERATING CONSTANTS. 5-14
5.6 BUILDING BLOCKS 5-15
5.6.1 Absolute Difference of Unsigned Numbers 5-15
5.6.2 Absolute Difference of Signed Numbers. 5-16
5.6.3 Absolute Value 5-16
5.6.4 Pixel Format Conversion 5-17
5.6.5 Endian Conversion 5-18
5.6.6 Clipping to an Arbitrary Range [High, Low] 5-19
5.6.6.1 Highly Efficient Clipping 5-19
5.6.6.2 Clipping to an Arbitrary Unsigned Range [High, Low]. 5-21
5.6.7 Packed Max/Min of Byte, Word and Dword 5-21
5.6.8 Packed Multiply Integers 5-21
5.6.9 Packed Sum of Absolute Differences. 5-22
5.6.10 MPSADBW and PHMINPOSUW 5-22
5.6.11 Packed Average (Byte/Word). 5-22
5.6.12 Complex Multiply by a Constant. 5-22
5.6.13 Packed 64-bit Add/Subtract 5-23
5.6.14 128-bit Shifts 5-23
5.6.15 PTEST and Conditional Branch. 5-23
5.6.16 Vectorization of Heterogeneous Computations across Loop Iterations. 5-24
5.6.17 Vectorization of Control Flows in Nested Loops 5-25
5.7 MEMORY OPTIMIZATIONS 5-27
5.7.1 Partial Memory Accesses. 5-28
5.7.1.1 Supplemental Techniques for Avoiding Cache Line Splits 5-29
5.7.2 Increasing Bandwidth of Memory Fills and Video Fills 5-30
5.7.2.1 Increasing Memory Bandwidth Using the MOVDQ Instruction. 5-30
5.7.2.2 Increasing Memory Bandwidth by Loading and Storing to and from the Same DRAM Page 5-30
5.7.2.3 Increasing UC and WC Store Bandwidth by Using Aligned Stores. 5-31
5.7.3 Reverse Memory Copy 5-31
5.8 CONVERTING FROM 64-BIT TO 128-BIT SIMD INTEGERS 5-34
5.8.1 SIMD Optimizations and Microarchitectures. 5-34
5.8.1.1 Packed SSE2 Integer versus MMX Instructions 5-34
5.8.1.2 Work-around for False Dependency Issue 5-35
5.9 TUNING PARTIALLY VECTORIZABLE CODE 5-35
5.10 PARALLEL MODE AES ENCRYPTION AND DECRYPTION. 5-38
5.10.1 AES Counter Mode of Operation 5-38

PAGE

5.10.2 AES Key Expansion Alternative . . . 5-46
5.10.3 Enhancement in Intel Microarchitecture Code Name Haswell . . . 5-48
5.10.3.1 AES and Multi-Buffer Cryptographic Throughput. . . . 5-48
5.10.3.2 PCLMULQDQ Improvement . . . 5-48
5.11 LIGHT-WEIGHT DECOMPRESSION AND DATABASE PROCESSING . . . 5-48
5.11.1 Reduced Dynamic Range Datasets . . . 5-49
5.11.2 Compression and Decompression Using SIMD Instructions . . . 5-49

CHAPTER 6
OPTIMIZING FOR SIMD FLOATING-POINT APPLICATIONS

6.1 GENERAL RULES FOR SIMD FLOATING-POINT CODE . . . 6-1
6.2 PLANNING CONSIDERATIONS . . . 6-1
6.3 USING SIMD FLOATING-POINT WITH X87 FLOATING-POINT. . . . 6-2
6.4 SCALAR FLOATING-POINT CODE . . . 6-2
6.5 DATA ALIGNMENT . . . 6-2
6.5.1 Data Arrangement . . . 6-2
6.5.1.1 Vertical versus Horizontal Computation . . . 6-3
6.5.1.2 Data Swizzling . . . 6-5
6.5.1.3 Data Deswizzling . . . 6-7
6.5.1.4 Horizontal ADD Using SSE . . . 6-8
6.5.2 Use of CVTTPS2PI/CVTTSS2SI Instructions . . . 6-10
6.5.3 Flush-to-Zero and Denormals-are-Zero Modes . . . 6-10
6.6 SIMD OPTIMIZATIONS AND MICROARCHITECTURES . . . 6-11
6.6.1 SIMD Floating-point Programming Using SSE3 . . . 6-11
6.6.1.1 SSE3 and Complex Arithmetics . . . 6-12
6.6.1.2 Packed Floating-Point Performance in Intel Core Duo Processor . . . 6-14
6.6.2 Dot Product and Horizontal SIMD Instructions . . . 6-14
6.6.3 Vector Normalization . . . 6-16
6.6.4 Using Horizontal SIMD Instruction Sets and Data Layout . . . 6-18
6.6.4.1 SOA and Vector Matrix Multiplication . . . 6-20

CHAPTER 7
OPTIMIZING CACHE USAGE

7.1 GENERAL PREFETCH CODING GUIDELINES . . . 7-1
7.2 HARDWARE PREFETCHING OF DATA . . . 7-2
7.3 PREFETCH AND CACHEABILITY INSTRUCTIONS . . . 7-3
7.4 PREFETCH. . . . 7-3
7.4.1 Software Data Prefetch . . . 7-3
7.4.2 Prefetch Instructions – Pentium® 4 Processor Implementation . . . 7-4
7.4.3 Prefetch and Load Instructions . . . 7-4
7.5 CACHEABILITY CONTROL . . . 7-5
7.5.1 The Non-temporal Store Instructions . . . 7-5
7.5.1.1 Fencing . . . 7-5
7.5.1.2 Streaming Non-temporal Stores . . . 7-5
7.5.1.3 Memory Type and Non-temporal Stores . . . 7-6
7.5.1.4 Write-Combining . . . 7-6
7.5.2 Streaming Store Usage Models . . . 7-7
7.5.2.1 Coherent Requests . . . 7-7
7.5.2.2 Non-coherent requests . . . 7-7
7.5.3 Streaming Store Instruction Descriptions . . . 7-7
7.5.4 The Streaming Load Instruction . . . 7-8
7.5.5 FENCE Instructions . . . 7-8
7.5.5.1 SFENCE Instruction . . . 7-8
7.5.5.2 LFENCE Instruction . . . 7-8
7.5.5.3 MFENCE Instruction . . . 7-8
7.5.6 CLFLUSH Instruction . . . 7-9
7.5.7 CLFLUSHOPT Instruction . . . 7-10
7.6 MEMORY OPTIMIZATION USING PREFETCH . . . 7-11
7.6.1 Software-Controlled Prefetch . . . 7-11
7.6.2 Hardware Prefetch . . . 7-12
7.6.3 Example of Effective Latency Reduction with Hardware Prefetch . . . 7-12
7.6.4 Example of Latency Hiding with S/W Prefetch Instruction . . . 7-13

PAGE

7.6.5 Software Prefetching Usage Checklist . . . 7-15
7.6.6 Software Prefetch Scheduling Distance . . . 7-15
7.6.7 Software Prefetch Concatenation . . . 7-16
7.6.8 Minimize Number of Software Prefetches . . . 7-17
7.6.9 Mix Software Prefetch with Computation Instructions . . . 7-18
7.6.10 Software Prefetch and Cache Blocking Techniques . . . 7-19
7.6.11 Hardware Prefetching and Cache Blocking Techniques . . . 7-23
7.6.12 Single-pass versus Multi-pass Execution . . . 7-24
7.7 MEMORY OPTIMIZATION USING NON-TEMPORAL STORES . . . 7-25
7.7.1 Non-temporal Stores and Software Write-Combining . . . 7-25
7.7.2 Cache Management . . . 7-26
7.7.2.1 Video Encoder . . . 7-26
7.7.2.2 Video Decoder . . . 7-26
7.7.2.3 Conclusions from Video Encoder and Decoder Implementation . . . 7-27
7.7.2.4 Optimizing Memory Copy Routines . . . 7-27
7.7.2.5 TLB Priming . . . 7-28
7.7.2.6 Using the 8-byte Streaming Stores and Software Prefetch . . . 7-29
7.7.2.7 Using 16-byte Streaming Stores and Hardware Prefetch . . . 7-29
7.7.2.8 Performance Comparisons of Memory Copy Routines . . . 7-30
7.7.3 Deterministic Cache Parameters . . . 7-31
7.7.3.1 Cache Sharing Using Deterministic Cache Parameters . . . 7-32
7.7.3.2 Cache Sharing in Single-Core or Multicore . . . 7-32
7.7.3.3 Determine Prefetch Stride . . . 7-32

CHAPTER 8
MULTICORE AND HYPER-THREADING TECHNOLOGY

8.1 PERFORMANCE AND USAGE MODELS . . . 8-1
8.1.1 Multithreading . . . 8-1
8.1.2 Multitasking Environment . . . 8-2
8.2 PROGRAMMING MODELS AND MULTITHREADING . . . 8-3
8.2.1 Parallel Programming Models . . . 8-4
8.2.1.1 Domain Decomposition . . . 8-4
8.2.2 Functional Decomposition . . . 8-4
8.2.3 Specialized Programming Models . . . 8-4
8.2.3.1 Producer-Consumer Threading Models . . . 8-5
8.2.4 Tools for Creating Multithreaded Applications . . . 8-7
8.2.4.1 Programming with OpenMP Directives . . . 8-8
8.2.4.2 Automatic Parallelization of Code . . . 8-8
8.2.4.3 Supporting Development Tools . . . 8-8
8.3 OPTIMIZATION GUIDELINES . . . 8-8
8.3.1 Key Practices of Thread Synchronization . . . 8-8
8.3.2 Key Practices of System Bus Optimization . . . 8-9
8.3.3 Key Practices of Memory Optimization . . . 8-9
8.3.4 Key Practices of Execution Resource Optimization . . . 8-9
8.3.5 Generality and Performance Impact . . . 8-10
8.4 THREAD SYNCHRONIZATION . . . 8-10
8.4.1 Choice of Synchronization Primitives . . . 8-10
8.4.2 Synchronization for Short Periods . . . 8-11
8.4.3 Optimization with Spin-Locks . . . 8-13
8.4.4 Synchronization for Longer Periods . . . 8-13
8.4.4.1 Avoid Coding Pitfalls in Thread Synchronization . . . 8-14
8.4.5 Prevent Sharing of Modified Data and False-Sharing . . . 8-14
8.4.6 Placement of Shared Synchronization Variable . . . 8-15
8.4.7 Pause Latency in Skylake Microarchitecture . . . 8-16
8.5 SYSTEM BUS OPTIMIZATION . . . 8-17
8.5.1 Conserve Bus Bandwidth . . . 8-17
8.5.2 Understand the Bus and Cache Interactions . . . 8-18
8.5.3 Avoid Excessive Software Prefetches . . . 8-18
8.5.4 Improve Effective Latency of Cache Misses . . . 8-18
8.5.5 Use Full Write Transactions to Achieve Higher Data Rate . . . 8-19
8.6 MEMORY OPTIMIZATION . . . 8-19
8.6.1 Cache Blocking Technique . . . 8-19
8.6.2 Shared-Memory Optimization . . . 8-20
8.6.2.1 Minimize Sharing of Data between Physical Processors . . . 8-20

PAGE

8.6.2.2 Batched Producer-Consumer Model . . . 8-20
8.6.3 Eliminate 64-KByte Aliased Data Accesses . . . 8-22
8.7 FRONT END OPTIMIZATION . . . 8-22
8.7.1 Avoid Excessive Loop Unrolling . . . 8-22
8.8 AFFINITIES AND MANAGING SHARED PLATFORM RESOURCES . . . 8-22
8.8.1 Topology Enumeration of Shared Resources . . . 8-24
8.8.2 Non-Uniform Memory Access . . . 8-24
8.9 OPTIMIZATION OF OTHER SHARED RESOURCES . . . 8-25
8.9.1 Expanded Opportunity for HT Optimization . . . 8-26

CHAPTER 9
64-BIT MODE CODING GUIDELINES

9.1 INTRODUCTION . . . 9-1
9.2 CODING RULES AFFECTING 64-BIT MODE . . . 9-1
9.2.1 Use Legacy 32-Bit Instructions When Data Size Is 32 Bits . . . 9-1
9.2.2 Use Extra Registers to Reduce Register Pressure . . . 9-1
9.2.3 Effective Use of 64-Bit by 64-Bit Multiplies . . . 9-2
9.2.4 Replace 128-bit Integer Division with 128-bit Multiplies . . . 9-2
9.2.5 Sign Extension to Full 64-Bits . . . 9-4
9.3 ALTERNATE CODING RULES FOR 64-BIT MODE . . . 9-5
9.3.1 Use 64-Bit Registers Instead of Two 32-Bit Registers for 64-Bit Arithmetic Result . . . 9-5
9.3.2 CVTSI2SS and CVTSI2SD . . . 9-6
9.3.3 Using Software Prefetch . . . 9-6

CHAPTER 10 SSE4.2 AND SIMD PROGRAMMING FOR TEXT-PROCESSING/LEXING/PARSING

10.1 SSE4.2 STRING AND TEXT INSTRUCTIONS . . . 10-1
10.1.1 CRC32 . . . 10-4
10.2 USING SSE4.2 STRING AND TEXT INSTRUCTIONS . . . 10-5
10.2.1 Unaligned Memory Access and Buffer Size Management . . . 10-5
10.2.2 Unaligned Memory Access and String Library . . . 10-6
10.3 SSE4.2 APPLICATION CODING GUIDELINE AND EXAMPLES . . . 10-6
10.3.1 Null Character Identification (Strlen equivalent) . . . 10-6
10.3.2 White-Space-Like Character Identification . . . 10-9
10.3.3 Substring Searches . . . 10-11
10.3.4 String Token Extraction and Case Handling . . . 10-18
10.3.5 Unicode Processing and PCMPxSTRy . . . 10-22
10.3.6 Replacement String Library Function Using SSE4.2 . . . 10-26
10.4 SSE4.2 ENABLED NUMERICAL AND LEXICAL COMPUTATION . . . 10-28
10.5 NUMERICAL DATA CONVERSION TO ASCII FORMAT . . . 10-34
10.5.1 Large Integer Numeric Computation . . . 10-48
10.5.1.1 MULX Instruction and Large Integer Numeric Computation . . . 10-48

CHAPTER 11
OPTIMIZATIONS FOR INTEL® AVX, FMA AND AVX2

11.1 INTEL® AVX INTRINSICS CODING . . . 11-2
11.1.1 Intel® AVX Assembly Coding . . . 11-4
11.2 NON-DESTRUCTIVE SOURCE (NDS) . . . 11-5
11.3 MIXING AVX CODE WITH SSE CODE . . . 11-7
11.3.1 Mixing Intel® AVX and Intel SSE in Function Calls . . . 11-9
11.4 128-BIT LANE OPERATION AND AVX . . . 11-10
11.4.1 Programming With the Lane Concept . . . 11-11
11.4.2 Strided Load Technique . . . 11-11
11.4.3 The Register Overlap Technique . . . 11-14
11.5 DATA GATHER AND SCATTER . . . 11-15
11.5.1 Data Gather . . . 11-15
11.5.2 Data Scatter . . . 11-17
11.6 DATA ALIGNMENT FOR INTEL® AVX . . . 11-19
11.6.1 Align Data to 32 Bytes . . . 11-19
11.6.2 Consider 16-Byte Memory Access when Memory is Unaligned . . . 11-20
11.6.3 Prefer Aligned Stores Over Aligned Loads . . . 11-22

PAGE

11.7 L1D CACHE LINE REPLACEMENTS . . . 11-22
11.8 4K ALIASING. . . 11-22
11.9 CONDITIONAL SIMD PACKED LOADS AND STORES . . . 11-23
11.9.1 Conditional Loops . . . 11-24
11.10 MIXING INTEGER AND FLOATING-POINT CODE . . . 11-25
11.11 HANDLING PORT 5 PRESSURE . . . 11-28
11.11.1 Replace Shuffles with Blends . . . 11-28
11.11.2 Design Algorithm With Fewer Shuffles . . . 11-30
11.11.3 Perform Basic Shuffles on Load Ports . . . 11-32
11.12 DIVIDE AND SQUARE ROOT OPERATIONS . . . 11-34
11.12.1 Single-Precision Divide . . . 11-35
11.12.2 Single-Precision Reciprocal Square Root. . . . 11-37
11.12.3 Single-Precision Square Root . . . 11-39
11.13 OPTIMIZATION OF ARRAY SUB SUM EXAMPLE . . . 11-41
11.14 HALF-PRECISION FLOATING-POINT CONVERSIONS. . . . 11-43
11.14.1 Packed Single-Precision to Half-Precision Conversion . . . 11-43
11.14.2 Packed Half-Precision to Single-Precision Conversion . . . 11-44
11.14.3 Locality Consideration for using Half-Precision FP to Conserve Bandwidth . . . 11-45
11.15 FUSED MULTIPLY-ADD (FMA) INSTRUCTIONS GUIDELINES. . . . 11-46
11.15.1 Optimizing Throughput with FMA and Floating-Point Add/MUL . . . 11-47
11.15.2 Optimizing Throughput with Vector Shifts . . . 11-48
11.16 AVX2 OPTIMIZATION GUIDELINES . . . 11-49
11.16.1 Multi-Buffering and AVX2 . . . 11-54
11.16.2 Modular Multiplication and AVX2 . . . 11-54
11.16.3 Data Movement Considerations. . . . 11-54
11.16.3.1 SIMD Heuristics to implement Memcpy(). . . . 11-55
11.16.3.2 Memcpy() Implementation Using Enhanced REP MOVSB . . . 11-55
11.16.3.3 Memset() Implementation Considerations . . . 11-56
11.16.3.4 Hoisting Memcpy/Memset Ahead of Consuming Code . . . 11-57
11.16.3.5 256-bit Fetch versus Two 128-bit Fetches. . . . 11-57
11.16.3.6 Mixing MULX and AVX2 Instructions . . . 11-57
11.16.4 Considerations for Gather Instructions . . . 11-64
11.16.4.1 Strided Loads. . . . 11-67
11.16.4.2 Adjacent Loads . . . 11-68
11.16.5 AVX2 Conversion Remedy to MMX Instruction Throughput Limitation . . . 11-69

CHAPTER 12
INTEL® TSX RECOMMENDATIONS

12.1 INTRODUCTION . . . 12-1
12.1.1 Optimization Outline . . . 12-2
12.2 APPLICATION-LEVEL TUNING AND OPTIMIZATIONS. . . . 12-2
12.2.1 Existing TSX-enabled Locking Libraries. . . . 12-3
12.2.1.1 Libraries allowing lock elision for unmodified programs . . . 12-3
12.2.1.2 Libraries requiring program modifications . . . 12-3
12.2.2 Initial Checks . . . 12-3
12.2.3 Run and Profile the Application . . . 12-3
12.2.4 Minimize Transactional Aborts. . . . 12-4
12.2.4.1 Transactional Aborts due to Data Conflicts . . . 12-5
12.2.4.2 Transactional Aborts due to Limited Transactional Resources . . . 12-6
12.2.4.3 Lock Elision Specific Transactional Aborts . . . 12-7
12.2.4.4 HLE Specific Transactional Aborts . . . 12-7
12.2.4.5 Miscellaneous Transactional Aborts. . . . 12-8
12.2.5 Using Transactional-Only Code Paths . . . 12-9
12.2.6 Dealing with Transactional Regions or Paths that Abort at a High Rate . . . 12-9
12.2.6.1 Transitioning to Non-Elided Execution without Aborting . . . 12-9
12.2.6.2 Forcing an Early Abort . . . 12-10
12.2.6.3 Not Eliding Selected Locks. . . . 12-10
12.3 DEVELOPING AN INTEL TSX ENABLED SYNCHRONIZATION LIBRARY. . . . 12-10
12.3.1 Adding HLE Prefixes . . . 12-10
12.3.2 Elision Friendly Critical Section Locks. . . . 12-10
12.3.3 Using HLE or RTM for Lock Elision . . . 12-11
12.3.4 An example wrapper for lock elision using RTM. . . . 12-11
12.3.5 Guidelines for the RTM fallback handler . . . 12-12
12.3.6 Implementing Elision-Friendly Locks using Intel TSX . . . 12-13

PAGE

12.3.6.1 Implementing a Simple Spinlock using HLE. 12-13
12.3.6.2 Implementing Reader-Writer Locks using Intel TSX 12-15
12.3.6.3 Implementing Ticket Locks using Intel TSX 12-15
12.3.6.4 Implementing Queue-Based Locks using Intel TSX 12-15
12.3.7 Eliding Application-Specific Meta-Locks using Intel TSX. 12-16
12.3.8 Avoiding Persistent Non-Elided Execution 12-17
12.3.9 Reading the Value of an Elided Lock in RTM-based libraries 12-18
12.3.10 Intermixing HLE and RTM 12-19
12.4 USING THE PERFORMANCE MONITORING SUPPORT FOR INTEL TSX 12-20
12.4.1 Measuring Transactional Success 12-20
12.4.2 Finding locks to elide and verifying all locks are elided. 12-20
12.4.3 Sampling Transactional Aborts. 12-21
12.4.4 Classifying Aborts using a Profiling Tool. 12-21
12.4.5 XABORT Arguments for RTM fallback handlers 12-22
12.4.6 Call Graphs for Transactional Aborts 12-22
12.4.7 Last Branch Records and Transactional Aborts 12-22
12.4.8 Profiling and Testing Intel TSX Software using the Intel SDE. 12-23
12.4.9 HLE Specific Performance Monitoring Events 12-23
12.4.10 Computing Useful Metrics for Intel TSX 12-24
12.5 PERFORMANCE GUIDELINES 12-25
12.6 DEBUGGING GUIDELINES 12-25
12.7 COMMON INTRINSICS FOR INTEL TSX 12-26
12.7.1 RTM C intrinsics. 12-26
12.7.1.1 Emulated RTM intrinsics on older gcc compatible compilers. 12-26
12.7.2 HLE intrinsics on gcc and other Linux compatible compilers. 12-27
12.7.2.1 Generating HLE intrinsics with gcc4.8 12-28
12.7.2.2 C++11 atomic support 12-28
12.7.2.3 Emulating HLE intrinsics with older gcc-compatible compilers. 12-28
12.7.3 HLE intrinsics on Windows C/C++ compilers 12-29

CHAPTER 13
POWER OPTIMIZATION FOR MOBILE USAGES

13.1 OVERVIEW 13-1
13.2 MOBILE USAGE SCENARIOS. 13-1
13.2.1 Intelligent Energy Efficient Software 13-2
13.3 ACPI C-STATES 13-3
13.3.1 Processor-Specific C4 and Deep C4 States 13-4
13.3.2 Processor-Specific Deep C-States and Intel® Turbo Boost Technology. 13-4
13.3.3 Processor-Specific Deep C-States for Intel® Microarchitecture Code Name Sandy Bridge 13-5
13.3.4 Intel® Turbo Boost Technology 2.0. 13-6
13.4 GUIDELINES FOR EXTENDING BATTERY LIFE 13-6
13.4.1 Adjust Performance to Meet Quality of Features 13-6
13.4.2 Reducing Amount of Work. 13-7
13.4.3 Platform-Level Optimizations. 13-7
13.4.4 Handling Sleep State Transitions 13-8
13.4.5 Using Enhanced Intel SpeedStep® Technology 13-8
13.4.6 Enabling Intel® Enhanced Deeper Sleep. 13-9
13.4.7 Multicore Considerations 13-10
13.4.7.1 Enhanced Intel SpeedStep® Technology 13-10
13.4.7.2 Thread Migration Considerations 13-10
13.4.7.3 Multicore Considerations for C-States. 13-11
13.5 TUNING SOFTWARE FOR INTELLIGENT POWER CONSUMPTION 13-12
13.5.1 Reduction of Active Cycles 13-12
13.5.1.1 Multi-threading to reduce Active Cycles 13-12
13.5.1.2 Vectorization. 13-13
13.5.2 PAUSE and Sleep(0) Loop Optimization. 13-14
13.5.3 Spin-Wait Loops 13-15
13.5.4 Using Event Driven Service Instead of Polling in Code 13-15
13.5.5 Reducing Interrupt Rate. 13-15
13.5.6 Reducing Privileged Time. 13-15
13.5.7 Setting Context Awareness in the Code. 13-16
13.5.8 Saving Energy by Optimizing for Performance 13-17
13.6 PROCESSOR SPECIFIC POWER MANAGEMENT OPTIMIZATION FOR SYSTEM SOFTWARE 13-17
13.6.1 Power Management Recommendation of Processor-Specific Inactive State Configurations 13-17

PAGE
13.6.1.1 Balancing Power Management and Responsiveness of Inactive To Active State Transitions. 13-19

CHAPTER 14
INTEL® ATOM™ MICROARCHITECTURE AND SOFTWARE OPTIMIZATION

14.1 OVERVIEW 14-1
14.2 INTEL® ATOM™ MICROARCHITECTURE 14-1
14.2.1 Hyper-Threading Technology Support in Intel® Atom™ Microarchitecture 14-3
14.3 CODING RECOMMENDATIONS FOR INTEL® ATOM™ MICROARCHITECTURE. 14-3
14.3.1 Optimization for Front End of Intel® Atom™ Microarchitecture 14-3
14.3.2 Optimizing the Execution Core. 14-5
14.3.2.1 Integer Instruction Selection 14-5
14.3.2.2 Address Generation. 14-6
14.3.2.3 Integer Multiply. 14-6
14.3.2.4 Integer Shift Instructions. 14-7
14.3.2.5 Partial Register Access. 14-7
14.3.2.6 FP/SIMD Instruction Selection 14-7
14.3.3 Optimizing Memory Access 14-9
14.3.3.1 Store Forwarding 14-9
14.3.3.2 First-level Data Cache. 14-10
14.3.3.3 Segment Base 14-10
14.3.3.4 String Moves 14-10
14.3.3.5 Parameter Passing. 14-11
14.3.3.6 Function Calls. 14-11
14.3.3.7 Optimization of Multiply/Add Dependent Chains 14-11
14.3.3.8 Position Independent Code 14-13
14.4 INSTRUCTION LATENCY 14-14

CHAPTER 15
SILVERMONT MICROARCHITECTURE AND SOFTWARE OPTIMIZATION

15.1 OVERVIEW 15-1
15.1.1 Intel Atom Processor Family Based on the Silvermont Microarchitecture 15-1
15.2 SILVERMONT MICROARCHITECTURE 15-1
15.2.1 Integer Pipeline. 15-4
15.2.2 Floating-Point Pipeline 15-4
15.3 CODING RECOMMENDATIONS FOR SILVERMONT MICROARCHITECTURE. 15-4
15.3.1 Optimizing The Front End 15-4
15.3.1.1 Instruction Decoder. 15-4
15.3.1.2 Front End High IPC Considerations 15-4
15.3.1.3 Loop Unrolling and Loop Stream Detector 15-5
15.3.2 Optimizing The Execution Core 15-6
15.3.2.1 Scheduling 15-6
15.3.2.2 Address Generation. 15-6
15.3.2.3 FP Multiply-Accumulate-Store Execution 15-6
15.3.2.4 Integer Multiply Execution 15-7
15.3.2.5 Zeroing Idioms. 15-8
15.3.2.6 Flags usage 15-8
15.3.2.7 Instruction Selection 15-9
15.3.3 Optimizing Memory Accesses. 15-11
15.3.3.1 Memory Reissue/Sleep causes. 15-11
15.3.3.2 Store Forwarding 15-11
15.3.3.3 PrefetchW Instruction. 15-11
15.3.3.4 Cache Line Splits and Alignment 15-12
15.3.3.5 Segment Base 15-12
15.3.3.6 Copy and String Copy 15-12
15.4 INSTRUCTION LATENCY 15-12

APPENDIX A
APPLICATION PERFORMANCE TOOLS

A.1 COMPILERS. A-2
A.1.1 Recommended Optimization Settings for Intel® 64 and IA-32 Processors. A-2
A.1.2 Vectorization and Loop Optimization. A-2

PAGE

A.1.2.1 Multithreading with OpenMP* A-3
A.1.2.2 Automatic Multithreading A-3
A.1.3 Inline Expansion of Library Functions (/Oi, /Oi-) A-3
A.1.4 Interprocedural and Profile-Guided Optimizations A-3
A.1.4.1 Interprocedural Optimization (IPO) A-3
A.1.4.2 Profile-Guided Optimization (PGO) A-3
A.1.5 Intel® Cilk Plus A-4
A.2 PERFORMANCE LIBRARIES A-4
A.2.1 Intel® Integrated Performance Primitives (Intel® IPP) A-4
A.2.2 Intel® Math Kernel Library (Intel® MKL) A-5
A.2.3 Intel® Threading Building Blocks (Intel® TBB) A-5
A.2.4 Benefits Summary A-5
A.3 PERFORMANCE PROFILERS A-5
A.3.1 Intel® VTune™ Amplifier XE A-5
A.3.1.1 Hardware Event-Based Sampling Analysis A-6
A.3.1.2 Algorithm Analysis A-6
A.3.1.3 Platform Analysis A-6
A.4 THREAD AND MEMORY CHECKERS A-6
A.4.1 Intel® Inspector A-6
A.5 VECTORIZATION ASSISTANT A-7
A.5.1 Intel® Advisor A-7
A.6 CLUSTER TOOLS A-7
A.6.1 Intel® Trace Analyzer and Collector A-7
A.6.1.1 MPI Performance Snapshot A-7
A.6.2 Intel® MPI Library A-7
A.6.3 Intel® MPI Benchmarks A-8
A.7 INTEL® ACADEMIC COMMUNITY A-8

APPENDIX B
USING PERFORMANCE MONITORING EVENTS

B.1 TOP-DOWN ANALYSIS METHOD B-1
B.1.1 Top-Level B-2
B.1.2 Front End Bound B-3
B.1.3 Back End Bound B-4
B.1.4 Memory Bound B-4
B.1.5 Core Bound B-5
B.1.6 Bad Speculation B-5
B.1.7 Retiring B-6
B.1.8 TMAM and Skylake Microarchitecture B-6
B.1.8.1 TMAM Examples B-6
B.2 INTEL® XEON® PROCESSOR 5500 SERIES B-7
B.3 PERFORMANCE ANALYSIS TECHNIQUES FOR INTEL® XEON® PROCESSOR 5500 SERIES B-8
B.3.1 Cycle Accounting and Uop Flow Analysis B-9
B.3.1.1 Cycle Drill Down and Branch Mispredictions B-10
B.3.1.2 Basic Block Drill Down B-13
B.3.2 Stall Cycle Decomposition and Core Memory Accesses B-14
B.3.2.1 Measuring Costs of Microarchitectural Conditions B-14
B.3.3 Core PMU Precise Events B-15
B.3.3.1 Precise Memory Access Events B-16
B.3.3.2 Load Latency Event B-17
B.3.3.3 Precise Execution Events B-19
B.3.3.4 Last Branch Record (LBR) B-20
B.3.3.5 Measuring Core Memory Access Latency B-22
B.3.3.6 Measuring Per-Core Bandwidth B-24
B.3.3.7 Miscellaneous L1 and L2 Events for Cache Misses B-25
B.3.3.8 TLB Misses B-25
B.3.3.9 L1 Data Cache B-26
B.3.4 Front End Monitoring Events B-26
B.3.4.1 Branch Mispredictions B-26
B.3.4.2 Front End Code Generation Metrics B-26
B.3.5 Uncore Performance Monitoring Events B-27
B.3.5.1 Global Queue Occupancy B-27
B.3.5.2 Global Queue Port Events B-29
B.3.5.3 Global Queue Snoop Events B-29

PAGE

B.3.5.4 L3 Events B-30
B.3.6 Intel QuickPath Interconnect Home Logic (QHL)..... B-30
B.3.7 Measuring Bandwidth From the Uncore B-35
B.4 PERFORMANCE TUNING TECHNIQUES FOR INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE..... B-36
B.4.1 Correlating Performance Bottleneck to Source Location B-36
B.4.2 Hierarchical Top-Down Performance Characterization Methodology and Locating Performance Bottlenecks B-37
B.4.2.1 Back End Bound Characterization..... B-38
B.4.2.2 Core Bound Characterization B-38
B.4.2.3 Memory Bound Characterization..... B-38
B.4.3 Back End Stalls B-39
B.4.4 Memory Sub-System Stalls B-40
B.4.4.1 Accounting for Load Latency B-41
B.4.4.2 Cache-line Replacement Analysis B-42
B.4.4.3 Lock Contention Analysis..... B-43
B.4.4.4 Other Memory Access Issues B-43
B.4.5 Execution Stalls..... B-46
B.4.5.1 Longer Instruction Latencies B-46
B.4.5.2 Assists..... B-46
B.4.6 Bad Speculation B-47
B.4.6.1 Branch Mispredicts..... B-47
B.4.7 Front End Stalls..... B-47
B.4.7.1 Understanding the Micro-op Delivery Rate B-47
B.4.7.2 Understanding the Sources of the Micro-op Queue B-49
B.4.7.3 The Decoded ICache B-50
B.4.7.4 Issues in the Legacy Decode Pipeline B-51
B.4.7.5 Instruction Cache B-51
B.5 USING PERFORMANCE EVENTS OF INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS..... B-52
B.5.1 Understanding the Results in a Performance Counter..... B-52
B.5.2 Ratio Interpretation..... B-52
B.5.3 Notes on Selected Events B-53
B.6 DRILL-DOWN TECHNIQUES FOR PERFORMANCE ANALYSIS..... B-53
B.6.1 Cycle Composition at Issue Port..... B-55
B.6.2 Cycle Composition of OOO Execution..... B-55
B.6.3 Drill-Down on Performance Stalls B-56
B.7 EVENT RATIOS FOR INTEL CORE MICROARCHITECTURE..... B-57
B.7.1 Clocks Per Instructions Retired Ratio (CPI)..... B-57
B.7.2 Front End Ratios..... B-58
B.7.2.1 Code Locality B-58
B.7.2.2 Branching and Front End B-58
B.7.2.3 Stack Pointer Tracker..... B-58
B.7.2.4 Macro-fusion B-58
B.7.2.5 Length Changing Prefix (LCP) Stalls..... B-59
B.7.2.6 Self Modifying Code Detection..... B-59
B.7.3 Branch Prediction Ratios B-59
B.7.3.1 Branch Mispredictions..... B-59
B.7.3.2 Virtual Tables and Indirect Calls..... B-59
B.7.3.3 Mispredicted Returns B-60
B.7.4 Execution Ratios..... B-60
B.7.4.1 Resource Stalls B-60
B.7.4.2 ROB Read Port Stalls..... B-60
B.7.4.3 Partial Register Stalls B-60
B.7.4.4 Partial Flag Stalls B-60
B.7.4.5 Bypass Between Execution Domains..... B-60
B.7.4.6 Floating-Point Performance Ratios B-60
B.7.5 Memory Sub-System - Access Conflicts Ratios..... B-61
B.7.5.1 Loads Blocked by the L1 Data Cache..... B-61
B.7.5.2 4K Aliasing and Store Forwarding Block Detection..... B-61
B.7.5.3 Load Block by Preceding Stores B-61
B.7.5.4 Memory Disambiguation..... B-62
B.7.5.5 Load Operation Address Translation B-62
B.7.6 Memory Sub-System - Cache Misses Ratios..... B-62
B.7.6.1 Locating Cache Misses in the Code..... B-62
B.7.6.2 L1 Data Cache Misses..... B-62
B.7.6.3 L2 Cache Misses B-62

PAGE

B.7.7 Memory Sub-system - Prefetching B-63
B.7.7.1 L1 Data Prefetching B-63
B.7.7.2 L2 Hardware Prefetching B-63
B.7.7.3 Software Prefetching B-63
B.7.8 Memory Sub-system - TLB Miss Ratios B-63
B.7.9 Memory Sub-system - Core Interaction B-64
B.7.9.1 Modified Data Sharing B-64
B.7.9.2 Fast Synchronization Penalty B-64
B.7.9.3 Simultaneous Extensive Stores and Load Misses B-64
B.7.10 Memory Sub-system - Bus Characterization B-64
B.7.10.1 Bus Utilization B-64
B.7.10.2 Modified Cache Lines Eviction B-65

APPENDIX C
INSTRUCTION LATENCY AND THROUGHPUT

C.1 OVERVIEW C-1
C.2 DEFINITIONS C-2
C.3 LATENCY AND THROUGHPUT C-2
C.3.1 Latency and Throughput with Register Operands C-3
C.3.2 Table Footnotes C-19
C.3.3 Instructions with Memory Operands C-20
C.3.3.1 Software Observable Latency of Memory References C-20

EXAMPLES

Example 3-1. Assembly Code with an Unpredictable Branch 3-5
Example 3-2. Code Optimization to Eliminate Branches 3-5
Example 3-3. Eliminating Branch with CMOV Instruction 3-6
Example 3-4. Use of PAUSE Instruction 3-6
Example 3-5. Static Branch Prediction Algorithm 3-7
Example 3-6. Static Taken Prediction 3-7
Example 3-7. Static Not-Taken Prediction 3-7
Example 3-8. Indirect Branch With Two Favored Targets 3-10
Example 3-9. A Peeling Technique to Reduce Indirect Branch Misprediction 3-10
Example 3-10. Loop Unrolling 3-11
Example 3-11. Macro-fusion, Unsigned Iteration Count 3-14
Example 3-12. Macro-fusion, If Statement 3-14
Example 3-13. Macro-fusion, Signed Variable 3-15
Example 3-14. Macro-fusion, Signed Comparison 3-15
Example 3-15. Additional Macro-fusion Benefit in Intel Microarchitecture Code Name Sandy Bridge 3-16
Example 3-16. Avoiding False LCP Delays with 0xF7 Group Instructions 3-17
Example 3-17. Unrolling Loops in LSD to Optimize Emission Bandwidth 3-18
Example 3-18. Independent Two-Operand LEA Example 3-22
Example 3-19. Alternative to Three-Operand LEA 3-22
Example 3-20. Examples of 512-bit Additions 3-23
Example 3-21. Clearing Register to Break Dependency While Negating Array Elements 3-26
Example 3-22. Spill Scheduling Code 3-28
Example 3-23. Zero-Latency MOV Instructions 3-29
Example 3-24. Byte-Granular Data Computation Technique 3-29
Example 3-25. Re-ordering Sequence to Improve Effectiveness of Zero-Latency MOV Instructions 3-30
Example 3-26. Avoiding Partial Register Stalls in Integer Code 3-32
Example 3-27. Avoiding Partial Register Stalls in SIMD Code 3-33
Example 3-28. Avoiding Partial Flag Register Stalls 3-34
Example 3-29. Partial Flag Register Accesses in Intel Microarchitecture Code Name Sandy Bridge 3-34
Example 3-30. Reference Code Template for Partially Vectorizable Program 3-36
Example 3-31. Three Alternate Packing Methods for Avoiding Store Forwarding Difficulty 3-37
Example 3-32. Using Four Registers to Reduce Memory Spills and Simplify Result Passing 3-38
Example 3-33. Stack Optimization Technique to Simplify Parameter Passing 3-38
Example 3-34. Base Line Code Sequence to Estimate Loop Overhead 3-39

PAGE

Example 3-35. Optimize for Load Port Bandwidth in Intel Microarchitecture Code Name Sandy Bridge 3-41
Example 3-36. Index versus Pointers in Pointer-Chasing Code 3-42
Example 3-37. Example of Bank Conflicts in L1D Cache and Remedy 3-43
Example 3-38. Using XMM Register in Lieu of Memory for Register Spills 3-44
Example 3-39. Loads Blocked by Stores of Unknown Address 3-45
Example 3-40. Code That Causes Cache Line Split 3-46
Example 3-41. Situations Showing Small Loads After Large Store 3-49
Example 3-42. Non-forwarding Example of Large Load After Small Store 3-49
Example 3-43. A Non-forwarding Situation in Compiler Generated Code 3-49
Example 3-44. Two Ways to Avoid Non-forwarding Situation in Example 3-43. 3-50
Example 3-45. Large and Small Load Stalls. 3-50
Example 3-46. Loop-carried Dependence Chain 3-52
Example 3-47. Rearranging a Data Structure 3-52
Example 3-48. Decomposing an Array 3-53
Example 3-49. Examples of Dynamical Stack Alignment 3-54
Example 3-50. Aliasing Between Loads and Stores Across Loop Iterations. 3-56
Example 3-51. Instruction Pointer Query Techniques. 3-57
Example 3-52. Using Non-temporal Stores and 64-byte Bus Write Transactions 3-60
Example 3-53. On-temporal Stores and Partial Bus Write Transactions 3-60
Example 3-54. Using DCU Hardware Prefetch. 3-61
Example 3-55. Avoid Causing DCU Hardware Prefetch to Fetch Un-needed Lines 3-62
Example 3-56. Technique For Using L1 Hardware Prefetch 3-63
Example 3-57. REP STOSD with Arbitrary Count Size and 4-Byte-Aligned Destination 3-65
Example 3-58. Algorithm to Avoid Changing Rounding Mode 3-72
Example 4-1. Identification of MMX Technology with CPUID 4-2
Example 4-2. Identification of SSE with CPUID 4-2
Example 4-3. Identification of SSE2 with cpuid 4-3
Example 4-4. Identification of SSE3 with CPUID. 4-3
Example 4-5. Identification of SSSE3 with cpuid 4-3
Example 4-6. Identification of SSE4.1 with cpuid. 4-4
Example 4-7. Identification of SSE4.2 with cpuid. 4-4
Example 4-8. Detection of AESNI Instructions 4-5
Example 4-9. Detection of PCLMULQDQ Instruction 4-5
Example 4-10. Detection of AVX Instruction 4-6
Example 4-11. Detection of VEX-Encoded AESNI Instructions 4-7
Example 4-12. Detection of VEX-Encoded AESNI Instructions 4-7
Example 4-13. Simple Four-Iteration Loop 4-14
Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding 4-14
Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics 4-15
Example 4-16. C++ Code Using the Vector Classes 4-16
Example 4-17. Automatic Vectorization for a Simple Loop 4-16
Example 4-18. C Algorithm for 64-bit Data Alignment 4-18
Example 4-19. AoS Data Structure 4-21
Example 4-20. SoA Data Structure 4-21
Example 4-21. AoS and SoA Code Samples. 4-21
Example 4-22. Hybrid SoA Data Structure 4-22
Example 4-23. Pseudo-code Before Strip Mining 4-23
Example 4-24. Strip Mined Code. 4-24
Example 4-25. Loop Blocking. 4-24
Example 4-26. Emulation of Conditional Moves. 4-26
Example 5-1. Resetting Register Between __m64 and FP Data Types Code 5-3
Example 5-2. FIR Processing Example in C language Code 5-4
Example 5-3. SSE2 and SSSE3 Implementation of FIR Processing Code. 5-4
Example 5-4. Zero Extend 16-bit Values into 32 Bits Using Unsigned Unpack Instructions Code 5-5
Example 5-5. Signed Unpack Code 5-5
Example 5-6. Interleaved Pack with Saturation Code. 5-7
Example 5-7. Interleaved Pack without Saturation Code 5-7
Example 5-8. Unpacking Two Packed-word Sources in Non-interleaved Way Code 5-9
Example 5-9. PEXTRW Instruction Code 5-10

PAGE

Example 5-10. PINSRW Instruction Code 5-10
Example 5-11. Repeated PINSRW Instruction Code 5-11
Example 5-12. Non-Unit Stride Load/Store Using SSE4.1 Instructions 5-11
Example 5-13. Scatter and Gather Operations Using SSE4.1 Instructions 5-11
Example 5-14. PMOVMSKB Instruction Code 5-12
Example 5-15. Broadcast a Word Across XMM, Using 2 SSE2 Instructions. 5-13
Example 5-16. Swap/Reverse words in an XMM, Using 3 SSE2 Instructions 5-13
Example 5-17. Generating Constants 5-15
Example 5-18. Absolute Difference of Two Unsigned Numbers 5-15
Example 5-19. Absolute Difference of Signed Numbers. 5-16
Example 5-20. Computing Absolute Value 5-16
Example 5-21. Basic C Implementation of RGBA to BGRA Conversion 5-17
Example 5-22. Color Pixel Format Conversion Using SSE2 5-17
Example 5-23. Color Pixel Format Conversion Using SSSE3 5-18
Example 5-24. Big-Endian to Little-Endian Conversion 5-19
Example 5-25. Clipping to a Signed Range of Words [High, Low]. 5-20
Example 5-26. Clipping to an Arbitrary Signed Range [High, Low] 5-20
Example 5-27. Simplified Clipping to an Arbitrary Signed Range 5-20
Example 5-28. Clipping to an Arbitrary Unsigned Range [High, Low] 5-21
Example 5-29. Complex Multiply by a Constant. 5-23
Example 5-30. Using PTEST to Separate Vectorizable and non-Vectorizable Loop Iterations. 5-24
Example 5-31. Using PTEST and Variable BLEND to Vectorize Heterogeneous Loops 5-24
Example 5-32. Baseline C Code for Mandelbrot Set Map Evaluation. 5-25
Example 5-33. Vectorized Mandelbrot Set Map Evaluation Using SSE4.1 Intrinsics 5-26
Example 5-34. A Large Load after a Series of Small Stores (Penalty) 5-28
Example 5-35. Accessing Data Without Delay 5-28
Example 5-36. A Series of Small Loads After a Large Store 5-28
Example 5-37. Eliminating Delay for a Series of Small Loads after a Large Store 5-29
Example 5-38. An Example of Video Processing with Cache Line Splits 5-29
Example 5-39. Video Processing Using LDDQU to Avoid Cache Line Splits. 5-30
Example 5-40. Un-optimized Reverse Memory Copy in C. 5-31
Example 5-41. Using PSHUFB to Reverse Byte Ordering 16 Bytes at a Time. 5-33
Example 5-42. PMOVSX/PMOVZX Work-around to Avoid False Dependency 5-35
Example 5-43. Table Look-up Operations in C Code. 5-35
Example 5-44. Shift Techniques on Non-Vectorizable Table Look-up 5-36
Example 5-45. PEXTRD Techniques on Non-Vectorizable Table Look-up. 5-37
Example 5-46. Pseudo-Code Flow of AES Counter Mode Operation. 5-39
Example 5-47. AES128-CTR Implementation with Eight Block in Parallel. 5-39
Example 5-48. AES128 Key Expansion 5-46
Example 5-49. Compress 32-bit Integers into 5-bit Buckets. 5-49
Example 5-50. Decompression of a Stream of 5-bit Integers into 32-bit Elements 5-51
Example 6-1. Pseudocode for Horizontal (xyz, AoS) Computation 6-4
Example 6-2. Pseudocode for Vertical (xxxx, yyyy, zzzz, SoA) Computation 6-5
Example 6-3. Swizzling Data Using SHUFPS, MOVLHPS, MOVHLPS 6-5
Example 6-4. Swizzling Data Using UNPCKxxx Instructions 6-6
Example 6-5. Deswizzling Single-Precision SIMD Data 6-7
Example 6-6. Deswizzling Data Using SIMD Integer Instructions 6-8
Example 6-7. Horizontal Add Using MOVHLPS/MOVLHPS 6-9
Example 6-8. Horizontal Add Using Intrinsics with MOVHLPS/MOVLHPS. 6-10
Example 6-9. Multiplication of Two Pair of Single-precision Complex Number 6-12
Example 6-10. Division of Two Pair of Single-precision Complex Numbers 6-12
Example 6-11. Double-Precision Complex Multiplication of Two Pairs. 6-13
Example 6-12. Double-Precision Complex Multiplication Using Scalar SSE2. 6-13
Example 6-13. Dot Product of Vector Length 4 Using SSE/SSE2 6-14
Example 6-14. Dot Product of Vector Length 4 Using SSE3 6-15
Example 6-15. Dot Product of Vector Length 4 Using SSE4.1 6-15
Example 6-16. Unrolled Implementation of Four Dot Products 6-15
Example 6-17. Normalization of an Array of Vectors 6-16
Example 6-18. Normalize (x, y, z) Components of an Array of Vectors Using SSE2 6-17

PAGE

Example 6-19. Normalize (x, y, z) Components of an Array of Vectors Using SSE4.1 . . . 6-18
Example 6-20. Data Organization in Memory for AOS Vector-Matrix Multiplication . . . 6-19
Example 6-21. AOS Vector-Matrix Multiplication with HADDPS . . . 6-19
Example 6-22. AOS Vector-Matrix Multiplication with DPPS . . . 6-20
Example 6-23. Data Organization in Memory for SOA Vector-Matrix Multiplication . . . 6-21
Example 6-24. Vector-Matrix Multiplication with Native SOA Data Layout . . . 6-22
Example 7-1. Pseudo-code Using CLFLUSH . . . 7-9
Example 7-2. Flushing Cache Lines Using CLFLUSH or CLFLUSHOPT . . . 7-11
Example 7-3. Populating an Array for Circular Pointer Chasing with Constant Stride . . . 7-12
Example 7-4. Prefetch Scheduling Distance . . . 7-15
Example 7-5. Using Prefetch Concatenation . . . 7-16
Example 7-6. Concatenation and Unrolling the Last Iteration of Inner Loop . . . 7-17
Example 7-7. Data Access of a 3D Geometry Engine without Strip-mining . . . 7-21
Example 7-8. Data Access of a 3D Geometry Engine with Strip-mining . . . 7-22
Example 7-9. Using HW Prefetch to Improve Read-Once Memory Traffic . . . 7-23
Example 7-10. Basic Algorithm of a Simple Memory Copy . . . 7-27
Example 7-11. A Memory Copy Routine Using Software Prefetch . . . 7-28
Example 7-12. Memory Copy Using Hardware Prefetch and Bus Segmentation . . . 7-29
Example 8-1. Serial Execution of Producer and Consumer Work Items . . . 8-5
Example 8-2. Basic Structure of Implementing Producer Consumer Threads . . . 8-6
Example 8-3. Thread Function for an Interlaced Producer Consumer Model . . . 8-7
Example 8-4. Spin-wait Loop and PAUSE Instructions . . . 8-12
Example 8-5. Coding Pitfall using Spin Wait Loop . . . 8-14
Example 8-6. Placement of Synchronization and Regular Variables . . . 8-15
Example 8-7. Declaring Synchronization Variables without Sharing a Cache Line . . . 8-16
Example 8-8. Batched Implementation of the Producer Consumer Threads . . . 8-21
Example 8-9. Parallel Memory Initialization Technique Using OpenMP and NUMA . . . 8-25
Example 9-1. Compute 64-bit Quotient and Remainder with 64-bit Divisor . . . 9-3
Example 9-2. Quotient and Remainder of 128-bit Dividend with 64-bit Divisor . . . 9-4
Example 10-1. A Hash Function Examples . . . 10-4
Example 10-2. Hash Function Using CRC32 . . . 10-4
Example 10-3. Strlen() Using General-Purpose Instructions . . . 10-6
Example 10-4. Sub-optimal PCMPISTRI Implementation of EOS handling . . . 10-8
Example 10-5. Strlen() Using PCMPISTRI without Loop-Carry Dependency . . . 10-8
Example 10-6. WordCnt() Using C and Byte-Scanning Technique . . . 10-9
Example 10-7. WordCnt() Using PCMPISTRM . . . 10-10
Example 10-8. KMP Substring Search in C . . . 10-12
Example 10-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic . . . 10-13
Example 10-10.Substring Search Using PCMPISTRI and KMP Overlap Table . . . 10-15
Example 10-11.I Equivalent Strtok_s() Using PCMPISTRI Intrinsic . . . 10-19
Example 10-12.I Equivalent Strupr() Using PCMPISTRM Intrinsic . . . 10-21
Example 10-13.UTF16 VerStrlen() Using C and Table Lookup Technique . . . 10-22
Example 10-14.Assembly Listings of UTF16 VerStrlen() Using PCMPISTRI . . . 10-23
Example 10-15.Intrinsic Listings of UTF16 VerStrlen() Using PCMPISTRI . . . 10-25
Example 10-16.Replacement String Library Strcmp Using SSE4.2 . . . 10-27
Example 10-17.High-level flow of Character Subset Validation for String Conversion . . . 10-29
Example 10-18.Intrinsic Listings of atol() Replacement Using PCMPISTRI . . . 10-29
Example 10-19.Auxiliary Routines and Data Constants Used in sse4i_atol() listing . . . 10-31
Example 10-20.Conversion of 64-bit Integer to ASCII . . . 10-34
Example 10-21.Conversion of 64-bit Integer to ASCII without Integer Division . . . 10-35
Example 10-22.Conversion of 64-bit Integer to ASCII Using SSE4 . . . 10-37
Example 10-23.Conversion of 64-bit Integer to Wide Character String Using SSE4 . . . 10-43
Example 10-24. MULX and Carry Chain in Large Integer Numeric . . . 10-48
Example 10-25. Building-block Macro Used in Binary Decimal Floating-point Operations . . . 10-49
Example 11-1. Cartesian Coordinate Transformation with Intrinsics . . . 11-3
Example 11-2. Cartesian Coordinate Transformation with Assembly . . . 11-4
Example 11-3. Direct Polynomial Calculation . . . 11-6
Example 11-4. Function Calls and AVX/SSE transitions . . . 11-10
Example 11-5. AoS to SoA Conversion of Complex Numbers in C Code . . . 11-12

PAGE

Example 11-6. Aos to SoA Conversion of Complex Numbers Using AVX 11-13
Example 11-7. Register Overlap Method for Median of 3 Numbers 11-15
Example 11-8. Data Gather - AVX versus Scalar Code 11-16
Example 11-9. Scatter Operation Using AVX 11-18
Example 11-10.SAXPY using Intel AVX 11-19
Example 11-11.Using 16-Byte Memory Operations for Unaligned 32-Byte Memory Operation 11-21
Example 11-12.SAXPY Implementations for Unaligned Data Addresses 11-21
Example 11-13.Loop with Conditional Expression 11-24
Example 11-14.Handling Loop Conditional with VMASKMOV 11-24
Example 11-15.Three-Tap Filter in C Code 11-25
Example 11-16.Three-Tap Filter with 128-bit Mixed Integer and FP SIMD 11-26
Example 11-17.256-bit AVX Three-Tap Filter Code with VSHUFPS 11-26
Example 11-18.Three-Tap Filter Code with Mixed 256-bit AVX and 128-bit AVX Code 11-27
Example 11-19.8x8 Matrix Transpose - Replace Shuffles with Blends 11-29
Example 11-20.8x8 Matrix Transpose Using VINSRTPS 11-31
Example 11-21.Port 5 versus Load Port Shuffles 11-33
Example 11-22.Divide Using DIVPS for 24-bit Accuracy 11-36
Example 11-23.Divide Using RCPPS 11-bit Approximation 11-36
Example 11-24.Divide Using RCPPS and Newton-Raphson Iteration 11-36
Example 11-25.Reciprocal Square Root Using DIVPS+SQRTPS for 24-bit Accuracy 11-38
Example 11-26.Reciprocal Square Root Using RCPPS 11-bit Approximation 11-38
Example 11-27.Reciprocal Square Root Using RCPPS and Newton-Raphson Iteration 11-38
Example 11-28.Square Root Using SQRTPS for 24-bit Accuracy 11-39
Example 11-29. Square Root Using RCPPS 11-bit Approximation 11-40
Example 11-30. Square Root Using RCPPS and One Taylor Series Expansion 11-40
Example 11-31. Array Sub Sums Algorithm 11-42
Example 11-32. Single-Precision to Half-Precision Conversion 11-43
Example 11-33. Half-Precision to Single-Precision Conversion 11-44
Example 11-34. Performance Comparison of Median3 using Half-Precision vs. Single-Precision 11-45
Example 11-35. FP Mul/FP Add Versus FMA 11-47
Example 11-36. Unrolling to Hide Dependent FP Add Latency 11-47
Example 11-37. FP Mul/FP Add Versus FMA 11-49
Example 11-38. Macros for Separable KLT Intra-block Transformation Using AVX2 11-50
Example 11-39. Separable KLT Intra-block Transformation Using AVX2 11-52
Example 11-40. Macros for Parallel Moduli/Remainder Calculation 11-57
Example 11-41. Signed 64-bit Integer Conversion Utility 11-58
Example 11-42. Unsigned 63-bit Integer Conversion Utility 11-60
Example 11-43. Access Patterns Favoring Non-VGATHER Techniques 11-64
Example 11-44. Access Patterns Likely to Favor VGATHER Techniques 11-65
Example 11-45. Software AVX Sequence Equivalent to Full-Mask VPGATHERD 11-66
Example 11-46.AOS to SOA Transformation Alternatives 11-67
Example 11-47. Non-Strided AOS to SOA 11-68
Example 11-48. Conversion to Throughput-Reduced MMX sequence to AVX2 Alternative 11-70
Example 12-1. Reduce Data Conflict with Conditional Updates 12-6
Example 12-2. Transition from Non-Elided Execution without Aborting 12-10
Example 12-3. Exemplary Wrapper Using RTM for Lock/Unlock Primitives 12-12
Example 12-4. Spin Lock Example Using HLE in GCC 4.8 and Later 12-14
Example 12-5. Spin Lock Example Using HLE in Intel and Microsoft Compiler Intrinsic 12-14
Example 12-6. A Meta Lock Example 12-16
Example 12-7. A Meta Lock Example Using RTM 12-17
Example 12-8. HLE-enabled Lock-Acquire/ Lock-Release Sequence 12-17
Example 12-9. A Spin Wait Example Using HLE 12-18
Example 12-10. A Conceptual Example of Intermixed HLE and RTM 12-19
Example 12-11. Emulated RTM intrinsic for Older GCC compilers 12-27
Example 12-12. C++ Example of HLE Intrinsic 12-28
Example 12-13. Emulated HLE Intrinsic with Older GCC compiler 12-28
Example 12-14. HLE Intrinsic Supported by Intel and Microsoft Compilers 12-29
Example 13-1. Unoptimized Sleep Loop 13-14
Example 13-2. Power Consumption Friendly Sleep Loop Using PAUSE 13-14

PAGE

Example 14-1. Instruction Pairing and Alignment to Optimize Decode Throughput on Intel® Atom™ Microarchitecture . . . 14-4
Example 14-2. Alternative to Prevent AGU and Execution Unit Dependency . . . 14-6
Example 14-3. Pipeling Instruction Execution in Integer Computation . . . 14-7
Example 14-4. Memory Copy of 64-byte . . . 14-11
Example 14-5. Examples of Dependent Multiply and Add Computation . . . 14-12
Example 14-6. Instruction Pointer Query Techniques . . . 14-13
Example 15-1. Unrolled Loop Executes In-Order Due to Multiply-Store Port Conflict . . . 15-7
Example 15-2. Grouping Store Instructions Eliminates Bubbles and Improves IPC . . . 15-7

PAGE

FIGURES

Figure 2-1. CPU Core Pipeline Functionality of the Skylake Microarchitecture. 2-2
Figure 2-2. CPU Core Pipeline Functionality of the Haswell Microarchitecture 2-6
Figure 2-3. Four Core System Integration of the Haswell Microarchitecture. 2-7
Figure 2-4. An Example of the Haswell-E Microarchitecture Supporting 12 Processor Cores 2-11
Figure 2-5. Intel Microarchitecture Code Name Sandy Bridge Pipeline Functionality 2-14
Figure 2-6. Intel Core Microarchitecture Pipeline Functionality. 2-32
Figure 2-7. Execution Core of Intel Core Microarchitecture 2-38
Figure 2-8. Store-Forwarding Enhancements in Enhanced Intel Core Microarchitecture 2-41
Figure 2-9. Intel Advanced Smart Cache Architecture. 2-42
Figure 2-10. Intel Microarchitecture Code Name Nehalem Pipeline Functionality 2-45
Figure 2-11. Front End of Intel Microarchitecture Code Name Nehalem 2-46
Figure 2-12. Store-Forwarding Scenarios of 16-Byte Store Operations. 2-51
Figure 2-13. Store-Forwarding Enhancement in Intel Microarchitecture Code Name Nehalem. 2-52
Figure 2-14. Hyper-Threading Technology on an SMP 2-54
Figure 2-15. Typical SIMD Operations 2-57
Figure 2-16. SIMD Instruction Register Usage. 2-58
Figure 3-1. Generic Program Flow of Partially Vectorized Code 3-36
Figure 3-2. Cache Line Split in Accessing Elements in a Array. 3-46
Figure 3-3. Size and Alignment Restrictions in Store Forwarding 3-48
Figure 3-4. Memcpy Performance Comparison for Lengths up to 2KB. 3-66
Figure 4-1. General Procedural Flow of Application Detection of AVX. 4-6
Figure 4-2. General Procedural Flow of Application Detection of Float-16 4-8
Figure 4-3. Converting to Streaming SIMD Extensions Chart. 4-11
Figure 4-4. Hand-Coded Assembly and High-Level Compiler Performance Trade-offs 4-13
Figure 4-5. Loop Blocking Access Pattern 4-26
Figure 5-1. PACKSSDW mm, mm/mm64 Instruction 5-6
Figure 5-2. Interleaved Pack with Saturation 5-7
Figure 5-3. Result of Non-Interleaved Unpack Low in MM0 5-8
Figure 5-4. Result of Non-Interleaved Unpack High in MM1. 5-8
Figure 5-5. PEXTRW Instruction 5-9
Figure 5-6. PINSRW Instruction. 5-10
Figure 5-7. PMOVSMKB Instruction 5-12
Figure 5-8. Data Alignment of Loads and Stores in Reverse Memory Copy 5-32
Figure 5-9. A Technique to Avoid Cacheline Split Loads in Reverse Memory Copy Using Two Aligned Loads 5-33
Figure 6-1. Homogeneous Operation on Parallel Data Elements. 6-3
Figure 6-2. Horizontal Computation Model. 6-3
Figure 6-3. Dot Product Operation 6-4
Figure 6-4. Horizontal Add Using MOVHLPS/MOVLHPS. 6-9
Figure 6-5. Asymmetric Arithmetic Operation of the SSE3 Instruction 6-11
Figure 6-6. Horizontal Arithmetic Operation of the SSE3 Instruction HADDPD 6-11
Figure 7-1. CLFLUSHOPT versus CLFLUSH In SkyLake Microarchitecture 7-10
Figure 7-2. Effective Latency Reduction as a Function of Access Stride 7-13
Figure 7-3. Memory Access Latency and Execution Without Prefetch 7-14
Figure 7-4. Memory Access Latency and Execution With Prefetch 7-14
Figure 7-5. Prefetch and Loop Unrolling. 7-17
Figure 7-6. Memory Access Latency and Execution With Prefetch 7-18
Figure 7-7. Spread Prefetch Instructions 7-19
Figure 7-8. Cache Blocking – Temporally Adjacent and Non-adjacent Passes. 7-20
Figure 7-9. Examples of Prefetch and Strip-mining for Temporally Adjacent and Non-Adjacent Passes Loops . . . 7-21
Figure 7-10. Single-Pass Vs. Multi-Pass 3D Geometry Engines 7-25
Figure 8-1. Amdahl's Law and MP Speed-up 8-2
Figure 8-2. Single-threaded Execution of Producer-consumer Threading Model. 8-5
Figure 8-3. Execution of Producer-consumer Threading Model on a Multicore Processor 8-5
Figure 8-4. Interlaced Variation of the Producer Consumer Model. 8-6
Figure 8-5. Batched Approach of Producer Consumer Model. 8-21
Figure 10-1. SSE4.2 String/Text Instruction Immediate Operand Control 10-2
Figure 10-2. Retrace Inefficiency of Byte-Granular, Brute-Force Search 10-12

PAGE

Figure 10-3. SSE4.2 Speedup of SubString Searches 10-18
Figure 10-4. Compute Four Remainders of Unsigned Short Integer in Parallel. 10-37
Figure 11-1. AVX-SSE Transitions in the Broadwell, and Prior Generation Microarchitectures 11-8
Figure 11-2. AVX-SSE Transitions in the Skylake Microarchitecture 11-8
Figure 11-3. 4x4 Image Block Transformation. 11-50
Figure 11-4. Throughput Comparison of Gather Instructions 11-65
Figure 11-5. Comparison of HW GATHER Versus Software Sequence in Skylake Microarchitecture. 11-66
Figure 13-1. Performance History and State Transitions 13-2
Figure 13-2. Active Time Versus Halted Time of a Processor 13-3
Figure 13-3. Application of C-states to Idle Time 13-4
Figure 13-4. Profiles of Coarse Task Scheduling and Power Consumption 13-9
Figure 13-5. Thread Migration in a Multicore Processor. 13-11
Figure 13-6. Progression to Deeper Sleep 13-11
Figure 13-7. Energy Saving due to Performance Optimization 13-13
Figure 13-8. Energy Saving due to Vectorization. 13-13
Figure 13-9. Energy Saving Comparison of Synchronization Primitives 13-16
Figure 13-10. Power Saving Comparison of Power-Source-Aware Frame Rate Configurations 13-17
Figure 14-1. Intel Atom Microarchitecture Pipeline 14-2
Figure 15-1. Silvermont Microarchitecture Pipeline. 15-2
Figure B-1. General TMAM Hierarchy for Out-of-Order Microarchitectures. B-2
Figure B-2. TMAM's Top Level Drill Down Flowchart. B-3
Figure B-3. TMAM Hierarchy Supported by Skylake Microarchitecture. B-7
Figure B-4. System Topology Supported by Intel® Xeon® Processor 5500 Series. B-8
Figure B-5. PMU Specific Event Logic Within the Pipeline B-10
Figure B-6. LBR Records and Basic Blocks. B-21
Figure B-7. Using LBR Records to Rectify Skewed Sample Distribution B-21
Figure B-8. RdData Request after LLC Miss to Local Home (Clean Rsp). B-32
Figure B-9. RdData Request after LLC Miss to Remote Home (Clean Rsp) B-32
Figure B-11. RdData Request after LLC Miss to Local Home (Hitm Response) B-33
Figure B-10. RdData Request after LLC Miss to Remote Home (Hitm Response). B-33
Figure B-12. RdData Request after LLC Miss to Local Home (Hit Response) B-34
Figure B-13. RdInvOwn Request after LLC Miss to Remote Home (Clean Res) B-34
Figure B-15. RdInvOwn Request after LLC Miss to Local Home (Hit Res) B-35
Figure B-14. RdInvOwn Request after LLC Miss to Remote Home (Hitm Res). B-35
Figure B-16. Performance Events Drill-Down and Software Tuning Feedback Loop B-54

PAGE

TABLES

Table 2-1. Dispatch Port and Execution Stacks of the Skylake Microarchitecture 2-3
Table 2-2. Skylake Microarchitecture Execution Units and Representative Instructions 2-4
Table 2-3. Bypass Delay Between Producer and Consumer Micro-ops 2-4
Table 2-4. Cache Parameters of the Skylake Microarchitecture 2-5
Table 2-5. TLB Parameters of the Skylake Microarchitecture 2-6
Table 2-6. Dispatch Port and Execution Stacks of the Haswell Microarchitecture 2-8
Table 2-7. Haswell Microarchitecture Execution Units and Representative Instructions 2-9
Table 2-8. Bypass Delay Between Producer and Consumer Micro-ops (cycles) 2-9
Table 2-9. Cache Parameters of the Haswell Microarchitecture 2-10
Table 2-10. TLB Parameters of the Haswell Microarchitecture 2-10
Table 2-11. TLB Parameters of the Broadwell Microarchitecture 2-12
Table 2-12. Components of the Front End of Intel Microarchitecture Code Name Sandy Bridge 2-15
Table 2-13. ICache and ITLB of Intel Microarchitecture Code Name Sandy Bridge 2-15
Table 2-14. Dispatch Port and Execution Stacks 2-21
Table 2-15. Execution Core Writeback Latency (cycles) 2-22
Table 2-16. Cache Parameters 2-22
Table 2-17. Lookup Order and Load Latency 2-23
Table 2-18. L1 Data Cache Components 2-24
Table 2-19. Effect of Addressing Modes on Load Latency 2-25
Table 2-20. DTLB and STLB Parameters 2-25
Table 2-21. Store Forwarding Conditions (1 and 2 byte stores) 2-26
Table 2-22. Store Forwarding Conditions (4-16 byte stores) 2-26
Table 2-23. 32-byte Store Forwarding Conditions (0-15 byte alignment) 2-26
Table 2-24. 32-byte Store Forwarding Conditions (16-31 byte alignment) 2-27
Table 2-25. Components of the Front End 2-32
Table 2-26. Issue Ports of Intel Core Microarchitecture and Enhanced Intel Core Microarchitecture 2-37
Table 2-27. Cache Parameters of Processors based on Intel Core Microarchitecture 2-43
Table 2-28. Characteristics of Load and Store Operations in Intel Core Microarchitecture 2-43
Table 2-29. Bypass Delay Between Producer and Consumer Micro-ops (cycles) 2-47
Table 2-30. Issue Ports of Intel Microarchitecture Code Name Nehalem 2-48
Table 2-31. Cache Parameters of Intel Core i7 Processors 2-49
Table 2-32. Performance Impact of Address Alignments of MOVDQU from L1 2-50
Table 3-1. Macro-Fusible Instructions in Intel Microarchitecture Code Name Sandy Bridge 3-13
Table 3-2. Small Loop Criteria Detected by Sandy Bridge and Haswell Microarchitectures 3-18
Table 3-3. Store Forwarding Restrictions of Processors Based on Intel Core Microarchitecture 3-50
Table 3-4. Relative Performance of Memcpy() Using ERMSB Vs. 128-bit AVX 3-67
Table 3-5. Effect of Address Misalignment on Memcpy() Performance 3-67
Table 5-1. PSHUF Encoding 5-13
Table 6-1. SoA Form of Representing Vertices Data 6-4
Table 7-1. Software Prefetching Considerations into Strip-mining Code 7-23
Table 7-2. Relative Performance of Memory Copy Routines 7-30
Table 7-3. Deterministic Cache Parameters Leaf 7-31
Table 8-1. Properties of Synchronization Objects 8-11
Table 8-2. Design-Time Resource Management Choices 8-23
Table 8-3. Microarchitectural Resources Comparisons of HT Implementations 8-26
Table 10-1. SSE4.2 String/Text Instructions Compare Operation on N-elements 10-2
Table 10-2. SSE4.2 String/Text Instructions Unary Transformation on IntRes1 10-3
Table 10-3. SSE4.2 String/Text Instructions Output Selection Imm[6] 10-3
Table 10-4. SSE4.2 String/Text Instructions Element-Pair Comparison Definition 10-3
Table 10-5. SSE4.2 String/Text Instructions Eflags Behavior 10-3
Table 11-1. Features between 256-bit AVX, 128-bit AVX and Legacy SSE Extensions 11-2
Table 11-2. State Transitions of Mixing AVX and SSE Code 11-9
Table 11-3. Approximate Magnitude of AVX-SSE Transition Penalties in Different Microarchitectures 11-9
Table 11-4. Effect of VZEROUPPER with Inter-Function Calls Between AVX and SSE Code 11-10
Table 11-5. Comparison of Numeric Alternatives of Selected Linear Algebra in Skylake Microarchitecture 11-34
Table 11-6. Single-Precision Divide and Square Root Alternatives 11-35
Table 11-7. Comparison of Single-Precision Divide Alternatives 11-37

PAGE

Table 11-8. Comparison of Single-Precision Reciprocal Square Root Operation 11-39
Table 11-9. Comparison of Single-Precision Square Root Operation 11-41
Table 11-10. Comparison of AOS to SOA with Strided Access Pattern 11-68
Table 11-11. Comparison of Indexed AOS to SOA Transformation 11-69
Table 12-1. RTM Abort Status Definition. 12-22
Table 13-1. ACPI C-State Type Mappings to Processor Specific C-State for Mobile Processors Based on Intel Microarchitecture Code Name Nehalem 13-5
Table 13-2. ACPI C-State Type Mappings to Processor Specific C-State of Intel Microarchitecture Code Name Sandy Bridge. 13-5
Table 13-3. C-State Total Processor Exit Latency for Client Systems (Core+ Package Exit Latency) with Slow VR13-18
Table 13-4. C-State Total Processor Exit Latency for Client Systems (Core+ Package Exit Latency) with Fast VR13-18
Table 13-5. C-State Core-Only Exit Latency for Client Systems with Slow VR. 13-18
Table 13-6. POWER_CTL MSR in Next Generation Intel Processor (Intel® Microarchitecture Code Name Sandy Bridge) 13-19
Table 14-1. Instruction Latency/Throughput Summary of Intel® Atom™ Microarchitecture 14-7
Table 14-2. Intel® Atom™ Microarchitecture Instructions Latency Data. 14-14
Table 15-1. Function Unit Mapping of the Silvermont Microarchitecture 15-3
Table 15-2. Integer Multiply Operation Latency 15-8
Table 15-3. Floating-Point and SIMD Integer Latency. 15-9
Table 15-4. Unsigned Integer Division Operation Latency. 15-10
Table 15-5. Signed Integer Division Operation Latency 15-10
Table 15-6. Silvermont Microarchitecture Instructions Latency and Throughput 15-13
Table A-1. Recommended Processor Optimization Options. A-2
Table B-1. Cycle Accounting and Micro-ops Flow Recipe. B-9
Table B-2. CMask/Inv/Edge/Thread Granularity of Events for Micro-op Flow B-10
Table B-3. Cycle Accounting of Wasted Work Due to Misprediction B-11
Table B-4. Cycle Accounting of Instruction Starvation. B-12
Table B-5. CMask/Inv/Edge/Thread Granularity of Events for Micro-op Flow B-13
Table B-6. Approximate Latency of L2 Misses of Intel Xeon Processor 5500. . . . . .B-15
Table B-7. Load Latency Event Programming B-18
Table B-8. Data Source Encoding for Load Latency PEBS Record B-18
Table B-9. Core PMU Events to Drill Down L2 Misses B-22
Table B-10. Core PMU Events for Super Queue Operation B-23
Table B-11. Core PMU Event to Drill Down OFFCore Responses B-23
Table B-12. OFFCORE_RSP_0 MSR Programming. B-23
Table B-13. Common Request and Response Types for OFFCORE_RSP_0 MSR. B-24
Table B-14. Uncore PMU Events for Occupancy Cycles. B-29
Table B-15. Common QHL Opcode Matching Facility Programming. B-31
Table C-1. CPUID Signature Values of Of Recent Intel Microarchitectures. C-3
Table C-2. Instruction Extensions Introduction by Microarchitectures (CPUID Signature). C-4
Table C-3. BMI1, BMI2 and General Purpose Instructions C-4
Table C-4. 256-bit AVX2 Instructions C-4
Table C-5. Gather Timing Data from L1D* C-6
Table C-6. BMI1, BMI2 and General Purpose Instructions C-6
Table C-7. F16C,RDRAND Instructions. C-7
Table C-8. 256-bit AVX Instructions. C-7
Table C-9. AESNI and PCLMULQDQ Instructions C-9
Table C-10. SSE4.2 Instructions C-9
Table C-11. SSE4.1 Instructions C-10
Table C-12. Supplemental Streaming SIMD Extension 3 Instructions. C-11
Table C-13. Streaming SIMD Extension 3 SIMD Floating-point Instructions C-11
Table C-14. Streaming SIMD Extension 2 128-bit Integer Instructions C-13
Table C-15. Streaming SIMD Extension 2 Double-precision Floating-point Instructions. C-14
Table C-16. Streaming SIMD Extension Single-precision Floating-point Instructions. C-15
Table C-17. General Purpose Instructions. C-17
Table C-18. Pointer-Chasing Variability of Software Measurable Latency of L1 Data Cache Latency. C-20

PAGE

CHAPTER 1
INTRODUCTION

The *Intel® 64 and IA-32 Architectures Optimization Reference Manual* describes how to optimize software to take advantage of the performance characteristics of IA-32 and Intel 64 architecture processors. Optimizations described in this manual apply to processors based on the Intel® Core™ microarchitecture, Enhanced Intel® Core™ microarchitecture, Intel® microarchitecture code name Nehalem, Intel® microarchitecture code name Westmere, Intel® microarchitecture code name Sandy Bridge, Intel® microarchitecture code name Ivy Bridge, Intel® microarchitecture code name Haswell, Intel NetBurst® microarchitecture, the Intel® Core™ Duo, Intel® Core™ Solo, Pentium® M processor families.

The target audience for this manual includes software programmers and compiler writers. This manual assumes that the reader is familiar with the basics of the IA-32 architecture and has access to the *Intel® 64 and IA-32 Architectures Software Developer's Manual* (five volumes). A detailed understanding of Intel 64 and IA-32 processors is often required. In many cases, knowledge of the underlying microarchitectures is required.

The design guidelines that are discussed in this manual for developing high-performance software generally apply to current as well as to future IA-32 and Intel 64 processors. The coding rules and code optimization techniques listed target the Intel Core microarchitecture, the Intel NetBurst microarchitecture and the Pentium M processor microarchitecture. In most cases, coding rules apply to software running in 64-bit mode of Intel 64 architecture, compatibility mode of Intel 64 architecture, and IA-32 modes (IA-32 modes are supported in IA-32 and Intel 64 architectures). Coding rules specific to 64-bit modes are noted separately.

1.1 TUNING YOUR APPLICATION

Tuning an application for high performance on any Intel 64 or IA-32 processor requires understanding and basic skills in:

- Intel 64 and IA-32 architecture.
- C and Assembly language.
- Hot-spot regions in the application that have impact on performance.
- Optimization capabilities of the compiler.
- Techniques used to evaluate application performance.

The Intel® VTune™ Performance Analyzer can help you analyze and locate hot-spot regions in your applications. On the Intel® Core™ i7, Intel® Core™2 Duo, Intel® Core™ Duo, Intel® Core™ Solo, Pentium® 4, Intel® Xeon® and Pentium® M processors, this tool can monitor an application through a selection of performance monitoring events and analyze the performance event data that is gathered during code execution.

This manual also describes information that can be gathered using the performance counters through Pentium 4 processor's performance monitoring events.

1.2 ABOUT THIS MANUAL

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200 and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture. In this document, references to the Core 2 Duo processor refer to processors based on the Intel® Core™ microarchitecture.

The Intel® Xeon® processor 3100, 3300, 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q8000 series, and Intel® Core™2 Extreme processors QX9000 series are based on 45 nm Enhanced Intel® Core™microarchitecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Intel® microarchitecture code name Nehalem. Intel® microarchitecture code name Westmere is a 32 nm version of Intel® microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on Intel® microarchitecture code name Westmere.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge.

The 3rd generation Intel® Core™ processors and the Intel Xeon processor E3-1200 v2 product family are based on Intel® microarchitecture code name Ivy Bridge. The Intel® Xeon® processor E5 v2 and E7 v2 families are based on the Ivy Bridge-E microarchitecture, support Intel 64 architecture and multiple physical processor packages in a platform.

The 4th generation Intel® Core™ processors and the Intel® Xeon® processor E3-1200 v3 product family are based on Intel® microarchitecture code name Haswell. The Intel® Xeon® processor E5 26xx v3 family is based on the Haswell-E microarchitecture, supports Intel 64 architecture and multiple physical processor packages in a platform.

The Intel® Core™ M processor family and 5th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Broadwell and support Intel 64 architecture.

The 6th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Skylake and support Intel 64 architecture.

In this document, references to the Pentium 4 processor refer to processors based on the Intel NetBurst® microarchitecture. This includes the Intel Pentium 4 processor and many Intel Xeon processors based on Intel NetBurst microarchitecture. Where appropriate, differences are noted (for example, some Intel Xeon processors have third level cache).

The Dual-core Intel® Xeon® processor LV is based on the same architecture as Intel® Core™ Duo and Intel® Core™ Solo processors.

Intel® Atom™ processor is based on Intel® Atom™ microarchitecture.

The following bullets summarize chapters in this manual.

- **Chapter 1: Introduction** — Defines the purpose and outlines the contents of this manual.
- **Chapter 2: Intel® 64 and IA-32 Processor Architectures** — Describes the microarchitecture of recent IA-32 and Intel 64 processor families, and other features relevant to software optimization.
- **Chapter 3: General Optimization Guidelines** — Describes general code development and optimization techniques that apply to all applications designed to take advantage of the common features of the Intel Core microarchitecture, Enhanced Intel Core microarchitecture, Intel NetBurst microarchitecture and Pentium M processor microarchitecture.
- **Chapter 4: Coding for SIMD Architectures** — Describes techniques and concepts for using the SIMD integer and SIMD floating-point instructions provided by the MMX™ technology, Streaming SIMD Extensions, Streaming SIMD Extensions 2, Streaming SIMD Extensions 3, SSSE3, and SSE4.1.
- **Chapter 5: Optimizing for SIMD Integer Applications** — Provides optimization suggestions and common building blocks for applications that use the 128-bit SIMD integer instructions.
- **Chapter 6: Optimizing for SIMD Floating-point Applications** — Provides optimization suggestions and common building blocks for applications that use the single-precision and double-precision SIMD floating-point instructions.
- **Chapter 7: Optimizing Cache Usage** — Describes how to use the PREFETCH instruction, cache control management instructions to optimize cache usage, and the deterministic cache parameters.
- **Chapter 8: Multicore and Hyper-Threading Technology** — Describes guidelines and techniques for optimizing multithreaded applications to achieve optimal performance scaling. Use these when targeting multicore processor, processors supporting Hyper-Threading Technology, or multiprocessor (MP) systems.

- **Chapter 9: 64-Bit Mode Coding Guidelines** — This chapter describes a set of additional coding guidelines for application software written to run in 64-bit mode.
- **Chapter 10: SSE4.2 and SIMD Programming for Text-Processing/Lexing/Parsing**— Describes SIMD techniques of using SSE4.2 along with other instruction extensions to improve text/string processing and lexing/parsing applications.
- **Chapter 11: Optimizations for Intel® AVX, FMA and AVX2**— Provides optimization suggestions and common building blocks for applications that use Intel® Advanced Vector Extensions, FMA, and AVX2.
- **Chapter 12: Intel Transactional Synchronization Extensions** — Tuning recommendations to use lock elision techniques with Intel Transactional Synchronization Extensions to optimize multi-threaded software with contended locks.
- **Chapter 13: Power Optimization for Mobile Usages** — This chapter provides background on power saving techniques in mobile processors and makes recommendations that developers can leverage to provide longer battery life.
- **Chapter 14: Intel® Atom™ Microarchitecture and Software Optimization** — Describes the microarchitecture of processor families based on Intel Atom microarchitecture, and software optimization techniques targeting Intel Atom microarchitecture.
- **Chapter 15: Silvermont Microarchitecture and Software Optimization** — Describes the micro-architecture of processor families based on the Silvermont microarchitecture, and software optimization techniques targeting Intel processors based on the Silvermont microarchitecture.
- **Appendix A: Application Performance Tools** — Introduces tools for analyzing and enhancing application performance without having to write assembly code.
- **Appendix B: Using Performance Monitoring Events** — Provides information on the Top-Down Analysis Method and information on how to use performance events specific to the Intel Xeon processor 5500 series, processors based on Intel microarchitecture code name Sandy Bridge, and Intel Core Solo and Intel Core Duo processors.
- **Appendix C: IA-32 Instruction Latency and Throughput** — Provides latency and throughput data for the IA-32 instructions. Instruction timing data specific to recent processor families are provided.

1.3 RELATED INFORMATION

For more information on the Intel® architecture, techniques, and the processor architecture terminology, the following are of particular interest:

- *Intel® 64 and IA-32 Architectures Software Developer's Manual* (in five volumes)
- *Intel® Processor Identification with the CPUID Instruction, AP-485*
- *Developing Multi-threaded Applications: A Platform Consistent Approach*
- Intel® C++ Compiler documentation and online help
- Intel® Fortran Compiler documentation and online help
- Intel® VTune™ Performance Analyzer documentation and online help
- *Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor MP*

More relevant links are:

- Software network link:

  http://softwarecommunity.intel.com/isn/home/
- Developer centers:

  http://www3.intel.com/cd/ids/developer/asmo-na/eng/dc/index.htm
- Processor support general link:

  http://www.intel.com/support/processors/
- Software products and packages:

http://www3.intel.com/cd/software/products/asmo-na/eng/index.htm

- Intel 64 and IA-32 processor manuals (printed or PDF downloads):
  http://developer.intel.com/products/processor/manuals/index.htm
- Intel Multi-Core Technology:
  http://developer.intel.com/technology/multi-core/index.htm
- Hyper-Threading Technology (HT Technology):
  http://developer.intel.com/technology/hyperthread/
- SSE4.1 Application Note: Motion Estimation with Intel® Streaming SIMD Extensions 4:
  http://softwarecommunity.intel.com/articles/eng/1246.htm
- SSE4.1 Application Note: Increasing Memory Throughput with Intel® Streaming SIMD Extensions 4:
  http://softwarecommunity.intel.com/articles/eng/1248.htm
- Processor Topology and Cache Topology white paper and reference code
  http://software.intel.com/en-us/articles/intel-64-architecture-processor-topology-enumeration
- Multi-buffering techniques using SIMD extensions:
  https://www-ssl.intel.com/content/www/us/en/communications/communications-ia-multi-buffer-paper.html
- Parallel hashing using Multi-buffering techniques:
  http://www.scirp.org/journal/PaperInformation.aspx?paperID=23995
  http://eprint.iacr.org/2012/476.pdf
- AES Library of sample code:
  http://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library/
  http://software.intel.com/file/26898
- PCMMULQDQ resources:
  http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/fast-crc-computation-paper.html
  https://www-ssl.intel.com/content/www/us/en/intelligent-systems/wireless-infrastructure/aes-ipsec-performance-linux-paper.html
- Modular exponentiation using redundant representation and AVX2:
  http://rd.springer.com/chapter/10.1007%2F978-3-642-31662-3_9?LI=true

CHAPTER 2
INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

This chapter gives an overview of features relevant to software optimization for current generations of Intel 64 and IA-32 processors (processors based on Intel® microarchitecture code name Skylake, Intel® microarchitecture code name Broadwell, Intel® microarchitecture code name Haswell, Intel microarchitecture code name Ivy Bridge, Intel microarchitecture code name Sandy Bridge, processors based on the Intel Core microarchitecture, Enhanced Intel Core microarchitecture, Intel microarchitecture code name Nehalem). These features are:

- Microarchitectures that enable executing instructions with high throughput at high clock rates, a high speed cache hierarchy and high speed system bus.
- Multicore architecture available across Intel Core processor and Intel Xeon processor families.
- Hyper-Threading Technology[1] (HT Technology) support.
- Intel 64 architecture on Intel 64 processors.
- SIMD instruction extensions: MMX technology, Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), Supplemental Streaming SIMD Extensions 3 (SSSE3), SSE4.1, and SSE4.2.
- Intel® Advanced Vector Extensions (Intel® AVX).
- Half-precision floating-point conversion and RDRAND.
- Fused Multiply Add Extensions.
- Intel® Advanced Vector Extensions 2 (Intel® AVX2).
- ADX and RDSEED.

The Intel Core 2, Intel Core 2 Extreme, Intel Core 2 Quad processor family, Intel Xeon processor 3000, 3200, 5100, 5300, 7300 series are based on the high-performance and power-efficient Intel Core microarchitecture. Intel Xeon processor 3100, 3300, 5200, 5400, 7400 series, Intel Core 2 Extreme processor QX9600, QX9700 series, Intel Core 2 Quad Q9000 series, Q8000 series are based on the enhanced Intel Core microarchitecture. Intel Core i7 processor is based on Intel microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and Intel Core i7, i5, i3 processors are based on Intel microarchitecture code name Westmere.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge.

The Intel® Xeon® processor E3-1200 v2 product family and the 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture. The Intel® Xeon® processor E5 v2 and E7 v2 families are based on the Ivy Bridge-E microarchitecture, support Intel 64 architecture and multiple physical processor packages in a platform.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture. The Intel® Xeon® processor E5 26xx v3 family is based on the Haswell-E microarchitecture, supports Intel 64 architecture and multiple physical processor packages in a platform.

Intel® Core™ M processors, 5th generation Intel Core processors and Intel Xeon processor E3-1200 v4 series are based on the Broadwell microarchitecture and support Intel 64 architecture.

The 6th generation Intel Core processors, Intel Xeon processor E3-1500m v5 are based on the Skylake microarchitecture and support Intel 64 architecture.

---

1. Hyper-Threading Technology requires a computer system with an Intel processor supporting HT Technology and an HT Technology enabled chipset, BIOS and operating system. Performance varies depending on the hardware and software used.

2.1 THE SKYLAKE MICROARCHITECTURE

The Skylake microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake microarchitecture is depicted in Figure 2-1.

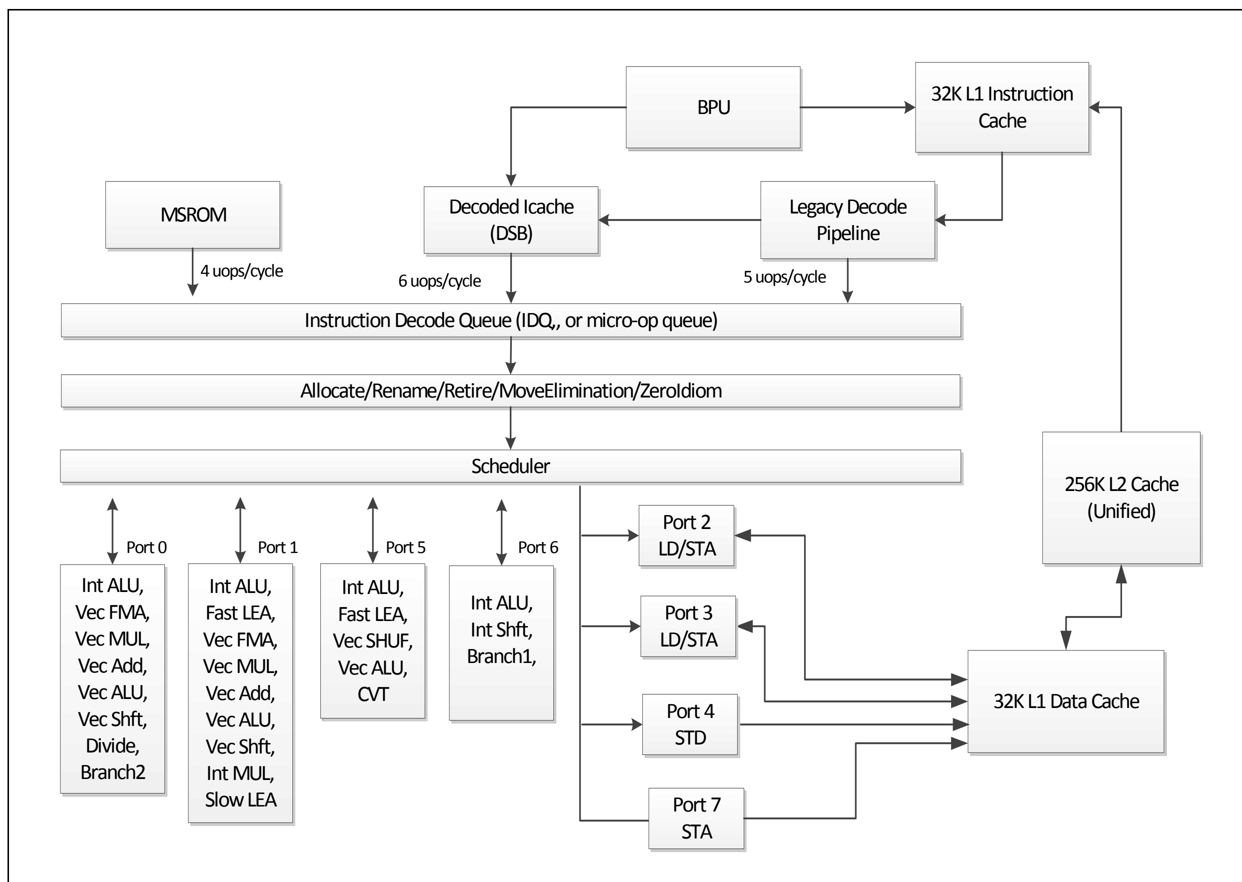


Figure 2-1. CPU Core Pipeline Functionality of the Skylake Microarchitecture

The Skylake microarchitecture offers the following enhancements:

- Larger internal buffers to enable deeper OOO execution and higher cache bandwidth.
- Improved front end throughput.
- Improved branch predictor.
- Improved divider throughput and latency.
- Lower power consumption.
- Improved SMT performance with Hyper-Threading Technology.
- Balanced floating-point ADD, MUL, FMA throughput and latency.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3 (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc. A four-core configuration can be supported similar to the arrangement shown in Figure 2-3.

2.1.1 The Front End

The front end in the Skylake microarchitecture provides the following improvements over previous generation microarchitectures:

- Legacy Decode Pipeline delivery of 5 uops per cycle to the IDQ compared to 4 uops in previous generations.
- The DSB delivers 6 uops per cycle to the IDQ compared to 4 uops in previous generations.
- The IDQ can hold 64 uops per logical processor vs. 28 uops per logical processor in previous generations when two sibling logical processors in the same core are active (2x64 vs. 2x28 per core). If only one logical processor is active in the core, the IDQ can hold 64 uops (64 vs. 56 uops in ST operation).
- The LSD in the IDQ can detect loops up to 64 uops per logical processor irrespective ST or SMT operation.
- Improved Branch Predictor.

2.1.2 The Out-of-Order Execution Engine

The Out of Order and execution engine changes in Skylake microarchitecture include:

- Larger buffers enable deeper OOO execution compared to previous generations.
- Improved throughput and latency for divide/sqrt and approximate reciprocals.
- Identical latency and throughput for all operations running on FMA units.
- Longer pause latency enables better power efficiency and better SMT performance resource utilization.

Table 2-1 summarizes the OOO engine's capability to dispatch different types of operations to various ports.

Table 2-1. Dispatch Port and Execution Stacks of the Skylake Microarchitecture

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Vec ALU	ALU, Fast LEA, Vec ALU	LD STA	STA	ALU, Fast LEA, Vec ALU,	ALU, Shft,	STA
Vec Shft, Vec Add,	Vec Shft, Vec Add,			Vec Shuffle,	Branch1	
Vec Mul, FMA,	Vec Mul, FMA					
DIV,	Slow Int					
Branch2	Slow LEA					

Table 2-2 lists execution units and common representative instructions that rely on these units. Throughput improvements across the SSE, AVX and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-2. Skylake Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*,
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm,
FP Mov	1	(v)movsd/ss, (v)movd gpr,
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 11.16.5 on MMX instruction throughput remedy.

A significant portion of the SSE, AVX and general-purpose instructions also have latency improvements. Appendix C lists the specific details. Software-visible latency exposure of an instruction sometimes may include additional contributions that depend on the relationship between micro-ops flows of the producer instruction and the micro-op flows of the ensuing consumer instruction. For example, a two-uop instruction like VPMULLD may experience two cumulative bypass delays of 1 cycle each from each of the two micro-ops of VPMULLD.

Table 2-3 describes the bypass delay in cycles between a producer uop and the consumer uop. The left-most column lists a variety of situations characteristic of the producer micro-op. The top row lists a variety of situations characteristic of the consumer micro-op.

Table 2-3. Bypass Delay Between Producer and Consumer Micro-ops

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA

Table 2-3. Bypass Delay Between Producer and Consumer Micro-ops (Contd.)

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
SHUF/5/1,3	0	0	1	0	0	0	NA
V2I/0/3	NA	NA	NA	NA	NA	NA	NA
I2V/5/1	0	0	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the uop. For example:

- "SIMD/0,1/1" applies to 1-cycle vector SIMD uop dispatched to either port 0 or port 1.
- "VIMUL/0,1/4" applies to 4-cycle vector integer multiply uop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either 1-cycle or 3-cycle non-shuffle uop dispatched to port 5.

2.1.3 Cache and Memory Subsystem

The cache hierarchy of the Skylake microarchitecture has the following enhancements:

- Higher Cache bandwidth compared to previous generations.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Processor can do two page walks in parallel compared to one in Haswell microarchitecture and earlier generations.
- Page split load penalty down from 100 cycles in previous generation to 5 cycles.
- L3 write bandwidth increased from 4 cycles per line in previous generation to 2 per line.
- Support for the CLFLUSHOPT instruction to flush cache lines and manage memory ordering of flushed data using SFENCE.
- Reduced performance penalty for a software prefetch that specifies a NULL pointer.
- L2 associativity changed from 8 ways to 4 ways.

Table 2-4. Cache Parameters of the Skylake Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency[1]	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	96 (2x32B Load + 1*32B Store)	~81	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/4	64	12 cycle	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44	32	~18	Writeback

NOTES:

1. Software-visible latency will vary depending on access patterns and other factors.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2. The partition column of Table 2-5 indicates the resource sharing policy when Hyper-Threading Technology is active.

Table 2-5. TLB Parameters of the Skylake Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	8 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1536	12	fixed
Second Level	1GB	16	4	fixed

2.2 THE HASWELL MICROARCHITECTURE

The Haswell microarchitecture builds on the successes of the Sandy Bridge and Ivy Bridge microarchitectures. The basic pipeline functionality of the Haswell microarchitecture is depicted in Figure 2-2. In general, most of the features described in Section 2.2.1 - Section 2.2.4 also apply to the Broadwell microarchitecture. Enhancements of the Broadwell microarchitecture are summarized in Section 2.2.6.

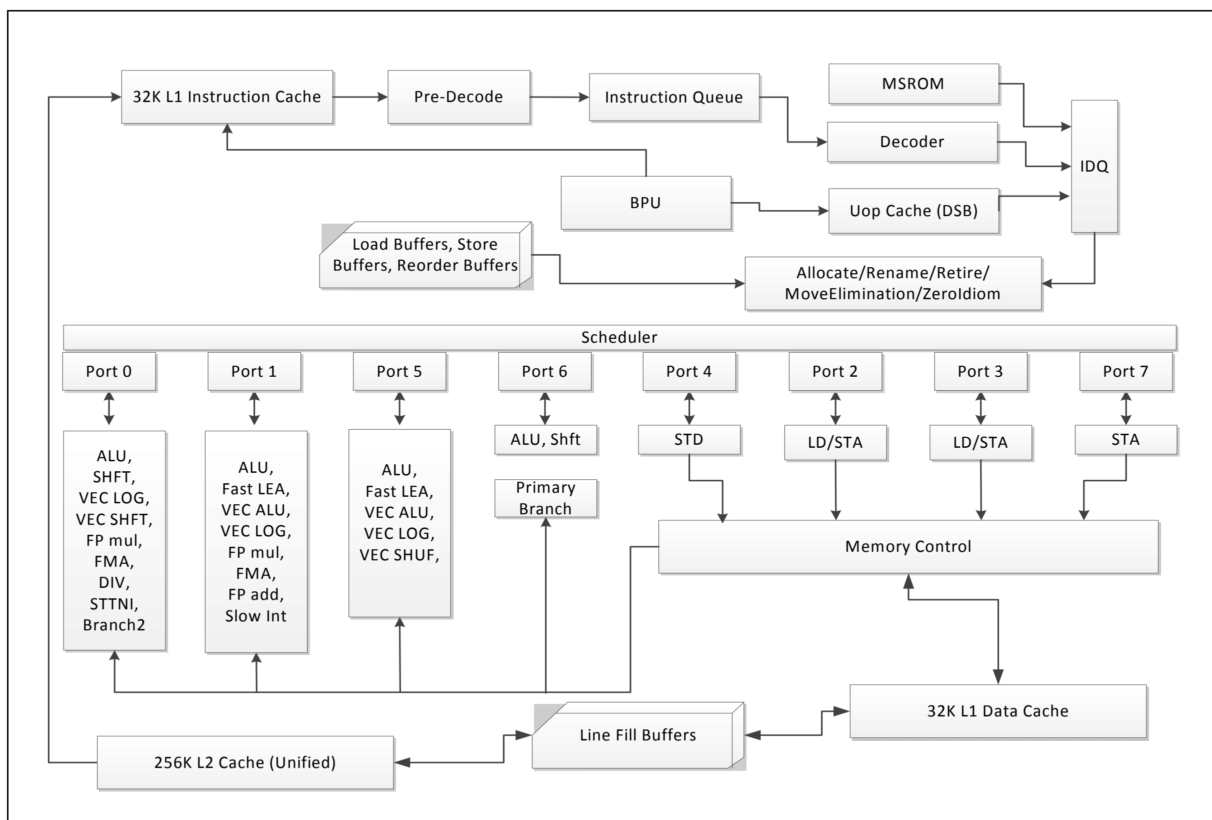


Figure 2-2. CPU Core Pipeline Functionality of the Haswell Microarchitecture

The Haswell microarchitecture offers the following innovative features:

- Support for Intel Advanced Vector Extensions 2 (Intel AVX2), FMA.

- Support for general-purpose, new instructions to accelerate integer numeric encryption.
- Support for Intel® Transactional Synchronization Extensions (Intel® TSX).
- Each core can dispatch up to 8 micro-ops per cycle.
- 256-bit data path for memory operation, FMA, AVX floating-point and AVX2 integer execution units.
- Improved L1D and L2 cache bandwidth.
- Two FMA execution pipelines.
- Four arithmetic logical units (ALUs).
- Three store address ports.
- Two branch execution units.
- Advanced power management features for IA processor core and uncore sub-systems.
- Support for optional fourth level cache.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3 (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc. An example of the system integration view of four CPU cores with uncore components is illustrated in Figure 2-3.

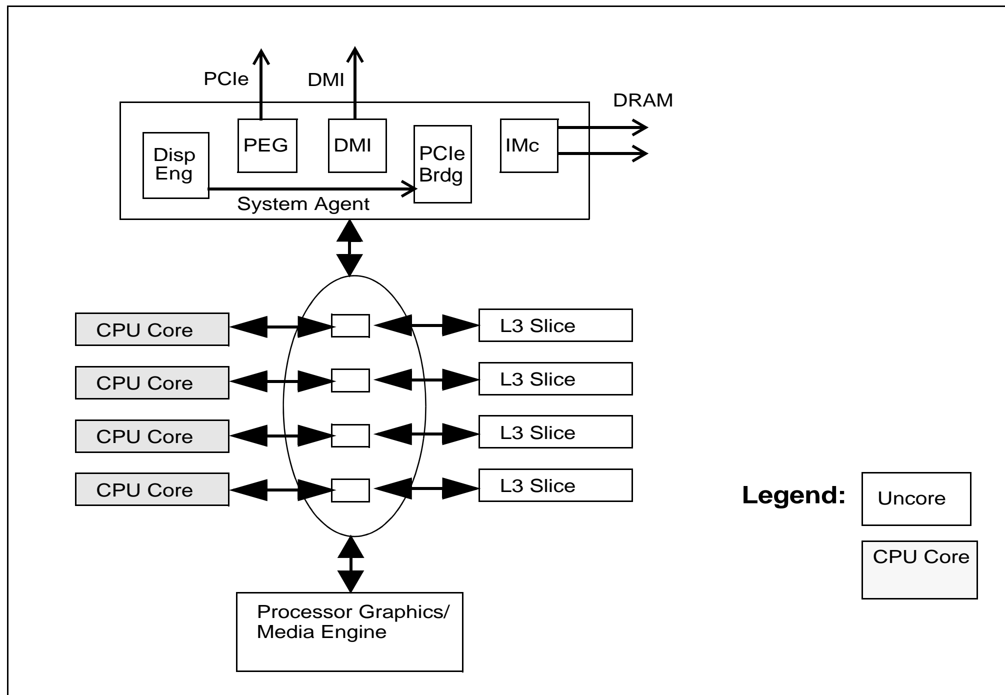


Figure 2-3. Four Core System Integration of the Haswell Microarchitecture

2.2.1 The Front End

The front end of Intel microarchitecture code name Haswell builds on that of Intel microarchitecture code name Sandy Bridge and Intel microarchitecture code name Ivy Bridge, see Section 2.3.2 and Section 2.3.7. Additional enhancements in the front end include:

- The uop cache (or decoded ICache) is partitioned equally between two logical processors.
- The instruction decoders will alternate between each active logical processor. If one sibling logical processor is idle, the active logical processor will use the decoders continuously.

- The LSD in the micro-op queue (or IDQ) can detect small loops up to 56 micro-ops. The 56-entry micro-op queue is shared by two logical processors if Hyper-Threading Technology is active (Intel microarchitecture Sandy Bridge provides duplicated 28-entry micro-op queue in each core).

2.2.2 The Out-of-Order Engine

The key components and significant improvements to the out-of-order engine are summarized below:

Renamer: The Renamer moves micro-ops from the micro-op queue to bind to the dispatch ports in the Scheduler with execution resources. Zero-idiom, one-idiom and zero-latency register move operations are performed by the Renamer to free up the Scheduler and execution core for improved performance.

Scheduler: The Scheduler controls the dispatch of micro-ops onto the dispatch ports. There are eight dispatch ports to support the out-of-order execution core. Four of the eight ports provided execution resources for computational operations. The other 4 ports support memory operations of up to two 256-bit load and one 256-bit store operation in a cycle.

Execution Core: The scheduler can dispatch up to eight micro-ops every cycle, one on each port. Of the four ports providing computational resources, each provides an ALU, two of these execution pipes provided dedicated FMA units. With the exception of division/square-root, STTNI/AESNI units, most floating-point and integer SIMD execution units are 256-bit wide. The four dispatch ports servicing memory operations consist with two dual-use ports for load and store-address operation. Plus a dedicated 3rd store-address port and one dedicated store-data port. All memory ports can handle 256-bit memory micro-ops. Peak floating-point throughput, at 32 single-precision operations per cycle and 16 double-precision operations per cycle using FMA, is twice that of Intel microarchitecture code name Sandy Bridge.

The out-of-order engine can handle 192 uops in flight compared to 168 in Intel microarchitecture code name Sandy Bridge.

2.2.3 Execution Engine

Table 2-6 summarizes which operations can be dispatched on which port.

Table 2-6. Dispatch Port and Execution Stacks of the Haswell Microarchitecture

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Shift	ALU, Fast LEA, BM	Load_Addr, Store_addr	Store_data	ALU, Fast LEA, BM	ALU, Shift, JEU	Store_addr, Simple_AGU
SIMD_Log, SIMD misc, SIMD_Shifts	SIMD_ALU, SIMD_Log			SIMD_ALU, SIMD_Log,		
FMA/FP_mul, Divide	FMA/FP_mul, FP_add			Shuffle		
2nd_Jeu	slow_int,			FP mov, AES		

Table 2-7 lists execution units and common representative instructions that rely on these units. Table 2-7 also includes some instructions that are available only on processors based on the Broadwell microarchitecture.

Table 2-7. Haswell Microarchitecture Execution Units and Representative Instructions

Execution Unit	# of Ports	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa
SHFT	2	sal, shl, rol, adc, sarx, (adcx, adox)[1] etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc
SIMD Log	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)blendp*, vpblendd
SIMD_Shft	1	(v)psl*, (v)psr*
SIMD ALU	2	(v)padd*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)pcmpgt*
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)pblendw
SIMD Misc	1	(v)pmul*, (v)pmadd*, STTNI, (v)pclmulqdq, (v)psadw, (v)pcmpgtq, vpsllvd, (v)bendv*, (v)plendw,
FP Add	1	(v)addp*, (v)cmpp*, (v)max*, (v)min*,
FP Mov	1	(v)movap*, (v)movup*, (v)movsd/ss, (v)movd gpr, (v)andp*, (v)orp*
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Only available in processors based on the Broadwell microarchitecture and support CPUID ADX feature flag.

The reservation station (RS) is expanded to 60 entries deep (compared to 54 entries in Intel microarchitecture code name Sandy Bridge). It can dispatch up to eight micro-ops in one cycle if the micro-ops are ready to execute. The RS dispatch a micro-op through an issue port to a specific execution cluster, arranged in several stacks to handle specific data types or granularity of data.

When a source of a micro-op executed in one stack comes from a micro-op executed in another stack, a delay can occur. The delay occurs also for transitions between Intel SSE integer and Intel SSE floating-point operations. In some of the cases the data transition is done using a micro-op that is added to the instruction flow. Table 2-29 describes how data, written back after execution, can bypass to micro-op execution in the following cycles.

Table 2-8. Bypass Delay Between Producer and Consumer Micro-ops (cycles)

From/To	INT	SSE-INT/ AVX-INT	SSE-FP/ AVX-FP_LOW	X87/ AVX-FP_High
INT		• micro-op (port 5) • micro-op (port 6) + 1 cycle	• micro-op (port 5) • micro-op (port 6) + 1 cycle	micro-op (port 5) + 3 cycle delay
SSE-INT/ AVX-INT	micro-op (port 1)		1 cycle delay	
SSE-FP/ AVX-FP_LOW	micro-op (port 1)	1 cycle delay		micro-op (port 5) + 1cycle delay

Table 2-8. Bypass Delay Between Producer and Consumer Micro-ops (cycles) (Contd.)

From/To	INT	SSE-INT/ AVX-INT	SSE-FP/ AVX-FP_LOW	X87/ AVX-FP_High
X87/ AVX-FP_High	micro-op (port 1) + 3 cycle delay		micro-op (port 5) + 1cycle delay	
Load		1 cycle delay	1 cycle delay	2 cycle delay

2.2.4 Cache and Memory Subsystem

The cache hierarchy is similar to prior generations, including an instruction cache, a first-level data cache and a second-level unified cache in each core, and a 3rd-level unified cache with size dependent on specific product configuration. The 3rd-level cache is organized as multiple cache slices, the size of each slice may depend on product configurations, connected by a ring interconnect. The exact details of the cache topology is reported by CPUID leaf 4. The 3rd level cache resides in the "uncore" sub-system that is shared by all the processor cores. In some product configurations, a fourth level cache is also supported. Table 2-27 provides more details of the cache hierarchy.

Table 2-9. Cache Parameters of the Haswell Microarchitecture

Level	Capacity/Associativity	Line Size (bytes)	Fastest Latency[1]	Throughput (clocks)	Peak Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	0.5[2]	64 (Load) + 32 (Store)	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/8	64	11 cycle	Varies	64	Writeback
Third Level (Shared L3)	Varies	64	~34	Varies		Writeback

NOTES:
1. Software-visible latency will vary depending on access patterns and other factors. L3 latency can vary due to clock ratios between the processor core and uncore.
2. First level data cache supports two load micro-ops each cycle; each micro-op can fetch up to 32-bytes of data.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2.

Table 2-10. TLB Parameters of the Haswell Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	4 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1024	8	fixed

2.2.4.1 Load and Store Operation Enhancements

The L1 data cache can handle two 256-bit load and one 256-bit store operations each cycle. The unified L2 can service one cache line (64 bytes) each cycle. Additionally, there are 72 load buffers and 42 store buffers available to support micro-ops execution in-flight.

2.2.5 The Haswell-E Microarchitecture

Intel processors based on the Haswell-E microarchitecture comprises the same processor cores as described in the Haswell microarchitecture, but provides more advanced uncore and integrated I/O capabilities. Processors based on the Haswell-E microarchitecture support platforms with multiple sockets.

The Haswell-E microarchitecture supports versatile processor architectures and platform configurations for scalability and high performance. Some of capabilities provided by the uncore and integrated I/O sub-system of the Haswell-E microarchitecture include:

- Support for multiple Intel QPI interconnects in multi-socket configurations.
- Up to two integrated memory controllers per physical processor.
- Up to 40 lanes of PCI Express* 3.0 links per physical processor.
- Up to 18 processor cores connected by two ring interconnects to the L3 in each physical processor.

An example of a possible 12-core processor implementation using the Haswell-E microarchitecture is illustrated in Figure 2-4. The capabilities of the uncore and integrated I/O sub-system vary across the processor family implementing the Haswell-E microarchitecture. For details, please consult the data sheets of respective Intel Xeon E5 v3 processors.

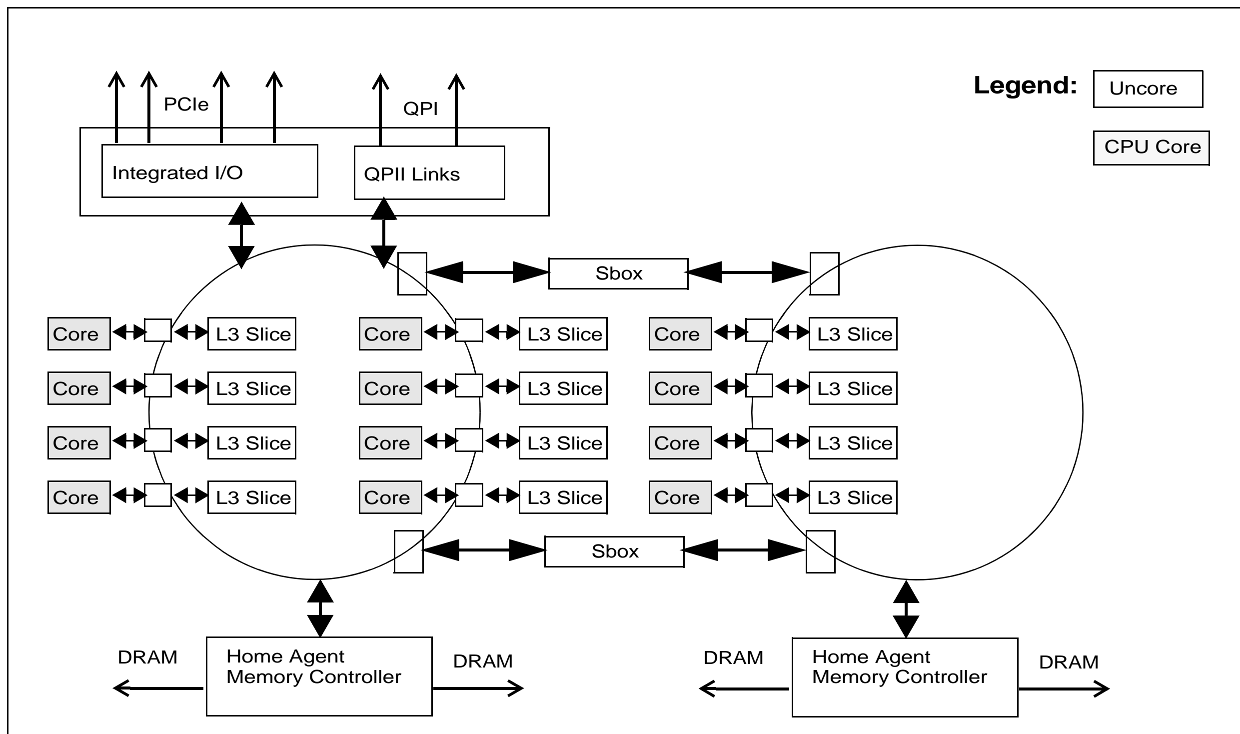


Figure 2-4. An Example of the Haswell-E Microarchitecture Supporting 12 Processor Cores

2.2.6 The Broadwell Microarchitecture

Intel Core M processors are based on the Broadwell microarchitecture. The Broadwell microarchitecture builds from the Haswell microarchitecture and provides several enhancements. This section covers enhanced features of the Broadwell microarchitecture.

- Floating-point multiply instruction latency is improved from 5 cycles in prior generation to 3 cycle in the Broadwell microarchitecture. This applies to AVX, SSE and FP instruction sets.
- The throughput of gather instructions has been improved significantly, see Table C-5.
- The PCLMULQDQ instruction implementation is a single uop in the Broadwell microarchitecture with improved latency and throughput.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2.

Table 2-11. TLB Parameters of the Broadwell Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	4 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2MB pages	1536	6	fixed
Second Level	1GB pages	16	4	fixed

2.3 INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Intel® microarchitecture code name Sandy Bridge builds on the successes of Intel® Core™ microarchitecture and Intel microarchitecture code name Nehalem. It offers the following innovative features:

- Intel Advanced Vector Extensions (Intel AVX)
  - 256-bit floating-point instruction set extensions to the 128-bit Intel Streaming SIMD Extensions, providing up to 2X performance benefits relative to 128-bit code.
  - Non-destructive destination encoding offers more flexible coding techniques.
  - Supports flexible migration and co-existence between 256-bit AVX code, 128-bit AVX code and legacy 128-bit SSE code.
- Enhanced front end and execution engine
  - New decoded ICache component that improves front end bandwidth and reduces branch misprediction penalty.
  - Advanced branch prediction.
  - Additional macro-fusion support.
  - Larger dynamic execution window.
  - Multi-precision integer arithmetic enhancements (ADC/SBB, MUL/IMUL).
  - LEA bandwidth improvement.
  - Reduction of general execution stalls (read ports, writeback conflicts, bypass latency, partial stalls).
  - Fast floating-point exception handling.
  - XSAVE/XRSTORE performance improvements and XSAVEOPT new instruction.

- Cache hierarchy improvements for wider data path
  - Doubling of bandwidth enabled by two symmetric ports for memory operation.
  - Simultaneous handling of more in-flight loads and stores enabled by increased buffers.
  - Internal bandwidth of two loads and one store each cycle.
  - Improved prefetching.
  - High bandwidth low latency LLC architecture.
  - High bandwidth ring architecture of on-die interconnect.
- System-on-a-chip support
  - Integrated graphics and media engine in second generation Intel Core processors.
  - Integrated PCIE controller.
  - Integrated memory controller.
- Next generation Intel Turbo Boost Technology
  - Leverage TDP headroom to boost performance of CPU cores and integrated graphic unit.

2.3.1 Intel® Microarchitecture Code Name Sandy Bridge Pipeline Overview

Figure 2-5 depicts the pipeline and major components of a processor core that's based on Intel microarchitecture code name Sandy Bridge. The pipeline consists of

- An in-order issue front end that fetches instructions and decodes them into micro-ops (micro-operations). The front end feeds the next pipeline stages with a continuous stream of micro-ops from the most likely path that the program will execute.
- An out-of-order, superscalar execution engine that dispatches up to six micro-ops to execution, per cycle. The allocate/rename block reorders micro-ops to "dataflow" order so they can execute as soon as their sources are ready and execution resources are available.
- An in-order retirement unit that ensures that the results of execution of the micro-ops, including any exceptions they may have encountered, are visible according to the original program order.

The flow of an instruction in the pipeline can be summarized in the following progression:

1. The Branch Prediction Unit chooses the next block of code to execute from the program. The processor searches for the code in the following resources, in this order:
   a. Decoded ICache.
   b. Instruction Cache, via activating the legacy decode pipeline.
   c. L2 cache, last level cache (LLC) and memory, as necessary.

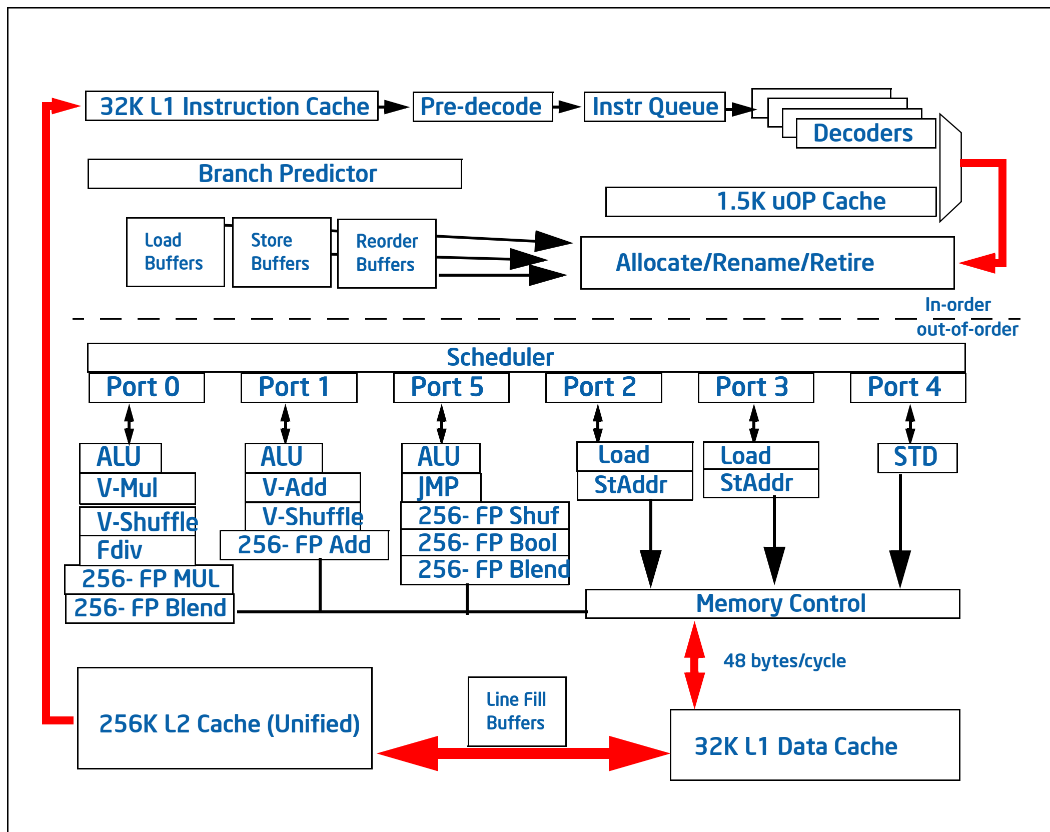


Figure 2-5. Intel Microarchitecture Code Name Sandy Bridge Pipeline Functionality

2. The micro-ops corresponding to this code are sent to the Rename/retirement block. They enter into the scheduler in program order, but execute and are de-allocated from the scheduler according to data-flow order. For simultaneously ready micro-ops, FIFO ordering is nearly always maintained.

   Micro-op execution is executed using execution resources arranged in three stacks. The execution units in each stack are associated with the data type of the instruction.

   Branch mispredictions are signaled at branch execution. It re-steers the front end which delivers micro-ops from the correct path. The processor can overlap work preceding the branch misprediction with work from the following corrected path.

3. Memory operations are managed and reordered to achieve parallelism and maximum performance. Misses to the L1 data cache go to the L2 cache. The data cache is non-blocking and can handle multiple simultaneous misses.

4. Exceptions (Faults, Traps) are signaled at retirement (or attempted retirement) of the faulting instruction.

Each processor core based on Intel microarchitecture code name Sandy Bridge can support two logical processor if Intel HyperThreading Technology is enabled.

2.3.2 The Front End

This section describes the key characteristics of the front end. Table 2-12 lists the components of the front end, their functions, and the problems they address.

Table 2-12. Components of the Front End of Intel Microarchitecture Code Name Sandy Bridge

Component	Functions	Performance Challenges
Instruction Cache	32-Kbyte backing store of instruction bytes	Fast access to hot code instruction bytes
Legacy Decode Pipeline	Decode instructions to micro-ops, delivered to the micro-op queue and the Decoded ICache.	Provides the same decode latency and bandwidth as prior Intel processors. Decoded ICache warm-up
Decoded ICache	Provide stream of micro-ops to the micro-op queue.	Provides higher micro-op bandwidth at lower latency and lower power than the legacy decode pipeline
MSROM	Complex instruction micro-op flow store, accessible from both Legacy Decode Pipeline and Decoded ICache	
Branch Prediction Unit (BPU)	Determine next block of code to be executed and drive lookup of Decoded ICache and legacy decode pipelines.	Improves performance and energy efficiency through reduced branch mispredictions.
Micro-op queue	Queues micro-ops from the Decoded ICache and the legacy decode pipeline.	Hide front end bubbles; provide execution micro-ops at a constant rate.

2.3.2.1 Legacy Decode Pipeline

The Legacy Decode Pipeline comprises the instruction translation lookaside buffer (ITLB), the instruction cache (ICache), instruction predecode, and instruction decode units.

Instruction Cache and ITLB

An instruction fetch is a 16-byte aligned lookup through the ITLB and into the instruction cache. The instruction cache can deliver every cycle 16 bytes to the instruction pre-decoder. Table 2-12 compares the ICache and ITLB with prior generation.

Table 2-13. ICache and ITLB of Intel Microarchitecture Code Name Sandy Bridge

Component	Intel microarchitecture code name Sandy Bridge	Intel microarchitecture code name Nehalem
ICache Size	32-Kbyte	32-Kbyte
ICache Ways	8	4
ITLB 4K page entries	128	128
ITLB large page (2M or 4M) entries	8	7

Upon ITLB miss there is a lookup to the Second level TLB (STLB) that is common to the DTLB and the ITLB. The penalty of an ITLB miss and a STLB hit is seven cycles.

Instruction PreDecode

The predecode unit accepts the 16 bytes from the instruction cache and determines the length of the instructions.

The following length changing prefixes (LCPs) imply instruction length that is different from the default length of instructions. Therefore they cause an additional penalty of three cycles per LCP during length decoding. Previous processors incur a six-cycle penalty for each 16-byte chunk that has one or more LCPs in it. Since usually there is no more than one LCP in a 16-byte chunk, in most cases, Intel microarchitecture code name Sandy Bridge introduces an improvement over previous processors.

- Operand Size Override (66H) preceding an instruction with a word/double immediate data. This prefix might appear when the code uses 16 bit data types, unicode processing, and image processing.
- Address Size Override (67H) preceding an instruction with a modr/m in real, big real, 16-bit protected or 32-bit protected modes. This prefix may appear in boot code sequences.
- The REX prefix (4xh) in the Intel® 64 instruction set can change the size of two classes of instructions: MOV offset and MOV immediate. Despite this capability, it does not cause an LCP penalty and hence is not considered an LCP.

Instruction Decode

There are four decoding units that decode instruction into micro-ops. The first can decode all IA-32 and Intel 64 instructions up to four micro-ops in size. The remaining three decoding units handle single-micro-op instructions. All four decoding units support the common cases of single micro-op flows including micro-fusion and macro-fusion.

Micro-ops emitted by the decoders are directed to the micro-op queue and to the Decoded ICache. Instructions longer than four micro-ops generate their micro-ops from the MSROM. The MSROM bandwidth is four micro-ops per cycle. Instructions whose micro-ops come from the MSROM can start from either the legacy decode pipeline or from the Decoded ICache.

MicroFusion

Micro-fusion fuses multiple micro-ops from the same instruction into a single complex micro-op. The complex micro-op is dispatched in the out-of-order execution core as many times as it would if it were not micro-fused.

Micro-fusion enables you to use memory-to-register operations, also known as the complex instruction set computer (CISC) instruction set, to express the actual program operation without worrying about a loss of decode bandwidth. Micro-fusion improves instruction bandwidth delivered from decode to retirement and saves power.

Coding an instruction sequence by using single-uop instructions will increases the code size, which can decrease fetch bandwidth from the legacy pipeline.

The following are examples of micro-fused micro-ops that can be handled by all decoders.

- All stores to memory, including store immediate. Stores execute internally as two separate functions, store-address and store-data.
- All instructions that combine load and computation operations (load+op), for example:
  - ADDPS XMM9, OWORD PTR [RSP+40]
  - FADD DOUBLE PTR [RDI+RSI*8]
  - XOR RAX, QWORD PTR [RBP+32]
- All instructions of the form "load and jump," for example:
  - JMP [RDI+200]
  - RET
- CMP and TEST with immediate operand and memory

An instruction with RIP relative addressing is not micro-fused in the following cases:

- An additional immediate is needed, for example:
  - CMP [RIP+400], 27
  - MOV [RIP+3000], 142
- The instruction is a control flow instruction with an indirect target specified using RIP-relative addressing, for example:
  - JMP [RIP+5000000]

In these cases, an instruction that can not be micro-fused will require decoder 0 to issue two micro-ops, resulting in a slight loss of decode bandwidth.

In 64-bit code, the usage of RIP Relative addressing is common for global data. Since there is no micro-fusion in these cases, performance may be reduced when porting 32-bit code to 64-bit code.

Macro-Fusion

Macro-fusion merges two instructions into a single micro-op. In Intel Core microarchitecture, this hardware optimization is limited to specific conditions specific to the first and second of the macro-fusable instruction pair.

- The first instruction of the macro-fused pair modifies the flags. The following instructions can be macro-fused:
  - In Intel microarchitecture code name Nehalem: CMP, TEST.
  - In Intel microarchitecture code name Sandy Bridge: CMP, TEST, ADD, SUB, AND, INC, DEC
  - These instructions can fuse if
    - The first source / destination operand is a register.
    - The second source operand (if exists) is one of: immediate, register, or non RIP-relative memory.
- The second instruction of the macro-fusable pair is a conditional branch. Table 3-1 describes, for each instruction, what branches it can fuse with.

Macro fusion does not happen if the first instruction ends on byte 63 of a cache line, and the second instruction is a conditional branch that starts at byte 0 of the next cache line.

Since these pairs are common in many types of applications, macro-fusion improves performance even on non-recompiled binaries.

Each macro-fused instruction executes with a single dispatch. This reduces latency and frees execution resources. You also gain increased rename and retire bandwidth, increased virtual storage, and power savings from representing more work in fewer bits.

2.3.2.2 Decoded ICache

The Decoded ICache is essentially an accelerator of the legacy decode pipeline. By storing decoded instructions, the Decoded ICache enables the following features:

- Reduced latency on branch mispredictions.
- Increased micro-op delivery bandwidth to the out-of-order engine.
- Reduced front end power consumption.

The Decoded ICache caches the output of the instruction decoder. The next time the micro-ops are consumed for execution the decoded micro-ops are taken from the Decoded ICache. This enables skipping the fetch and decode stages for these micro-ops and reduces power and latency of the Front End. The Decoded ICache provides average hit rates of above 80% of the micro-ops; furthermore, "hot spots" typically have hit rates close to 100%.

Typical integer programs average less than four bytes per instruction, and the front end is able to race ahead of the back end, filling in a large window for the scheduler to find instruction level parallelism. However, for high performance code with a basic block consisting of many instructions, for example, Intel SSE media algorithms or excessively unrolled loops, the 16 instruction bytes per cycle is occasionally a limitation. The 32-byte orientation of the Decoded ICache helps such code to avoid this limitation.

The Decoded ICache automatically improves performance of programs with temporal and spatial locality. However, to fully utilize the Decoded ICache potential, you might need to understand its internal organization.

The Decoded ICache consists of 32 sets. Each set contains eight Ways. Each Way can hold up to six micro-ops. The Decoded ICache can ideally hold up to 1536 micro-ops.

The following are some of the rules how the Decoded ICache is filled with micro-ops:

- All micro-ops in a Way represent instructions which are statically contiguous in the code and have their EIPs within the same aligned 32-byte region.

- Up to three Ways may be dedicated to the same 32-byte aligned chunk, allowing a total of 18 micro-ops to be cached per 32-byte region of the original IA program.
- A multi micro-op instruction cannot be split across Ways.
- Up to two branches are allowed per Way.
- An instruction which turns on the MSROM consumes an entire Way.
- A non-conditional branch is the last micro-op in a Way.
- Micro-fused micro-ops (load+op and stores) are kept as one micro-op.
- A pair of macro-fused instructions is kept as one micro-op.
- Instructions with 64-bit immediate require two slots to hold the immediate.

When micro-ops cannot be stored in the Decoded ICache due to these restrictions, they are delivered from the legacy decode pipeline. Once micro-ops are delivered from the legacy pipeline, fetching micro-ops from the Decoded ICache can resume only after the next branch micro-op. Frequent switches can incur a penalty.

The Decoded ICache is virtually included in the Instruction cache and ITLB. That is, any instruction with micro-ops in the Decoded ICache has its original instruction bytes present in the instruction cache. Instruction cache evictions must also be evicted from the Decoded ICache, which evicts only the necessary lines.

There are cases where the entire Decoded ICache is flushed. One reason for this can be an ITLB entry eviction. Other reasons are not usually visible to the application programmer, as they occur when important controls are changed, for example, mapping in CR3, or feature and mode enabling in CR0 and CR4. There are also cases where the Decoded ICache is disabled, for instance, when the CS base address is NOT set to zero.

2.3.2.3 Branch Prediction

Branch prediction predicts the branch target and enables the processor to begin executing instructions long before the branch true execution path is known. All branches utilize the branch prediction unit (BPU) for prediction. This unit predicts the target address not only based on the EIP of the branch but also based on the execution path through which execution reached this EIP. The BPU can efficiently predict the following branch types:

- Conditional branches.
- Direct calls and jumps.
- Indirect calls and jumps.
- Returns.

2.3.2.4 Micro-op Queue and the Loop Stream Detector (LSD)

The micro-op queue decouples the front end and the out-of order engine. It stays between the micro-op generation and the renamer as shown in Figure 2-5. This queue helps to hide bubbles which are introduced between the various sources of micro-ops in the front end and ensures that four micro-ops are delivered for execution, each cycle.

The micro-op queue provides post-decode functionality for certain instructions types. In particular, loads combined with computational operations and all stores, when used with indexed addressing, are represented as a single micro-op in the decoder or Decoded ICache. In the micro-op queue they are fragmented into two micro-ops through a process called un-lamination, one does the load and the other does the operation. A typical example is the following "load plus operation" instruction:

```
ADD                RAX, [RBP+RSI] ; rax := rax + LD( RBP+RSI )
```

Similarly, the following store instruction has three register sources and is broken into "generate store address" and "generate store data" sub-components.

```
MOV                [ESP+ECX*4+12345678], AL
```

The additional micro-ops generated by unlamination use the rename and retirement bandwidth. However, it has an overall power benefit. For code that is dominated by indexed addressing (as often happens with array processing), recoding algorithms to use base (or base+displacement) addressing can sometimes improve performance by keeping the load plus operation and store instructions fused.

The Loop Stream Detector (LSD)

The Loop Stream Detector was introduced in Intel® Core microarchitectures. The LSD detects small loops that fit in the micro-op queue and locks them down. The loop streams from the micro-op queue, with no more fetching, decoding, or reading micro-ops from any of the caches, until a branch mis-prediction inevitably ends it.

The loops with the following attributes qualify for LSD/micro-op queue replay:

- Up to eight chunk fetches of 32-instruction-bytes.
- Up to 28 micro-ops (~28 instructions).
- All micro-ops are also resident in the Decoded ICache.
- Can contain no more than eight taken branches and none of them can be a CALL or RET.
- Cannot have mismatched stack operations. For example, more PUSH than POP instructions.

Many calculation-intensive loops, searches and software string moves match these characteristics.

Use the loop cache functionality opportunistically. For high performance code, loop unrolling is generally preferable for performance even when it overflows the LSD capability.

2.3.3 The Out-of-Order Engine

The Out-of-Order engine provides improved performance over prior generations with excellent power characteristics. It detects dependency chains and sends them to execution out-of-order while maintaining the correct data flow. When a dependency chain is waiting for a resource, such as a second-level data cache line, it sends micro-ops from another chain to the execution core. This increases the overall rate of instructions executed per cycle (IPC).

The out-of-order engine consists of two blocks, shown in Figure 2-5: Core Functional Diagram, the Rename/retirement block, and the Scheduler.

The Out-of-Order engine contains the following major components:

Renamer. The Renamer component moves micro-ops from the front end to the execution core. It eliminates false dependencies among micro-ops, thereby enabling out-of-order execution of micro-ops.

Scheduler. The Scheduler component queues micro-ops until all source operands are ready. Schedules and dispatches ready micro-ops to the available execution units in as close to a first in first out (FIFO) order as possible.

Retirement. The Retirement component retires instructions and micro-ops in order and handles faults and exceptions.

2.3.3.1 Renamer

The Renamer is the bridge between the in-order part in Figure 2-5, and the dataflow world of the Scheduler. It moves up to four micro-ops every cycle from the micro-op queue to the out-of-order engine. Although the renamer can send up to 4 micro-ops (unfused, micro-fused, or macro-fused) per cycle, this is equivalent to the issue port can dispatch six micro-ops per cycle. In this process, the out-of-order core carries out the following steps:

- Renames architectural sources and destinations of the micro-ops to micro-architectural sources and destinations.
- Allocates resources to the micro-ops. For example, load or store buffers.
- Binds the micro-op to an appropriate dispatch port.

Some micro-ops can execute to completion during rename and are removed from the pipeline at that point, effectively costing no execution bandwidth. These include:

- Zero idioms (dependency breaking idioms).
- NOP.
- VZEROUPPER.
- FXCHG.

The renamer can allocate two branches each cycle, compared to one branch each cycle in the previous microarchitecture. This can eliminate some bubbles in execution.

Micro-fused load and store operations that use an index register are decomposed to two micro-ops, hence consume two out of the four slots the Renamer can use every cycle.

Dependency Breaking Idioms

Instruction parallelism can be improved by using common instructions to clear register contents to zero. The renamer can detect them on the zero evaluation of the destination register.

Use one of these dependency breaking idioms to clear a register when possible.

- XOR REG,REG
- SUB REG,REG
- PXOR/VPXOR XMMREG,XMMREG
- PSUBB/W/D/Q XMMREG,XMMREG
- VPSUBB/W/D/Q XMMREG,XMMREG
- XORPS/PD XMMREG,XMMREG
- VXORPS/PD YMMREG, YMMREG

Since zero idioms are detected and removed by the renamer, they have no execution latency.

There is another dependency breaking idiom - the "ones idiom".

- CMPEQ XMM1, XMM1; "ones idiom" set all elements to all "ones"

In this case, the micro-op must execute, however, since it is known that regardless of the input data the output data is always "all ones" the micro-op dependency upon its sources does not exist as with the zero idiom and it can execute as soon as it finds a free execution port.

2.3.3.2 Scheduler

The scheduler controls the dispatch of micro-ops onto their execution ports. In order to do this, it must identify which micro-ops are ready and where its sources come from: a register file entry, or a bypass directly from an execution unit. Depending on the availability of dispatch ports and writeback buses, and the priority of ready micro-ops, the scheduler selects which micro-ops are dispatched every cycle.

2.3.4 The Execution Core

The execution core is superscalar and can process instructions out of order. The execution core optimizes overall performance by handling the most common operations efficiently, while minimizing potential delays.

The out-of-order execution core improves execution unit organization over prior generation in the following ways:

- Reduction in read port stalls.
- Reduction in writeback conflicts and delays.
- Reduction in power.
- Reduction of SIMD FP assists dealing with denormal inputs and underflow outputs.

Some high precision FP algorithms need to operate with FTZ=0 and DAZ=0, i.e. permitting underflowed intermediate results and denormal inputs to achieve higher numerical precision at the expense of reduced performance on prior generation microarchitectures due to SIMD FP assists. The reduction of SIMD FP assists in Intel microarchitecture code name Sandy Bridge applies to the following SSE instructions (and AVX variants): ADDPD/ADDPS, MULPD/MULPS, DIVPD/DIVPS, and CVTPD2PS.

The out-of-order core consist of three execution stacks, where each stack encapsulates a certain type of data. The execution core contains the following execution stacks:

- General purpose integer.
- SIMD integer and floating-point.
- X87.

The execution core also contains connections to and from the cache hierarchy. The loaded data is fetched from the caches and written back into one of the stacks.

The scheduler can dispatch up to six micro-ops every cycle, one on each port. The following table summarizes which operations can be dispatched on which port.

Table 2-14. Dispatch Port and Execution Stacks

	Port 0	**Port 1**	**Port 2**	**Port 3**	**Port 4**	**Port 5**
Integer	ALU, Shift	ALU, Fast LEA, Slow LEA, MUL	Load_Addr, Store_addr	Load_Addr Store_addr	Store_data	ALU, Shift, Branch, Fast LEA
SSE-Int, AVX-Int, MMX	Mul, Shift, STTNI, Int-Div, 128b-Mov	ALU, Shuf, Blend, 128b-Mov			Store_data	ALU, Shuf, Shift, Blend, 128b-Mov
SSE-FP, AVX-FP_low	Mul, Div, Blend, 256b-Mov	Add, CVT			Store_data	Shuf, Blend, 256b-Mov
X87, AVX-FP_High	Mul, Div, Blend, 256b-Mov	Add, CVT			Store_data	Shuf, Blend, 256b-Mov

After execution, the data is written back on a writeback bus corresponding to the dispatch port and the data type of the result. Micro-ops that are dispatched on the same port but have different latencies may need the write back bus at the same cycle. In these cases the execution of one of the micro-ops is delayed until the writeback bus is available. For example, MULPS (five cycles) and BLENDPS (one cycle) may collide if both are ready for execution on port 0: first the MULPS and four cycles later the BLENDPS. Intel microarchitecture code name Sandy Bridge eliminates such collisions as long as the micro-ops write the results to different stacks. For example, integer ADD (one cycle) can be dispatched four cycles after MULPS (five cycles) since the integer ADD uses the integer stack while the MULPS uses the FP stack.

When a source of a micro-op executed in one stack comes from a micro-op executed in another stack, a one- or two-cycle delay can occur. The delay occurs also for transitions between Intel SSE integer and Intel SSE floating-point operations. In some of the cases the data transition is done using a micro-op that is added to the instruction flow. The following table describes how data, written back after execution, can bypass to micro-op execution in the following cycles.

Table 2-15. Execution Core Writeback Latency (cycles)

	Integer	SSE-Int, AVX-Int, MMX	SSE-FP, AVX-FP_low	X87, AVX-FP_High
Integer	0	micro-op (port 0)	micro-op (port 0)	micro-op (port 0) + 1 cycle
SSE-Int, AVX-Int, MMX	micro-op (port 5) or micro-op (port 5) +1 cycle	0	1 cycle delay	0
SSE-FP, AVX-FP_low	micro-op (port 5) or micro-op (port 5) +1 cycle	1 cycle delay	0	micro-op (port 5) +1 cycle
X87, AVX-FP_High	micro-op (port 5) +1 cycle	0	micro-op (port 5) +1 cycle	0
Load	0	1 cycle delay	1 cycle delay	2 cycle delay

2.3.5 Cache Hierarchy

The cache hierarchy contains a first level instruction cache, a first level data cache (L1 DCache) and a second level (L2) cache, in each core. The L1D cache may be shared by two logical processors if the processor support Intel HyperThreading Technology. The L2 cache is shared by instructions and data. All cores in a physical processor package connect to a shared last level cache (LLC) via a ring connection.

The caches use the services of the Instruction Translation Lookaside Buffer (ITLB), Data Translation Lookaside Buffer (DTLB) and Shared Translation Lookaside Buffer (STLB) to translate linear addresses to physical address. Data coherency in all cache levels is maintained using the MESI protocol. For more information, see the Intel® 64 IA-32 Architectures Software Developer's Manual, Volume 3. Cache hierarchy details can be obtained at run-time using the CPUID instruction. see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Table 2-16. Cache Parameters

Level	Capacity	Associativity (ways)	Line Size (bytes)	Write Update Policy	Inclusive
L1 Data	32 KB	8	64	Writeback	-
Instruction	32 KB	8	N/A	N/A	-
L2 (Unified)	256 KB	8	64	Writeback	No
Third Level (LLC)	Varies, query CPUID leaf 4	Varies with cache size	64	Writeback	Yes

2.3.5.1 Load and Store Operation Overview

This section provides an overview of the load and store operations.

Loads

When an instruction reads data from a memory location that has write-back (WB) type, the processor looks for it in the caches and memory. Table 2-17 shows the access lookup order and best case latency. The actual latency can vary depending on the cache queue occupancy, LLC ring occupancy, memory components, and their parameters.

Table 2-17. Lookup Order and Load Latency

Level	Latency (cycles)	Bandwidth (per core per cycle)
L1 Data	4[1]	2 x16 bytes
L2 (Unified)	12	1 x 32 bytes
Third Level (LLC)	26-31[2]	1 x 32 bytes
L2 and L1 DCache in other cores if applicable	43- clean hit; 60 - dirty hit	

NOTES:

1. Subject to execution core bypass restriction shown in Table 2-15.
2. Latency of L3 varies with product segment and sku. The values apply to second generation Intel Core processor families.

The LLC is inclusive of all cache levels above it - data contained in the core caches must also reside in the LLC. Each cache line in the LLC holds an indication of the cores that may have this line in their L2 and L1 caches. If there is an indication in the LLC that other cores may hold the line of interest and its state might have to modify, there is a lookup into the L1 DCache and L2 of these cores too. The lookup is called "clean" if it does not require fetching data from the other core caches. The lookup is called "dirty" if modified data has to be fetched from the other core caches and transferred to the loading core.

The latencies shown above are the best-case scenarios. Sometimes a modified cache line has to be evicted to make space for a new cache line. The modified cache line is evicted in parallel to bringing the new data and does not require additional latency. However, when data is written back to memory, the eviction uses cache bandwidth and possibly memory bandwidth as well. Therefore, when multiple cache misses require the eviction of modified lines within a short time, there is an overall degradation in cache response time. Memory access latencies vary based on occupancy of the memory controller queues, DRAM configuration, DDR parameters, and DDR paging behavior (if the requested page is a page-hit, page-miss or page-empty).

Stores

When an instruction writes data to a memory location that has a write back memory type, the processor first ensures that it has the line containing this memory location in its L1 DCache, in Exclusive or Modified MESI state. If the cache line is not there, in the right state, the processor fetches it from the next levels of the memory hierarchy using a Read for Ownership request. The processor looks for the cache line in the following locations, in the specified order:

1. L1 DCache
2. L2
3. Last Level Cache
4. L2 and L1 DCache in other cores, if applicable
5. Memory

Once the cache line is in the L1 DCache, the new data is written to it, and the line is marked as Modified.

Reading for ownership and storing the data happens after instruction retirement and follows the order of store instruction retirement. Therefore, the store latency usually does not affect the store instruction itself. However, several sequential stores that miss the L1 DCache may have cumulative latency that can affect performance. As long as the store does not complete, its entry remains occupied in the store buffer. When the store buffer becomes full, new micro-ops cannot enter the execution pipe and execution might stall.

2.3.5.2 L1 DCache

The L1 DCache is the first level data cache. It manages all load and store requests from all types through its internal data structures. The L1 DCache:

- Enables loads and stores to issue speculatively and out of order.
- Ensures that retired loads and stores have the correct data upon retirement.
- Ensures that loads and stores follow the memory ordering rules of the IA-32 and Intel 64 instruction set architecture.

Table 2-18. L1 Data Cache Components

Component	Intel microarchitecture code name Sandy Bridge	Intel microarchitecture code name Nehalem
Data Cache Unit (DCU)	32KB, 8 ways	32KB, 8 ways
Load buffers	64 entries	48 entries
Store buffers	36 entries	32 entries
Line fill buffers (LFB)	10 entries	10 entries

The DCU is organized as 32 KBytes, eight-way set associative. Cache line size is 64-bytes arranged in eight banks.

Internally, accesses are up to 16 bytes, with 256-bit Intel AVX instructions utilizing two 16-byte accesses. Two load operations and one store operation can be handled each cycle.

The L1 DCache maintains requests which cannot be serviced immediately to completion. Some reasons for requests that are delayed: cache misses, unaligned access that splits across cache lines, data not ready to be forwarded from a preceding store, loads experiencing bank collisions, and load block due to cache line replacement.

The L1 DCache can maintain up to 64 load micro-ops from allocation until retirement. It can maintain up to 36 store operations from allocation until the store value is committed to the cache, or written to the line fill buffers (LFB) in the case of non-temporal stores.

The L1 DCache can handle multiple outstanding cache misses and continue to service incoming stores and loads. Up to 10 requests of missing cache lines can be managed simultaneously using the LFB.

The L1 DCache is a write-back write-allocate cache. Stores that hit in the DCU do not update the lower levels of the memory hierarchy. Stores that miss the DCU allocate a cache line.

Loads

The L1 DCache architecture can service two loads per cycle, each of which can be up to 16 bytes. Up to 32 loads can be maintained at different stages of progress, from their allocation in the out of order engine until the loaded value is returned to the execution core.

Loads can:

- Read data before preceding stores when the load address and store address ranges are known not to conflict.
- Be carried out speculatively, before preceding branches are resolved.
- Take cache misses out of order and in an overlapped manner.

Loads cannot:

- Speculatively take any sort of fault or trap.
- Speculatively access uncacheable memory.

The common load latency is five cycles. When using a simple addressing mode, base plus offset that is smaller than 2048, the load latency can be four cycles. This technique is especially useful for pointer-chasing code. However, overall latency varies depending on the target register data type due to stack bypass. See Section 2.3.4 for more information.

The following table lists overall load latencies. These latencies assume the common case of flat segment, that is, segment base address is zero. If segment base is not zero, load latency increases.

Table 2-19. Effect of Addressing Modes on Load Latency

Data Type/Addressing Mode	Base + Offset > 2048; Base + Index [+ Offset]	Base + Offset < 2048
Integer	5	4
MMX, SSE, 128-bit AVX	6	5
X87	7	6
256-bit AVX	7	7

Stores

Stores to memory are executed in two phases:

- Execution phase. Fills the store buffers with linear and physical address and data. Once store address and data are known, the store data can be forwarded to the following load operations that need it.
- Completion phase. After the store retires, the L1 DCache moves its data from the store buffers to the DCU, up to 16 bytes per cycle.

Address Translation

The DTLB can perform three linear to physical address translations every cycle, two for load addresses and one for a store address. If the address is missing in the DTLB, the processor looks for it in the STLB, which holds data and instruction address translations. The penalty of a DTLB miss that hits the STLB is seven cycles. Large page support include 1G byte pages, in addition to 4K and 2M/4M pages.

The DTLB and STLB are four way set associative. The following table specifies the number of entries in the DTLB and STLB.

Table 2-20. DTLB and STLB Parameters

TLB	Page Size	Entries
DTLB	4KB	64
	2MB/4MB	32
	1GB	4
STLB	4KB	512

Store Forwarding

If a load follows a store and reloads the data that the store writes to memory, the data can forward directly from the store operation to the load. This process, called store to load forwarding, saves cycles by enabling the load to obtain the data directly from the store operation instead of through memory. You can take advantage of store forwarding to quickly move complex structures without losing the ability to forward the subfields. The memory control unit can handle store forwarding situations with less restrictions compared to previous micro-architectures.

The following rules must be met to enable store to load forwarding:

- The store must be the last store to that address, prior to the load.
- The store must contain all data being loaded.
- The load is from a write-back memory type and neither the load nor the store are non-temporal accesses.

Stores cannot forward to loads in the following cases:

- Four byte and eight byte loads that cross eight byte boundary, relative to the preceding 16- or 32-byte store.
- Any load that crosses a 16-byte boundary of a 32-byte store.

Table 2-21 to Table 2-24 detail the store to load forwarding behavior. For a given store size, all the loads that may overlap are shown and specified by ‘F’. Forwarding from 32 byte store is similar to forwarding from each of the 16 byte halves of the store. Cases that cannot forward are shown as ‘N’.

Table 2-21. Store Forwarding Conditions (1 and 2 byte stores)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	F															
2	1	F	F														
	2	F	N														

Table 2-22. Store Forwarding Conditions (4-16 byte stores)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	1	F	F	F	F												
	2	F	F	F	N												
	4	F	N	N	N												
8	1	F	F	F	F	F	F	F	F								
	2	F	F	F	F	F	F	F	N								
	4	F	F	F	F	F	N	N	N								
	8	F	N	N	N	N	N	N	N								
16	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Table 2-23. 32-byte Store Forwarding Conditions (0-15 byte alignment)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	32	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Table 2-24. 32-byte Store Forwarding Conditions (16-31 byte alignment)

		Load Alignment															
Store Size	Load Size	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	32	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Memory Disambiguation

A load operation may depend on a preceding store. Many microarchitectures block loads until all preceding store addresses are known. The memory disambiguator predicts which loads will not depend on any previous stores. When the disambiguator predicts that a load does not have such a dependency, the load takes its data from the L1 data cache even when the store address is unknown. This hides the load latency. Eventually, the prediction is verified. If an actual conflict is detected, the load and all succeeding instructions are re-executed.

The following loads are not disambiguated. The execution of these loads is stalled until addresses of all previous stores are known.

- Loads that cross the 16-byte boundary
- 32-byte Intel AVX loads that are not 32-byte aligned.

The memory disambiguator always assumes dependency between loads and earlier stores that have the same address bits 0:11.

Bank Conflict

Since 16-byte loads can cover up to three banks, and two loads can happen every cycle, it is possible that six of the eight banks may be accessed per cycle, for loads. A bank conflict happens when two load accesses need the same bank (their address has the same 2-4 bit value) in different sets, at the same time. When a bank conflict occurs, one of the load accesses is recycled internally.

In many cases two loads access exactly the same bank in the same cache line, as may happen when popping operands off the stack, or any sequential accesses. In these cases, conflict does not occur and the loads are serviced simultaneously.

2.3.5.3 Ring Interconnect and Last Level Cache

The system-on-a-chip design provides a high bandwidth bi-directional ring bus to connect between the IA cores and various sub-systems in the uncore. In the second generation Intel Core processor 2xxx series, the uncore subsystem include a system agent, the graphics unit (GT) and the last level cache (LLC).

The LLC consists of multiple cache slices. The number of slices is equal to the number of IA cores. Each slice has logic portion and data array portion. The logic portion handles data coherency, memory ordering, access to the data array portion, LLC misses and writeback to memory, and more. The data array portion stores cache lines. Each slice contains a full cache port that can supply 32 bytes/cycle.

The physical addresses of data kept in the LLC data arrays are distributed among the cache slices by a hash function, such that addresses are uniformly distributed. The data array in a cache block may have 4/8/12/16 ways corresponding to 0.5M/1M/1.5M/2M block size. However, due to the address distribution among the cache blocks from the software point of view, this does not appear as a normal N-way cache.

From the processor cores and the GT view, the LLC act as one shared cache with multiple ports and bandwidth that scales with the number of cores. The LLC hit latency, ranging between 26-31 cycles, depends on the core location relative to the LLC block, and how far the request needs to travel on the ring.

The number of cache-slices increases with the number of cores, therefore the ring and LLC are not likely to be a bandwidth limiter to core operation.

The GT sits on the same ring interconnect, and uses the LLC for its data operations as well. In this respect it is very similar to an IA core. Therefore, high bandwidth graphic applications using cache bandwidth and significant cache footprint, can interfere, to some extent, with core operations.

All the traffic that cannot be satisfied by the LLC, such as LLC misses, dirty line writeback, non-cacheable operations, and MMIO/IO operations, still travels through the cache-slice logic portion and the ring, to the system agent.

In the Intel Xeon Processor E5 Family, the uncore subsystem does not include the graphics unit (GT). Instead, the uncore subsystem contains many more components, including an LLC with larger capacity and snooping capabilities to support multiple processors, Intel® QuickPath Interconnect interfaces that can support multi-socket platforms, power management control hardware, and a system agent capable of supporting high-bandwidth traffic from memory and I/O devices.

In the Intel Xeon processor E5 2xxx or 4xxx families, the LLC capacity generally scales with the number of processor cores with 2.5 MBytes per core.

2.3.5.4 Data Prefetching

Data can be speculatively loaded to the L1 DCache using software prefetching, hardware prefetching, or any combination of the two.

You can use the four Streaming SIMD Extensions (SSE) prefetch instructions to enable software-controlled prefetching. These instructions are hints to bring a cache line of data into the desired levels of the cache hierarchy. The software-controlled prefetch is intended for prefetching data, but not for prefetching code.

The rest of this section describes the various hardware prefetching mechanisms provided by Intel microarchitecture code name Sandy Bridge and their improvement over previous processors. The goal of the prefetchers is to automatically predict which data the program is about to consume. If this data is not close-by to the execution core or inner cache, the prefetchers bring it from the next levels of cache hierarchy and memory. Prefetching has the following effects:

- Improves performance if data is arranged sequentially in the order used in the program.
- May cause slight performance degradation due to bandwidth issues, if access patterns are sparse instead of local.
- On rare occasions, if the algorithm's working set is tuned to occupy most of the cache and unneeded prefetches evict lines required by the program, hardware prefetcher may cause severe performance degradation due to cache capacity of L1.

Data Prefetch to L1 Data Cache

Data prefetching is triggered by load operations when the following conditions are met:

- Load is from writeback memory type.
- The prefetched data is within the same 4K byte page as the load instruction that triggered it.
- No fence is in progress in the pipeline.
- Not many other load misses are in progress.
- There is not a continuous stream of stores.

Two hardware prefetchers load data to the L1 DCache:

- **Data cache unit (DCU) prefetcher**. This prefetcher, also known as the streaming prefetcher, is triggered by an ascending access to very recently loaded data. The processor assumes that this access is part of a streaming algorithm and automatically fetches the next line.

- **Instruction pointer (IP)-based stride prefetcher**. This prefetcher keeps track of individual load instructions. If a load instruction is detected to have a regular stride, then a prefetch is sent to the next address which is the sum of the current address and the stride. This prefetcher can prefetch forward or backward and can detect strides of up to 2K bytes.

Data Prefetch to the L2 and Last Level Cache

The following two hardware prefetchers fetched data from memory to the L2 cache and last level cache:

Spatial Prefetcher: This prefetcher strives to complete every cache line fetched to the L2 cache with the pair line that completes it to a 128-byte aligned chunk.

Streamer: This prefetcher monitors read requests from the L1 cache for ascending and descending sequences of addresses. Monitored read requests include L1 DCache requests initiated by load and store operations and by the hardware prefetchers, and L1 ICache requests for code fetch. When a forward or backward stream of requests is detected, the anticipated cache lines are prefetched. Prefetched cache lines must be in the same 4K page.

The streamer and spatial prefetcher prefetch the data to the last level cache. Typically data is brought also to the L2 unless the L2 cache is heavily loaded with missing demand requests.

Enhancement to the streamer includes the following features:

- The streamer may issue two prefetch requests on every L2 lookup. The streamer can run up to 20 lines ahead of the load request.
- Adjusts dynamically to the number of outstanding requests per core. If there are not many outstanding requests, the streamer prefetches further ahead. If there are many outstanding requests it prefetches to the LLC only and less far ahead.
- When cache lines are far ahead, it prefetches to the last level cache only and not to the L2. This method avoids replacement of useful cache lines in the L2 cache.
- Detects and maintains up to 32 streams of data accesses. For each 4K byte page, you can maintain one forward and one backward stream can be maintained.

2.3.6 System Agent

The system agent implemented in the second generation Intel Core processor family contains the following components:

- An arbiter that handles all accesses from the ring domain and from I/O (PCIe* and DMI) and routes the accesses to the right place.
- PCIe controllers connect to external PCIe devices. The PCIe controllers have different configuration possibilities the varies with product segment specifics: x16+x4, x8+x8+x4, x8+x4+x4+x4.
- DMI controller connects to the PCH chipset.
- Integrated display engine, Flexible Display Interconnect, and Display Port, for the internal graphic operations.
- Memory controller.

All main memory traffic is routed from the arbiter to the memory controller. The memory controller in the second generation Intel Core processor 2xxx series support two channels of DDR, with data rates of 1066MHz, 1333MHz and 1600MHz, and 8 bytes per cycle, depending on the unit type, system configuration and DRAMs. Addresses are distributed between memory channels based on a local hash function that attempts to balance the load between the channels in order to achieve maximum bandwidth and minimum hotspot collisions.

For best performance, populate both channels with equal amounts of memory, preferably the exact same types of DIMMs. In addition, using more ranks for the same amount of memory, results in somewhat better memory bandwidth, since more DRAM pages can be open simultaneously. For best performance, populate the system with the highest supported speed DRAM (1333MHz or 1600MHz data rates, depending on the max supported frequency) with the best DRAM timings.

The two channels have separate resources and handle memory requests independently. The memory controller contains a high-performance out-of-order scheduler that attempts to maximize memory band-

width while minimizing latency. Each memory channel contains a 32 cache-line write-data-buffer. Writes to the memory controller are considered completed when they are written to the write-data-buffer. The write-data-buffer is flushed out to main memory at a later time, not impacting write latency.

Partial writes are not handled efficiently on the memory controller and may result in read-modify-write operations on the DDR channel if the partial-writes do not complete a full cache-line in time. Software should avoid creating partial write transactions whenever possible and consider alternative, such as buffering the partial writes into full cache line writes.

The memory controller also supports high-priority isochronous requests (such as USB isochronous, and Display isochronous requests). High bandwidth of memory requests from the integrated display engine takes up some of the memory bandwidth and impacts core access latency to some degree.

2.3.7 Intel® Microarchitecture Code Name Ivy Bridge

Third generation Intel Core processors are based on Intel microarchitecture code name Ivy Bridge. Most of the features described in Section 2.3.1 - Section 2.3.6 also apply to Intel microarchitecture code name Ivy Bridge. This section covers feature differences in microarchitecture that can affect coding and performance.

Support for new instructions enabling include:

- Numeric conversion to and from half-precision floating-point values.
- Hardware-based random number generator compliant to NIST SP 800-90A.
- Reading and writing to FS/GS base registers in any ring to improve user-mode threading support.

For details about using the hardware based random number generator instruction RDRAND, please refer to the article available from Intel Software Network at http://software.intel.com/en-us/articles/download-the-latest-bull-mountain-software-implementation-guide/?wapkw=bull+mountain.

A small number of microarchitectural enhancements that can be beneficial to software:

- Hardware prefetch enhancement: A next-page prefetcher (NPP) is added in Intel microarchitecture code name Ivy Bridge. The NPP is triggered by sequential accesses to cache lines approaching the page boundary, either upwards or downwards.
- Zero-latency register move operation: A subset of register-to-register MOV instructions are executed at the front end, conserving scheduling and execution resource in the out-of-order engine.
- Front end enhancement: In Intel microarchitecture code name Sandy Bridge, the micro-op queue is statically partitioned to provide 28 entries for each logical processor, irrespective of software executing in single thread or multiple threads. If one logical processor is not active in Intel microarchitecture code name Ivy Bridge, then a single thread executing on that processor core can use the 56 entries in the micro-op queue. In this case, the LSD can handle larger loop structure that would require more than 28 entries.
- The latency and throughput of some instructions have been improved over those of Intel microarchitecture code name Sandy Bridge. For example, 256-bit packed floating-point divide and square root operations are faster; ROL and ROR instructions are also improved.

2.4 INTEL® CORE™ MICROARCHITECTURE AND ENHANCED INTEL® CORE™ MICROARCHITECTURE

Intel Core microarchitecture introduces the following features that enable high performance and power-efficient performance for single-threaded as well as multi-threaded workloads:

- **Intel® Wide Dynamic Execution** enables each processor core to fetch, dispatch, execute with high bandwidths and retire up to four instructions per cycle. Features include:
  — Fourteen-stage efficient pipeline.
  — Three arithmetic logical units.

- — Four decoders to decode up to five instruction per cycle.
- — Macro-fusion and micro-fusion to improve front end throughput.
- — Peak issue rate of dispatching up to six micro-ops per cycle.
- — Peak retirement bandwidth of up to four micro-ops per cycle.
- — Advanced branch prediction.
- — Stack pointer tracker to improve efficiency of executing function/procedure entries and exits.

- **Intel® Advanced Smart Cache** delivers higher bandwidth from the second level cache to the core, optimal performance and flexibility for single-threaded and multi-threaded applications. Features include:
  - — Optimized for multicore and single-threaded execution environments.
  - — 256 bit internal data path to improve bandwidth from L2 to first-level data cache.
  - — Unified, shared second-level cache of 4 Mbyte, 16 way (or 2 MByte, 8 way).
- **Intel® Smart Memory Access** prefetches data from memory in response to data access patterns and reduces cache-miss exposure of out-of-order execution. Features include:
  - — Hardware prefetchers to reduce effective latency of second-level cache misses.
  - — Hardware prefetchers to reduce effective latency of first-level data cache misses.
  - — Memory disambiguation to improve efficiency of speculative execution engine.
- **Intel® Advanced Digital Media Boost** improves most 128-bit SIMD instructions with single-cycle throughput and floating-point operations. Features include:
  - — Single-cycle throughput of most 128-bit SIMD instructions (except 128-bit shuffle, pack, unpack operations)
  - — Up to eight floating-point operations per cycle
  - — Three issue ports available to dispatching SIMD instructions for execution.

The Enhanced Intel Core microarchitecture supports all of the features of Intel Core microarchitecture and provides a comprehensive set of enhancements.

- **Intel® Wide Dynamic Execution** includes several enhancements:
  - — A radix-16 divider replacing previous radix-4 based divider to speedup long-latency operations such as divisions and square roots.
  - — Improved system primitives to speedup long-latency operations such as RDTSC, STI, CLI, and VM exit transitions.
- **Intel® Advanced Smart Cache** provides up to 6 MBytes of second-level cache shared between two processor cores (quad-core processors have up to 12 MBytes of L2); up to 24 way/set associativity.
- **Intel® Smart Memory Access** supports high-speed system bus up 1600 MHz and provides more efficient handling of memory operations such as split cache line load and store-to-load forwarding situations.
- **Intel® Advanced Digital Media Boost** provides 128-bit shuffler unit to speedup shuffle, pack, unpack operations; adds support for 47 SSE4.1 instructions.

In the sub-sections of 2.1.x, most of the descriptions on Intel Core microarchitecture also applies to Enhanced Intel Core microarchitecture. Differences between them are note explicitly.

2.4.1 Intel® Core™ Microarchitecture Pipeline Overview

The pipeline of the Intel Core microarchitecture contains:

- An in-order issue front end that fetches instruction streams from memory, with four instruction decoders to supply decoded instruction (micro-ops) to the out-of-order execution core.

- An out-of-order superscalar execution core that can issue up to six micro-ops per cycle (see Table 2-26) and reorder micro-ops to execute as soon as sources are ready and execution resources are available.
- An in-order retirement unit that ensures the results of execution of micro-ops are processed and architectural states are updated according to the original program order.

Intel Core 2 Extreme processor X6800, Intel Core 2 Duo processors and Intel Xeon processor 3000, 5100 series implement two processor cores based on the Intel Core microarchitecture. Intel Core 2 Extreme quad-core processor, Intel Core 2 Quad processors and Intel Xeon processor 3200 series, 5300 series implement four processor cores. Each physical package of these quad-core processors contains two processor dies, each die containing two processor cores. The functionality of the subsystems in each core are depicted in Figure 2-6.

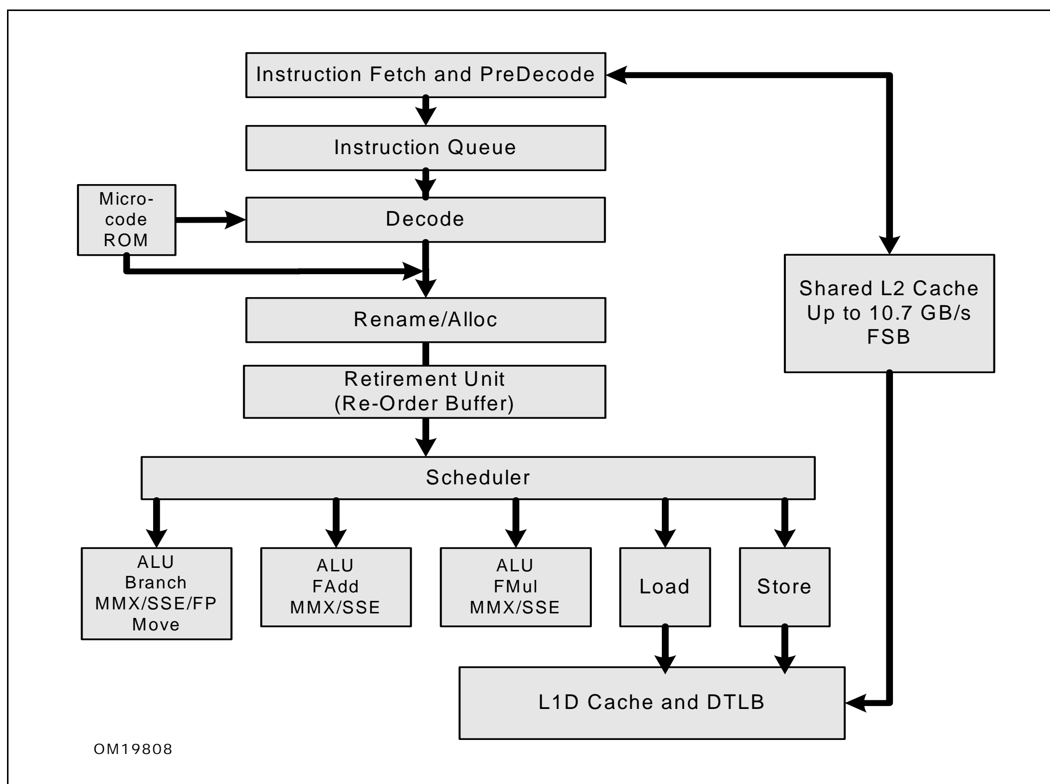


Figure 2-6. Intel Core Microarchitecture Pipeline Functionality

2.4.2 Front End

The front ends needs to supply decoded instructions (micro-ops) and sustain the stream to a six-issue wide out-of-order engine. The components of the front end, their functions, and the performance challenges to microarchitectural design are described in Table 2-25.

Table 2-25. Components of the Front End

Component	Functions	Performance Challenges
Branch Prediction Unit (BPU)	• Helps the instruction fetch unit fetch the most likely instruction to be executed by predicting the various branch types: conditional, indirect, direct, call, and return. Uses dedicated hardware for each type.	• Enables speculative execution. • Improves speculative execution efficiency by reducing the amount of code in the "non-architected path"[1] to be fetched into the pipeline.

Table 2-25. Components of the Front End (Contd.)

Component	Functions	Performance Challenges
Instruction Fetch Unit	• Prefetches instructions that are likely to be executed • Caches frequently-used instructions • Predecodes and buffers instructions, maintaining a constant bandwidth despite irregularities in the instruction stream	• Variable length instruction format causes unevenness (bubbles) in decode bandwidth. • Taken branches and misaligned targets causes disruptions in the overall bandwidth delivered by the fetch unit.
Instruction Queue and Decode Unit	• Decodes up to four instructions, or up to five with macro-fusion • Stack pointer tracker algorithm for efficient procedure entry and exit • Implements the Macro-Fusion feature, providing higher performance and efficiency • The Instruction Queue is also used as a loop cache, enabling some loops to be executed with both higher bandwidth and lower power	• Varying amounts of work per instruction requires expansion into variable numbers of micro-ops. • Prefix adds a dimension of decoding complexity. • Length Changing Prefix (LCP) can cause front end bubbles.

NOTES:

1. Code paths that the processor thought it should execute but then found out it should go in another path and therefore reverted from its initial intention.

2.4.2.1 Branch Prediction Unit

Branch prediction enables the processor to begin executing instructions long before the branch outcome is decided. All branches utilize the BPU for prediction. The BPU contains the following features:

- 16-entry Return Stack Buffer (RSB). It enables the BPU to accurately predict RET instructions.
- Front end queuing of BPU lookups. The BPU makes branch predictions for 32 bytes at a time, twice the width of the fetch engine. This enables taken branches to be predicted with no penalty.

  Even though this BPU mechanism generally eliminates the penalty for taken branches, software should still regard taken branches as consuming more resources than do not-taken branches.

The BPU makes the following types of predictions:

- Direct Calls and Jumps. Targets are read as a target array, without regarding the taken or not-taken prediction.
- Indirect Calls and Jumps. These may either be predicted as having a monotonic target or as having targets that vary in accordance with recent program behavior.
- Conditional branches. Predicts the branch target and whether or not the branch will be taken.

For information about optimizing software for the BPU, see Section 3.4, "Optimizing the Front End."

2.4.2.2 Instruction Fetch Unit

The instruction fetch unit comprises the instruction translation lookaside buffer (ITLB), an instruction prefetcher, the instruction cache and the predecode logic of the instruction queue (IQ).

Instruction Cache and ITLB

An instruction fetch is a 16-byte aligned lookup through the ITLB into the instruction cache and instruction prefetch buffers. A hit in the instruction cache causes 16 bytes to be delivered to the instruction predecoder. Typical programs average slightly less than 4 bytes per instruction, depending on the code

being executed. Since most instructions can be decoded by all decoders, an entire fetch can often be consumed by the decoders in one cycle.

A misaligned target reduces the number of instruction bytes by the amount of offset into the 16 byte fetch quantity. A taken branch reduces the number of instruction bytes delivered to the decoders since the bytes after the taken branch are not decoded. Branches are taken approximately every 10 instructions in typical integer code, which translates into a "partial" instruction fetch every 3 or 4 cycles.

Due to stalls in the rest of the machine, front end starvation does not usually cause performance degradation. For extremely fast code with larger instructions (such as SSE2 integer media kernels), it may be beneficial to use targeted alignment to prevent instruction starvation.

Instruction PreDecode

The predecode unit accepts the sixteen bytes from the instruction cache or prefetch buffers and carries out the following tasks:

- Determine the length of the instructions.
- Decode all prefixes associated with instructions.
- Mark various properties of instructions for the decoders (for example, "is branch.").

The predecode unit can write up to six instructions per cycle into the instruction queue. If a fetch contains more than six instructions, the predecoder continues to decode up to six instructions per cycle until all instructions in the fetch are written to the instruction queue. Subsequent fetches can only enter predecoding after the current fetch completes.

For a fetch of seven instructions, the predecoder decodes the first six in one cycle, and then only one in the next cycle. This process would support decoding 3.5 instructions per cycle. Even if the instruction per cycle (IPC) rate is not fully optimized, it is higher than the performance seen in most applications. In general, software usually does not have to take any extra measures to prevent instruction starvation.

The following instruction prefixes cause problems during length decoding. These prefixes can dynamically change the length of instructions and are known as length changing prefixes (LCPs):

- Operand Size Override (66H) preceding an instruction with a word immediate data.
- Address Size Override (67H) preceding an instruction with a mod R/M in real, 16-bit protected or 32-bit protected modes.

When the predecoder encounters an LCP in the fetch line, it must use a slower length decoding algorithm. With the slower length decoding algorithm, the predecoder decodes the fetch in 6 cycles, instead of the usual 1 cycle.

Normal queuing within the processor pipeline usually cannot hide LCP penalties.

The REX prefix (4xh) in the Intel 64 architecture instruction set can change the size of two classes of instruction: MOV offset and MOV immediate. Nevertheless, it does not cause an LCP penalty and hence is not considered an LCP.

2.4.2.3 Instruction Queue (IQ)

The instruction queue is 18 instructions deep. It sits between the instruction predecode unit and the instruction decoders. It sends up to five instructions per cycle, and supports one macro-fusion per cycle. It also serves as a loop cache for loops smaller than 18 instructions. The loop cache operates as described below.

A Loop Stream Detector (LSD) resides in the BPU. The LSD attempts to detect loops which are candidates for streaming from the instruction queue (IQ). When such a loop is detected, the instruction bytes are locked down and the loop is allowed to stream from the IQ until a misprediction ends it. When the loop plays back from the IQ, it provides higher bandwidth at reduced power (since much of the rest of the front end pipeline is shut off).

The LSD provides the following benefits:

- No loss of bandwidth due to taken branches.

- No loss of bandwidth due to misaligned instructions.
- No LCP penalties, as the pre-decode stage has already been passed.
- Reduced front end power consumption, because the instruction cache, BPU and predecode unit can be idle.

Software should use the loop cache functionality opportunistically. Loop unrolling and other code optimizations may make the loop too big to fit into the LSD. For high performance code, loop unrolling is generally preferable for performance even when it overflows the loop cache capability.

2.4.2.4 Instruction Decode

The Intel Core microarchitecture contains four instruction decoders. The first, Decoder 0, can decode Intel 64 and IA-32 instructions up to 4 micro-ops in size. Three other decoders handle single micro-op instructions. The microsequencer can provide up to 3 micro-ops per cycle, and helps decode instructions larger than 4 micro-ops.

All decoders support the common cases of single micro-op flows, including: micro-fusion, stack pointer tracking and macro-fusion. Thus, the three simple decoders are not limited to decoding single micro-op instructions. Packing instructions into a 4-1-1-1 template is not necessary and not recommended.

Macro-fusion merges two instructions into a single micro-op. Intel Core microarchitecture is capable of one macro-fusion per cycle in 32-bit operation (including compatibility sub-mode of the Intel 64 architecture), but not in 64-bit mode because code that uses longer instructions (length in bytes) more often is less likely to take advantage of hardware support for macro-fusion.

2.4.2.5 Stack Pointer Tracker

The Intel 64 and IA-32 architectures have several commonly used instructions for parameter passing and procedure entry and exit: PUSH, POP, CALL, LEAVE and RET. These instructions implicitly update the stack pointer register (RSP), maintaining a combined control and parameter stack without software intervention. These instructions are typically implemented by several micro-ops in previous microarchitectures.

The Stack Pointer Tracker moves all these implicit RSP updates to logic contained in the decoders themselves. The feature provides the following benefits:

- Improves decode bandwidth, as PUSH, POP and RET are single micro-op instructions in Intel Core microarchitecture.
- Conserves execution bandwidth as the RSP updates do not compete for execution resources.
- Improves parallelism in the out of order execution engine as the implicit serial dependencies between micro-ops are removed.
- Improves power efficiency as the RSP updates are carried out on small, dedicated hardware.

2.4.2.6 Micro-fusion

Micro-fusion fuses multiple micro-ops from the same instruction into a single complex micro-op. The complex micro-op is dispatched in the out-of-order execution core. Micro-fusion provides the following performance advantages:

- Improves instruction bandwidth delivered from decode to retirement.
- Reduces power consumption as the complex micro-op represents more work in a smaller format (in terms of bit density), reducing overall “bit-toggling” in the machine for a given amount of work and virtually increasing the amount of storage in the out-of-order execution engine.

Many instructions provide register flavors and memory flavors. The flavor involving a memory operand will decodes into a longer flow of micro-ops than the register version. Micro-fusion enables software to use memory to register operations to express the actual program behavior without worrying about a loss of decode bandwidth.

2.4.3 Execution Core

The execution core of the Intel Core microarchitecture is superscalar and can process instructions out of order. When a dependency chain causes the machine to wait for a resource (such as a second-level data cache line), the execution core executes other instructions. This increases the overall rate of instructions executed per cycle (IPC).

The execution core contains the following three major components:

- **Renamer** — Moves micro-ops from the front end to the execution core. Architectural registers are renamed to a larger set of microarchitectural registers. Renaming eliminates false dependencies known as read-after-read and write-after-read hazards.
- **Reorder buffer** (ROB) — Holds micro-ops in various stages of completion, buffers completed micro-ops, updates the architectural state in order, and manages ordering of exceptions. The ROB has 96 entries to handle instructions in flight.
- **Reservation station** (RS) — Queues micro-ops until all source operands are ready, schedules and dispatches ready micro-ops to the available execution units. The RS has 32 entries.

The initial stages of the out of order core move the micro-ops from the front end to the ROB and RS. In this process, the out of order core carries out the following steps:

- Allocates resources to micro-ops (for example: these resources could be load or store buffers).
- Binds the micro-op to an appropriate issue port.
- Renames sources and destinations of micro-ops, enabling out of order execution.
- Provides data to the micro-op when the data is either an immediate value or a register value that has already been calculated.

The following list describes various types of common operations and how the core executes them efficiently:

- **Micro-ops with single-cycle latency** — Most micro-ops with single-cycle latency can be executed by multiple execution units, enabling multiple streams of dependent operations to be executed quickly.
- **Frequently-used μops with longer latency** — These micro-ops have pipelined execution units so that multiple micro-ops of these types may be executing in different parts of the pipeline simultaneously.
- **Operations with data-dependent latencies** — Some operations, such as division, have data dependent latencies. Integer division parses the operands to perform the calculation only on significant portions of the operands, thereby speeding up common cases of dividing by small numbers.
- **Floating-point operations with fixed latency for operands that meet certain restrictions** — Operands that do not fit these restrictions are considered exceptional cases and are executed with higher latency and reduced throughput. The lower-throughput cases do not affect latency and throughput for more common cases.
- **Memory operands with variable latency, even in the case of an L1 cache hit** — Loads that are not known to be safe from forwarding may wait until a store-address is resolved before executing. The memory order buffer (MOB) accepts and processes all memory operations. See Section 2.4.4 for more information about the MOB.

2.4.3.1 Issue Ports and Execution Units

The scheduler can dispatch up to six micro-ops per cycle through the issue ports. The issue ports of Intel Core microarchitecture and Enhanced Intel Core microarchitecture are depicted in Table 2-26, the former is denoted by its CPUID signature of DisplayFamily_DisplayModel value of 06_0FH, the latter denoted by the corresponding signature value of 06_17H. The table provides latency and throughput data of common integer and floating-point (FP) operations for each issue port in cycles.

Table 2-26. Issue Ports of Intel Core Microarchitecture and Enhanced Intel Core Microarchitecture

Executable operations	Latency, Throughput		Comment[1]
	Signature = 06_0FH	Signature = 06_17H	
Integer ALU Integer SIMD ALU FP/SIMD/SSE2 Move and Logic	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	Includes 64-bit mode integer MUL; Issue port 0; Writeback port 0;
Single-precision (SP) FP MUL Double-precision FP MUL	4, 1 5, 1	4, 1 5, 1	Issue port 0; Writeback port 0
FP MUL (X87) FP Shuffle DIV/SQRT	5, 2 1, 1	5, 2 1, 1	Issue port 0; Writeback port 0 FP shuffle does not handle QW shuffle.
Integer ALU Integer SIMD ALU FP/SIMD/SSE2 Move and Logic	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	Excludes 64-bit mode integer MUL; Issue port 1; Writeback port 1;
FP ADD QW Shuffle	3, 1 1, 1[2]	3, 1 1, 1[3]	Issue port 1; Writeback port 1;
Integer loads FP loads	3, 1 4, 1	3, 1 4, 1	Issue port 2; Writeback port 2;
Store address[4]	3, 1	3, 1	Issue port 3;
Store data[5].			Issue Port 4;
Integer ALU Integer SIMD ALU FP/SIMD/SSE2 Move and Logic	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	Issue port 5; Writeback port 5;
QW shuffles 128-bit Shuffle/Pack/Unpack	1, 1[2] 2-4, 2-4[6]	1, 1[3] 1-3, 1[7]	Issue port 5; Writeback port 5;

NOTES:

1. Mixing operations of different latencies that use the same port can result in writeback bus conflicts; this can reduce overall throughput.
2. 128-bit instructions executes with longer latency and reduced throughput.
3. Uses 128-bit shuffle unit in port 5.
4. Prepares the store forwarding and store retirement logic with the address of the data being stored.
5. Prepares the store forwarding and store retirement logic with the data being stored.
6. Varies with instructions; 128-bit instructions are executed using QW shuffle units.
7. Varies with instructions, 128-bit shuffle unit replaces QW shuffle units in Intel Core microarchitecture.

In each cycle, the RS can dispatch up to six micro-ops. Each cycle, up to 4 results may be written back to the RS and ROB, to be used as early as the next cycle by the RS. This high execution bandwidth enables execution bursts to keep up with the functional expansion of the micro-fused micro-ops that are decoded and retired.

The execution core contains the following three execution stacks:

- SIMD integer.
- Regular integer.
- x87/SIMD floating-point.

The execution core also contains connections to and from the memory cluster. See Figure 2-7.

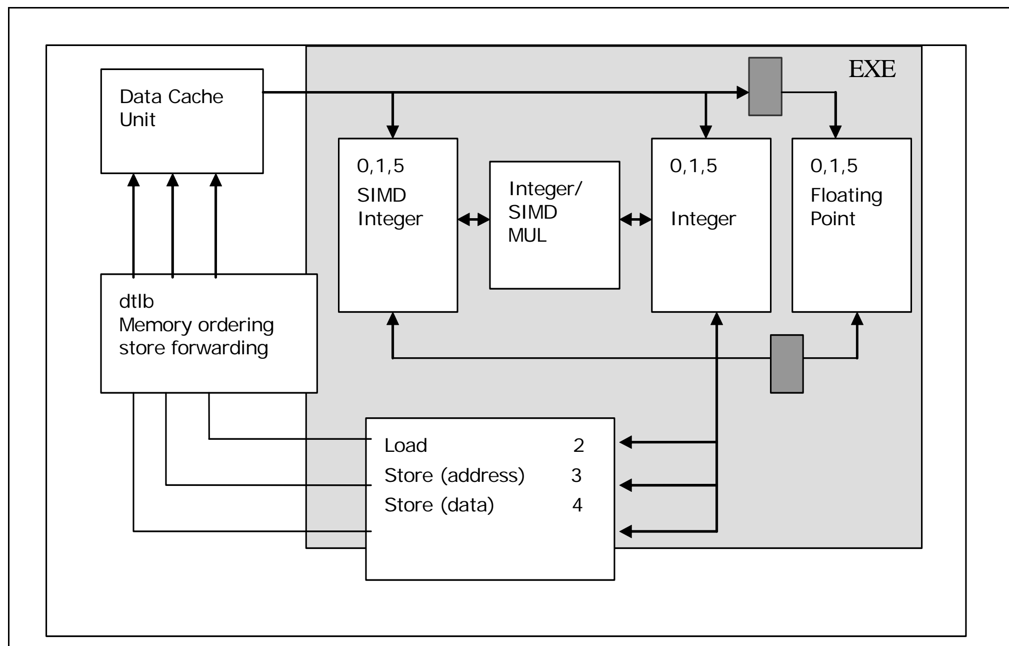


Figure 2-7. Execution Core of Intel Core Microarchitecture

Notice that the two dark squares inside the execution block (in grey color) and appear in the path connecting the integer and SIMD integer stacks to the floating-point stack. This delay shows up as an extra cycle called a bypass delay. Data from the L1 cache has one extra cycle of latency to the floating-point unit. The dark-colored squares in Figure 2-7 represent the extra cycle of latency.

2.4.4 Intel® Advanced Memory Access

The Intel Core microarchitecture contains an instruction cache and a first-level data cache in each core. The two cores share a 2 or 4-MByte L2 cache. All caches are writeback and non-inclusive. Each core contains:

- **L1 data cache, known as the data cache unit (DCU)** — The DCU can handle multiple outstanding cache misses and continue to service incoming stores and loads. It supports maintaining cache coherency. The DCU has the following specifications:
  - — 32-KBytes size.
  - — 8-way set associative.
  - — 64-bytes line size.
- **Data translation lookaside buffer (DTLB)** — The DTLB in Intel Core microarchitecture implements two levels of hierarchy. Each level of the DTLB have multiple entries and can support either 4-KByte pages or large pages. The entries of the inner level (DTLB0) is used for loads. The entries in the outer level (DTLB1) support store operations and loads that missed DTLB0. All entries are 4-way associative. Here is a list of entries in each DTLB:
  - — DTLB1 for large pages: 32 entries.
  - — DTLB1 for 4-KByte pages: 256 entries.
  - — DTLB0 for large pages: 16 entries.
  - — DTLB0 for 4-KByte pages: 16 entries.

An DTLB0 miss and DTLB1 hit causes a penalty of 2 cycles. Software only pays this penalty if the DTLB0 is used in some dispatch cases. The delays associated with a miss to the DTLB1 and PMH are largely non-blocking due to the design of Intel Smart Memory Access.

- **Page miss handler (PMH)**
- **A memory ordering buffer (MOB)** — Which:
  - Enables loads and stores to issue speculatively and out of order.
  - Ensures retired loads and stores have the correct data upon retirement.
  - Ensures loads and stores follow memory ordering rules of the Intel 64 and IA-32 architectures.

The memory cluster of the Intel Core microarchitecture uses the following to speed up memory operations:

- 128-bit load and store operations.
- Data prefetching to L1 caches.
- Data prefetch logic for prefetching to the L2 cache.
- Store forwarding.
- Memory disambiguation.
- 8 fill buffer entries.
- 20 store buffer entries.
- Out of order execution of memory operations.
- Pipelined read-for-ownership operation (RFO).

For information on optimizing software for the memory cluster, see Section 3.6, "Optimizing Memory Accesses."

2.4.4.1 Loads and Stores

The Intel Core microarchitecture can execute up to one 128-bit load and up to one 128-bit store per cycle, each to different memory locations. The microarchitecture enables execution of memory operations out of order with respect to other instructions and with respect to other memory operations.

Loads can:

- Issue before preceding stores when the load address and store address are known not to conflict.
- Be carried out speculatively, before preceding branches are resolved.
- Take cache misses out of order and in an overlapped manner.
- Issue before preceding stores, speculating that the store is not going to be to a conflicting address.

Loads cannot:

- Speculatively take any sort of fault or trap.
- Speculatively access the uncacheable memory type.

Faulting or uncacheable loads are detected and wait until retirement, when they update the programmer visible state. x87 and floating-point SIMD loads add 1 additional clock latency.

Stores to memory are executed in two phases:

- **Execution phase** — Prepares the store buffers with address and data for store forwarding. Consumes dispatch ports, which are ports 3 and 4.
- **Completion phase** — The store is retired to programmer-visible memory. It may compete for cache banks with executing loads. Store retirement is maintained as a background task by the memory order buffer, moving the data from the store buffers to the L1 cache.

2.4.4.2 Data Prefetch to L1 caches

Intel Core microarchitecture provides two hardware prefetchers to speed up data accessed by a program by prefetching to the L1 data cache:

- **Data cache unit (DCU) prefetcher** — This prefetcher, also known as the streaming prefetcher, is triggered by an ascending access to very recently loaded data. The processor assumes that this access is part of a streaming algorithm and automatically fetches the next line.
- **Instruction pointer (IP)- based strided prefetcher** — This prefetcher keeps track of individual load instructions. If a load instruction is detected to have a regular stride, then a prefetch is sent to the next address which is the sum of the current address and the stride. This prefetcher can prefetch forward or backward and can detect strides of up to half of a 4KB-page, or 2 KBytes.

Data prefetching works on loads only when the following conditions are met:

- Load is from writeback memory type.
- Prefetch request is within the page boundary of 4 Kbytes.
- No fence or lock is in progress in the pipeline.
- Not many other load misses are in progress.
- The bus is not very busy.
- There is not a continuous stream of stores.

DCU Prefetching has the following effects:

- Improves performance if data in large structures is arranged sequentially in the order used in the program.
- May cause slight performance degradation due to bandwidth issues if access patterns are sparse instead of local.
- On rare occasions, if the algorithm's working set is tuned to occupy most of the cache and unneeded prefetches evict lines required by the program, hardware prefetcher may cause severe performance degradation due to cache capacity of L1.

In contrast to hardware prefetchers relying on hardware to anticipate data traffic, software prefetch instructions relies on the programmer to anticipate cache miss traffic, software prefetch act as hints to bring a cache line of data into the desired levels of the cache hierarchy. The software-controlled prefetch is intended for prefetching data, but not for prefetching code.

2.4.4.3 Data Prefetch Logic

Data prefetch logic (DPL) prefetches data to the second-level (L2) cache based on past request patterns of the DCU from the L2. The DPL maintains two independent arrays to store addresses from the DCU: one for upstreams (12 entries) and one for down streams (4 entries). The DPL tracks accesses to one 4K byte page in each entry. If an accessed page is not in any of these arrays, then an array entry is allocated.

The DPL monitors DCU reads for incremental sequences of requests, known as streams. Once the DPL detects the second access of a stream, it prefetches the next cache line. For example, when the DCU requests the cache lines A and A+1, the DPL assumes the DCU will need cache line A+2 in the near future. If the DCU then reads A+2, the DPL prefetches cache line A+3. The DPL works similarly for "downward" loops.

The Intel Pentium M processor introduced DPL. The Intel Core microarchitecture added the following features to DPL:

- The DPL can detect more complicated streams, such as when the stream skips cache lines. DPL may issue 2 prefetch requests on every L2 lookup. The DPL in the Intel Core microarchitecture can run up to 8 lines ahead from the load request.
- DPL in the Intel Core microarchitecture adjusts dynamically to bus bandwidth and the number of requests. DPL prefetches far ahead if the bus is not busy, and less far ahead if the bus is busy.
- DPL adjusts to various applications and system configurations.

Entries for the two cores are handled separately.

2.4.4.4 Store Forwarding

If a load follows a store and reloads the data that the store writes to memory, the Intel Core microarchitecture can forward the data directly from the store to the load. This process, called store to load forwarding, saves cycles by enabling the load to obtain the data directly from the store operation instead of through memory.

The following rules must be met for store to load forwarding to occur:

- The store must be the last store to that address prior to the load.
- The store must be equal or greater in size than the size of data being loaded.
- The load cannot cross a cache line boundary.
- The load cannot cross an 8-Byte boundary. 16-Byte loads are an exception to this rule.
- The load must be aligned to the start of the store address, except for the following exceptions:
  - — An aligned 64-bit store may forward either of its 32-bit halves.
  - — An aligned 128-bit store may forward any of its 32-bit quarters.
  - — An aligned 128-bit store may forward either of its 64-bit halves.

Software can use the exceptions to the last rule to move complex structures without losing the ability to forward the subfields.

In Enhanced Intel Core microarchitecture, the alignment restrictions to permit store forwarding to proceed have been relaxed. Enhanced Intel Core microarchitecture permits store-forwarding to proceed in several situations that the succeeding load is not aligned to the preceding store. Figure 2-8 shows six situations (in gradient-filled background) of store-forwarding that are permitted in Enhanced Intel Core microarchitecture but not in Intel Core microarchitecture. The cases with backward slash background depicts store-forwarding that can proceed in both Intel Core microarchitecture and Enhanced Intel Core microarchitecture.

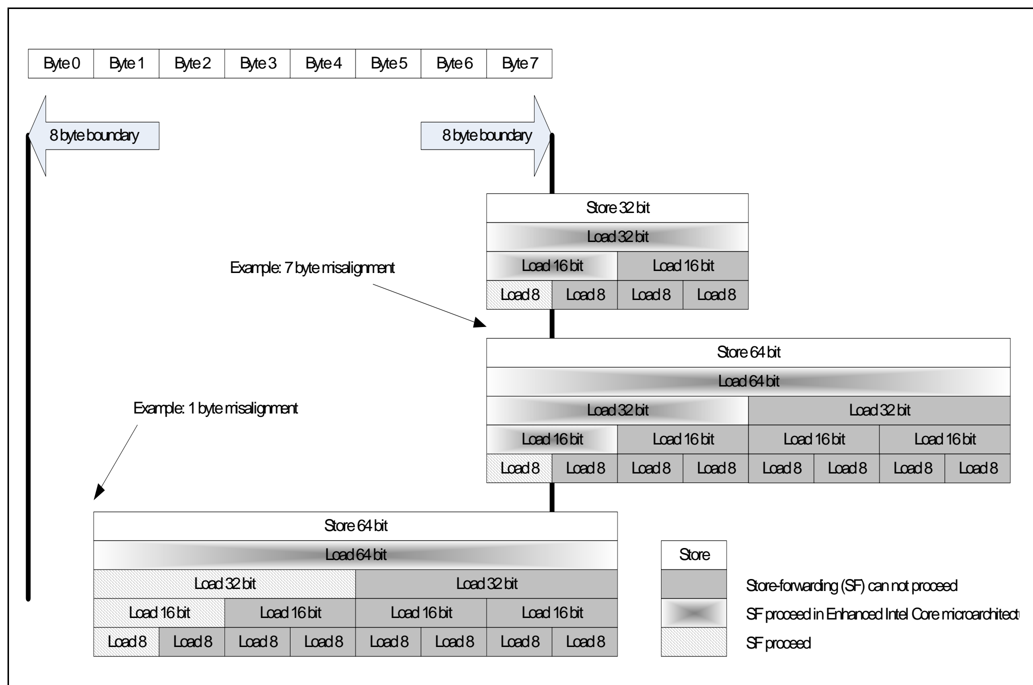


Figure 2-8. Store-Forwarding Enhancements in Enhanced Intel Core Microarchitecture

2.4.4.5 Memory Disambiguation

A load instruction micro-op may depend on a preceding store. Many microarchitectures block loads until all preceding store address are known.

The memory disambiguator predicts which loads will not depend on any previous stores. When the disambiguator predicts that a load does not have such a dependency, the load takes its data from the L1 data cache.

Eventually, the prediction is verified. If an actual conflict is detected, the load and all succeeding instructions are re-executed.

2.4.5 Intel® Advanced Smart Cache

The Intel Core microarchitecture optimized a number of features for two processor cores on a single die. The two cores share a second-level cache and a bus interface unit, collectively known as Intel Advanced Smart Cache. This section describes the components of Intel Advanced Smart Cache. Figure 2-9 illustrates the architecture of the Intel Advanced Smart Cache.

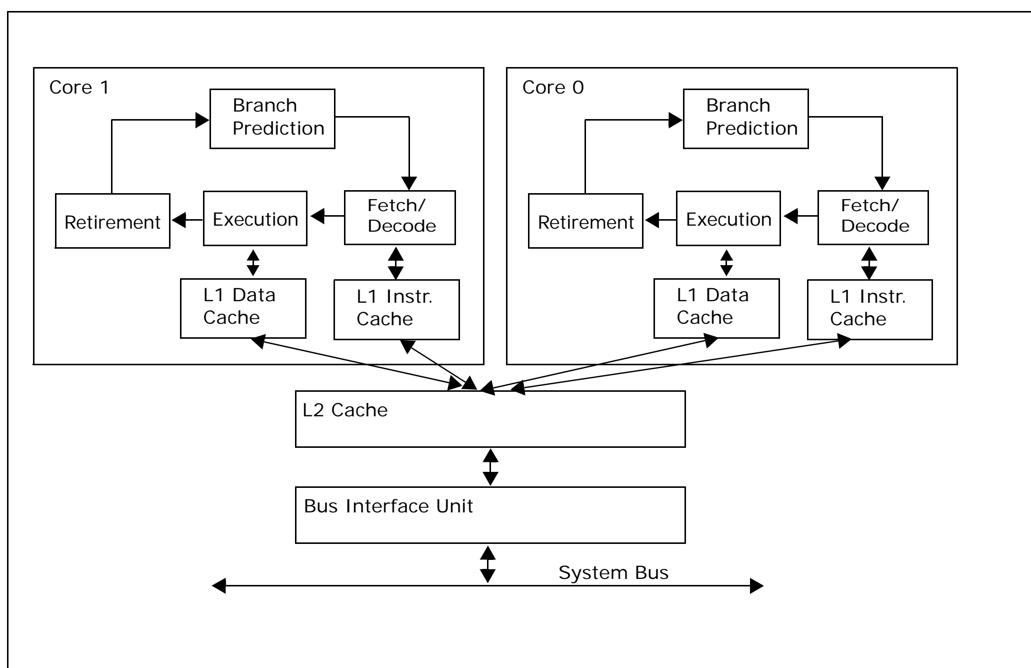


Figure 2-9. Intel Advanced Smart Cache Architecture

Table 2-27 details the parameters of caches in the Intel Core microarchitecture. For information on enumerating the cache hierarchy identification using the deterministic cache parameter leaf of CPUID instruction, see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Table 2-27. Cache Parameters of Processors based on Intel Core Microarchitecture

Level	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency (clocks)	Access Throughput (clocks)	Write Update Policy
First Level	32 KB	8	64	3	1	Writeback
Instruction	32 KB	8	N/A	N/A	N/A	N/A
Second Level (Shared L2)[1]	2, 4 MB	8 or 16	64	14[2]	2	Writeback
Second Level (Shared L2)[3]	3, 6MB	12 or 24	64	15[2]	2	Writeback
Third Level[4]	8, 12, 16 MB	16	64	~110	12	Writeback

NOTES:

1. Intel Core microarchitecture (CPUID signature DisplayFamily = 06H, DisplayModel = 0FH).
2. Software-visible latency will vary depending on access patterns and other factors.
3. Enhanced Intel Core microarchitecture (CPUID signature DisaplyFamily = 06H, DisplayModel = 17H or 1DH).
4. Enhanced Intel Core microarchitecture (CPUID signature DisaplyFamily = 06H, DisplayModel = 1DH).

2.4.5.1 Loads

When an instruction reads data from a memory location that has write-back (WB) type, the processor looks for the cache line that contains this data in the caches and memory in the following order:

1. DCU of the initiating core.
2. DCU of the other core and second-level cache.
3. System memory.

The cache line is taken from the DCU of the other core only if it is modified, ignoring the cache line availability or state in the L2 cache.

Table 2-28 shows the characteristics of fetching the first four bytes of different localities from the memory cluster. The latency column provides an estimate of access latency. However, the actual latency can vary depending on the load of cache, memory components, and their parameters.

Table 2-28. Characteristics of Load and Store Operations in Intel Core Microarchitecture

Data Locality	Load		Store	
	Latency	Throughput	Latency	Throughput
DCU	3	1	2	1
DCU of the other core in modified state	14 + 5.5 bus cycles	14 + 5.5 bus cycles	14 + 5.5 bus cycles	
2nd-level cache	14	3	14	3
Memory	14 + 5.5 bus cycles + memory	Depends on bus read protocol	14 + 5.5 bus cycles + memory	Depends on bus write protocol

Sometimes a modified cache line has to be evicted to make space for a new cache line. The modified cache line is evicted in parallel to bringing the new data and does not require additional latency. However, when data is written back to memory, the eviction uses cache bandwidth and possibly bus bandwidth as well. Therefore, when multiple cache misses require the eviction of modified lines within a short time, there is an overall degradation in cache response time.

2.4.5.2 Stores

When an instruction writes data to a memory location that has WB memory type, the processor first ensures that the line is in Exclusive or Modified state in its own DCU. The processor looks for the cache line in the following locations, in the specified order:

1. DCU of initiating core.
2. DCU of the other core and L2 cache.
3. System memory.

The cache line is taken from the DCU of the other core only if it is modified, ignoring the cache line availability or state in the L2 cache. After reading for ownership is completed, the data is written to the first-level data cache and the line is marked as modified.

Reading for ownership and storing the data happens after instruction retirement and follows the order of retirement. Therefore, the store latency does not effect the store instruction itself. However, several sequential stores may have cumulative latency that can affect performance. Table 2-28 presents store latencies depending on the initial cache line location.

2.5 INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Intel microarchitecture code name Nehalem provides the foundation for many innovative features of Intel Core i7 processors and Intel Xeon processor 3400, 5500, and 7500 series. It builds on the success of 45 nm enhanced Intel Core microarchitecture and provides the following feature enhancements:

- **Enhanced processor core**
  - Improved branch prediction and recovery from misprediction.
  - Enhanced loop streaming to improve front end performance and reduce power consumption.
  - Deeper buffering in out-of-order engine to extract parallelism.
  - Enhanced execution units to provide acceleration in CRC, string/text processing and data shuffling.
- **Hyper-Threading Technology**
  - Provides two hardware threads (logical processors) per core.
  - Takes advantage of 4-wide execution engine, large L3, and massive memory bandwidth.
- **Smart Memory Access**
  - Integrated memory controller provides low-latency access to system memory and scalable memory bandwidth.
  - New cache hierarchy organization with shared, inclusive L3 to reduce snoop traffic.
  - Two level TLBs and increased TLB size.
  - Fast unaligned memory access.
- **Dedicated Power management Innovations**
  - Integrated microcontroller with optimized embedded firmware to manage power consumption.
  - Embedded real-time sensors for temperature, current, and power.
  - Integrated power gate to turn off/on per-core power consumption.
  - Versatility to reduce power consumption of memory, link subsystems.

Intel microarchitecture code name Westmere is a 32 nm version of Intel microarchitecture code name Nehalem. All of the features of latter also apply to the former.

2.5.1 Microarchitecture Pipeline

Intel microarchitecture code name Nehalem continues the four-wide microarchitecture pipeline pioneered by the 65nm Intel Core microarchitecture. Figure 2-10 illustrates the basic components of the pipeline of Intel microarchitecture code name Nehalem as implemented in Intel Core i7 processor, only two of the four cores are sketched in the Figure 2-10 pipeline diagram.

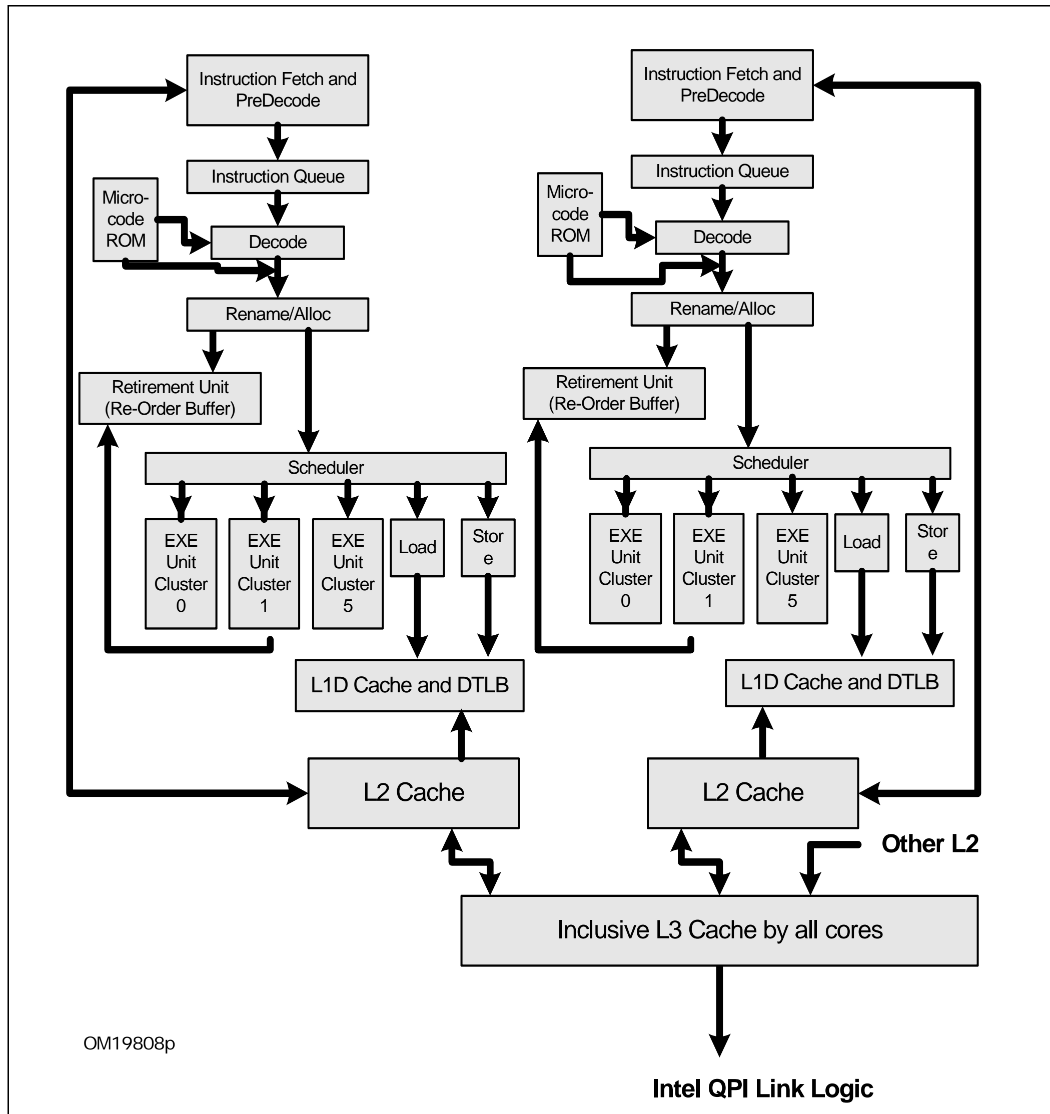


Figure 2-10. Intel Microarchitecture Code Name Nehalem Pipeline Functionality

The length of the pipeline in Intel microarchitecture code name Nehalem is two cycles longer than its predecessor in 45 nm Intel Core 2 processor family, as measured by branch misprediction delay. The front end can decode up to 4 instructions in one cycle and supports two hardware threads by decoding the instruction streams between two logical processors in alternate cycles. The front end includes enhancement in branch handling, loop detection, MSROM throughput, etc. These are discussed in subsequent sections.

The scheduler (or reservation station) can dispatch up to six micro-ops in one cycle through six issue ports (five issue ports are shown in Figure 2-10; store operation involves separate ports for store address and store data but is depicted as one in the diagram).

The out-of-order engine has many execution units that are arranged in three execution clusters shown in Figure 2-10. It can retire four micro-ops in one cycle, same as its predecessor.

2.5.2 Front End Overview

Figure 2-11 depicts the key components of the front end of the microarchitecture. The instruction fetch unit (IFU) can fetch up to 16 bytes of aligned instruction bytes each cycle from the instruction cache to the instruction length decoder (ILD). The instruction queue (IQ) buffers the ILD-processed instructions and can deliver up to four instructions in one cycle to the instruction decoder.

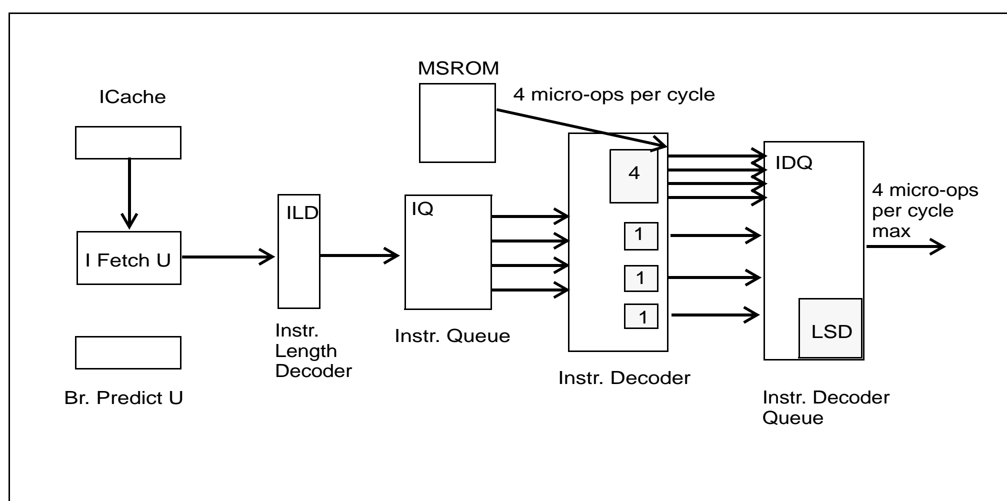


Figure 2-11. Front End of Intel Microarchitecture Code Name Nehalem

The instruction decoder has three decoder units that can decode one simple instruction per cycle per unit. The other decoder unit can decode one instruction every cycle, either simple instruction or complex instruction made up of several micro-ops. Instructions made up of more than four micro-ops are delivered from the MSROM. Up to four micro-ops can be delivered each cycle to the instruction decoder queue (IDQ).

The loop stream detector is located inside the IDQ to improve power consumption and front end efficiency for loops with a short sequence of instructions.

The instruction decoder supports micro-fusion to improve front end throughput, increase the effective size of queues in the scheduler and re-order buffer (ROB). The rules for micro-fusion are similar to those of Intel Core microarchitecture.

The instruction queue also supports macro-fusion to combine adjacent instructions into one micro-ops where possible. In previous generations of Intel Core microarchitecture, macro-fusion support for CMP/Jcc sequence is limited to the CF and ZF flag, and macrofusion is not supported in 64-bit mode.

In Intel microarchitecture code name Nehalem , macro-fusion is supported in 64-bit mode, and the following instruction sequences are supported:

- CMP or TEST can be fused when comparing (unchanged):

  REG-REG. For example: CMP EAX,ECX; JZ label
  REG-IMM. For example: CMP EAX,0x80; JZ label
  REG-MEM. For example: CMP EAX,[ECX]; JZ label
  MEM-REG. For example: CMP [EAX],ECX; JZ label

- TEST can fused with all conditional jumps (unchanged).
- CMP can be fused with the following conditional jumps. These conditional jumps check carry flag (CF) or zero flag (ZF). The list of macro-fusion-capable conditional jumps are (unchanged):

JA or JNBE
JAE or JNB or JNC
JE or JZ
JNA or JBE
JNAE or JC or JB
JNE or JNZ

- CMP can be fused with the following conditional jumps in Intel microarchitecture code name Nehalem, (this is an enhancement):

JL or JNGE
JGE or JNL
JLE or JNG
JG or JNLE

The hardware improves branch handling in several ways. Branch target buffer has increased to increase the accuracy of branch predictions. Renaming is supported with return stack buffer to reduce mispredictions of return instructions in the code. Furthermore, hardware enhancement improves the handling of branch misprediction by expediting resource reclamation so that the front end would not be waiting to decode instructions in an architected code path (the code path in which instructions will reach retirement) while resources were allocated to executing mispredicted code path. Instead, new micro-ops stream can start forward progress as soon as the front end decodes the instructions in the architected code path.

2.5.3 Execution Engine

The IDQ (Figure 2-11) delivers micro-op stream to the allocation/renaming stage (Figure 2-10) of the pipeline. The out-of-order engine supports up to 128 micro-ops in flight. Each micro-ops must be allocated with the following resources: an entry in the re-order buffer (ROB), an entry in the reservation station (RS), and a load/store buffer if a memory access is required.

The allocator also renames the register file entry of each micro-op in flight. The input data associated with a micro-op are generally either read from the ROB or from the retired register file.

The RS is expanded to 36 entry deep (compared to 32 entries in previous generation). It can dispatch up to six micro-ops in one cycle if the micro-ops are ready to execute. The RS dispatch a micro-op through an issue port to a specific execution cluster, each cluster may contain a collection of integer/FP/SIMD execution units.

The result from the execution unit executing a micro-op is written back to the register file, or forwarded through a bypass network to a micro-op in-flight that needs the result. Intel microarchitecture code name Nehalem can support write back throughput of one register file write per cycle per port. The bypass network consists of three domains of integer/FP/SIMD. Forwarding the result within the same bypass domain from a producer micro-op to a consumer micro is done efficiently in hardware without delay. Forwarding the result across different bypass domains may be subject to additional bypass delays. The bypass delays may be visible to software in addition to the latency and throughput characteristics of individual execution units. The bypass delays between a producer micro-op and a consumer micro-op across different bypass domains are shown in Table 2-29.

Table 2-29. Bypass Delay Between Producer and Consumer Micro-ops (cycles)

	FP	**Integer**	**SIMD**
FP	0	2	2
Integer	2	0	1
SIMD	2	1	0

2.5.3.1 Issue Ports and Execution Units

Table 2-30 summarizes the key characteristics of the issue ports and the execution unit latency/throughputs for common operations in the microarchitecture.

Table 2-30. Issue Ports of Intel Microarchitecture Code Name Nehalem

Port	Executable operations	Latency	Throughput	Domain	Comment
Port 0	Integer ALU Integer Shift	1 1	1 1	Integer	
Port 0	Integer SIMD ALU Integer SIMD Shuffle	1 1	1 1	SIMD	
Port 0	Single-precision (SP) FP MUL Double-precision FP MUL FP MUL (X87) FP/SIMD/SSE2 Move and Logic FP Shuffle DIV/SQRT	4 5 5 1 1	1 1 1 1 1	FP	
Port 1	Integer ALU Integer LEA Integer Mul	1 1 3	1 1 1	Integer	
Port 1	Integer SIMD MUL Integer SIMD Shift PSAD StringCompare	1 1 3	1 1 1	SIMD	
Port 1	FP ADD	3	1	FP	
Port 2	Integer loads	4	1	Integer	
Port 3	Store address	5	1	Integer	
Port 4	Store data			Integer	
Port 5	Integer ALU Integer Shift Jmp	1 1 1	1 1 1	Integer	
Port 5	Integer SIMD ALU Integer SIMD Shuffle	1 1	1 1	SIMD	
Port 5	FP/SIMD/SSE2 Move and Logic	1	1	FP	

2.5.4 Cache and Memory Subsystem

Intel microarchitecture code name Nehalem contains an instruction cache, a first-level data cache and a second-level unified cache in each core (see Figure 2-10). Each physical processor may contain several processor cores and a shared collection of sub-systems that are referred to as “uncore“. Specifically in Intel Core i7 processor, the uncore provides a unified third-level cache shared by all cores in the physical processor, Intel QuickPath Interconnect links and associated logic. The L1 and L2 caches are writeback and non-inclusive.

The shared L3 cache is writeback and inclusive, such that a cache line that exists in either L1 data cache, L1 instruction cache, unified L2 cache also exists in L3. The L3 is designed to use the inclusive nature to minimize snoop traffic between processor cores. Table 2-31 lists characteristics of the cache hierarchy. The latency of L3 access may vary as a function of the frequency ratio between the processor and the uncore sub-system.

Table 2-31. Cache Parameters of Intel Core i7 Processors

Level	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency (clocks)	Access Throughput (clocks)	Write Update Policy
First Level Data	32 KB	8	64	4	1	Writeback
Instruction	32 KB	4	N/A	N/A	N/A	N/A
Second Level	256KB	8	64	10[1]	Varies	Writeback
Third Level (Shared L3)[2]	8MB	16	64	35-40+[2]	Varies	Writeback

NOTES:

1. Software-visible latency will vary depending on access patterns and other factors.
2. Minimal L3 latency is 35 cycles if the frequency ratio between core and uncore is unity.

The Intel microarchitecture code name Nehalem implements two levels of translation lookaside buffer (TLB). The first level consists of separate TLBs for data and code. DTLB0 handles address translation for data accesses, it provides 64 entries to support 4KB pages and 32 entries for large pages. The ITLB provides 64 entries (per thread) for 4KB pages and 7 entries (per thread) for large pages.

The second level TLB (STLB) handles both code and data accesses for 4KB pages. It support 4KB page translation operation that missed DTLB0 or ITLB. All entries are 4-way associative. Here is a list of entries in each DTLB:

- STLB for 4-KByte pages: 512 entries (services both data and instruction look-ups).
- DTLB0 for large pages: 32 entries.
- DTLB0 for 4-KByte pages: 64 entries.

An DTLB0 miss and STLB hit causes a penalty of 7cycles. Software only pays this penalty if the DTLB0 is used in some dispatch cases. The delays associated with a miss to the STLB and PMH are largely non-blocking.

2.5.5 Load and Store Operation Enhancements

The memory cluster of Intel microarchitecture code name Nehalem provides the following enhancements to speed up memory operations:

- Peak issue rate of one 128-bit load and one 128-bit store operation per cycle.
- Deeper buffers for load and store operations: 48 load buffers, 32 store buffers and 10 fill buffers.
- Fast unaligned memory access and robust handling of memory alignment hazards.
- Improved store-forwarding for aligned and non-aligned scenarios.
- Store forwarding for most address alignments.

2.5.5.1 Efficient Handling of Alignment Hazards

The cache and memory subsystems handles a significant percentage of instructions in every workload. Different address alignment scenarios will produce varying performance impact for memory and cache operations. For example, 1-cycle throughput of L1 (see Table 2-32) generally applies to naturally-aligned loads from L1 cache. But using unaligned load instructions (e.g. MOVUPS, MOVUPD, MOVDQU, etc.) to

access data from L1 will experience varying amount of delays depending on specific microarchitectures and alignment scenarios.

Table 2-32. Performance Impact of Address Alignments of MOVDQU from L1

Throughput (cycle)	Intel Core i7 Processor	45 nm Intel Core Microarchitecture	65 nm Intel Core Microarchitecture
Alignment Scenario	**06_1AH**	**06_17H**	**06_0FH**
16B aligned	1	2	2
Not-16B aligned, not cache split	1	~2	~2
Split cache line boundary	~4.5	~20	~20

Table 2-32 lists approximate throughput of issuing MOVDQU instructions with different address alignment scenarios to load data from the L1 cache. If a 16-byte load spans across cache line boundary, previous microarchitecture generations will experience significant software-visible delays.

Intel microarchitecture code name Nehalem provides hardware enhancements to reduce the delays of handling different address alignment scenarios including cache line splits.

2.5.5.2 Store Forwarding Enhancement

When a load follows a store and reloads the data that the store writes to memory, the microarchitecture can forward the data directly from the store to the load in many cases. This situation, called store to load forwarding, saves several cycles by enabling the load to obtain the data directly from the store operation instead of through the memory system.

Several general rules must be met for store to load forwarding to proceed without delay:

- The store must be the last store to that address prior to the load.
- The store must be equal or greater in size than the size of data being loaded.
- The load data must be completely contained in the preceding store.

Specific address alignment and data sizes between the store and load operations will determine whether a store-forward situation may proceed with data forwarding or experience a delay via the cache/memory sub-system. The 45 nm Enhanced Intel Core microarchitecture offers more flexible address alignment and data sizes requirement than previous microarchitectures. Intel microarchitecture code name Nehalem offers additional enhancement with allowing more situations to forward data expeditiously.

The store-forwarding situations for with respect to store operations of 16 bytes are illustrated in Figure 2-12.

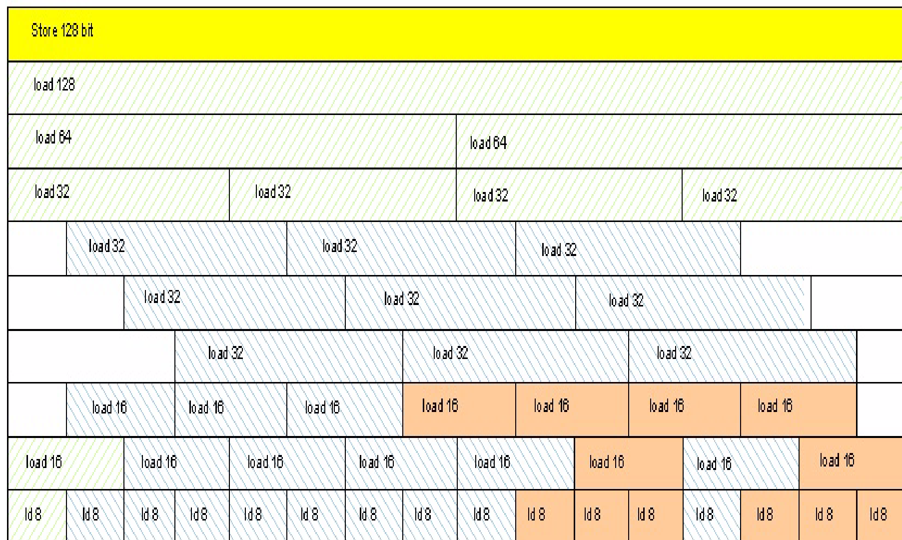


Figure 2-12. Store-Forwarding Scenarios of 16-Byte Store Operations

Intel microarchitecture code name Nehalem allows store-to-load forwarding to proceed regardless of store address alignment (The white space in the diagram does not correspond to an applicable store-to-load scenario). Figure 2-13 illustrates situations for store operation of 8 bytes or less.

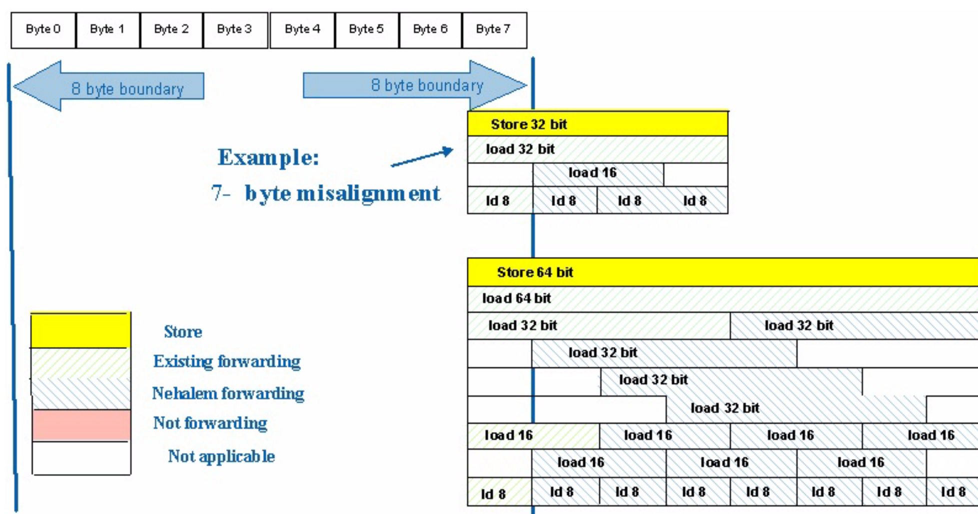


Figure 2-13. Store-Forwarding Enhancement in Intel Microarchitecture Code Name Nehalem

2.5.6 REP String Enhancement

REP prefix in conjunction with MOVS/STOS instruction and a count value in ECX are frequently used to implement library functions such as memcpy()/memset(). These are referred to as "REP string" instructions. Each iteration of these instruction can copy/write constant a value in byte/word/dword/qword granularity The performance characteristics of using REP string can be attributed to two components: startup overhead and data transfer throughput.

The two components of performance characteristics of REP String varies further depending on granularity, alignment, and/or count values. Generally, MOVSB is used to handle very small chunks of data. Therefore, processor implementation of REP MOVSB is optimized to handle ECX < 4. Using REP MOVSB with ECX > 3 will achieve low data throughput due to not only byte-granular data transfer but also additional startup overhead. The latency for MOVSB, is 9 cycles if ECX < 4; otherwise REP MOVSB with ECX >9 have a 50-cycle startup cost.

For REP string of larger granularity data transfer, as ECX value increases, the startup overhead of REP String exhibit step-wise increase:

- Short string (ECX <= 12): the latency of REP MOVSW/MOVSD/MOVSQ is about 20 cycles.
- Fast string (ECX >= 76: excluding REP MOVSB): the processor implementation provides hardware optimization by moving as many pieces of data in 16 bytes as possible. The latency of REP string latency will vary if one of the 16-byte data transfer spans across cache line boundary:
  — Split-free: the latency consists of a startup cost of about 40 cycles and each 64 bytes of data adds 4 cycles.
  — Cache splits: the latency consists of a startup cost of about 35 cycles and each 64 bytes of data adds 6cycles.
- Intermediate string lengths: the latency of REP MOVSW/MOVSD/MOVSQ has a startup cost of about 15 cycles plus one cycle for each iteration of the data movement in word/dword/qword.

Intel microarchitecture code name Nehalem improves the performance of REP strings significantly over previous microarchitectures in several ways:

- Startup overhead have been reduced in most cases relative to previous microarchitecture.
- Data transfer throughput are improved over previous generation.

- In order for REP string to operate in “fast string” mode, previous microarchitectures requires address alignment. In Intel microarchitecture code name Nehalem, REP string can operate in “fast string” mode even if address is not aligned to 16 bytes.

2.5.7 Enhancements for System Software

In addition to microarchitectural enhancements that can benefit both application-level and system-level software, Intel microarchitecture code name Nehalem enhances several operations that primarily benefit system software.

Lock primitives: Synchronization primitives using the Lock prefix (e.g. XCHG, CMPXCHG8B) executes with significantly reduced latency than previous microarchitectures.

VMM overhead improvements: VMX transitions between a Virtual Machine (VM) and its supervisor (the VMM) can take thousands of cycle each time on previous microarchitectures. The latency of VMX transitions has been reduced in processors based on Intel microarchitecture code name Nehalem.

2.5.8 Efficiency Enhancements for Power Consumption

Intel microarchitecture code name Nehalem is not only designed for high performance and power-efficient performance under wide range of loading situations, it also features enhancement for low power consumption while the system idles. Intel microarchitecture code name Nehalem supports processor-specific C6 states, which have the lowest leakage power consumption that OS can manage through ACPI and OS power management mechanisms.

2.5.9 Hyper-Threading Technology Support in Intel® Microarchitecture Code Name Nehalem

Intel microarchitecture code name Nehalem supports Hyper-Threading Technology (HT). Its implementation of HT provides two logical processors sharing most execution/cache resources in each core. The HT implementation in Intel microarchitecture code name Nehalem differs from previous generations of HT implementations using Intel NetBurst microarchitecture in several areas:

- Intel microarchitecture code name Nehalem provides four-wide execution engine, more functional execution units coupled to three issue ports capable of issuing computational operations.
- Intel microarchitecture code name Nehalem supports integrated memory controller that can provide peak memory bandwidth of up to 25.6 GB/sec in Intel Core i7 processor.
- Deeper buffering and enhanced resource sharing/partition policies:
  — Replicated resource for HT operation: register state, renamed return stack buffer, large-page ITLB.
  — Partitioned resources for HT operation: load buffers, store buffers, re-order buffers, small-page ITLB are statically allocated between two logical processors.
  — Competitively-shared resource during HT operation: the reservation station, cache hierarchy, fill buffers, both DTLB0 and STLB.
  — Alternating during HT operation: front end operation generally alternates between two logical processors to ensure fairness.
  — HT unaware resources: execution units.

2.6 INTEL® HYPER-THREADING TECHNOLOGY

Intel® Hyper-Threading Technology (HT Technology) is supported by specific members of the Intel Pentium 4 and Xeon processor families. The technology enables software to take advantage of task-level, or thread-level parallelism by providing multiple logical processors within a physical processor package.

In its first implementation in Intel Xeon processor, Hyper-Threading Technology makes a single physical processor appear as two logical processors.

The two logical processors each have a complete set of architectural registers while sharing one single physical processor's resources. By maintaining the architecture state of two processors, an HT Technology capable processor looks like two processors to software, including operating system and application code.

By sharing resources needed for peak demands between two logical processors, HT Technology is well suited for multiprocessor systems to provide an additional performance boost in throughput when compared to traditional MP systems.

Figure 2-14 shows a typical bus-based symmetric multiprocessor (SMP) based on processors supporting HT Technology. Each logical processor can execute a software thread, allowing a maximum of two software threads to execute simultaneously on one physical processor. The two software threads execute simultaneously, meaning that in the same clock cycle an “add” operation from logical processor 0 and another “add” operation and load from logical processor 1 can be executed simultaneously by the execution engine.

In the first implementation of HT Technology, the physical execution resources are shared and the architecture state is duplicated for each logical processor. This minimizes the die area cost of implementing HT Technology while still achieving performance gains for multithreaded applications or multitasking workloads.

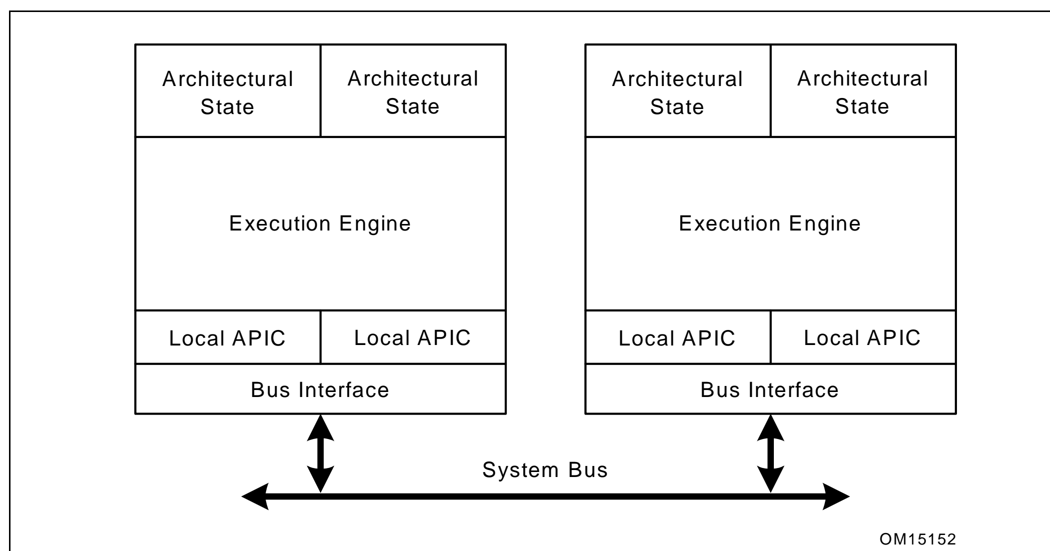


Figure 2-14. Hyper-Threading Technology on an SMP

The performance potential due to HT Technology is due to:

- The fact that operating systems and user programs can schedule processes or threads to execute simultaneously on the logical processors in each physical processor.
- The ability to use on-chip execution resources at a higher level than when only a single thread is consuming the execution resources; higher level of resource utilization can lead to higher system throughput.

2.6.1 Processor Resources and HT Technology

The majority of microarchitecture resources in a physical processor are shared between the logical processors. Only a few small data structures were replicated for each logical processor. This section describes how resources are shared, partitioned or replicated.

2.6.1.1 Replicated Resources

The architectural state is replicated for each logical processor. The architecture state consists of registers that are used by the operating system and application code to control program behavior and store data for computations. This state includes the eight general-purpose registers, the control registers, machine state registers, debug registers, and others. There are a few exceptions, most notably the memory type range registers (MTRRs) and the performance monitoring resources. For a complete list of the architecture state and exceptions, see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B & 3C*.

Other resources such as instruction pointers and register renaming tables were replicated to simultaneously track execution and state changes of the two logical processors. The return stack predictor is replicated to improve branch prediction of return instructions.

In addition, a few buffers (for example, the 2-entry instruction streaming buffers) were replicated to reduce complexity.

2.6.1.2 Partitioned Resources

Several buffers are shared by limiting the use of each logical processor to half the entries. These are referred to as partitioned resources. Reasons for this partitioning include:

- Operational fairness.
- Permitting the ability to allow operations from one logical processor to bypass operations of the other logical processor that may have stalled.

For example: a cache miss, a branch misprediction, or instruction dependencies may prevent a logical processor from making forward progress for some number of cycles. The partitioning prevents the stalled logical processor from blocking forward progress.

In general, the buffers for staging instructions between major pipe stages are partitioned. These buffers include µop queues after the execution trace cache, the queues after the register rename stage, the reorder buffer which stages instructions for retirement, and the load and store buffers.

In the case of load and store buffers, partitioning also provided an easier implementation to maintain memory ordering for each logical processor and detect memory ordering violations.

2.6.1.3 Shared Resources

Most resources in a physical processor are fully shared to improve the dynamic utilization of the resource, including caches and all the execution units. Some shared resources which are linearly addressed, like the DTLB, include a logical processor ID bit to distinguish whether the entry belongs to one logical processor or the other.

The first level cache can operate in two modes depending on a context-ID bit:

- Shared mode: The L1 data cache is fully shared by two logical processors.
- Adaptive mode: In adaptive mode, memory accesses using the page directory is mapped identically across logical processors sharing the L1 data cache.

The other resources are fully shared.

2.6.2 Microarchitecture Pipeline and HT Technology

This section describes the HT Technology microarchitecture and how instructions from the two logical processors are handled between the front end and the back end of the pipeline.

Although instructions originating from two programs or two threads execute simultaneously and not necessarily in program order in the execution core and memory hierarchy, the front end and back end contain several selection points to select between instructions from the two logical processors. All selection points alternate between the two logical processors unless one logical processor cannot make use of a pipeline stage. In this case, the other logical processor has full use of every cycle of the pipeline stage.

Reasons why a logical processor may not use a pipeline stage include cache misses, branch mispredictions, and instruction dependencies.

2.6.3 Front End Pipeline

The execution trace cache is shared between two logical processors. Execution trace cache access is arbitrated by the two logical processors every clock. If a cache line is fetched for one logical processor in one clock cycle, the next clock cycle a line would be fetched for the other logical processor provided that both logical processors are requesting access to the trace cache.

If one logical processor is stalled or is unable to use the execution trace cache, the other logical processor can use the full bandwidth of the trace cache until the initial logical processor's instruction fetches return from the L2 cache.

After fetching the instructions and building traces of µops, the µops are placed in a queue. This queue decouples the execution trace cache from the register rename pipeline stage. As described earlier, if both logical processors are active, the queue is partitioned so that both logical processors can make independent forward progress.

2.6.4 Execution Core

The core can dispatch up to six µops per cycle, provided the µops are ready to execute. Once the µops are placed in the queues waiting for execution, there is no distinction between instructions from the two logical processors. The execution core and memory hierarchy is also oblivious to which instructions belong to which logical processor.

After execution, instructions are placed in the re-order buffer. The re-order buffer decouples the execution stage from the retirement stage. The re-order buffer is partitioned such that each uses half the entries.

2.6.5 Retirement

The retirement logic tracks when instructions from the two logical processors are ready to be retired. It retires the instruction in program order for each logical processor by alternating between the two logical processors. If one logical processor is not ready to retire any instructions, then all retirement bandwidth is dedicated to the other logical processor.

Once stores have retired, the processor needs to write the store data into the level-one data cache. Selection logic alternates between the two logical processors to commit store data to the cache.

2.7 INTEL® 64 ARCHITECTURE

Intel 64 architecture supports almost all features in the IA-32 Intel architecture and extends support to run 64-bit OS and 64-bit applications in 64-bit linear address space. Intel 64 architecture provides a new operating mode, referred to as IA-32e mode, and increases the linear address space for software to 64 bits and supports physical address space up to 40 bits.

IA-32e mode consists of two sub-modes: (1) compatibility mode enables a 64-bit operating system to run most legacy 32-bit software unmodified, (2) 64-bit mode enables a 64-bit operating system to run applications written to access 64-bit linear address space.

In the 64-bit mode of Intel 64 architecture, software may access:

- 64-bit flat linear addressing.
- 8 additional general-purpose registers (GPRs).
- 8 additional registers (XMM) for streaming SIMD extensions (SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AESNI, PCLMULQDQ).

— Sixteen 256-bit YMM registers (whose lower 128 bits are overlaid to the respective XMM registers) if AVX, F16C, AVX2 or FMA are supported.

- 64-bit-wide GPRs and instruction pointers.
- Uniform byte-register addressing.
- Fast interrupt-prioritization mechanism.
- A new instruction-pointer relative-addressing mode.

2.8 SIMD TECHNOLOGY

SIMD computations (see Figure 2-15) were introduced to the architecture with MMX technology. MMX technology allows SIMD computations to be performed on packed byte, word, and doubleword integers. The integers are contained in a set of eight 64-bit registers called MMX registers (see Figure 2-16).

The Pentium III processor extended the SIMD computation model with the introduction of the Streaming SIMD Extensions (SSE). SSE allows SIMD computations to be performed on operands that contain four packed single-precision floating-point data elements. The operands can be in memory or in a set of eight 128-bit XMM registers (see Figure 2-16). SSE also extended SIMD computational capability by adding additional 64-bit MMX instructions.

Figure 2-15 shows a typical SIMD computation. Two sets of four packed data elements (X1, X2, X3, and X4, and Y1, Y2, Y3, and Y4) are operated on in parallel, with the same operation being performed on each corresponding pair of data elements (X1 and Y1, X2 and Y2, X3 and Y3, and X4 and Y4). The results of the four parallel computations are sorted as a set of four packed data elements.

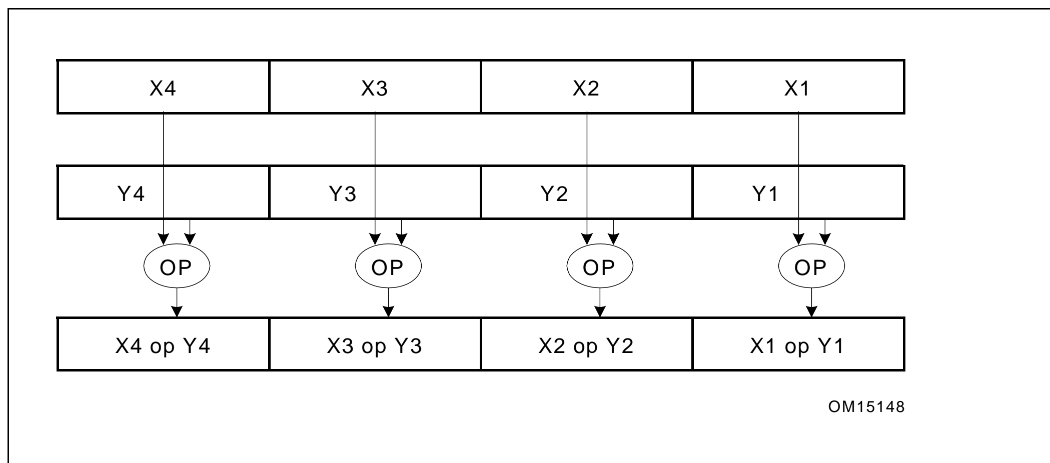


Figure 2-15. Typical SIMD Operations

The Pentium 4 processor further extended the SIMD computation model with the introduction of Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), and Intel Xeon processor 5100 series introduced Supplemental Streaming SIMD Extensions 3 (SSSE3).

SSE2 works with operands in either memory or in the XMM registers. The technology extends SIMD computations to process packed double-precision floating-point data elements and 128-bit packed integers. There are 144 instructions in SSE2 that operate on two packed double-precision floating-point data elements or on 16 packed byte, 8 packed word, 4 doubleword, and 2 quadword integers.

SSE3 enhances x87, SSE and SSE2 by providing 13 instructions that can accelerate application performance in specific areas. These include video processing, complex arithmetics, and thread synchronization. SSE3 complements SSE and SSE2 with instructions that process SIMD data asymmetrically, facilitate horizontal computation, and help avoid loading cache line splits. See Figure 2-16.

SSSE3 provides additional enhancement for SIMD computation with 32 instructions on digital video and signal processing.

SSE4.1, SSE4.2 and AESNI are additional SIMD extensions that provide acceleration for applications in media processing, text/lexical processing, and block encryption/decryption.

The SIMD extensions operates the same way in Intel 64 architecture as in IA-32 architecture, with the following enhancements:

- 128-bit SIMD instructions referencing XMM register can access 16 XMM registers in 64-bit mode.
- Instructions that reference 32-bit general purpose registers can access 16 general purpose registers in 64-bit mode.

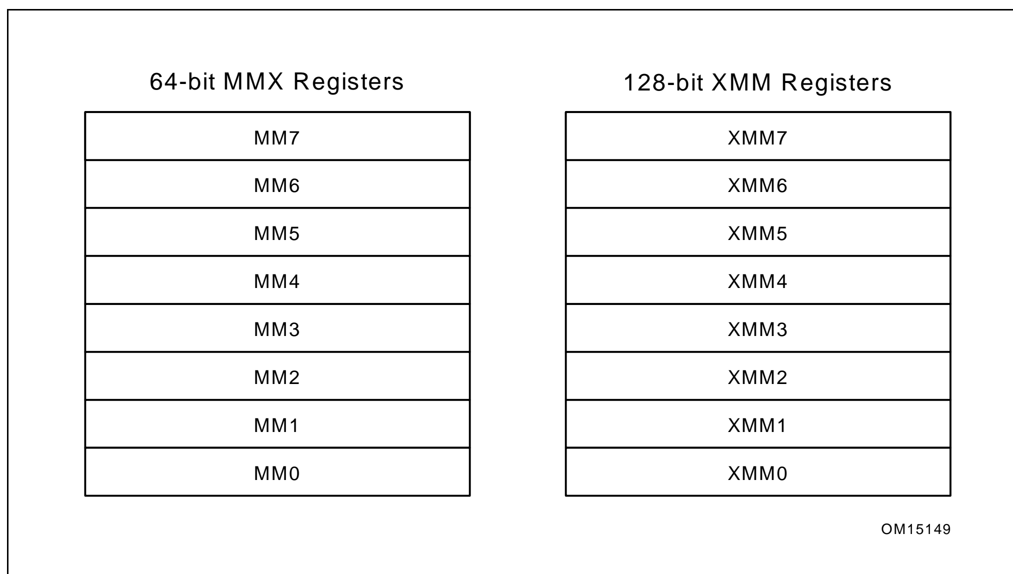


Figure 2-16. SIMD Instruction Register Usage

SIMD improves the performance of 3D graphics, speech recognition, image processing, scientific applications and applications that have the following characteristics:

- Inherently parallel.
- Recurring memory access patterns.
- Localized recurring operations performed on the data.
- Data-independent control flow.

2.9 SUMMARY OF SIMD TECHNOLOGIES AND APPLICATION LEVEL EXTENSIONS

SIMD floating-point instructions fully support the IEEE Standard 754 for Binary Floating-Point Arithmetic. They are accessible from all IA-32 execution modes: protected mode, real address mode, and Virtual 8086 mode.

SSE, SSE2, and MMX technologies are architectural extensions. Existing software will continue to run correctly, without modification on Intel microprocessors that incorporate these technologies. Existing software will also run correctly in the presence of applications that incorporate SIMD technologies.

SSE and SSE2 instructions also introduced cacheability and memory ordering instructions that can improve cache usage and application performance.

For more on SSE, SSE2, SSE3 and MMX technologies, see the following chapters in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*:

- Chapter 9, "Programming with Intel® MMX™ Technology".
- Chapter 10, "Programming with Streaming SIMD Extensions (SSE)".
- Chapter 11, "Programming with Streaming SIMD Extensions 2 (SSE2)".
- Chapter 12, "Programming with SSE3, SSSE3 and SSE4".
- Chapter 14, "Programming with AVX, FMA and AVX2".
- Chapter 15, "Programming with Intel® Transactional Synchronization Extensions".

2.9.1 MMX™ Technology

MMX Technology introduced:

- 64-bit MMX registers.
- Support for SIMD operations on packed byte, word, and doubleword integers.

MMX instructions are useful for multimedia and communications software.

2.9.2 Streaming SIMD Extensions

Streaming SIMD extensions introduced:

- 128-bit XMM registers.
- 128-bit data type with four packed single-precision floating-point operands.
- Data prefetch instructions.
- Non-temporal store instructions and other cacheability and memory ordering instructions.
- Extra 64-bit SIMD integer support.

SSE instructions are useful for 3D geometry, 3D rendering, speech recognition, and video encoding and decoding.

2.9.3 Streaming SIMD Extensions 2

Streaming SIMD extensions 2 add the following:

- 128-bit data type with two packed double-precision floating-point operands.
- 128-bit data types for SIMD integer operation on 16-byte, 8-word, 4-doubleword, or 2-quadword integers.
- Support for SIMD arithmetic on 64-bit integer operands.
- Instructions for converting between new and existing data types.
- Extended support for data shuffling.
- Extended support for cacheability and memory ordering operations.

SSE2 instructions are useful for 3D graphics, video decoding/encoding, and encryption.

2.9.4 Streaming SIMD Extensions 3

Streaming SIMD extensions 3 add the following:

- SIMD floating-point instructions for asymmetric and horizontal computation.
- A special-purpose 128-bit load instruction to avoid cache line splits.
- An x87 FPU instruction to convert to integer independent of the floating-point control word (FCW).

- Instructions to support thread synchronization.

SSE3 instructions are useful for scientific, video and multi-threaded applications.

2.9.5 Supplemental Streaming SIMD Extensions 3

The Supplemental Streaming SIMD Extensions 3 introduces 32 new instructions to accelerate eight types of computations on packed integers. These include:

- 12 instructions that perform horizontal addition or subtraction operations.
- 6 instructions that evaluate the absolute values.
- 2 instructions that perform multiply and add operations and speed up the evaluation of dot products.
- 2 instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- 2 instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.
- 6 instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- 2 instructions that align data from the composite of two operands.

2.9.6 SSE4.1

SSE4.1 introduces 47 new instructions to accelerate video, imaging and 3D applications. SSE4.1 also improves compiler vectorization and significantly increase support for packed dword computation. These include:

- Two instructions perform packed dword multiplies.
- Two instructions perform floating-point dot products with input/output selects.
- One instruction provides a streaming hint for WC loads.
- Six instructions simplify packed blending.
- Eight instructions expand support for packed integer MIN/MAX.
- Four instructions support floating-point round with selectable rounding mode and precision exception override.
- Seven instructions improve data insertion and extractions from XMM registers
- Twelve instructions improve packed integer format conversions (sign and zero extensions).
- One instruction improves SAD (sum absolute difference) generation for small block sizes.
- One instruction aids horizontal searching operations of word integers.
- One instruction improves masked comparisons.
- One instruction adds qword packed equality comparisons.
- One instruction adds dword packing with unsigned saturation.

2.9.7 SSE4.2

SSE4.2 introduces 7 new instructions. These include:

- A 128-bit SIMD integer instruction for comparing 64-bit integer data elements.
- Four string/text processing instructions providing a rich set of primitives, these primitives can accelerate:
  - Basic and advanced string library functions from strlen, strcmp, to strcspn.
  - Delimiter processing, token extraction for lexing of text streams.

— Parser, schema validation including XML processing.

- A general-purpose instruction for accelerating cyclic redundancy checksum signature calculations.
- A general-purpose instruction for calculating bit count population of integer numbers.

2.9.8 AESNI and PCLMULQDQ

AESNI introduces 7 new instructions, six of them are primitives for accelerating algorithms based on AES encryption/decryption standard, referred to as AESNI.

The PCLMULQDQ instruction accelerates general-purpose block encryption, which can perform carry-less multiplication for two binary numbers up to 64-bit wide.

Typically, algorithm based on AES standard involve transformation of block data over multiple iterations via several primitives. The AES standard supports cipher key of sizes 128, 192, and 256 bits. The respective cipher key sizes correspond to 10, 12, and 14 rounds of iteration.

AES encryption involves processing 128-bit input data (plaintext) through a finite number of iterative operation, referred to as "AES round", into a 128-bit encrypted block (ciphertext). Decryption follows the reverse direction of iterative operation using the "equivalent inverse cipher" instead of the "inverse cipher".

The cryptographic processing at each round involves two input data, one is the "state", the other is the "round key". Each round uses a different "round key". The round keys are derived from the cipher key using a "key schedule" algorithm. The "key schedule" algorithm is independent of the data processing of encryption/decryption, and can be carried out independently from the encryption/decryption phase.

The AES extensions provide two primitives to accelerate AES rounds on encryption, two primitives for AES rounds on decryption using the equivalent inverse cipher, and two instructions to support the AES key expansion procedure.

2.9.9 Intel® Advanced Vector Extensions

Intel® Advanced Vector Extensions offers comprehensive architectural enhancements over previous generations of Streaming SIMD Extensions. Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors and SIMD register set.
- 256-bit floating-point instruction set enhancement with up to 2X performance gain relative to 128-bit Streaming SIMD extensions.
- Instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.
- Support flexible deployment of 256-bit AVX code, 128-bit AVX code, legacy 128-bit code and scalar code.

Intel AVX instruction set and 256-bit register state management detail are described in IA-32 Intel® Architecture Software Developer's Manual, Volumes 2A, 2B and 3A. Optimization techniques for Intel AVX is discussed in Chapter 11, "Optimization for Intel AVX, FMA, and AVX2".

2.9.10 Half-Precision Floating-Point Conversion (F16C)

VCVTPH2PS and VCVTPS2PH are two instructions supporting half-precision floating-point data type conversion to and from single-precision floating-point data types. These two instruction extends on the same programming model as Intel AVX.

2.9.11 RDRAND

The RDRAND instruction retrieves a random number supplied by a cryptographically secure, deterministic random bit generator (DBRG). The DBRG is designed to meet NIST SP 800-90A standard.

2.9.12 Fused-Multiply-ADD (FMA) Extensions

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract operations. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

2.9.13 Intel AVX2

Intel AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. AVX2 instructions follow the same programming model as AVX instructions.

In addition, AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

2.9.14 General-Purpose Bit-Processing Instructions

The fourth generation Intel Core processor family introduces a collection of bit processing instructions that operate on the general purpose registers. The majority of these instructions uses the VEX-prefix encoding scheme to provide non-destructive source operand syntax.

There instructions are enumerated by three separate feature flags reported by CPUID. For details, see Section 5.1 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* and CHAPTER 3, 4 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B & 2C*.

2.9.15 Intel® Transactional Synchronization Extensions

The fourth generation Intel Core processor family introduces Intel® Transactional Synchronization Extensions (Intel TSX), which aim to improve the performance of lock-protected critical sections of multi-threaded applications while maintaining the lock-based programming model.

For background and details, see Chapter 15, "Programming with Intel® Transactional Synchronization Extensions" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Software tuning recommendations for using Intel TSX on lock-protected critical sections of multithreaded applications are described in Chapter 12, "Intel® TSX Recommendations".

2.9.16 RDSEED

The Intel Core M processor family introduces the RDSEED, ADCX and ADOX instructions.

The RDSEED instruction retrieves a random number supplied by a cryptographically secure, enhanced deterministic random bit generator Enhanced NRBG). The NRBG is designed to meet the NIST SP 800-90B and NIST SP 800-90C standards.

2.9.17 ADCX and ADOX Instructions

The ADCX and ADOX instructions, in conjunction with MULX instruction, enable software to speed up calculations that require large integer numerics. Details can be found at https://www-

ssl.intel.com/content/www/us/en/intelligent-systems/intel-technology/ia-large-integer-arithmetic-paper.html? and http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/large-integer-squaring-ia-paper.html.

CHAPTER 3
GENERAL OPTIMIZATION GUIDELINES

This chapter discusses general optimization techniques that can improve the performance of applications running on processors based on Intel microarchitecture code name Haswell, Ivy Bridge, Sandy Bridge, Westmere, Nehalem, Enhanced Intel Core microarchitecture and Intel Core microarchitectures. These techniques take advantage of microarchitectural described in Chapter 2, "Intel® 64 and IA-32 Processor Architectures." Optimization guidelines focusing on Intel multi-core processors, Hyper-Threading Technology and 64-bit mode applications are discussed in Chapter 8, "Multicore and Hyper-Threading Technology," and Chapter 9, "64-bit Mode Coding Guidelines."

Practices that optimize performance focus on three areas:

- Tools and techniques for code generation.
- Analysis of the performance characteristics of the workload and its interaction with microarchitectural sub-systems.
- Tuning code to the target microarchitecture (or families of microarchitecture) to improve performance.

Some hints on using tools are summarized first to simplify the first two tasks. the rest of the chapter will focus on recommendations of code generation or code tuning to the target microarchitectures.

This chapter explains optimization techniques for the Intel C++ Compiler, the Intel Fortran Compiler, and other compilers.

3.1 PERFORMANCE TOOLS

Intel offers several tools to help optimize application performance, including compilers, performance analyzer and multithreading tools.

3.1.1 Intel® C++ and Fortran Compilers

Intel compilers support multiple operating systems (Windows*, Linux*, Mac OS* and embedded). The Intel compilers optimize performance and give application developers access to advanced features:

- Flexibility to target 32-bit or 64-bit Intel processors for optimization
- Compatibility with many integrated development environments or third-party compilers.
- Automatic optimization features to take advantage of the target processor's architecture.
- Automatic compiler optimization reduces the need to write different code for different processors.
- Common compiler features that are supported across Windows, Linux and Mac OS include:
  - — General optimization settings.
  - — Cache-management features.
  - — Interprocedural optimization (IPO) methods.
  - — Profile-guided optimization (PGO) methods.
  - — Multithreading support.
  - — Floating-point arithmetic precision and consistency support.
  - — Compiler optimization and vectorization reports.

3.1.2 General Compiler Recommendations

Generally speaking, a compiler that has been tuned for the target microarchitecture can be expected to match or outperform hand-coding. However, if performance problems are noted with the compiled code, some compilers (like Intel C++ and Fortran Compilers) allow the coder to insert intrinsics or inline assembly in order to exert control over what code is generated. If inline assembly is used, the user must verify that the code generated is of good quality and yields good performance.

Default compiler switches are targeted for common cases. An optimization may be made to the compiler default if it is beneficial for most programs. If the root cause of a performance problem is a poor choice on the part of the compiler, using different switches or compiling the targeted module with a different compiler may be the solution.

3.1.3 VTune™ Performance Analyzer

VTune uses performance monitoring hardware to collect statistics and coding information of your application and its interaction with the microarchitecture. This allows software engineers to measure performance characteristics of the workload for a given microarchitecture. VTune supports all current and past Intel processor families.

The VTune Performance Analyzer provides two kinds of feedback:

- Indication of a performance improvement gained by using a specific coding recommendation or microarchitectural feature.
- Information on whether a change in the program has improved or degraded performance with respect to a particular metric.

The VTune Performance Analyzer also provides measures for a number of workload characteristics, including:

- Retirement throughput of instruction execution as an indication of the degree of extractable instruction-level parallelism in the workload.
- Data traffic locality as an indication of the stress point of the cache and memory hierarchy.
- Data traffic parallelism as an indication of the degree of effectiveness of amortization of data access latency.

NOTE

Improving performance in one part of the machine does not necessarily bring significant gains to overall performance. It is possible to degrade overall performance by improving performance for some particular metric.

Where appropriate, coding recommendations in this chapter include descriptions of the VTune Performance Analyzer events that provide measurable data on the performance gain achieved by following the recommendations. For more on using the VTune analyzer, refer to the application's online help.

3.2 PROCESSOR PERSPECTIVES

Many coding recommendations for work well across modern microarchitectures from Intel Core microarchitecture to the Haswell microarchitecture. However, there are situations where a recommendation may benefit one microarchitecture more than another. Some of these are:

- Instruction decode throughput is important. Additionally, taking advantage of decoded ICache, Loop Stream Detector and macrofusion can further improve front end performance.
- Generating code to take advantage 4 decoders and employ micro-fusion and macro-fusion so that each of three simple decoders are not restricted to handling simple instructions consisting of one micro-op.

- On processors based on Sandy Bridge, Ivy Bridge and Haswell microarchitectures, the code size for optimal front end performance is associated with the decode ICache.
- Dependencies for partial register writes can incur varying degree of penalties To avoid false dependences from partial register updates, use full register updates and extended moves.
- Use appropriate instructions that support dependence-breaking (e.g. PXOR, SUB, XOR, XORPS).
- Hardware prefetching can reduce the effective memory latency for data and instruction accesses in general. But different microarchitectures may require some custom modifications to adapt to the specific hardware prefetch implementation of each microarchitecture.

3.2.1 CPUID Dispatch Strategy and Compatible Code Strategy

When optimum performance on all processor generations is desired, applications can take advantage of the CPUID instruction to identify the processor generation and integrate processor-specific instructions into the source code. The Intel C++ Compiler supports the integration of different versions of the code for different target processors. The selection of which code to execute at runtime is made based on the CPU identifiers. Binary code targeted for different processor generations can be generated under the control of the programmer or by the compiler.

For applications that target multiple generations of microarchitectures, and where minimum binary code size and single code path is important, a compatible code strategy is the best. Optimizing applications using techniques developed for the Intel Core microarchitecture and combined with Intel microarchitecture code name Nehalem are likely to improve code efficiency and scalability when running on processors based on current and future generations of Intel 64 and IA-32 processors.

3.2.2 Transparent Cache-Parameter Strategy

If the CPUID instruction supports function leaf 4, also known as deterministic cache parameter leaf, the leaf reports cache parameters for each level of the cache hierarchy in a deterministic and forward-compatible manner across Intel 64 and IA-32 processor families.

For coding techniques that rely on specific parameters of a cache level, using the deterministic cache parameter allows software to implement techniques in a way that is forward-compatible with future generations of Intel 64 and IA-32 processors, and cross-compatible with processors equipped with different cache sizes.

3.2.3 Threading Strategy and Hardware Multithreading Support

Intel 64 and IA-32 processor families offer hardware multithreading support in two forms: dual-core technology and HT Technology.

To fully harness the performance potential of hardware multithreading in current and future generations of Intel 64 and IA-32 processors, software must embrace a threaded approach in application design. At the same time, to address the widest range of installed machines, multi-threaded software should be able to run without failure on a single processor without hardware multithreading support and should achieve performance on a single logical processor that is comparable to an unthreaded implementation (if such comparison can be made). This generally requires architecting a multi-threaded application to minimize the overhead of thread synchronization. Additional guidelines on multithreading are discussed in Chapter 8, “Multicore and Hyper-Threading Technology.”

3.3 CODING RULES, SUGGESTIONS AND TUNING HINTS

This section includes rules, suggestions and hints. They are targeted for engineers who are:

- Modifying source code to enhance performance (user/source rules).
- Writing assemblers or compilers (assembly/compiler rules).

- Doing detailed performance tuning (tuning suggestions).

Coding recommendations are ranked in importance using two measures:

- Local impact (high, medium, or low) refers to a recommendation’s affect on the performance of a given instance of code.
- Generality (high, medium, or low) measures how often such instances occur across all application domains. Generality may also be thought of as “frequency”.

These recommendations are approximate. They can vary depending on coding style, application domain, and other factors.

The purpose of the high, medium, and low (H, M, and L) priorities is to suggest the relative level of performance gain one can expect if a recommendation is implemented.

Because it is not possible to predict the frequency of a particular code instance in applications, priority hints cannot be directly correlated to application-level performance gain. In cases in which application-level performance gain has been observed, we have provided a quantitative characterization of the gain (for information only). In cases in which the impact has been deemed inapplicable, no priority is assigned.

3.4 OPTIMIZING THE FRONT END

Optimizing the front end covers two aspects:

- Maintaining steady supply of micro-ops to the execution engine — Mispredicted branches can disrupt streams of micro-ops, or cause the execution engine to waste execution resources on executing streams of micro-ops in the non-architected code path. Much of the tuning in this respect focuses on working with the Branch Prediction Unit. Common techniques are covered in Section 3.4.1, “Branch Prediction Optimization.”
- Supplying streams of micro-ops to utilize the execution bandwidth and retirement bandwidth as much as possible — For Intel Core microarchitecture and Intel Core Duo processor family, this aspect focuses maintaining high decode throughput. In Intel microarchitecture code name Sandy Bridge, this aspect focuses on keeping the hod code running from Decoded ICache. Techniques to maximize decode throughput for Intel Core microarchitecture are covered in Section 3.4.2, “Fetch and Decode Optimization.”

3.4.1 Branch Prediction Optimization

Branch optimizations have a significant impact on performance. By understanding the flow of branches and improving their predictability, you can increase the speed of code significantly.

Optimizations that help branch prediction are:

- Keep code and data on separate pages. This is very important; see Section 3.6, “Optimizing Memory Accesses,” for more information.
- Eliminate branches whenever possible.
- Arrange code to be consistent with the static branch prediction algorithm.
- Use the PAUSE instruction in spin-wait loops.
- Inline functions and pair up calls and returns.
- Unroll as necessary so that repeatedly-executed loops have sixteen or fewer iterations (unless this causes an excessive code size increase).
- Avoid putting two conditional branch instructions in a loop so that both have the same branch target address and, at the same time, belong to (i.e. have their last bytes' addresses within) the same 16-byte aligned code block.

3.4.1.1 Eliminating Branches

Eliminating branches improves performance because:

- It reduces the possibility of mispredictions.
- It reduces the number of required branch target buffer (BTB) entries. Conditional branches, which are never taken, do not consume BTB resources.

There are four principal ways of eliminating branches:

- Arrange code to make basic blocks contiguous.
- Unroll loops, as discussed in Section 3.4.1.7, “Loop Unrolling.”
- Use the CMOV instruction.
- Use the SETCC instruction.

The following rules apply to branch elimination:

Assembly/Compiler Coding Rule 1. (MH impact, M generality) *Arrange code to make basic blocks contiguous and eliminate unnecessary branches.*

Assembly/Compiler Coding Rule 2. (M impact, ML generality) *Use the SETCC and CMOV instructions to eliminate unpredictable conditional branches where possible. Do not do this for predictable branches. Do not use these instructions to eliminate all unpredictable conditional branches (because using these instructions will incur execution overhead due to the requirement for executing both paths of a conditional branch). In addition, converting a conditional branch to SETCC or CMOV trades off control flow dependence for data dependence and restricts the capability of the out-of-order engine. When tuning, note that all Intel 64 and IA-32 processors usually have very high branch prediction rates. Consistently mispredicted branches are generally rare. Use these instructions only if the increase in computation time is less than the expected cost of a mispredicted branch.*

Consider a line of C code that has a condition dependent upon one of the constants:

```
X = (A < B) ? CONST1 : CONST2;
```

This code conditionally compares two values, A and B. If the condition is true, X is set to CONST1; otherwise it is set to CONST2. An assembly code sequence equivalent to the above C code can contain branches that are not predictable if there are no correlation in the two values.

Example 3-1 shows the assembly code with unpredictable branches. The unpredictable branches can be removed with the use of the SETCC instruction. Example 3-2 shows optimized code that has no branches.

Example 3-1. Assembly Code with an Unpredictable Branch

```
    cmp a, b                ; Condition
    jbe L30                 ; Conditional branch
    mov ebx const1          ; ebx holds X
    jmp L31                 ; Unconditional branch
L30:
    mov ebx, const2
L31:
```

Example 3-2. Code Optimization to Eliminate Branches

```
xor   ebx, ebx      ; Clear ebx (X in the C code)
cmp   A, B
setge bl            ; When ebx = 0 or 1
                    ; OR the complement condition
sub   ebx, 1        ; ebx=11...11 or 00...00
and   ebx, CONST3; CONST3 = CONST1-CONST2
add   ebx, CONST2; ebx=CONST1 or CONST2
```

The optimized code in Example 3-2 sets EBX to zero, then compares A and B. If A is greater than or equal to B, EBX is set to one. Then EBX is decreased and AND'd with the difference of the constant values. This sets EBX to either zero or the difference of the values. By adding CONST2 back to EBX, the correct value is written to EBX. When CONST2 is equal to zero, the last instruction can be deleted.

Another way to remove branches is to use the CMOV and FCMOV instructions. Example 3-3 shows how to change a TEST and branch instruction sequence using CMOV to eliminate a branch. If the TEST sets the equal flag, the value in EBX will be moved to EAX. This branch is data-dependent, and is representative of an unpredictable branch.

Example 3-3. Eliminating Branch with CMOV Instruction

```
    test ecx, ecx
    jne  1H
    mov  eax, ebx


1H:
; To optimize code, combine jne and mov into one cmovcc instruction that checks the equal flag
    test      ecx, ecx              ; Test the flags
    cmoveq    eax, ebx              ; If the equal flag is set, move
                                    ; ebx to eax- the 1H: tag no longer needed
```

3.4.1.2 Spin-Wait and Idle Loops

The Pentium 4 processor introduces a new PAUSE instruction; the instruction is architecturally a NOP on Intel 64 and IA-32 processor implementations.

To the Pentium 4 and later processors, this instruction acts as a hint that the code sequence is a spin-wait loop. Without a PAUSE instruction in such loops, the Pentium 4 processor may suffer a severe penalty when exiting the loop because the processor may detect a possible memory order violation. Inserting the PAUSE instruction significantly reduces the likelihood of a memory order violation and as a result improves performance.

In Example 3-4, the code spins until memory location A matches the value stored in the register EAX. Such code sequences are common when protecting a critical section, in producer-consumer sequences, for barriers, or other synchronization.

Example 3-4. Use of PAUSE Instruction

```
lock:   cmp eax, a
        jne loop
        ; Code in critical section:
loop:   pause
        cmp eax, a
        jne loop
        jmp lock
```

3.4.1.3 Static Prediction

Branches that do not have a history in the BTB (see Section 3.4.1, "Branch Prediction Optimization") are predicted using a static prediction algorithm:

- Predict unconditional branches to be taken.
- Predict indirect branches to be NOT taken.

The following rule applies to static elimination:

Assembly/Compiler Coding Rule 3. (M impact, H generality) *Arrange code to be consistent with the static branch prediction algorithm: make the fall-through code following a conditional branch be the likely target for a branch with a forward target, and make the fall-through code following a conditional branch be the unlikely target for a branch with a backward target.*

Example 3-5 illustrates the static branch prediction algorithm. The body of an IF-THEN conditional is predicted.

Example 3-5. Static Branch Prediction Algorithm

```
//Forward condition branches not taken (fall through)
    IF<condition> {....
    ↓
    }

IF<condition> {...
    ↓
    }

//Backward conditional branches are taken
    LOOP {...
    ↑ —— }<condition>

//Unconditional branches taken
     JMP
    ------→
```

Example 3-6 and Example 3-7 provide basic rules for a static prediction algorithm. In Example 3-6, the backward branch (JC BEGIN) is not in the BTB the first time through; therefore, the BTB does not issue a prediction. The static predictor, however, will predict the branch to be taken, so a misprediction will not occur.

Example 3-6. Static Taken Prediction

```
Begin:  mov     eax, mem32
        and     eax, ebx
        imul    eax, edx
        shld    eax, 7
        jc      Begin
```

The first branch instruction (JC BEGIN) in Example 3-7 is a conditional forward branch. It is not in the BTB the first time through, but the static predictor will predict the branch to fall through. The static prediction algorithm correctly predicts that the CALL CONVERT instruction will be taken, even before the branch has any branch history in the BTB.

Example 3-7. Static Not-Taken Prediction

```
        mov     eax, mem32
        and     eax, ebx
        imul    eax, edx
        shld    eax, 7
        jc      Begin
        mov     eax, 0
Begin:  call    Convert
```

The Intel Core microarchitecture does not use the static prediction heuristic. However, to maintain consistency across Intel 64 and IA-32 processors, software should maintain the static prediction heuristic as the default.

3.4.1.4 Inlining, Calls and Returns

The return address stack mechanism augments the static and dynamic predictors to optimize specifically for calls and returns. It holds 16 entries, which is large enough to cover the call depth of most programs. If there is a chain of more than 16 nested calls and more than 16 returns in rapid succession, performance may degrade.

The trace cache in Intel NetBurst microarchitecture maintains branch prediction information for calls and returns. As long as the trace with the call or return remains in the trace cache and the call and return targets remain unchanged, the depth limit of the return address stack described above will not impede performance.

To enable the use of the return stack mechanism, calls and returns must be matched in pairs. If this is done, the likelihood of exceeding the stack depth in a manner that will impact performance is very low.

The following rules apply to inlining, calls, and returns:

Assembly/Compiler Coding Rule 4. (MH impact, MH generality) *Near calls must be matched with near returns, and far calls must be matched with far returns. Pushing the return address on the stack and jumping to the routine to be called is not recommended since it creates a mismatch in calls and returns.*

Calls and returns are expensive; use inlining for the following reasons:

- Parameter passing overhead can be eliminated.
- In a compiler, inlining a function exposes more opportunity for optimization.
- If the inlined routine contains branches, the additional context of the caller may improve branch prediction within the routine.
- A mispredicted branch can lead to performance penalties inside a small function that are larger than those that would occur if that function is inlined.

Assembly/Compiler Coding Rule 5. (MH impact, MH generality) *Selectively inline a function if doing so decreases code size or if the function is small and the call site is frequently executed.*

Assembly/Compiler Coding Rule 6. (H impact, H generality) *Do not inline a function if doing so increases the working set size beyond what will fit in the trace cache.*

Assembly/Compiler Coding Rule 7. (ML impact, ML generality) *If there are more than 16 nested calls and returns in rapid succession; consider transforming the program with inline to reduce the call depth.*

Assembly/Compiler Coding Rule 8. (ML impact, ML generality) *Favor inlining small functions that contain branches with poor prediction rates. If a branch misprediction results in a RETURN being prematurely predicted as taken, a performance penalty may be incurred.*

Assembly/Compiler Coding Rule 9. (L impact, L generality) *If the last statement in a function is a call to another function, consider converting the call to a jump. This will save the call/return overhead as well as an entry in the return stack buffer.*

Assembly/Compiler Coding Rule 10. (M impact, L generality) *Do not put more than four branches in a 16-byte chunk.*

Assembly/Compiler Coding Rule 11. (M impact, L generality) *Do not put more than two end loop branches in a 16-byte chunk.*

3.4.1.5 Code Alignment

Careful arrangement of code can enhance cache and memory locality. Likely sequences of basic blocks should be laid out contiguously in memory. This may involve removing unlikely code, such as code to handle error conditions, from the sequence. See Section 3.7, “Prefetching,” on optimizing the instruction prefetcher.

Assembly/Compiler Coding Rule 12. (M impact, H generality) *All branch targets should be 16-byte aligned.*

Assembly/Compiler Coding Rule 13. (M impact, H generality) *If the body of a conditional is not likely to be executed, it should be placed in another part of the program. If it is highly unlikely to be executed and code locality is an issue, it should be placed on a different code page.*

3.4.1.6 Branch Type Selection

The default predicted target for indirect branches and calls is the fall-through path. Fall-through prediction is overridden if and when a hardware prediction is available for that branch. The predicted branch target from branch prediction hardware for an indirect branch is the previously executed branch target.

The default prediction to the fall-through path is only a significant issue if no branch prediction is available, due to poor code locality or pathological branch conflict problems. For indirect calls, predicting the fall-through path is usually not an issue, since execution will likely return to the instruction after the associated return.

Placing data immediately following an indirect branch can cause a performance problem. If the data consists of all zeros, it looks like a long stream of ADDs to memory destinations and this can cause resource conflicts and slow down branch recovery. Also, data immediately following indirect branches may appear as branches to the branch predication hardware, which can branch off to execute other data pages. This can lead to subsequent self-modifying code problems.

Assembly/Compiler Coding Rule 14. (M impact, L generality) *When indirect branches are present, try to put the most likely target of an indirect branch immediately following the indirect branch. Alternatively, if indirect branches are common but they cannot be predicted by branch prediction hardware, then follow the indirect branch with a UD2 instruction, which will stop the processor from decoding down the fall-through path.*

Indirect branches resulting from code constructs (such as switch statements, computed GOTOs or calls through pointers) can jump to an arbitrary number of locations. If the code sequence is such that the target destination of a branch goes to the same address most of the time, then the BTB will predict accurately most of the time. Since only one taken (non-fall-through) target can be stored in the BTB, indirect branches with multiple taken targets may have lower prediction rates.

The effective number of targets stored may be increased by introducing additional conditional branches. Adding a conditional branch to a target is fruitful if:

- The branch direction is correlated with the branch history leading up to that branch; that is, not just the last target, but how it got to this branch.
- The source/target pair is common enough to warrant using the extra branch prediction capacity. This may increase the number of overall branch mispredictions, while improving the misprediction of indirect branches. The profitability is lower if the number of mispredicting branches is very large.

User/Source Coding Rule 1. (M impact, L generality) *If an indirect branch has two or more common taken targets and at least one of those targets is correlated with branch history leading up to the branch, then convert the indirect branch to a tree where one or more indirect branches are preceded by conditional branches to those targets. Apply this "peeling" procedure to the common target of an indirect branch that correlates to branch history.*

The purpose of this rule is to reduce the total number of mispredictions by enhancing the predictability of branches (even at the expense of adding more branches). The added branches must be predictable for this to be worthwhile. One reason for such predictability is a strong correlation with preceding branch history. That is, the directions taken on preceding branches are a good indicator of the direction of the branch under consideration.

Example 3-8 shows a simple example of the correlation between a target of a preceding conditional branch and a target of an indirect branch.

Example 3-8. Indirect Branch With Two Favored Targets

```
function ()
{
int n = rand();             // random integer 0 to RAND_MAX
     if ( ! (n & 0x01) ) {  // n will be 0 half the times
          n = 0;            // updates branch history to predict taken
     }
     // indirect branches with multiple taken targets
     // may have lower prediction rates

 switch (n) {
     case 0: handle_0(); break;     // common target, correlated with
                                    // branch history that is forward taken
     case 1: handle_1(); break;     // uncommon
     case 3: handle_3(); break;     // uncommon
     default: handle_other();       // common target
      }
}
```

Correlation can be difficult to determine analytically, for a compiler and for an assembly language programmer. It may be fruitful to evaluate performance with and without peeling to get the best performance from a coding effort.

An example of peeling out the most favored target of an indirect branch with correlated branch history is shown in Example 3-9.

Example 3-9. A Peeling Technique to Reduce Indirect Branch Misprediction

```
function ()
{
   int n = rand();                          // Random integer 0 to RAND_MAX
     if( ! (n & 0x01) ) THEN
           n = 0;                           // n will be 0 half the times
     if (!n) THEN
          handle_0();                       // Peel out the most common target
                                            // with correlated branch history

      {
        switch (n) {
          case 1: handle_1(); break;        // Uncommon
          case 3: handle_3(); break;        // Uncommon

          default: handle_other();          // Make the favored target in
                                            // the fall-through path
          }
     }
}
```

3.4.1.7 Loop Unrolling

Benefits of unrolling loops are:

- Unrolling amortizes the branch overhead, since it eliminates branches and some of the code to manage induction variables.
- Unrolling allows one to aggressively schedule (or pipeline) the loop to hide latencies. This is useful if you have enough free registers to keep variables live as you stretch out the dependence chain to expose the critical path.
- Unrolling exposes the code to various other optimizations, such as removal of redundant loads, common subexpression elimination, and so on.

The potential costs of unrolling loops are:

- Excessive unrolling or unrolling of very large loops can lead to increased code size. This can be harmful if the unrolled loop no longer fits in the trace cache (TC).
- Unrolling loops whose bodies contain branches increases demand on BTB capacity. If the number of iterations of the unrolled loop is 16 or fewer, the branch predictor should be able to correctly predict branches in the loop body that alternate direction.

Assembly/Compiler Coding Rule 15. (H impact, M generality) *Unroll small loops until the overhead of the branch and induction variable accounts (generally) for less than 10% of the execution time of the loop.*

Assembly/Compiler Coding Rule 16. (H impact, M generality) *Avoid unrolling loops excessively; this may thrash the trace cache or instruction cache.*

Assembly/Compiler Coding Rule 17. (M impact, M generality) *Unroll loops that are frequently executed and have a predictable number of iterations to reduce the number of iterations to 16 or fewer. Do this unless it increases code size so that the working set no longer fits in the trace or instruction cache. If the loop body contains more than one conditional branch, then unroll so that the number of iterations is 16/(# conditional branches).*

Example 3-10 shows how unrolling enables other optimizations.

Example 3-10. Loop Unrolling

```
Before unrolling:
    do i = 1, 100
        if ( i mod 2 == 0 ) then a( i ) = x
            else a( i ) = y
    enddo
After unrolling
    do i = 1, 100, 2
        a( i ) = y
        a( i+1 ) = x
    enddo
```

In this example, the loop that executes 100 times assigns X to every even-numbered element and Y to every odd-numbered element. By unrolling the loop you can make assignments more efficiently, removing one branch in the loop body.

3.4.1.8 Compiler Support for Branch Prediction

Compilers generate code that improves the efficiency of branch prediction in Intel processors. The Intel C++ Compiler accomplishes this by:

- Keeping code and data on separate pages.
- Using conditional move instructions to eliminate branches.
- Generating code consistent with the static branch prediction algorithm.
- Inlining where appropriate.
- Unrolling if the number of iterations is predictable.

With profile-guided optimization, the compiler can lay out basic blocks to eliminate branches for the most frequently executed paths of a function or at least improve their predictability. Branch prediction need not be a concern at the source level. For more information, see Intel C++ Compiler documentation.

3.4.2 Fetch and Decode Optimization

Intel Core microarchitecture provides several mechanisms to increase front end throughput. Techniques to take advantage of some of these features are discussed below.

3.4.2.1 Optimizing for Micro-fusion

An Instruction that operates on a register and a memory operand decodes into more micro-ops than its corresponding register-register version. Replacing the equivalent work of the former instruction using the register-register version usually require a sequence of two instructions. The latter sequence is likely to result in reduced fetch bandwidth.

Assembly/Compiler Coding Rule 18. (ML impact, M generality) *For improving fetch/decode throughput, Give preference to memory flavor of an instruction over the register-only flavor of the same instruction, if such instruction can benefit from micro-fusion.*

The following examples are some of the types of micro-fusions that can be handled by all decoders:

- All stores to memory, including store immediate. Stores execute internally as two separate micro-ops: store-address and store-data.
- All "read-modify" (load+op) instructions between register and memory, for example:

```
ADDPS   XMM9, OWORD PTR [RSP+40]
FADD    DOUBLE PTR [RDI+RSI*8]
XOR     RAX, QWORD PTR [RBP+32]
```

- All instructions of the form "load and jump," for example:

```
JMP     [RDI+200]
RET
```

- CMP and TEST with immediate operand and memory.

An Intel 64 instruction with RIP relative addressing is not micro-fused in the following cases:

- When an additional immediate is needed, for example:

```
CMP     [RIP+400], 27
MOV     [RIP+3000], 142
```

- When an RIP is needed for control flow purposes, for example:

```
JMP     [RIP+5000000]
```

In these cases, Intel Core microarchitecture and Intel microarchitecture code name Sandy Bridge provides a 2 micro-op flow from decoder 0, resulting in a slight loss of decode bandwidth since 2 micro-op flow must be steered to decoder 0 from the decoder with which it was aligned.

RIP addressing may be common in accessing global data. Since it will not benefit from micro-fusion, compiler may consider accessing global data with other means of memory addressing.

3.4.2.2 Optimizing for Macro-fusion

Macro-fusion merges two instructions to a single micro-op. Intel Core microarchitecture performs this hardware optimization under limited circumstances.

The first instruction of the macro-fused pair must be a CMP or TEST instruction. This instruction can be REG-REG, REG-IMM, or a micro-fused REG-MEM comparison. The second instruction (adjacent in the instruction stream) should be a conditional branch.

Since these pairs are common ingredient in basic iterative programming sequences, macro-fusion improves performance even on un-recompiled binaries. All of the decoders can decode one macro-fused

pair per cycle, with up to three other instructions, resulting in a peak decode bandwidth of 5 instructions per cycle.

Each macro-fused instruction executes with a single dispatch. This process reduces latency, which in this case shows up as a cycle removed from branch mispredict penalty. Software also gain all other fusion benefits: increased rename and retire bandwidth, more storage for instructions in-flight, and power savings from representing more work in fewer bits.

The following list details when you can use macro-fusion:

- CMP or TEST can be fused when comparing:

  REG-REG. For example: CMP EAX,ECX; JZ label
  REG-IMM. For example: CMP EAX,0x80; JZ label
  REG-MEM. For example: CMP EAX,[ECX]; JZ label
  MEM-REG. For example: CMP [EAX],ECX; JZ label

- TEST can fused with all conditional jumps.
- CMP can be fused with only the following conditional jumps in Intel Core microarchitecture. These conditional jumps check carry flag (CF) or zero flag (ZF). jump. The list of macro-fusion-capable conditional jumps are:

  JA or JNBE
  JAE or JNB or JNC
  JE or JZ
  JNA or JBE
  JNAE or JC or JB
  JNE or JNZ

CMP and TEST can not be fused when comparing MEM-IMM (e.g. CMP [EAX],0x80; JZ label). Macro-fusion is not supported in 64-bit mode for Intel Core microarchitecture.

- Intel microarchitecture code name Nehalem supports the following enhancements in macrofusion:
  — CMP can be fused with the following conditional jumps (that was not supported in Intel Core microarchitecture):
    - JL or JNGE
    - JGE or JNL
    - JLE or JNG
    - JG or JNLE
  — Macro-fusion is support in 64-bit mode.
- Enhanced macrofusion support in Intel microarchitecture code name Sandy Bridge is summarized in Table 3-1 with additional information in Section 2.3.2.1 and Example 3-15:

Table 3-1. Macro-Fusible Instructions in Intel Microarchitecture Code Name Sandy Bridge

Instructions	TEST	AND	CMP	ADD	SUB	INC	DEC
JO/JNO	Y	Y	N	N	N	N	N
JC/JB/JAE/JNB	Y	Y	Y	Y	Y	N	N
JE/JZ/JNE/JNZ	Y	Y	Y	Y	Y	Y	Y
JNA/JBE/JA/JNBE	Y	Y	Y	Y	Y	N	N
JS/JNS/JP/JPE/JNP/JPO	Y	Y	N	N	N	N	N
JL/JNGE/JGE/JNL/JLE/JNG/JG/JNLE	Y	Y	Y	Y	Y	Y	Y

Assembly/Compiler Coding Rule 19. (M impact, ML generality) *Employ macro-fusion where possible using instruction pairs that support macro-fusion. Prefer TEST over CMP if possible. Use unsigned variables and unsigned jumps when possible. Try to logically verify that a variable is non-negative at the time of comparison. Avoid CMP or TEST of MEM-IMM flavor when possible. However, do not add other instructions to avoid using the MEM-IMM flavor.*

Example 3-11. Macro-fusion, Unsigned Iteration Count

	Without Macro-fusion	**With Macro-fusion**
C code	for (int[1] i = 0; i < 1000; i++) a++;	for (unsigned int[2] i = 0; i < 1000; i++) a++;

Disassembly	for (int i = 0; i < 1000; i++)
	mov dword ptr [i], 0
	jmp First
	Loop:
	mov eax, dword ptr [i]
	add eax, 1
	mov dword ptr [i], eax
	
	First:
	cmp dword ptr [i], 3E8H[3]
	jge End
	a++;
	mov eax, dword ptr [a]
	addqq eax,1
	mov dword ptr [a], eax
	jmp Loop
	End:
		for (unsigned int i = 0; i < 1000; i++)
		xor eax, eax
		mov dword ptr [i], eax
		jmp First
		Loop:
		mov eax, dword ptr [i]
		add eax, 1
		mov dword ptr [i], eax
		First:
		cmp eax, 3E8H [4]
		jae End
		a++;
		mov eax, dword ptr [a]
		add eax, 1
		mov dword ptr [a], eax
		jmp Loop
		End:

NOTES:

1. Signed iteration count inhibits macro-fusion.
2. Unsigned iteration count is compatible with macro-fusion.
3. CMP MEM-IMM, JGE inhibit macro-fusion.
4. CMP REG-IMM, JAE permits macro-fusion.

Example 3-12. Macro-fusion, If Statement

	Without Macro-fusion	**With Macro-fusion**

C code	int[1] a = 7;
	if (a < 77)
	a++;
	else
	a--;
		unsigned int[2] a = 7;
		if (a < 77)
		a++;
		else
		a--;

Disassembly	int a = 7;
	mov dword ptr [a], 7
	if (a < 77)
	cmp dword ptr [a], 4DH [3]
	jge Dec
		unsigned int a = 7;
		mov dword ptr [a], 7
		if (a < 77)
		mov eax, dword ptr [a]
		cmp eax, 4DH
		jae Dec

Example 3-12. Macro-fusion, If Statement (Contd.)

	Without Macro-fusion	With Macro-fusion

	a++;
	mov eax, dword ptr [a]
	add eax, 1
	mov dword ptr [a], eax
	else
	jmp End
	a--;
	Dec:
	mov eax, dword ptr [a]
	sub eax, 1
	mov dword ptr [a], eax
	End::
		a++;
		add eax,1
		mov dword ptr [a], eax
		else
		jmp End
		a--;
		Dec:
		sub eax, 1
		mov dword ptr [a], eax
		End::

NOTES:

1. Signed iteration count inhibits macro-fusion.
2. Unsigned iteration count is compatible with macro-fusion.
3. CMP MEM-IMM, JGE inhibit macro-fusion.

Assembly/Compiler Coding Rule 20. (M impact, ML generality) *Software can enable macro fusion when it can be logically determined that a variable is non-negative at the time of comparison; use TEST appropriately to enable macro-fusion when comparing a variable with 0.*

Example 3-13. Macro-fusion, Signed Variable

Without Macro-fusion	With Macro-fusion
test ecx, ecx jle OutSideTheIF cmp ecx, 64H jge OutSideTheIF <IF BLOCK CODE> OutSideTheIF:	test ecx, ecx jle OutSideTheIF cmp ecx, 64H jae OutSideTheIF <IF BLOCK CODE> OutSideTheIF:

For either signed or unsigned variable ‘a’; “CMP a,0” and “TEST a,a” produce the same result as far as the flags are concerned. Since TEST can be macro-fused more often, software can use “TEST a,a” to replace “CMP a,0” for the purpose of enabling macro-fusion.

Example 3-14. Macro-fusion, Signed Comparison

C Code	Without Macro-fusion	With Macro-fusion
if (a == 0)	cmp a, 0 jne lbl ... lbl:	test a, a jne lbl ... lbl:
if (a >= 0)	cmp a, 0 jl lbl; ... lbl:	test a, a jl lbl ... lbl:

Intel microarchitecture code name Sandy Bridge enables more arithmetic and logic instructions to macro-fuse with conditional branches. In loops where the ALU ports are already congested, performing one of these macro-fusions can relieve the pressure, as the macro-fused instruction consumes only port 5, instead of an ALU port plus port 5.

In Example 3-15, the “add/cmp/jnz” loop contains two ALU instructions that can be dispatched via either port 0, 1, 5. So there is higher probability of port 5 might bind to either ALU instruction causing JNZ to

wait a cycle. The "sub/jnz" loop, the likelihood of ADD/SUB/JNZ can be dispatched in the same cycle is increased because only SUB is free to bind with either port 0, 1, 5.

Example 3-15. Additional Macro-fusion Benefit in Intel Microarchitecture Code Name Sandy Bridge

Add + cmp + jnz alternative		Loop control with sub + jnz	
lea	rdx, buff	lea	rdx, buff - 4
xor	rcx, rcx	xor	rcx, LEN
xor	eax, eax	xor	eax, eax
loop:		loop:	
add	eax, [rdx + 4 * rcx]	add	eax, [rdx + 4 * rcx]
add	rcx, 1	sub	rcx, 1
cmp	rcx, LEN	jnz	loop
jnz	loop		

3.4.2.3 Length-Changing Prefixes (LCP)

The length of an instruction can be up to 15 bytes in length. Some prefixes can dynamically change the length of an instruction that the decoder must recognize. Typically, the pre-decode unit will estimate the length of an instruction in the byte stream assuming the absence of LCP. When the predecoder encounters an LCP in the fetch line, it must use a slower length decoding algorithm. With the slower length decoding algorithm, the predecoder decodes the fetch in 6 cycles, instead of the usual 1 cycle. Normal queuing throughout of the machine pipeline generally cannot hide LCP penalties.

The prefixes that can dynamically change the length of a instruction include:

- Operand size prefix (0x66).
- Address size prefix (0x67).

The instruction MOV DX, 01234h is subject to LCP stalls in processors based on Intel Core microarchitecture, and in Intel Core Duo and Intel Core Solo processors. Instructions that contain imm16 as part of their fixed encoding but do not require LCP to change the immediate size are not subject to LCP stalls. The REX prefix (4xh) in 64-bit mode can change the size of two classes of instruction, but does not cause an LCP penalty.

If the LCP stall happens in a tight loop, it can cause significant performance degradation. When decoding is not a bottleneck, as in floating-point heavy code, isolated LCP stalls usually do not cause performance degradation.

Assembly/Compiler Coding Rule 21. (MH impact, MH generality) *Favor generating code using imm8 or imm32 values instead of imm16 values.*

If imm16 is needed, load equivalent imm32 into a register and use the word value in the register instead.

Double LCP Stalls

Instructions that are subject to LCP stalls and cross a 16-byte fetch line boundary can cause the LCP stall to trigger twice. The following alignment situations can cause LCP stalls to trigger twice:

- An instruction is encoded with a MODR/M and SIB byte, and the fetch line boundary crossing is between the MODR/M and the SIB bytes.
- An instruction starts at offset 13 of a fetch line references a memory location using register and immediate byte offset addressing mode.

The first stall is for the 1st fetch line, and the 2nd stall is for the 2nd fetch line. A double LCP stall causes a decode penalty of 11 cycles.

The following examples cause LCP stall once, regardless of their fetch-line location of the first byte of the instruction:

```
ADD DX, 01234H
ADD word ptr [EDX], 01234H
```

```
ADD word ptr 012345678H[EDX], 01234H
ADD word ptr [012345678H], 01234H
```

The following instructions cause a double LCP stall when starting at offset 13 of a fetch line:

```
ADD word ptr [EDX+ESI], 01234H
ADD word ptr 012H[EDX], 01234H
ADD word ptr 012345678H[EDX+ESI], 01234H
```

To avoid double LCP stalls, do not use instructions subject to LCP stalls that use SIB byte encoding or addressing mode with byte displacement.

False LCP Stalls

False LCP stalls have the same characteristics as LCP stalls, but occur on instructions that do not have any imm16 value.

False LCP stalls occur when (a) instructions with LCP that are encoded using the F7 opcodes, and (b) are located at offset 14 of a fetch line. These instructions are: not, neg, div, idiv, mul, and imul. False LCP experiences delay because the instruction length decoder can not determine the length of the instruction before the next fetch line, which holds the exact opcode of the instruction in its MODR/M byte.

The following techniques can help avoid false LCP stalls:

- Upcast all short operations from the F7 group of instructions to long, using the full 32 bit version.
- Ensure that the F7 opcode never starts at offset 14 of a fetch line.

Assembly/Compiler Coding Rule 22. (M impact, ML generality) *Ensure instructions using 0xF7 opcode byte does not start at offset 14 of a fetch line; and avoid using these instruction to operate on 16-bit data, upcast short data to 32 bits.*

Example 3-16. Avoiding False LCP Delays with 0xF7 Group Instructions

A Sequence Causing Delay in the Decoder	Alternate Sequence to Avoid Delay
neg word ptr a	movsx eax, word ptr a neg eax mov word ptr a, AX

3.4.2.4 Optimizing the Loop Stream Detector (LSD)

Loops that fit the following criteria are detected by the LSD and replayed from the instruction queue to feed the decoder in Intel Core microarchitecture:

- Must be less than or equal to four 16-byte fetches.
- Must be less than or equal to 18 instructions.
- Can contain no more than four taken branches and none of them can be a RET.
- Should usually have more than 64 iterations.

Loop Stream Detector in Intel microarchitecture code name Nehalem is improved by:

- Caching decoded micro-operations in the instruction decoder queue (IDQ, see Section 2.5.2) to feed the rename/alloc stage.
- The size of the LSD is increased to 28 micro-ops.

The LSD and micro-op queue implementation continue to improve in Sandy Bridge and Haswell microarchitectures. They have the following characteristics:

Table 3-2. Small Loop Criteria Detected by Sandy Bridge and Haswell Microarchitectures

Sandy Bridge and Ivy Bridge microarchitectures	Haswell microarchitecture
Up to 8 chunk fetches of 32 instruction bytes	8 chunk fetches if HTT active, 11 chunk fetched if HTT off
Up to 28 micro ops	28 micro-ops if HTT active, 56 micro-ops if HTT off
All micro-ops resident in Decoded Icache (i.e. DSB), but not from MSROM	All micro-ops resident in DSB, including micro-ops from MSRROM
No more than 8 taken branches	Relaxed
Exclude CALL and RET	Exclude CALL and RET
Mismatched stack operation disqualify	Same

Many calculation-intensive loops, searches and software string moves match these characteristics. These loops exceed the BPU prediction capacity and always terminate in a branch misprediction.

Assembly/Compiler Coding Rule 23. (MH impact, MH generality) *Break up a loop long sequence of instructions into loops of shorter instruction blocks of no more than the size of LSD.*

Assembly/Compiler Coding Rule 24. (MH impact, M generality) *Avoid unrolling loops containing LCP stalls, if the unrolled block exceeds the size of LSD.*

3.4.2.5 Exploit LSD Micro-op Emission Bandwidth in Intel® Microarchitecture Code Name Sandy Bridge

The LSD holds micro-ops that construct small "infinite" loops. Micro-ops from the LSD are allocated in the out-of-order engine. The loop in the LSD ends with a taken branch to the beginning of the loop. The taken branch at the end of the loop is always the last micro-op allocated in the cycle. The instruction at the beginning of the loop is always allocated at the next cycle. If the code performance is bound by front end bandwidth, unused allocation slots result in a bubble in allocation, and can cause performance degradation.

Allocation bandwidth in Intel microarchitecture code name Sandy Bridge is four micro-ops per cycle. Performance is best, when the number of micro-ops in the LSD result in the least number of unused allocation slots. You can use loop unrolling to control the number of micro-ops that are in the LSD.

In the Example 3-17, the code sums all array elements. The original code adds one element per iteration. It has three micro-ops per iteration, all allocated in one cycle. Code throughput is one load per cycle.

When unrolling the loop once there are five micro-ops per iteration, which are allocated in two cycles. Code throughput is still one load per cycle. Therefore there is no performance gain.

When unrolling the loop twice there are seven micro-ops per iteration, still allocated in two cycles. Since two loads can be executed in each cycle this code has a potential throughput of three load operations in two cycles.

.

Example 3-17. Unrolling Loops in LSD to Optimize Emission Bandwidth

No Unrolling	Unroll once	Unroll Twice
lp: add eax, [rsi + 4* rcx] dec rcx jnz lp	lp: add eax, [rsi + 4* rcx] add eax, [rsi + 4* rcx +4] add rcx, -2 jnz lp	lp: add eax, [rsi + 4* rcx] add eax, [rsi + 4* rcx +4] add eax, [rsi + 4* rcx + 8] add rcx, -3 jnz lp

3.4.2.6 Optimization for Decoded ICache

The decoded ICache is a new feature in Intel microarchitecture code name Sandy Bridge. Running the code from the Decoded ICache has two advantages:

- Higher bandwidth of micro-ops feeding the out-of-order engine.
- The front end does not need to decode the code that is in the Decoded ICache. This saves power.

There is overhead in switching between the Decoded ICache and the legacy decode pipeline. If your code switches frequently between the front end and the Decoded ICache, the penalty may be higher than running only from the legacy pipeline

To ensure “hot” code is feeding from the decoded ICache:

- Make sure each hot code block is less than about 500 instructions. Specifically, do not unroll to more than 500 instructions in a loop. This should enable Decoded ICache residency even when hyper-threading is enabled.
- For applications with very large blocks of calculations inside a loop, consider loop-fission: split the loop into multiple loops that fit in the Decoded ICache, rather than a single loop that overflows.
- If an application can be sure to run with only one thread per core, it can increase hot code block size to about 1000 instructions.

Dense Read-Modify-Write Code

The Decoded ICache can hold only up to 18 micro-ops per each 32 byte aligned memory chunk. Therefore, code with a high concentration of instructions that are encoded in a small number of bytes, yet have many micro-ops, may overflow the 18 micro-op limitation and not enter the Decoded ICache. Read-modify-write (RMW) instructions are a good example of such instructions.

RMW instructions accept one memory source operand, one register source operand, and use the source memory operand as the destination. The same functionality can be achieved by two or three instructions: the first reads the memory source operand, the second performs the operation with the second register source operand, and the last writes the result back to memory. These instructions usually result in the same number of micro-ops but use more bytes to encode the same functionality.

One case where RMW instructions may be used extensively is when the compiler optimizes aggressively for code size.

Here are some possible solutions to fit the hot code in the Decoded ICache:

- Replace RMW instructions with two or three instructions that have the same functionality. For example, “adc [rdi], rcx“ is only three bytes long; the equivalent sequence “adc rax, [rdi]“ + “mov [rdi], rax“ has a footprint of six bytes.
- Align the code so that the dense part is broken down among two different 32-byte chunks. This solution is useful when using a tool that aligns code automatically, and is indifferent to code changes.
- Spread the code by adding multiple byte NOPs in the loop. Note that this solution adds micro-ops for execution.

Align Unconditional Branches for Decoded ICache

For code entering the Decoded ICache, each unconditional branch is the last micro-op occupying a Decoded ICache Way. Therefore, only three unconditional branches per a 32 byte aligned chunk can enter the Decoded ICache.

Unconditional branches are frequent in jump tables and switch declarations. Below are examples for these constructs, and methods for writing them so that they fit in the Decoded ICache.

Compilers create jump tables for C++ virtual class methods or DLL dispatch tables. Each unconditional branch consumes five bytes; therefore up to seven of them can be associated with a 32-byte chunk. Thus jump tables may not fit in the Decoded ICache if the unconditional branches are too dense in each 32Byte-aligned chunk. This can cause performance degradation for code executing before and after the branch table.

The solution is to add multi-byte NOP instructions among the branches in the branch table. This may increases code size and should be used cautiously. However, these NOPs are not executed and therefore have no penalty in later pipe stages.

Switch-Case constructs represents a similar situation. Each evaluation of a case condition results in an unconditional branch. The same solution of using multi-byte NOP can apply for every three consecutive unconditional branches that fits inside an aligned 32-byte chunk.

Two Branches in a Decoded ICache Way

The Decoded ICache can hold up to two branches in a way. Dense branches in a 32 byte aligned chunk, or their ordering with other instructions may prohibit all the micro-ops of the instructions in the chunk from entering the Decoded ICache. This does not happen often. When it does happen, you can space the code with NOP instructions where appropriate. Make sure that these NOP instructions are not part of hot code.

Assembly/Compiler Coding Rule 25. (M impact, M generality) *Avoid putting explicit references to* ESP *in a sequence of stack operations (*POP, PUSH, CALL, RET*).*

3.4.2.7 Other Decoding Guidelines

Assembly/Compiler Coding Rule 26. (ML impact, L generality) *Use simple instructions that are less than eight bytes in length.*

Assembly/Compiler Coding Rule 27. (M impact, MH generality) *Avoid using prefixes to change the size of immediate and displacement.*

Long instructions (more than seven bytes) may limit the number of decoded instructions per cycle. Each prefix adds one byte to the length of instruction, possibly limiting the decoder’s throughput. In addition, multiple prefixes can only be decoded by the first decoder. These prefixes also incur a delay when decoded. If multiple prefixes or a prefix that changes the size of an immediate or displacement cannot be avoided, schedule them behind instructions that stall the pipe for some other reason.

3.5 OPTIMIZING THE EXECUTION CORE

The superscalar, out-of-order execution core(s) in recent generations of microarchitectures contain multiple execution hardware resources that can execute multiple micro-ops in parallel. These resources generally ensure that micro-ops execute efficiently and proceed with fixed latencies. General guidelines to make use of the available parallelism are:

- Follow the rules (see Section 3.4) to maximize useful decode bandwidth and front end throughput. These rules include favouring single micro-op instructions and taking advantage of micro-fusion, Stack pointer tracker and macro-fusion.
- Maximize rename bandwidth. Guidelines are discussed in this section and include properly dealing with partial registers, ROB read ports and instructions which causes side-effects on flags.
- Scheduling recommendations on sequences of instructions so that multiple dependency chains are alive in the reservation station (RS) simultaneously, thus ensuring that your code utilizes maximum parallelism.
- Avoid hazards, minimize delays that may occur in the execution core, allowing the dispatched micro-ops to make progress and be ready for retirement quickly.

3.5.1 Instruction Selection

Some execution units are not pipelined, this means that micro-ops cannot be dispatched in consecutive cycles and the throughput is less than one per cycle.

It is generally a good starting point to select instructions by considering the number of micro-ops associated with each instruction, favoring in the order of: single micro-op instructions, simple instruction with less then 4 micro-ops, and last instruction requiring microsequencer ROM (micro-ops which are executed out of the microsequencer involve extra overhead).

Assembly/Compiler Coding Rule 28. (M impact, H generality) *Favor single-micro-operation instructions. Also favor instruction with shorter latencies.*

A compiler may be already doing a good job on instruction selection. If so, user intervention usually is not necessary.

Assembly/Compiler Coding Rule 29. (M impact, L generality) *Avoid prefixes, especially multiple non-0F-prefixed opcodes.*

Assembly/Compiler Coding Rule 30. (M impact, L generality) *Do not use many segment registers.*

Assembly/Compiler Coding Rule 31. (M impact, M generality) *Avoid using complex instructions (for example, enter, leave, or loop) that have more than four µops and require multiple cycles to decode. Use sequences of simple instructions instead.*

Assembly/Compiler Coding Rule 32. (MH impact, M generality) *Use push/pop to manage stack space and address adjustments between function calls/returns instead of enter/leave. Using enter instruction with non-zero immediates can experience significant delays in the pipeline in addition to misprediction.*

Theoretically, arranging instructions sequence to match the 4-1-1-1 template applies to processors based on Intel Core microarchitecture. However, with macro-fusion and micro-fusion capabilities in the front end, attempts to schedule instruction sequences using the 4-1-1-1 template will likely provide diminishing returns.

Instead, software should follow these additional decoder guidelines:

- If you need to use multiple micro-op, non-microsequenced instructions, try to separate by a few single micro-op instructions. The following instructions are examples of multiple micro-op instruction not requiring micro-sequencer:

  ```
  ADC/SBB
  CMOVcc
  Read-modify-write instructions
  ```

- If a series of multiple micro-op instructions cannot be separated, try breaking the series into a different equivalent instruction sequence. For example, a series of read-modify-write instructions may go faster if sequenced as a series of read-modify + store instructions. This strategy could improve performance even if the new code sequence is larger than the original one.

3.5.1.1 Use of the INC and DEC Instructions

The INC and DEC instructions modify only a subset of the bits in the flag register. This creates a dependence on all previous writes of the flag register. This is especially problematic when these instructions are on the critical path because they are used to change an address for a load on which many other instructions depend.

Assembly/Compiler Coding Rule 33. (M impact, H generality) *INC and DEC instructions should be replaced with ADD or SUB instructions, because ADD and SUB overwrite all flags, whereas INC and DEC do not, therefore creating false dependencies on earlier instructions that set the flags.*

3.5.1.2 Integer Divide

Typically, an integer divide is preceded by a CWD or CDQ instruction. Depending on the operand size, divide instructions use DX:AX or EDX:EAX for the dividend. The CWD or CDQ instructions sign-extend AX or EAX into DX or EDX, respectively. These instructions have denser encoding than a shift and move would be, but they generate the same number of micro-ops. If AX or EAX is known to be positive, replace these instructions with:

```
xor dx, dx
```

or

```
xor edx, edx
```

Modern compilers typically can transform high-level language expression involving integer division where the divisor is a known integer constant at compile time into a faster sequence using IMUL instruction instead. Thus programmers should minimize integer division expression with divisor whose value can not be known at compile time.

Alternately, if certain known divisor value are favored over other unknown ranges, software may consider isolating the few favored, known divisor value into constant-divisor expressions.

Section 9.2.4 describes more detail of using MUL/IMUL to replace integer divisions.

3.5.1.3 Using LEA

In Intel microarchitecture code name Sandy Bridge, there are two significant changes to the performance characteristics of LEA instruction:

- LEA can be dispatched via port 1 and 5 in most cases, doubling the throughput over prior generations. However this apply only to LEA instructions with one or two source operands.

Example 3-18. Independent Two-Operand LEA Example

```
    mov     edx, N
    mov     eax, X
    mov     ecx, Y


loop:
    lea     ecx, [ecx = ecx *2]
    lea     eax, [eax = eax *5]
    and     ecx, 0xff
    and     eax, 0xff
    dec     edx
    jg      loop
```

- For LEA instructions with three source operands and some specific situations, instruction latency has increased to 3 cycles, and must dispatch via port 1:
  - — LEA that has all three source operands: base, index, and offset.
  - — LEA that uses base and index registers where the base is EBP, RBP, or R13.
  - — LEA that uses RIP relative addressing mode.
  - — LEA that uses 16-bit addressing mode.

Example 3-19. Alternative to Three-Operand LEA

3 operand LEA is slower	Two-operand LEA alternative	Alternative 2

#define K 1
uint32 an = 0;
uint32 N= mi_N;
mov ecx, N
xor esi, esi;
xor edx, edx;
cmp ecx, 2;
jb finished;
dec ecx;
	#define K 1
	uint32 an = 0;
	uint32 N= mi_N;
	mov ecx, N
	xor esi, esi;
	xor edx, edx;
	cmp ecx, 2;
	jb finished;
	dec ecx;
		#define K 1
		uint32 an = 0;
		uint32 N= mi_N;
		mov ecx, N
		xor esi, esi;
		mov edx, K;
		cmp ecx, 2;
		jb finished;
		mov eax, 2
		dec ecx;

Example 3-19. Alternative to Three-Operand LEA

3 operand LEA is slower	Two-operand LEA alternative	Alternative 2

loop1:
mov edi, esi;
lea esi, [K+esi+edx];
and esi, 0xFF;
mov edx, edi;
dec ecx;
jnz loop1;
finished:
mov [an] ,esi;
	loop1:
	mov edi, esi;
	lea esi, [K+edx];
	lea esi, [esi+edx];
	and esi, 0xFF;
	mov edx, edi;
	dec ecx;
	jnz loop1;
	finished:
	mov [an] ,esi;
		loop1:
		mov edi, esi;
		lea esi, [esi+edx];
		and esi, 0xFF;
		lea edx, [edi +K];
		dec ecx;
		jnz loop1;
		finished:
		mov [an] ,esi;

In some cases with processor based on Intel NetBurst microarchitecture, the LEA instruction or a sequence of LEA, ADD, SUB and SHIFT instructions can replace constant multiply instructions. The LEA instruction can also be used as a multiple operand addition instruction, for example:

```
LEA ECX, [EAX + EBX + 4 + A]
```

Using LEA in this way may avoid register usage by not tying up registers for operands of arithmetic instructions. This use may also save code space.

If the LEA instruction uses a shift by a constant amount then the latency of the sequence of μops is shorter if adds are used instead of a shift, and the LEA instruction may be replaced with an appropriate sequence of μops. This, however, increases the total number of μops, leading to a trade-off.

Assembly/Compiler Coding Rule 34. (ML impact, L generality) *If an LEA instruction using the scaled index is on the critical path, a sequence with ADDs may be better. If code density and bandwidth out of the trace cache are the critical factor, then use the LEA instruction.*

3.5.1.4 ADC and SBB in Intel® Microarchitecture Code Name Sandy Bridge

The throughput of ADC and SBB in Intel microarchitecture code name Sandy Bridge is 1 cycle, compared to 1.5-2 cycles in prior generation. These two instructions are useful in numeric handling of integer data types that are wider than the maximum width of native hardware.

Example 3-20. Examples of 512-bit Additions

//Add 64-bit to 512 Number	// 512-bit Addition

lea rsi, gLongCounter
lea rdi, gStepValue
mov rax, [rdi]
xor rcx, rcx
oop_start:
mov r10, [rsi+rcx]
add r10, rax
mov [rsi+rcx], r10
 	
mov r10, [rsi+rcx+8]
adc r10, 0
mov [rsi+rcx+8], r10
	loop1:
	mov rax, [StepValue]
	add rax, [LongCounter]
	mov LongCounter, rax
	mov rax, [StepValue+8]
	adc rax, [LongCounter+8]
	mov LongCounter+8, rax
	mov rax, [StepValue+16]
	adc rax, [LongCounter+16]

Example 3-20. Examples of 512-bit Additions (Contd.)

<pre>l   mov   r10, [rsi+rcx+16]
    adc   r10, 0
    mov   [rsi+rcx+16], r10
    mov   r10, [rsi+rcx+24]
    adc   r10, 0
    mov   [rsi+rcx+24], r10
 	
    mov   r10, [rsi+rcx+32]
    adc   r10, 0
    mov   [rsi+rcx+32], r10
 	
    mov r10, [rsi+rcx+40]
    adc r10, 0
    mov [rsi+rcx+40], r10
 	
mov r10, [rsi+rcx+48]
 adc r10, 0
 mov [rsi+rcx+48], r10
 	
 mov r10, [rsi+rcx+56]
 adc r10, 0
 mov [rsi+rcx+56], r10
 add rcx, 64
 cmp rcx, SIZE
 jnz loop_start</pre>
	<pre>mov LongCounter+16, rax
	mov rax, [StepValue+24]
	adc rax, [LongCounter+24]
	
	mov LongCounter+24, rax
	mov rax, [StepValue+32]
	adc rax, [LongCounter+32]
	
	mov LongCounter+32, rax
	mov rax, [StepValue+40]
	adc rax, [LongCounter+40]
	
	mov LongCounter+40, rax
	mov rax, [StepValue+48]
	adc rax, [LongCounter+48]
	
	mov LongCounter+48, rax
	mov rax, [StepValue+56]
	adc rax, [LongCounter+56]
	
	mov LongCounter+56, rax
	dec rcx
	jnz loop1</pre>

3.5.1.5 Bitwise Rotation

Bitwise rotation can choose between rotate with count specified in the CL register, an immediate constant and by 1 bit. Generally, The rotate by immediate and rotate by register instructions are slower than rotate by 1 bit. The rotate by 1 instruction has the same latency as a shift.

Assembly/Compiler Coding Rule 35. (ML impact, L generality) *Avoid ROTATE by register or ROTATE by immediate instructions. If possible, replace with a ROTATE by 1 instruction.*

In Intel microarchitecture code name Sandy Bridge, ROL/ROR by immediate has 1-cycle throughput, SHLD/SHRD using the same register as source and destination by an immediate constant has 1-cycle latency with 0.5 cycle throughput. The "ROL/ROR reg, imm8" instruction has two micro-ops with the latency of 1-cycle for the rotate register result and 2-cycles for the flags, if used.

In Intel microarchitecture code name Ivy Bridge, The "ROL/ROR reg, imm8" instruction with immediate greater than 1, is one micro-op with one-cycle latency when the overflow flag result is used. When the immediate is one, dependency on the overflow flag result of ROL/ROR by a subsequent instruction will see the ROL/ROR instruction with two-cycle latency.

3.5.1.6 Variable Bit Count Rotation and Shift

In Intel microarchitecture code name Sandy Bridge, The "ROL/ROR/SHL/SHR reg, cl" instruction has three micro-ops. When the flag result is not needed, one of these micro-ops may be discarded, providing better performance in many common usages. When these instructions update partial flag results that are subsequently used, the full three micro-ops flow must go through the execution and retirement pipeline, experiencing slower performance. In Intel microarchitecture code name Ivy Bridge, executing the full three micro-ops flow to use the updated partial flag result has additional delay. Consider the looped sequence below:

```
loop:
        shl eax, cl
        add ebx, eax
        dec edx ; DEC does not update carry, causing SHL to execute slower three micro-ops flow
        jnz loop
```

The DEC instruction does not modify the carry flag. Consequently, the SHL EAX, CL instruction needs to execute the three micro-ops flow in subsequent iterations. The SUB instruction will update all flags. So replacing DEC with SUB will allow SHL EAX, CL to execute the two micro-ops flow.

3.5.1.7 Address Calculations

For computing addresses, use the addressing modes rather than general-purpose computations. Internally, memory reference instructions can have four operands:

- Relocatable load-time constant.
- Immediate constant.
- Base register.
- Scaled index register.

Note that the latency and throughput of LEA with more than two operands are slower (see Section 3.5.1.3) in Intel microarchitecture code name Sandy Bridge. Addressing modes that uses both base and index registers will consume more read port resource in the execution engine and may experience more stalls due to availability of read port resources. Software should take care by selecting the speedy version of address calculation.

In the segmented model, a segment register may constitute an additional operand in the linear address calculation. In many cases, several integer instructions can be eliminated by fully using the operands of memory references.

3.5.1.8 Clearing Registers and Dependency Breaking Idioms

Code sequences that modifies partial register can experience some delay in its dependency chain, but can be avoided by using dependency breaking idioms.

In processors based on Intel Core microarchitecture, a number of instructions can help clear execution dependency when software uses these instruction to clear register content to zero. The instructions include:

```
XOR REG, REG
SUB REG, REG
XORPS/PD XMMREG, XMMREG
PXOR XMMREG, XMMREG
SUBPS/PD XMMREG, XMMREG
PSUBB/W/D/Q XMMREG, XMMREG
```

In processors based on Intel microarchitecture code name Sandy Bridge, the instruction listed above plus equivalent AVX counter parts are also zero idioms that can be used to break dependency chains. Furthermore, they do not consume an issue port or an execution unit. So using zero idioms are preferable than moving 0’s into the register. The AVX equivalent zero idioms are:

```
VXORPS/PD XMMREG, XMMREG
VXORPS/PD YMMREG, YMMREG
VPXOR XMMREG, XMMREG
VSUBPS/PD XMMREG, XMMREG
VSUBPS/PD YMMREG, YMMREG
VPSUBB/W/D/Q XMMREG, XMMREG
```

In Intel Core Solo and Intel Core Duo processors, the XOR, SUB, XORPS, or PXOR instructions can be used to clear execution dependencies on the zero evaluation of the destination register.

The Pentium 4 processor provides special support for XOR, SUB, and PXOR operations when executed within the same register. This recognizes that clearing a register does not depend on the old value of the register. The XORPS and XORPD instructions do not have this special support. They cannot be used to break dependence chains.

Assembly/Compiler Coding Rule 36. (M impact, ML generality) *Use dependency-breaking-idiom instructions to set a register to* 0*, or to break a false dependence chain resulting from re-use of registers. In contexts where the condition codes must be preserved, move* 0 *into the register instead. This requires more code space than using* XOR *and* SUB*, but avoids setting the condition codes.*

Example 3-21 of using pxor to break dependency idiom on a XMM register when performing negation on the elements of an array.

```
int a[4096], b[4096], c[4096];
For ( int i = 0; i < 4096; i++ )
        C[i] = - ( a[i] + b[i] );
```

Example 3-21. Clearing Register to Break Dependency While Negating Array Elements

Negation (-x = (x XOR (-1)) - (-1) without breaking dependency	Negation (-x = 0 -x) using PXOR reg, reg breaks dependency
Lea eax, a lea ecx, b lea edi, c xor edx, edx movdqa xmm7, allone lp:	lea eax, a lea ecx, b lea edi, c xor edx, edx lp:

movdqa xmm0, [eax + edx]
paddd xmm0, [ecx + edx]
pxor xmm0, xmm7
psubd xmm0, xmm7
movdqa [edi + edx], xmm0
add edx, 16
cmp edx, 4096
jl lp
	movdqa xmm0, [eax + edx]
	paddd xmm0, [ecx + edx]
	pxor xmm7, xmm7
	psubd xmm7, xmm0
	movdqa [edi + edx], xmm7
	add edx,16
	cmp edx, 4096
	jl lp

Assembly/Compiler Coding Rule 37. (M impact, MH generality) *Break dependences on portions of registers between instructions by operating on 32-bit registers instead of partial registers. For moves, this can be accomplished with 32-bit moves or by using MOVZX.*

Sometimes sign-extended semantics can be maintained by zero-extending operands. For example, the C code in the following statements does not need sign extension, nor does it need prefixes for operand size overrides:

```
static short INT a, b;
IF (a == b) {
  . . .
}
```

Code for comparing these 16-bit operands might be:

```
MOVZW  EAX, [a]
MOVZW  EBX, [b]
CMP    EAX, EBX
```

These circumstances tend to be common. However, the technique will not work if the compare is for greater than, less than, greater than or equal, and so on, or if the values in eax or ebx are to be used in another operation where sign extension is required.

Assembly/Compiler Coding Rule 38. (M impact, M generality) *Try to use zero extension or operate on 32-bit operands instead of using moves with sign extension.*

The trace cache can be packed more tightly when instructions with operands that can only be represented as 32 bits are not adjacent.

Assembly/Compiler Coding Rule 39. (ML impact, L generality) *Avoid placing instructions that use 32-bit immediates which cannot be encoded as sign-extended 16-bit immediates near each other. Try to schedule μops that have no immediate immediately before or after μops with 32-bit immediates.*

3.5.1.9 Compares

Use TEST when comparing a value in a register with zero. TEST essentially ANDs operands together without writing to a destination register. TEST is preferred over AND because AND produces an extra result register. TEST is better than CMP ..., 0 because the instruction size is smaller.

Use TEST when comparing the result of a logical AND with an immediate constant for equality or inequality if the register is EAX for cases such as:

```
IF (AVAR & 8) { }
```

The TEST instruction can also be used to detect rollover of modulo of a power of 2. For example, the C code:

```
IF ( (AVAR % 16) == 0 ) { }
```

can be implemented using:

```
TEST    EAX, 0x0F
JNZ     AfterIf
```

Using the TEST instruction between the instruction that may modify part of the flag register and the instruction that uses the flag register can also help prevent partial flag register stall.

Assembly/Compiler Coding Rule 40. (ML impact, M generality) *Use the* TEST *instruction instead of* AND *when the result of the logical AND is not used. This saves μops in execution. Use a TEST of a register with itself instead of a CMP of the register to zero, this saves the need to encode the zero and saves encoding space. Avoid comparing a constant to a memory operand. It is preferable to load the memory operand and compare the constant to a register.*

Often a produced value must be compared with zero, and then used in a branch. Because most Intel architecture instructions set the condition codes as part of their execution, the compare instruction may be eliminated. Thus the operation can be tested directly by a JCC instruction. The notable exceptions are MOV and LEA. In these cases, use TEST.

Assembly/Compiler Coding Rule 41. (ML impact, M generality) *Eliminate unnecessary compare with zero instructions by using the appropriate conditional jump instruction when the flags are already set by a preceding arithmetic instruction. If necessary, use a* TEST *instruction instead of a compare. Be certain that any code transformations made do not introduce problems with overflow.*

3.5.1.10 Using NOPs

Code generators generate a no-operation (NOP) to align instructions. Examples of NOPs of different lengths in 32-bit mode are shown below:

1-byte: XCHG EAX, EAX
2-byte: 66 NOP
3-byte: LEA REG, 0 (REG) (8-bit displacement)
4-byte: NOP DWORD PTR [EAX + 0] (8-bit displacement)
5-byte: NOP DWORD PTR [EAX + EAX*1 + 0] (8-bit displacement)
6-byte: LEA REG, 0 (REG) (32-bit displacement)
7-byte: NOP DWORD PTR [EAX + 0] (32-bit displacement)
8-byte: NOP DWORD PTR [EAX + EAX*1 + 0] (32-bit displacement)
9-byte: NOP WORD PTR [EAX + EAX*1 + 0] (32-bit displacement)

These are all true NOPs, having no effect on the state of the machine except to advance the EIP. Because NOPs require hardware resources to decode and execute, use the fewest number to achieve the desired padding.

The one byte NOP: [XCHG EAX,EAX] has special hardware support. Although it still consumes a µop and its accompanying resources, the dependence upon the old value of EAX is removed. This µop can be executed at the earliest possible opportunity, reducing the number of outstanding instructions and is the lowest cost NOP.

The other NOPs have no special hardware support. Their input and output registers are interpreted by the hardware. Therefore, a code generator should arrange to use the register containing the oldest value as input, so that the NOP will dispatch and release RS resources at the earliest possible opportunity.

Try to observe the following NOP generation priority:

- Select the smallest number of NOPs and pseudo-NOPs to provide the desired padding.
- Select NOPs that are least likely to execute on slower execution unit clusters.
- Select the register arguments of NOPs to reduce dependencies.

3.5.1.11 Mixing SIMD Data Types

Previous microarchitectures (before Intel Core microarchitecture) do not have explicit restrictions on mixing integer and floating-point (FP) operations on XMM registers. For Intel Core microarchitecture, mixing integer and floating-point operations on the content of an XMM register can degrade performance. Software should avoid mixed-use of integer/FP operation on XMM registers. Specifically:

- Use SIMD integer operations to feed SIMD integer operations. Use PXOR for idiom.
- Use SIMD floating-point operations to feed SIMD floating-point operations. Use XORPS for idiom.
- When floating-point operations are bitwise equivalent, use PS data type instead of PD data type. MOVAPS and MOVAPD do the same thing, but MOVAPS takes one less byte to encode the instruction.

3.5.1.12 Spill Scheduling

The spill scheduling algorithm used by a code generator will be impacted by the memory subsystem. A spill scheduling algorithm is an algorithm that selects what values to spill to memory when there are too many live values to fit in registers. Consider the code in Example 3-22, where it is necessary to spill either A, B, or C.

Example 3-22. Spill Scheduling Code

```
LOOP
    C := ...
    B := ...
    A := A + ...
```

For modern microarchitectures, using dependence depth information in spill scheduling is even more important than in previous processors. The loop-carried dependence in A makes it especially important that A not be spilled. Not only would a store/load be placed in the dependence chain, but there would also be a data-not-ready stall of the load, costing further cycles.

Assembly/Compiler Coding Rule 42. (H impact, MH generality) *For small loops, placing loop invariants in memory is better than spilling loop-carried dependencies.*

A possibly counter-intuitive result is that in such a situation it is better to put loop invariants in memory than in registers, since loop invariants never have a load blocked by store data that is not ready.

3.5.1.13 Zero-Latency MOV Instructions

In processors based on Intel microarchitecture code name Ivy Bridge, a subset of register-to-register move operations are executed in the front end (similar to zero-idioms, see Section 3.5.1.8). This

conserves scheduling/execution resources in the out-of-order engine. Most forms of register-to-register MOV instructions can benefit from zero-latency MOV. Example 3-23 list the details of those forms that qualify and a small set that do not.

Example 3-23. Zero-Latency MOV Instructions

MOV instructions latency that can be eliminated	MOV instructions latency that cannot be eliminated

MOV reg32, reg32
MOV reg64, reg64
MOVUPD/MOVAPD xmm, xmm
MOVUPD/MOVAPD ymm, ymm
MOVUPS?MOVAPS xmm, xmm
MOVUPS/MOVAPS ymm, ymm
MOVDQA/MOVDQU xmm, xmm
MOVDQA/MOVDQU ymm, ymm
MOVZX reg32, reg8 (if not AH/BH/CH/DH)
MOVZX reg64, reg8 (if not AH/BH/CH/DH)
	MOV reg8, reg8
	MOV reg16, reg16
	MOVZX reg32, reg8 (if AH/BH/CH/DH)
	MOVZX reg64, reg8 (if AH/BH/CH/DH)
	MOVSX

Example 3-24 shows how to process 8-bit integers using MOVZX to take advantage of zero-latency MOV enhancement. Consider

X = (X * 3^N) MOD 256;

Y = (Y * 3^N) MOD 256;

When "MOD 256" is implemented using the "AND 0xff" technique, its latency is exposed in the result-dependency chain. Using a form of MOVZX on a truncated byte input, it can take advantage of zero-latency MOV enhancement and gain about 45% in speed.

Example 3-24. Byte-Granular Data Computation Technique

Use AND Reg32, 0xff	Use MOVZX

mov rsi, N
mov rax, X
mov rcx, Y
loop:
lea rcx, [rcx+rcx*2]
lea rax, [rax+rax*4]
and rcx, 0xff
and rax, 0xff
 	
lea rcx, [rcx+rcx*2]
lea rax, [rax+rax*4]
and rcx, 0xff
and rax, 0xff
sub rsi, 2
jg loop
	mov rsi, N
	mov rax, X
	mov rcx, Y
	loop:
	lea rbx, [rcx+rcx*2]
	movzx, rcx, bl
	lea rbx, [rcx+rcx*2]
	movzx, rcx, bl
	
	lea rdx, [rax+rax*4]
	movzx, rax, dl
	llea rdx, [rax+rax*4]
	movzx, rax, dl
	sub rsi, 2
	jg loop

The effectiveness of coding a dense sequence of instructions to rely on a zero-latency MOV instruction must also consider internal resource constraints in the microarchitecture.

Example 3-25. Re-ordering Sequence to Improve Effectiveness of Zero-Latency MOV Instructions

Needing more internal resource for zero-latency MOVs	Needing less internal resource for zero-latency MOVs

mov rsi, N
mov rax, X
mov rcx, Y
	mov rsi, N
	mov rax, X
	mov rcx, Y

loop: lea rbx, [rcx+rcx*2] movzx, rcx, bl lea rdx, [rax+rax*4] movzx, rax, dl lea rbx, [rcx+rcx*2] movzx, rcx, bl llea rdx, [rax+rax*4] movzx, rax, dl sub rsi, 2 jg loop	loop: lea rbx, [rcx+rcx*2] movzx, rcx, bl lea rbx, [rcx+rcx*2] movzx, rcx, bl lea rdx, [rax+rax*4] movzx, rax, dl llea rdx, [rax+rax*4] movzx, rax, dl sub rsi, 2 jg loop

In Example 3-25, RBX/RCX and RDX/RAX are pairs of registers that are shared and continuously overwritten. In the right-hand sequence, registers are overwritten with new results immediately, consuming less internal resources provided by the underlying microarchitecture. As a result, it is about 8% faster than the left-hand sequence where internal resources could only support 50% of the attempt to take advantage of zero-latency MOV instructions.

3.5.2 Avoiding Stalls in Execution Core

Although the design of the execution core is optimized to make common cases executes quickly. A micro-op may encounter various hazards, delays, or stalls while making forward progress from the front end to the ROB and RS. The significant cases are:

- ROB Read Port Stalls.
- Partial Register Reference Stalls.
- Partial Updates to XMM Register Stalls.
- Partial Flag Register Reference Stalls.

3.5.2.1 ROB Read Port Stalls

As a micro-op is renamed, it determines whether its source operands have executed and been written to the reorder buffer (ROB), or whether they will be captured “in flight” in the RS or in the bypass network. Typically, the great majority of source operands are found to be “in flight” during renaming. Those that have been written back to the ROB are read through a set of read ports.

Since the Intel Core microarchitecture is optimized for the common case where the operands are “in flight”, it does not provide a full set of read ports to enable all renamed micro-ops to read all sources from the ROB in the same cycle.

When not all sources can be read, a micro-op can stall in the rename stage until it can get access to enough ROB read ports to complete renaming the micro-op. This stall is usually short-lived. Typically, a micro-op will complete renaming in the next cycle, but it appears to the application as a loss of rename bandwidth.

Some of the software-visible situations that can cause ROB read port stalls include:

- Registers that have become cold and require a ROB read port because execution units are doing other independent calculations.

- Constants inside registers.
- Pointer and index registers.

In rare cases, ROB read port stalls may lead to more significant performance degradations. There are a couple of heuristics that can help prevent over-subscribing the ROB read ports:

- Keep common register usage clustered together. Multiple references to the same written-back register can be "folded" inside the out of order execution core.
- Keep short dependency chains intact. This practice ensures that the registers will not have been written back when the new micro-ops are written to the RS.

These two scheduling heuristics may conflict with other more common scheduling heuristics. To reduce demand on the ROB read port, use these two heuristics only if both the following situations are met:

- Short latency operations.
- Indications of actual ROB read port stalls can be confirmed by measurements of the performance event (the relevant event is RAT_STALLS.ROB_READ_PORT, see Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*).

If the code has a long dependency chain, these two heuristics should not be used because they can cause the RS to fill, causing damage that outweighs the positive effects of reducing demands on the ROB read port.

Starting with Intel microarchitecture code name Sandy Bridge, ROB port stall no longer applies because data is read from the physical register file.

3.5.2.2 Writeback Bus Conflicts

The writeback bus inside the execution engine is a common resource needed to facilitate out-of-order execution of micro-ops in flight. When the writeback bus is needed at the same time by two micro-ops executing in the same stack of execution units (see Table 2-15), the younger micro-op will have to wait for the writeback bus to be available. This situation typically will be more likely for short-latency instructions experience a delay when it might have been otherwise ready for dispatching into the execution engine.

Consider a repeating sequence of independent floating-point ADDs with a single-cycle MOV bound to the same dispatch port. When the MOV finds the dispatch port available, the writeback bus can be occupied by the ADD. This delays the MOV operation.

If this problem is detected, you can sometimes change the instruction selection to use a different dispatch port and reduce the writeback contention.

3.5.2.3 Bypass between Execution Domains

Floating-point (FP) loads have an extra cycle of latency. Moves between FP and SIMD stacks have another additional cycle of latency.

Example:

```
ADDPS  XMM0, XMM1
PAND  XMM0, XMM3
ADDPS  XMM2, XMM0
```

The overall latency for the above calculation is 9 cycles:

- 3 cycles for each ADDPS instruction.
- 1 cycle for the PAND instruction.
- 1 cycle to bypass between the ADDPS floating-point domain to the PAND integer domain.
- 1 cycle to move the data from the PAND integer to the second floating-point ADDPS domain.

To avoid this penalty, you should organize code to minimize domain changes. Sometimes you cannot avoid bypasses.

Account for bypass cycles when counting the overall latency of your code. If your calculation is latency-bound, you can execute more instructions in parallel or break dependency chains to reduce total latency.

Code that has many bypass domains and is completely latency-bound may run slower on the Intel Core microarchitecture than it did on previous microarchitectures.

3.5.2.4 Partial Register Stalls

General purpose registers can be accessed in granularities of bytes, words, doublewords; 64-bit mode also supports quadword granularity. Referencing a portion of a register is referred to as a partial register reference.

A partial register stall happens when an instruction refers to a register, portions of which were previously modified by other instructions. For example, partial register stalls occurs with a read to AX while previous instructions stored AL and AH, or a read to EAX while previous instruction modified AX.

The delay of a partial register stall is small in processors based on Intel Core and NetBurst microarchitectures, and in Pentium M processor (with CPUID signature family 6, model 13), Intel Core Solo, and Intel Core Duo processors. Pentium M processors (CPUID signature with family 6, model 9) and the P6 family incur a large penalty.

Note that in Intel 64 architecture, an update to the lower 32 bits of a 64 bit integer register is architecturally defined to zero extend the upper 32 bits. While this action may be logically viewed as a 32 bit update, it is really a 64 bit update (and therefore does not cause a partial stall).

Referencing partial registers frequently produces code sequences with either false or real dependencies. Example 3-18 demonstrates a series of false and real dependencies caused by referencing partial registers.

If instructions 4 and 6 (in Example 3-18) are changed to use a movzx instruction instead of a mov, then the dependences of instruction 4 on 2 (and transitively 1 before it), and instruction 6 on 5 are broken. This creates two independent chains of computation instead of one serial one.

Example 3-26 illustrates the use of MOVZX to avoid a partial register stall when packing three byte values into a register.

Example 3-26. Avoiding Partial Register Stalls in Integer Code

A Sequence Causing Partial Register Stall	Alternate Sequence Using MOVZX to Avoid Delay
mov al, byte ptr a[2] shl eax,16 mov ax, word ptr a movd mm0, eax ret	movzx eax, byte ptr a[2] shl eax, 16 movzx ecx, word ptr a or eax,ecx movd mm0, eax ret

In Intel microarchitecture code name Sandy Bridge, partial register access is handled in hardware by inserting a micro-op that merges the partial register with the full register in the following cases:

- After a write to one of the registers AH, BH, CH or DH and before a following read of the 2-, 4- or 8-byte form of the same register. In these cases a merge micro-op is inserted. The insertion consumes a full allocation cycle in which other micro-ops cannot be allocated.
- After a micro-op with a destination register of 1 or 2 bytes, which is not a source of the instruction (or the register's bigger form), and before a following read of a 2-,4- or 8-byte form of the same register. In these cases the merge micro-op is part of the flow. For example:
  - MOV AX, [BX]

    When you want to load from memory to a partial register, consider using MOVZX or MOVSX to avoid the additional merge micro-op penalty.
  - LEA AX, [BX+CX]

For optimal performance, use of zero idioms, before the use of the register, eliminates the need for partial register merge micro-ops.

3.5.2.5 Partial XMM Register Stalls

Partial register stalls can also apply to XMM registers. The following SSE and SSE2 instructions update only part of the destination register:

```
MOVL/HPD XMM, MEM64
MOVL/HPS XMM, MEM32
MOVSS/SD between registers
```

Using these instructions creates a dependency chain between the unmodified part of the register and the modified part of the register. This dependency chain can cause performance loss.

Example 3-27 illustrates the use of MOVZX to avoid a partial register stall when packing three byte values into a register.

Follow these recommendations to avoid stalls from partial updates to XMM registers:

- Avoid using instructions which update only part of the XMM register.
- If a 64-bit load is needed, use the MOVSD or MOVQ instruction.
- If 2 64-bit loads are required to the same register from non continuous locations, use MOVSD/MOVHPD instead of MOVLPD/MOVHPD.
- When copying the XMM register, use the following instructions for full register copy, even if you only want to copy some of the source register data:

```
MOVAPS
MOVAPD
MOVDQA
```

Example 3-27. Avoiding Partial Register Stalls in SIMD Code

Using movlpd for memory transactions and movsd between register copies Causing Partial Register Stall	Using movsd for memory and movapd between register copies Avoid Delay

mov edx, x
mov ecx, count
movlpd xmm3,_1_
movlpd xmm2,_1pt5_
align 16
lp:
movlpd xmm0, [edx]
addsd xmm0, xmm3
movsd xmm1, xmm2
subsd xmm1, [edx]
mulsd xmm0, xmm1
movsd [edx], xmm0
add edx, 8
dec ecx
jnz lp
	mov edx, x
	mov ecx, count
	movsd xmm3,_1_
	movsd xmm2, _1pt5_
	align 16
	lp:
	movsd xmm0, [edx]
	addsd xmm0, xmm3
	movapd xmm1, xmm2
	subsd xmm1, [edx]
	mulsd xmm0, xmm1
	movsd [edx], xmm0
	add edx, 8
	dec ecx
	jnz lp

3.5.2.6 Partial Flag Register Stalls

A "partial flag register stall" occurs when an instruction modifies a part of the flag register and the following instruction is dependent on the outcome of the flags. This happens most often with shift instructions (SAR, SAL, SHR, SHL). The flags are not modified in the case of a zero shift count, but the shift count is usually known only at execution time. The front end stalls until the instruction is retired.

Other instructions that can modify some part of the flag register include CMPXCHG8B, various rotate instructions, STC, and STD. An example of assembly with a partial flag register stall and alternative code without the stall is shown in Example 3-28.

In processors based on Intel Core microarchitecture, shift immediate by 1 is handled by special hardware such that it does not experience partial flag stall.

Example 3-28. Avoiding Partial Flag Register Stalls

Partial Flag Register Stall	Avoiding Partial Flag Register Stall
xor eax, eax mov ecx, a sar ecx, 2 setz al ;SAR can update carry causing a stall	or eax, eax mov ecx, a sar ecx, 2 test ecx, ecx ; test always updates all flags setz al ;No partial reg or flag stall,

In Intel microarchitecture code name Sandy Bridge, the cost of partial flag access is replaced by the insertion of a micro-op instead of a stall. However, it is still recommended to use less of instructions that write only to some of the flags (such as INC, DEC, SET CL) before instructions that can write flags conditionally (such as SHIFT CL).

Example 3-29 compares two techniques to implement the addition of very large integers (e.g. 1024 bits). The alternative sequence on the right side of Example 3-29 will be faster than the left side on Intel microarchitecture code name Sandy Bridge, but it will experience partial flag stalls on prior microarchitectures.

Example 3-29. Partial Flag Register Accesses in Intel Microarchitecture Code Name Sandy Bridge

Save partial flag register to avoid stall	Simplified code sequence

lea rsi, [A]
lea rdi, [B]
xor rax, rax
mov rcx, 16 ; 16*64 =1024 bit
 	
lp_64bit:
add rax, [rsi]
adc rax, [rdi]
mov [rdi], rax
setc al ;save carry for next iteration
movzx rax, al
add rsi, 8
add rdi, 8
dec rcx
jnz lp_64bit
	lea rsi, [A]
	lea rdi, [B]
	xor rax, rax
	mov rcx, 16
	
	lp_64bit:
	add rax, [rsi]
	adc rax, [rdi]
	mov [rdi], rax
	lea rsi, [rsi+8]
	lea rdi, [rdi+8]
	dec rcx
	jnz lp_64bit

3.5.2.7 Floating-Point/SIMD Operands

Moves that write a portion of a register can introduce unwanted dependences. The MOVSD REG, REG instruction writes only the bottom 64 bits of a register, not all 128 bits. This introduces a dependence on the preceding instruction that produces the upper 64 bits (even if those bits are not longer wanted). The dependence inhibits register renaming, and thereby reduces parallelism.

Use MOVAPD as an alternative; it writes all 128 bits. Even though this instruction has a longer latency, the μops for MOVAPD use a different execution port and this port is more likely to be free. The change can impact performance. There may be exceptional cases where the latency matters more than the dependence or the execution port.

Assembly/Compiler Coding Rule 43. (M impact, ML generality) *Avoid introducing dependences with partial floating-point register writes, e.g. from the* MOVSD XMMREG1, XMMREG2 *instruction. Use the* MOVAPD XMMREG1, XMMREG2 *instruction instead.*

The MOVSD XMMREG, MEM instruction writes all 128 bits and breaks a dependence.

The MOVUPD from memory instruction performs two 64-bit loads, but requires additional µops to adjust the address and combine the loads into a single register. This same functionality can be obtained using MOVSD XMMREG1, MEM; MOVSD XMMREG2, MEM+8; UNPCKLPD XMMREG1, XMMREG2, which uses fewer µops and can be packed into the trace cache more effectively. The latter alternative has been found to provide a several percent performance improvement in some cases. Its encoding requires more instruction bytes, but this is seldom an issue for the Pentium 4 processor. The store version of MOVUPD is complex and slow, so much so that the sequence with two MOVSD and a UNPCKHPD should always be used.

Assembly/Compiler Coding Rule 44. (ML impact, L generality) *Instead of using* MOVUPD XMMREG1, MEM *for a unaligned 128-bit load, use* MOVSD XMMREG1, MEM; MOVSD XMMREG2, MEM+8; UNPCKLPD XMMREG1, XMMREG2*. If the additional register is not available, then use* MOVSD XMMREG1, MEM; MOVHPD XMMREG1, MEM+8.

Assembly/Compiler Coding Rule 45. (M impact, ML generality) *Instead of using* MOVUPD MEM, XMMREG1 *for a store, use* MOVSD MEM, XMMREG1; UNPCKHPD XMMREG1, XMMREG1; MOVSD MEM+8, XMMREG1 *instead.*

3.5.3 Vectorization

This section provides a brief summary of optimization issues related to vectorization. There is more detail in the chapters that follow.

Vectorization is a program transformation that allows special hardware to perform the same operation on multiple data elements at the same time. Successive processor generations have provided vector support through the MMX technology, Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3) and Supplemental Streaming SIMD Extensions 3 (SSSE3).

Vectorization is a special case of SIMD, a term defined in Flynn's architecture taxonomy to denote a single instruction stream capable of operating on multiple data elements in parallel. The number of elements which can be operated on in parallel range from four single-precision floating-point data elements in Streaming SIMD Extensions and two double-precision floating-point data elements in Streaming SIMD Extensions 2 to sixteen byte operations in a 128-bit register in Streaming SIMD Extensions 2. Thus, vector length ranges from 2 to 16, depending on the instruction extensions used and on the data type.

The Intel C++ Compiler supports vectorization in three ways:

- The compiler may be able to generate SIMD code without intervention from the user.
- The can user insert pragmas to help the compiler realize that it can vectorize the code.
- The user can write SIMD code explicitly using intrinsics and C++ classes.

To help enable the compiler to generate SIMD code, avoid global pointers and global variables. These issues may be less troublesome if all modules are compiled simultaneously, and whole-program optimization is used.

User/Source Coding Rule 2. (H impact, M generality) *Use the smallest possible floating-point or SIMD data type, to enable more parallelism with the use of a (longer) SIMD vector. For example, use single precision instead of double precision where possible.*

User/Source Coding Rule 3. (M impact, ML generality) *Arrange the nesting of loops so that the innermost nesting level is free of inter-iteration dependencies. Especially avoid the case where the store of data in an earlier iteration happens lexically after the load of that data in a future iteration, something which is called a lexically backward dependence.*

The integer part of the SIMD instruction set extensions cover 8-bit,16-bit and 32-bit operands. Not all SIMD operations are supported for 32 bits, meaning that some source code will not be able to be vectorized at all unless smaller operands are used.

User/Source Coding Rule 4. (M impact, ML generality) *Avoid the use of conditional branches inside loops and consider using SSE instructions to eliminate branches.*

User/Source Coding Rule 5. (M impact, ML generality) *Keep induction (loop) variable expressions simple.*

3.5.4 Optimization of Partially Vectorizable Code

Frequently, a program contains a mixture of vectorizable code and some routines that are non-vectorizable. A common situation of partially vectorizable code involves a loop structure which include mixtures of vectorized code and unvectorizable code. This situation is depicted in Figure 3-1.

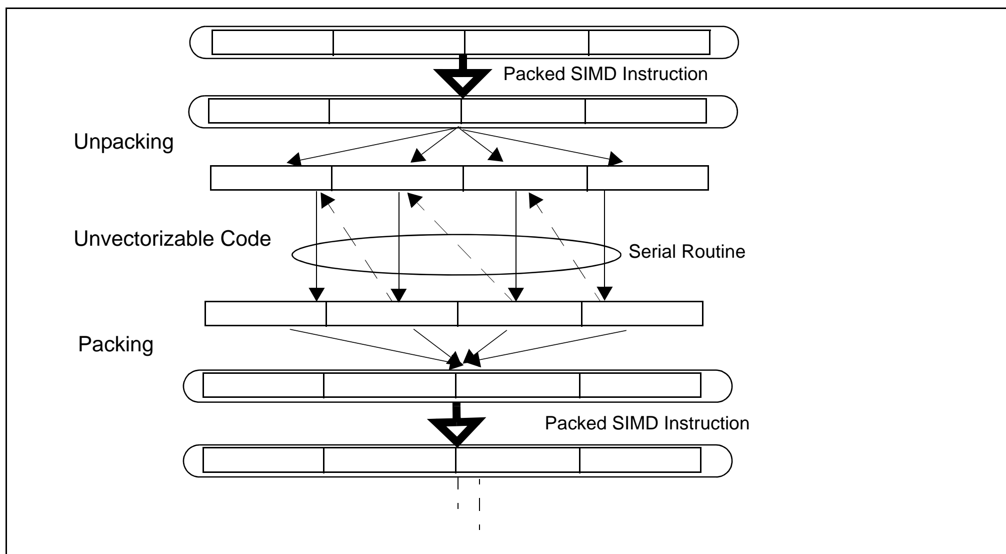


Figure 3-1. Generic Program Flow of Partially Vectorized Code

It generally consists of five stages within the loop:

- Prolog.
- Unpacking vectorized data structure into individual elements.
- Calling a non-vectorizable routine to process each element serially.
- Packing individual result into vectorized data structure.
- Epilog.

This section discusses techniques that can reduce the cost and bottleneck associated with the packing/unpacking stages in these partially vectorize code.

Example 3-30 shows a reference code template that is representative of partially vectorizable coding situations that also experience performance issues. The unvectorizable portion of code is represented generically by a sequence of calling a serial function named "foo" multiple times. This generic example is referred to as "shuffle with store forwarding", because the problem generally involves an unpacking stage that shuffles data elements between register and memory, followed by a packing stage that can experience store forwarding issue.

There are more than one useful techniques that can reduce the store-forwarding bottleneck between the serialized portion and the packing stage. The following sub-sections presents alternate techniques to deal with the packing, unpacking, and parameter passing to serialized function calls.

Example 3-30. Reference Code Template for Partially Vectorizable Program

```
// Prolog ///////////////////////////////
push ebp
mov ebp, esp

// Unpacking ////////////////////////////
sub ebp, 32
and ebp, 0xfffffff0
movaps [ebp], xmm0
```

Example 3-30. Reference Code Template for Partially Vectorizable Program (Contd.)

```
// Serial operations on components ///////
sub ebp, 4

mov eax, [ebp+4]
mov [ebp], eax
call foo
mov [ebp+16+4], eax

mov eax, [ebp+8]
mov [ebp], eax
call foo
mov [ebp+16+4+4], eax


mov eax, [ebp+12]
mov [ebp], eax
call foo
mov [ebp+16+8+4], eax


mov eax, [ebp+12+4]
mov [ebp], eax
call foo
mov [ebp+16+12+4], eax


// Packing ///////////////////////////////
movaps xmm0, [ebp+16+4]

// Epilog ////////////////////////////////
pop ebp
ret
```

3.5.4.1 Alternate Packing Techniques

The packing method implemented in the reference code of Example 3-30 will experience delay as it assembles 4 doubleword result from memory into an XMM register due to store-forwarding restrictions.

Three alternate techniques for packing, using different SIMD instruction to assemble contents in XMM registers are shown in Example 3-31. All three techniques avoid store-forwarding delay by satisfying the restrictions on data sizes between a preceding store and subsequent load operations.

Example 3-31. Three Alternate Packing Methods for Avoiding Store Forwarding Difficulty

Packing Method 1	Packing Method 2	Packing Method 3

movd xmm0, [ebp+16+4]
movd xmm1, [ebp+16+8]
movd xmm2, [ebp+16+12]
movd xmm3, [ebp+12+16+4]
punpckldq xmm0, xmm1
punpckldq xmm2, xmm3
punpckldq xmm0, xmm2
	movd xmm0, [ebp+16+4]
	movd xmm1, [ebp+16+8]
	movd xmm2, [ebp+16+12]
	movd xmm3, [ebp+12+16+4]
	psllq xmm3, 32
	orps xmm2, xmm3
	psllq xmm1, 32
	orps xmm0, xmm1movlhps xmm0, xmm2
		movd xmm0, [ebp+16+4]
		movd xmm1, [ebp+16+8]
		movd xmm2, [ebp+16+12]
		movd xmm3, [ebp+12+16+4]
		movlhps xmm1,xmm3
		psllq xmm1, 32
		movlhps xmm0, xmm2
		orps xmm0, xmm1

3.5.4.2 Simplifying Result Passing

In Example 3-30, individual results were passed to the packing stage by storing to contiguous memory locations. Instead of using memory spills to pass four results, result passing may be accomplished by using either one or more registers. Using registers to simplify result passing and reduce memory spills can improve performance by varying degrees depending on the register pressure at runtime.

Example 3-32 shows the coding sequence that uses four extra XMM registers to reduce all memory spills of passing results back to the parent routine. However, software must observe the following conditions when using this technique:

- There is no register shortage.
- If the loop does not have many stores or loads but has many computations, this technique does not help performance. This technique adds work to the computational units, while the store and loads ports are idle.

Example 3-32. Using Four Registers to Reduce Memory Spills and Simplify Result Passing

```
mov eax, [ebp+4]
mov [ebp], eax
call foo
movd xmm0, eax

mov eax, [ebp+8]
mov [ebp], eax
call foo
movd xmm1, eax


mov eax, [ebp+12]
mov [ebp], eax
call foo
movd xmm2, eax

mov eax, [ebp+12+4]
mov [ebp], eax
call foo
movd xmm3, eax
```

3.5.4.3 Stack Optimization

In Example 3-30, an input parameter was copied in turn onto the stack and passed to the non-vectorizable routine for processing. The parameter passing from consecutive memory locations can be simplified by a technique shown in Example 3-33.

Example 3-33. Stack Optimization Technique to Simplify Parameter Passing

```
call foo
mov [ebp+16], eax

add ebp, 4
call foo
mov [ebp+16], eax
```

Example 3-33. Stack Optimization Technique to Simplify Parameter Passing (Contd.)

```
add ebp, 4
call foo
mov [ebp+16], eax

add ebp, 4
call foo
```

Stack Optimization can only be used when:

- The serial operations are function calls. The function "foo" is declared as: INT FOO(INT A). The parameter is passed on the stack.
- The order of operation on the components is from last to first.

Note the call to FOO and the advance of EDP when passing the vector elements to FOO one by one from last to first.

3.5.4.4 Tuning Considerations

Tuning considerations for situations represented by looping of Example 3-30 include:

- Applying one of more of the following combinations:
  - Choose an alternate packing technique.
  - Consider a technique to simply result-passing.
  - Consider the stack optimization technique to simplify parameter passing.
- Minimizing the average number of cycles to execute one iteration of the loop.
- Minimizing the per-iteration cost of the unpacking and packing operations.

The speed improvement by using the techniques discussed in this section will vary, depending on the choice of combinations implemented and characteristics of the non-vectorizable routine. For example, if the routine "foo" is short (representative of tight, short loops), the per-iteration cost of unpacking/packing tend to be smaller than situations where the non-vectorizable code contain longer operation or many dependencies. This is because many iterations of short, tight loop can be in flight in the execution core, so the per-iteration cost of packing and unpacking is only partially exposed and appear to cause very little performance degradation.

Evaluation of the per-iteration cost of packing/unpacking should be carried out in a methodical manner over a selected number of test cases, where each case may implement some combination of the techniques discussed in this section. The per-iteration cost can be estimated by:

- Evaluating the average cycles to execute one iteration of the test case.
- Evaluating the average cycles to execute one iteration of a base line loop sequence of non-vectorizable code.

Example 3-34 shows the base line code sequence that can be used to estimate the average cost of a loop that executes non-vectorizable routines.

Example 3-34. Base Line Code Sequence to Estimate Loop Overhead

```
push ebp
mov ebp, esp
sub ebp, 4

mov [ebp], edi
call foo
```

Example 3-34. Base Line Code Sequence to Estimate Loop Overhead (Contd.)

```
mov [ebp], edi
call foo

mov [ebp], edi
call foo


mov [ebp], edi
call foo

add ebp, 4
pop ebp
ret
```

The average per-iteration cost of packing/unpacking can be derived from measuring the execution times of a large number of iterations by:

((Cycles to run TestCase) - (Cycles to run equivalent baseline sequence)) / (Iteration count).

For example, using a simple function that returns an input parameter (representative of tight, short loops), the per-iteration cost of packing/unpacking may range from slightly more than 7 cycles (the shuffle with store forwarding case, Example 3-30) to ~0.9 cycles (accomplished by several test cases). Across 27 test cases (consisting of one of the alternate packing methods, no result-simplification/simplification of either 1 or 4 results, no stack optimization or with stack optimization), the average per-iteration cost of packing/unpacking is about 1.7 cycles.

Generally speaking, packing method 2 and 3 (see Example 3-31) tend to be more robust than packing method 1; the optimal choice of simplifying 1 or 4 results will be affected by register pressure of the runtime and other relevant microarchitectural conditions.

Note that the numeric discussion of per-iteration cost of packing/packing is illustrative only. It will vary with test cases using a different base line code sequence and will generally increase if the non-vectorizable routine requires longer time to execute because the number of loop iterations that can reside in flight in the execution core decreases.

3.6 OPTIMIZING MEMORY ACCESSES

This section discusses guidelines for optimizing code and data memory accesses. The most important recommendations are:

- Execute load and store operations within available execution bandwidth.
- Enable forward progress of speculative execution.
- Enable store forwarding to proceed.
- Align data, paying attention to data layout and stack alignment.
- Place code and data on separate pages.
- Enhance data locality.
- Use prefetching and cacheability control instructions.
- Enhance code locality and align branch targets.
- Take advantage of write combining.

Alignment and forwarding problems are among the most common sources of large delays on processors based on Intel NetBurst microarchitecture.

3.6.1 Load and Store Execution Bandwidth

Typically, loads and stores are the most frequent operations in a workload, up to 40% of the instructions in a workload carrying load or store intent are not uncommon. Each generation of microarchitecture provides multiple buffers to support executing load and store operations while there are instructions in flight.

Software can maximize memory performance by not exceeding the issue or buffering limitations of the machine. In the Intel Core microarchitecture, only 20 stores and 32 loads may be in flight at once. In Intel microarchitecture code name Nehalem, there are 32 store buffers and 48 load buffers. Since only one load can issue per cycle, algorithms which operate on two arrays are constrained to one operation every other cycle unless you use programming tricks to reduce the amount of memory usage.

Intel Core Duo and Intel Core Solo processors have less buffers. Nevertheless the general heuristic applies to all of them.

3.6.1.1 Make Use of Load Bandwidth in Intel® Microarchitecture Code Name Sandy Bridge

While prior microarchitecture has one load port (port 2), Intel microarchitecture code name Sandy Bridge can load from port 2 and port 3. Thus two load operations can be performed every cycle and doubling the load throughput of the code. This improves code that reads a lot of data and does not need to write out results to memory very often (Port 3 also handles store-address operation). To exploit this bandwidth, the data has to stay in the L1 data cache or it should be accessed sequentially, enabling the hardware prefetchers to bring the data to the L1 data cache in time.

Consider the following C code example of adding all the elements of an array:

```
int buff[BUFF_SIZE];

int sum = 0;


for (i=0;i<BUFF_SIZE;i++){

        sum+=buff[i];

}
```

Alternative 1 is the assembly code generated by the Intel compiler for this C code, using the optimization flag for Intel microarchitecture code name Nehalem. The compiler vectorizes execution using Intel SSE instructions. In this code, each ADD operation uses the result of the previous ADD operation. This limits the throughput to one load and ADD operation per cycle. Alternative 2 is optimized for Intel microarchitecture code name Sandy Bridge by enabling it to use the additional load bandwidth. The code removes the dependency among ADD operations, by using two registers to sum the array values. Two load and two ADD operations can be executed every cycle.

Example 3-35. Optimize for Load Port Bandwidth in Intel Microarchitecture Code Name Sandy Bridge

Register dependency inhibits PADD execution	Reduce register dependency allow two load port to supply PADD execution
xor eax, eax pxor xmm0, xmm0 lea rsi, buff	xor eax, eax pxor xmm0, xmm0 pxor xmm1, xmm1 lea rsi, buff

Example 3-35. Optimize for Load Port Bandwidth in Intel Microarchitecture Code Name Sandy Bridge (Contd.)

Register dependency inhibits PADD execution	Reduce register dependency allow two load port to supply PADD execution

loop_start:
paddd xmm0, [rsi+4*rax]
paddd xmm0, [rsi+4*rax+16]
paddd xmm0, [rsi+4*rax+32]
paddd xmm0, [rsi+4*rax+48]
paddd xmm0, [rsi+4*rax+64]
paddd xmm0, [rsi+4*rax+80]
paddd xmm0, [rsi+4*rax+96]
paddd xmm0, [rsi+4*rax+112]
add eax, 32
cmp eax, BUFF_SIZE
jl loop_start
sum_partials:
movdqa xmm1, xmm0
psrldq xmm1, 8
paddd xmm0, xmm1
movdqa xmm2, xmm0
psrldq xmm2, 4
paddd xmm0, xmm2
movd [sum], xmm0
	loop_start:
	paddd xmm0, [rsi+4*rax]
	paddd xmm1, [rsi+4*rax+16]
	paddd xmm0, [rsi+4*rax+32]
	paddd xmm1, [rsi+4*rax+48]
	paddd xmm0, [rsi+4*rax+64]
	paddd xmm1, [rsi+4*rax+80]
	paddd xmm0, [rsi+4*rax+96]
	paddd xmm1, [rsi+4*rax+112]
	add eax, 32
	cmp eax, BUFF_SIZE
	jl loop_start
	sum_partials:
	paddd xmm0, xmm1
	movdqa xmm1, xmm0
	psrldq xmm1, 8
	paddd xmm0, xmm1
	movdqa xmm2, xmm0
	psrldq xmm2, 4
	paddd xmm0, xmm2
	movd [sum], xmm0

3.6.1.2 L1D Cache Latency in Intel® Microarchitecture Code Name Sandy Bridge

Load latency from L1D cache may vary (see Table 2-19). The best case if 4 cycles, which apply to load operations to general purpose registers using one of the following:

- One register.
- A base register plus an offset that is smaller than 2048.

Consider the pointer-chasing code example in Example 3-36.

Example 3-36. Index versus Pointers in Pointer-Chasing Code

Traversing through indexes	Traversing through pointers

// C code example
index = buffer.m_buff[index].next_index;
// ASM example
loop:
shl rbx, 6
mov rbx, 0x20(rbx+rcx)
dec rax
cmp rax, -1
jne loop
	// C code example
	node = node->pNext;
	// ASM example
	loop:
	mov rdx, [rdx]
	dec rax
	cmp rax, -1
	jne loop

The left side implements pointer chasing via traversing an index. Compiler then generates the code shown below addressing memory using base+index with an offset. The right side shows compiler generated code from pointer de-referencing code and uses only a base register.

The code on the right side is faster than the left side across Intel microarchitecture code name Sandy Bridge and prior microarchitecture. However the code that traverses index will be slower on Intel microarchitecture code name Sandy Bridge relative to prior microarchitecture.

3.6.1.3 Handling L1D Cache Bank Conflict

In Intel microarchitecture code name Sandy Bridge, the internal organization of the L1D cache may manifest a situation when two load micro-ops whose addresses have a bank conflict. When a bank conflict is present between two load operations, the more recent one will be delayed until the conflict is resolved. A bank conflict happens when two simultaneous load operations have the same bit 2-5 of their linear address but they are not from the same set in the cache (bits 6 - 12).

Bank conflicts should be handled only if the code is bound by load bandwidth. Some bank conflicts do not cause any performance degradation since they are hidden by other performance limiters. Eliminating such bank conflicts does not improve performance.

The following example demonstrates bank conflict and how to modify the code and avoid them. It uses two source arrays with a size that is a multiple of cache line size. When loading an element from A and the counterpart element from B the elements have the same offset in their cache lines and therefore a bank conflict may happen.

With the Haswell microarchitecture, the L1 DCache bank conflict issue does not apply.

.

Example 3-37. Example of Bank Conflicts in L1D Cache and Remedy

```
int A[128];
int B[128];
int C[128];
for (i=0;i<128;i+=4){
    C[i]=A[i]+B[i];       the loads from A[i] and B[i] collide
    C[i+1]=A[i+1]+B[i+1];
    C[i+2]=A[i+2]+B[i+2];
    C[i+3]=A[i+3]+B[i+3];
}
```

// Code with Bank Conflicts	// Code without Bank Conflicts

xor rcx, rcx
lea r11, A
lea r12, B
lea r13, C
loop:
lea esi, [rcx*4]
movsxd rsi, esi
mov edi, [r11+rsi*4]
add edi, [r12+rsi*4]
mov r8d, [r11+rsi*4+4]
add r8d, [r12+rsi*4+4]
mov r9d, [r11+rsi*4+8]
add r9d, [r12+rsi*4+8]
mov r10d, [r11+rsi*4+12]
add r10d, [r12+rsi*4+12]
	xor rcx, rcx
	lea r11, A
	lea r12, B
	lea r13, C
	loop:
	lea esi, [rcx*4]
	movsxd rsi, esi
	mov edi, [r11+rsi*4]
	mov r8d, [r11+rsi*4+4]
	add edi, [r12+rsi*4]
	add r8d, [r12+rsi*4+4]
	mov r9d, [r11+rsi*4+8]
	mov r10d, [r11+rsi*4+12]
	add r9d, [r12+rsi*4+8]
	add r10d, [r12+rsi*4+12]

Example 3-37. Example of Bank Conflicts in L1D Cache and Remedy (Contd.)

mov [r13+rsi*4], edi
inc ecx
mov [r13+rsi*4+4], r8d
mov [r13+rsi*4+8], r9d
mov [r13+rsi*4+12], r10d
cmp ecx, LEN
jb loop
	inc ecx
	mov [r13+rsi*4], edi
	mov [r13+rsi*4+4], r8d
	mov [r13+rsi*4+8], r9d
	mov [r13+rsi*4+12], r10d
	cmp ecx, LEN
	jb loop

3.6.2 Minimize Register Spills

When a piece of code has more live variables than the processor can keep in general purpose registers, a common method is to hold some of the variables in memory. This method is called register spill. The effect of L1D cache latency can negatively affect the performance of this code. The effect can be more pronounced if the address of register spills uses the slower addressing modes.

One option is to spill general purpose registers to XMM registers. This method is likely to improve performance also on previous processor generations. The following example shows how to spill a register to an XMM register rather than to memory.

Example 3-38. Using XMM Register in Lieu of Memory for Register Spills

Register spills into memory	Register spills into XMM

loop:
mov rdx, [rsp+0x18]
movdqa xmm0, [rdx]
movdqa xmm1, [rsp+0x20]
pcmpeqd xmm1, xmm0
pmovmskb eax, xmm1
test eax, eax
jne end_loop
movzx rcx, [rbx+0x60]
 	
add qword ptr[rsp+0x18], 0x10
add rdi, 0x4
movzx rdx, di
sub rcx, 0x4
add rsi, 0x1d0
cmp rdx, rcx
jle loop
	movq xmm4, [rsp+0x18]
	mov rcx, 0x10
	movq xmm5, rcx
	loop:
	movq rdx, xmm4
	movdqa xmm0, [rdx]
	movdqa xmm1, [rsp+0x20]
	pcmpeqd xmm1, xmm0
	pmovmskb eax, xmm1
	test eax, eax
	jne end_loop
	movzx rcx, [rbx+0x60]
	
	padd xmm4, xmm5
	add rdi, 0x4
	movzx rdx, di
	sub rcx, 0x4
	add rsi, 0x1d0
	cmp rdx, rcx
	jle loop

3.6.3 Enhance Speculative Execution and Memory Disambiguation

Prior to Intel Core microarchitecture, when code contains both stores and loads, the loads cannot be issued before the address of the store is resolved. This rule ensures correct handling of load dependencies on preceding stores.

The Intel Core microarchitecture contains a mechanism that allows some loads to be issued early speculatively. The processor later checks if the load address overlaps with a store. If the addresses do overlap, then the processor re-executes the instructions.

Example 3-39 illustrates a situation that the compiler cannot be sure that "Ptr->Array" does not change during the loop. Therefore, the compiler cannot keep "Ptr->Array" in a register as an invariant and must read it again in every iteration. Although this situation can be fixed in software by a rewriting the code to require the address of the pointer is invariant, memory disambiguation provides performance gain without rewriting the code.

Example 3-39. Loads Blocked by Stores of Unknown Address

C code	Assembly sequence

struct AA {
AA ** array;
};
void nullify_array (AA *Ptr, DWORD Index, AA *ThisPtr)
{
while (Ptr->Array[--Index] != ThisPtr)
    {
    Ptr->Array[Index] = NULL ;
    } ;
} ;
	nullify_loop:
	mov dword ptr [eax], 0
	mov edx, dword ptr [edi]
	sub ecx, 4
	cmp dword ptr [ecx+edx], esi
	lea eax, [ecx+edx]
	jne nullify_loop

3.6.4 Alignment

Alignment of data concerns all kinds of variables:

- Dynamically allocated variables.
- Members of a data structure.
- Global or local variables.
- Parameters passed on the stack.

Misaligned data access can incur significant performance penalties. This is particularly true for cache line splits. The size of a cache line is 64 bytes in the Pentium 4 and other recent Intel processors, including processors based on Intel Core microarchitecture.

An access to data unaligned on 64-byte boundary leads to two memory accesses and requires several µops to be executed (instead of one). Accesses that span 64-byte boundaries are likely to incur a large performance penalty, the cost of each stall generally are greater on machines with longer pipelines.

Double-precision floating-point operands that are eight-byte aligned have better performance than operands that are not eight-byte aligned, since they are less likely to incur penalties for cache and MOB splits. Floating-point operation on a memory operands require that the operand be loaded from memory. This incurs an additional µop, which can have a minor negative impact on front end bandwidth. Additionally, memory operands may cause a data cache miss, causing a penalty.

Assembly/Compiler Coding Rule 46. (H impact, H generality) *Align data on natural operand size address boundaries. If the data will be accessed with vector instruction loads and stores, align the data on 16-byte boundaries.*

For best performance, align data as follows:

- Align 8-bit data at any address.
- Align 16-bit data to be contained within an aligned 4-byte word.
- Align 32-bit data so that its base address is a multiple of four.
- Align 64-bit data so that its base address is a multiple of eight.

- Align 80-bit data so that its base address is a multiple of sixteen.
- Align 128-bit data so that its base address is a multiple of sixteen.

A 64-byte or greater data structure or array should be aligned so that its base address is a multiple of 64. Sorting data in decreasing size order is one heuristic for assisting with natural alignment. As long as 16-byte boundaries (and cache lines) are never crossed, natural alignment is not strictly necessary (though it is an easy way to enforce this).

Example 3-40 shows the type of code that can cause a cache line split. The code loads the addresses of two DWORD arrays. 029E70FEH is not a 4-byte-aligned address, so a 4-byte access at this address will get 2 bytes from the cache line this address is contained in, and 2 bytes from the cache line that starts at 029E700H. On processors with 64-byte cache lines, a similar cache line split will occur every 8 iterations.

Example 3-40. Code That Causes Cache Line Split

```
    mov     esi, 029e70feh
    mov     edi, 05be5260h
Blockmove:
    mov     eax, DWORD PTR [esi]
    mov     ebx, DWORD PTR [esi+4]
    mov     DWORD PTR [edi], eax
    mov     DWORD PTR [edi+4], ebx
    add     esi, 8
    add     edi, 8
    sub     edx, 1
    jnz     Blockmove
```

Figure 3-2 illustrates the situation of accessing a data element that span across cache line boundaries.

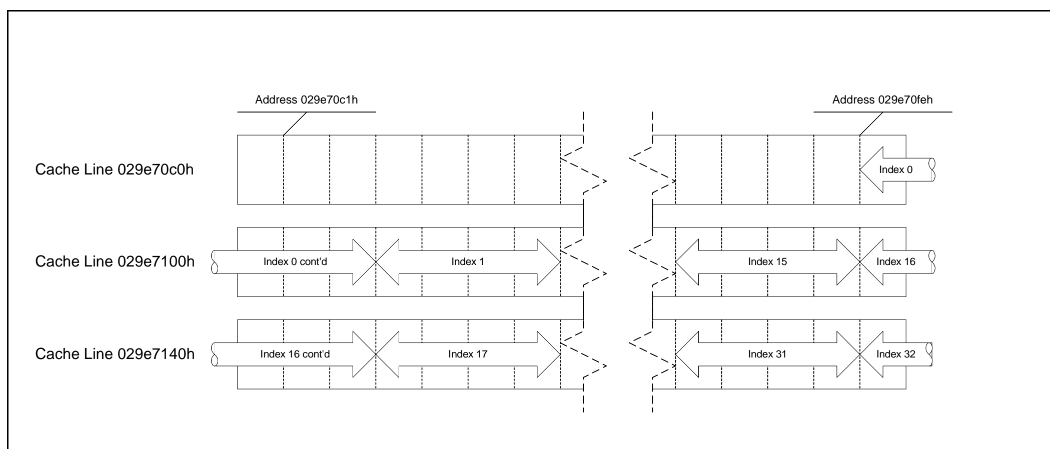


Figure 3-2. Cache Line Split in Accessing Elements in a Array

Alignment of code is less important for processors based on Intel NetBurst microarchitecture. Alignment of branch targets to maximize bandwidth of fetching cached instructions is an issue only when not executing out of the trace cache.

Alignment of code can be an issue for the Pentium M, Intel Core Duo and Intel Core 2 Duo processors. Alignment of branch targets will improve decoder throughput.

3.6.5 Store Forwarding

The processor's memory system only sends stores to memory (including cache) after store retirement. However, store data can be forwarded from a store to a subsequent load from the same address to give a much shorter store-load latency.

There are two kinds of requirements for store forwarding. If these requirements are violated, store forwarding cannot occur and the load must get its data from the cache (so the store must write its data back to the cache first). This incurs a penalty that is largely related to pipeline depth of the underlying micro-architecture.

The first requirement pertains to the size and alignment of the store-forwarding data. This restriction is likely to have high impact on overall application performance. Typically, a performance penalty due to violating this restriction can be prevented. The store-to-load forwarding restrictions vary from one micro-architecture to another. Several examples of coding pitfalls that cause store-forwarding stalls and solutions to these pitfalls are discussed in detail in Section 3.6.5.1, "Store-to-Load-Forwarding Restriction on Size and Alignment." The second requirement is the availability of data, discussed in Section 3.6.5.2, "Store-forwarding Restriction on Data Availability." A good practice is to eliminate redundant load operations.

It may be possible to keep a temporary scalar variable in a register and never write it to memory. Generally, such a variable must not be accessible using indirect pointers. Moving a variable to a register eliminates all loads and stores of that variable and eliminates potential problems associated with store forwarding. However, it also increases register pressure.

Load instructions tend to start chains of computation. Since the out-of-order engine is based on data dependence, load instructions play a significant role in the engine's ability to execute at a high rate. Eliminating loads should be given a high priority.

If a variable does not change between the time when it is stored and the time when it is used again, the register that was stored can be copied or used directly. If register pressure is too high, or an unseen function is called before the store and the second load, it may not be possible to eliminate the second load.

Assembly/Compiler Coding Rule 47. (H impact, M generality) *Pass parameters in registers instead of on the stack where possible. Passing arguments on the stack requires a store followed by a reload. While this sequence is optimized in hardware by providing the value to the load directly from the memory order buffer without the need to access the data cache if permitted by store-forwarding restrictions, floating-point values incur a significant latency in forwarding. Passing floating-point arguments in (preferably XMM) registers should save this long latency operation.*

Parameter passing conventions may limit the choice of which parameters are passed in registers which are passed on the stack. However, these limitations may be overcome if the compiler has control of the compilation of the whole binary (using whole-program optimization).

3.6.5.1 Store-to-Load-Forwarding Restriction on Size and Alignment

Data size and alignment restrictions for store-forwarding apply to processors based on Intel NetBurst microarchitecture, Intel Core microarchitecture, Intel Core 2 Duo, Intel Core Solo and Pentium M processors. The performance penalty for violating store-forwarding restrictions is less for shorter-pipelined machines than for Intel NetBurst microarchitecture.

Store-forwarding restrictions vary with each microarchitecture. Intel NetBurst microarchitecture places more constraints than Intel Core microarchitecture on code generation to enable store-forwarding to make progress instead of experiencing stalls. Fixing store-forwarding problems for Intel NetBurst microarchitecture generally also avoids problems on Pentium M, Intel Core Duo and Intel Core 2 Duo processors. The size and alignment restrictions for store-forwarding in processors based on Intel NetBurst microarchitecture are illustrated in Figure 3-3.

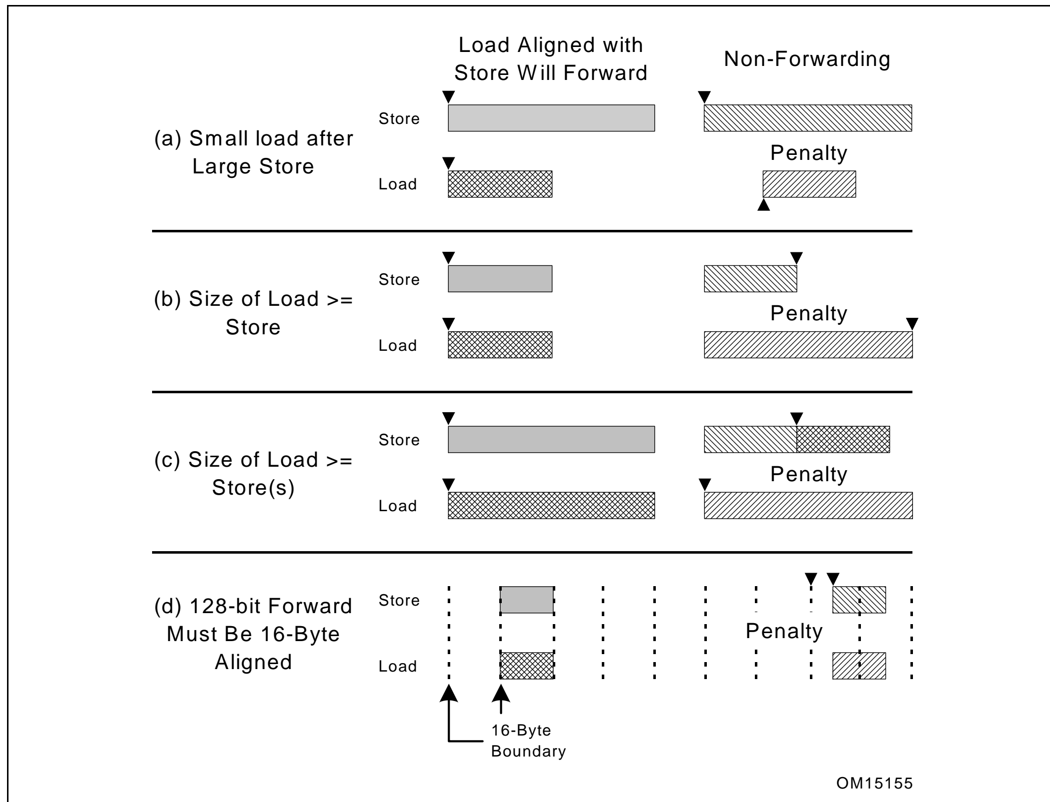


Figure 3-3. Size and Alignment Restrictions in Store Forwarding

The following rules help satisfy size and alignment restrictions for store forwarding:

Assembly/Compiler Coding Rule 48. (H impact, M generality) *A load that forwards from a store must have the same address start point and therefore the same alignment as the store data.*

Assembly/Compiler Coding Rule 49. (H impact, M generality) *The data of a load which is forwarded from a store must be completely contained within the store data.*

A load that forwards from a store must wait for the store's data to be written to the store buffer before proceeding, but other, unrelated loads need not wait.

Assembly/Compiler Coding Rule 50. (H impact, ML generality) *If it is necessary to extract a non-aligned portion of stored data, read out the smallest aligned portion that completely contains the data and shift/mask the data as necessary. This is better than incurring the penalties of a failed store-forward.*

Assembly/Compiler Coding Rule 51. (MH impact, ML generality) *Avoid several small loads after large stores to the same area of memory by using a single large read and register copies as needed.*

Example 3-41 depicts several store-forwarding situations in which small loads follow large stores. The first three load operations illustrate the situations described in Rule 51. However, the last load operation gets data from store-forwarding without problem.

Example 3-41. Situations Showing Small Loads After Large Store

```
mov [EBP],'abcd'
mov AL, [EBP]       ; Not blocked - same alignment
mov BL, [EBP + 1]   ; Blocked
mov CL, [EBP + 2]   ; Blocked
mov DL, [EBP + 3]   ; Blocked
mov AL, [EBP]       ; Not blocked - same alignment
                    ; n.b. passes older blocked loads
```

Example 3-42 illustrates a store-forwarding situation in which a large load follows several small stores. The data needed by the load operation cannot be forwarded because all of the data that needs to be forwarded is not contained in the store buffer. Avoid large loads after small stores to the same area of memory.

Example 3-42. Non-forwarding Example of Large Load After Small Store

```
mov [EBP], 'a'
mov [EBP + 1], 'b'
mov [EBP + 2], 'c'
mov [EBP + 3], 'd'
mov EAX, [EBP]    ; Blocked
    ; The first 4 small store can be consolidated into
    ; a single DWORD store to prevent this non-forwarding
    ; situation.
```

Example 3-43 illustrates a stalled store-forwarding situation that may appear in compiler generated code. Sometimes a compiler generates code similar to that shown in Example 3-43 to handle a spilled byte to the stack and convert the byte to an integer value.

Example 3-43. A Non-forwarding Situation in Compiler Generated Code

```
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
mov eax, DWORD PTR [esp+10h]  ; Stall
and eax, 0xff                 ; Converting back to byte value
```

Example 3-44 offers two alternatives to avoid the non-forwarding situation shown in Example 3-43.

Example 3-44. Two Ways to Avoid Non-forwarding Situation in Example 3-43

```
; A. Use MOVZ instruction to avoid large load after small
; store, when spills are ignored.
movz eax, bl                        ; Replaces the last three instructions
; B. Use MOVZ instruction and handle spills to the stack
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
movz eax, BYTE PTR [esp+10h]    ; Not blocked
```

When moving data that is smaller than 64 bits between memory locations, 64-bit or 128-bit SIMD register moves are more efficient (if aligned) and can be used to avoid unaligned loads. Although floating-point registers allow the movement of 64 bits at a time, floating-point instructions should not be used for this purpose, as data may be inadvertently modified.

As an additional example, consider the cases in Example 3-45.

Example 3-45. Large and Small Load Stalls

```
; A. Large load stall
mov        mem, eax        ; Store dword to address "MEM"
mov        mem + 4, ebx    ; Store dword to address "MEM + 4"
fld        mem             ; Load qword at address "MEM", stalls
; B. Small Load stall
fstp    mem                ; Store qword to address "MEM"
mov     bx, mem+2          ; Load word at address "MEM + 2", stalls
mov     cx, mem+4          ; Load word at address "MEM + 4", stalls
```

In the first case (A), there is a large load after a series of small stores to the same area of memory (beginning at memory address MEM). The large load will stall.

The FLD must wait for the stores to write to memory before it can access all the data it requires. This stall can also occur with other data types (for example, when bytes or words are stored and then words or doublewords are read from the same area of memory).

In the second case (B), there is a series of small loads after a large store to the same area of memory (beginning at memory address MEM). The small loads will stall.

The word loads must wait for the quadword store to write to memory before they can access the data they require. This stall can also occur with other data types (for example, when doublewords or words are stored and then words or bytes are read from the same area of memory). This can be avoided by moving the store as far from the loads as possible.

Store forwarding restrictions for processors based on Intel Core microarchitecture is listed in Table 3-3.

Table 3-3. Store Forwarding Restrictions of Processors Based on Intel Core Microarchitecture

Store Alignment	Width of Store (bits)	Load Alignment (byte)	Width of Load (bits)	Store Forwarding Restriction
To Natural size	16	word aligned	8, 16	not stalled
To Natural size	16	not word aligned	8	stalled
To Natural size	32	dword aligned	8, 32	not stalled
To Natural size	32	not dword aligned	8	stalled
To Natural size	32	word aligned	16	not stalled
To Natural size	32	not word aligned	16	stalled

Table 3-3. Store Forwarding Restrictions of Processors Based on Intel Core Microarchitecture (Contd.)

Store Alignment	Width of Store (bits)	Load Alignment (byte)	Width of Load (bits)	Store Forwarding Restriction
To Natural size	64	qword aligned	8, 16, 64	not stalled
To Natural size	64	not qword aligned	8, 16	stalled
To Natural size	64	dword aligned	32	not stalled
To Natural size	64	not dword aligned	32	stalled
To Natural size	128	dqword aligned	8, 16, 128	not stalled
To Natural size	128	not dqword aligned	8, 16	stalled
To Natural size	128	dword aligned	32	not stalled
To Natural size	128	not dword aligned	32	stalled
To Natural size	128	qword aligned	64	not stalled
To Natural size	128	not qword aligned	64	stalled
Unaligned, start byte 1	32	byte 0 of store	8, 16, 32	not stalled
Unaligned, start byte 1	32	not byte 0 of store	8, 16	stalled
Unaligned, start byte 1	64	byte 0 of store	8, 16, 32	not stalled
Unaligned, start byte 1	64	not byte 0 of store	8, 16, 32	stalled
Unaligned, start byte 1	64	byte 0 of store	64	stalled
Unaligned, start byte 7	32	byte 0 of store	8	not stalled
Unaligned, start byte 7	32	not byte 0 of store	8	not stalled
Unaligned, start byte 7	32	don’t care	16, 32	stalled
Unaligned, start byte 7	64	don’t care	16, 32, 64	stalled

3.6.5.2 Store-forwarding Restriction on Data Availability

The value to be stored must be available before the load operation can be completed. If this restriction is violated, the execution of the load will be delayed until the data is available. This delay causes some execution resources to be used unnecessarily, and that can lead to sizable but non-deterministic delays. However, the overall impact of this problem is much smaller than that from violating size and alignment requirements.

In modern microarchitectures, hardware predicts when loads are dependent on and get their data forwarded from preceding stores. These predictions can significantly improve performance. However, if a load is scheduled too soon after the store it depends on or if the generation of the data to be stored is delayed, there can be a significant penalty.

There are several cases in which data is passed through memory, and the store may need to be separated from the load:

- Spills, save and restore registers in a stack frame.
- Parameter passing.
- Global and volatile variables.
- Type conversion between integer and floating-point.

- When compilers do not analyze code that is inlined, forcing variables that are involved in the interface with inlined code to be in memory, creating more memory variables and preventing the elimination of redundant loads.

Assembly/Compiler Coding Rule 52. (H impact, MH generality) *Where it is possible to do so without incurring other penalties, prioritize the allocation of variables to registers, as in register allocation and for parameter passing, to minimize the likelihood and impact of store-forwarding problems. Try not to store-forward data generated from a long latency instruction - for example, MUL or DIV. Avoid store-forwarding data for variables with the shortest store-load distance. Avoid store-forwarding data for variables with many and/or long dependence chains, and especially avoid including a store forward on a loop-carried dependence chain.*

Example 3-46 shows an example of a loop-carried dependence chain.

Example 3-46. Loop-carried Dependence Chain

```
for ( i = 0; i < MAX; i++ ) {
    a[i] = b[i] * foo;
    foo = a[i] / 3;
}                    // foo is a loop-carried dependence.
```

Assembly/Compiler Coding Rule 53. (M impact, MH generality) *Calculate store addresses as early as possible to avoid having stores block loads.*

3.6.6 Data Layout Optimizations

User/Source Coding Rule 6. (H impact, M generality) *Pad data structures defined in the source code so that every data element is aligned to a natural operand size address boundary.*

If the operands are packed in a SIMD instruction, align to the packed element size (64-bit or 128-bit).

Align data by providing padding inside structures and arrays. Programmers can reorganize structures and arrays to minimize the amount of memory wasted by padding. However, compilers might not have this freedom. The C programming language, for example, specifies the order in which structure elements are allocated in memory. For more information, see Section 4.4, “Stack and Data Alignment”.

Example 3-47 shows how a data structure could be rearranged to reduce its size.

Example 3-47. Rearranging a Data Structure

```
struct unpacked { /* Fits in 20 bytes due to padding */
    int      a;
    char     b;
    int      c;
    char     d;
    int      e;
};

struct packed {  /* Fits in 16 bytes */
    int      a;
    int      c;
    int      e;
    char     b;
    char     d;
}
```

Cache line size of 64 bytes can impact streaming applications (for example, multimedia). These reference and use data only once before discarding it. Data accesses which sparsely utilize the data within a cache line can result in less efficient utilization of system memory bandwidth. For example, arrays of structures can be decomposed into several arrays to achieve better packing, as shown in Example 3-48.

Example 3-48. Decomposing an Array

```
struct {        /* 1600 bytes */
    int    a, c, e;
    char b, d;
} array_of_struct [100];

struct {        /* 1400 bytes */
    int    a[100], c[100], e[100];
    char b[100], d[100];
} struct_of_array;


struct {        /* 1200 bytes */
    int   a, c, e;
} hybrid_struct_of_array_ace[100];


struct {        /* 200 bytes */
    char b, d;
} hybrid_struct_of_array_bd[100];
```

The efficiency of such optimizations depends on usage patterns. If the elements of the structure are all accessed together but the access pattern of the array is random, then ARRAY_OF_STRUCT avoids unnecessary prefetch even though it wastes memory.

However, if the access pattern of the array exhibits locality (for example, if the array index is being swept through) then processors with hardware prefetchers will prefetch data from STRUCT_OF_ARRAY, even if the elements of the structure are accessed together.

When the elements of the structure are not accessed with equal frequency, such as when element A is accessed ten times more often than the other entries, then STRUCT_OF_ARRAY not only saves memory, but it also prevents fetching unnecessary data items B, C, D, and E.

Using STRUCT_OF_ARRAY also enables the use of the SIMD data types by the programmer and the compiler.

Note that STRUCT_OF_ARRAY can have the disadvantage of requiring more independent memory stream references. This can require the use of more prefetches and additional address generation calculations. It can also have an impact on DRAM page access efficiency. An alternative, HYBRID_STRUCT_OF_ARRAY blends the two approaches. In this case, only 2 separate address streams are generated and referenced: 1 for HYBRID_STRUCT_OF_ARRAY_ACE and 1 for HYBRID_STRUCT_OF_ARRAY_BD. The second alterative also prevents fetching unnecessary data — assuming that (1) the variables A, C and E are always used together, and (2) the variables B and D are always used together, but not at the same time as A, C and E.

The hybrid approach ensures:

- Simpler/fewer address generations than STRUCT_OF_ARRAY.
- Fewer streams, which reduces DRAM page misses.
- Fewer prefetches due to fewer streams.
- Efficient cache line packing of data elements that are used concurrently.

Assembly/Compiler Coding Rule 54. (H impact, M generality) *Try to arrange data structures such that they permit sequential access.*

If the data is arranged into a set of streams, the automatic hardware prefetcher can prefetch data that will be needed by the application, reducing the effective memory latency. If the data is accessed in a non-sequential manner, the automatic hardware prefetcher cannot prefetch the data. The prefetcher can

recognize up to eight concurrent streams. See Chapter 7, “Optimizing Cache Usage,” for more information on the hardware prefetcher.

User/Source Coding Rule 7. (M impact, L generality) *Beware of false sharing within a cache line (64 bytes).*

3.6.7 Stack Alignment

Performance penalty of unaligned access to the stack happens when a memory reference splits a cache line. This means that one out of eight spatially consecutive unaligned quadword accesses is always penalized, similarly for one out of 4 consecutive, non-aligned double-quadword accesses, etc.

Aligning the stack may be beneficial any time there are data objects that exceed the default stack alignment of the system. For example, on 32/64bit Linux, and 64bit Windows, the default stack alignment is 16 bytes, while 32bit Windows is 4 bytes.

Assembly/Compiler Coding Rule 55. (H impact, M generality) *Make sure that the stack is aligned at the largest multi-byte granular data type boundary matching the register width.*

Aligning the stack typically requires the use of an additional register to track across a padded area of unknown amount. There is a trade-off between causing unaligned memory references that spanned across a cache line and causing extra general purpose register spills.

The assembly level technique to implement dynamic stack alignment may depend on compilers, and specific OS environment. The reader may wish to study the assembly output from a compiler of interest.

Example 3-49. Examples of Dynamical Stack Alignment

```
// 32-bit environment
    push    ebp ; save ebp
    mov     ebp, esp ; ebp now points to incoming parameters
    andl    esp, $-<N> ;align esp to N byte boundary
    sub     esp, $<stack_size>; reserve space for new stack frame
    .        ; parameters must be referenced off of ebp
    mov     esp, ebp ; restore esp
    pop     ebp ; restore ebp

// 64-bit environment
    sub     esp, $<stack_size +N>
    mov     r13, $<offset_of_aligned_section_in_stack>
    andl    r13, $-<N> ; r13 point to aligned section in stack
    .        ;use r13 as base for aligned data
```

If for some reason it is not possible to align the stack for 64-bits, the routine should access the parameter and save it into a register or known aligned storage, thus incurring the penalty only once.

3.6.8 Capacity Limits and Aliasing in Caches

There are cases in which addresses with a given stride will compete for some resource in the memory hierarchy.

Typically, caches are implemented to have multiple ways of set associativity, with each way consisting of multiple sets of cache lines (or sectors in some cases). Multiple memory references that compete for the same set of each way in a cache can cause a capacity issue. There are aliasing conditions that apply to specific microarchitectures. Note that first-level cache lines are 64 bytes. Thus, the least significant 6 bits

are not considered in alias comparisons. For processors based on Intel NetBurst microarchitecture, data is loaded into the second level cache in a sector of 128 bytes, so the least significant 7 bits are not considered in alias comparisons.

3.6.8.1 Capacity Limits in Set-Associative Caches

Capacity limits may be reached if the number of outstanding memory references that are mapped to the same set in each way of a given cache exceeds the number of ways of that cache. The conditions that apply to the first-level data cache and second level cache are listed below:

- **L1 Set Conflicts** — Multiple references map to the same first-level cache set. The conflicting condition is a stride determined by the size of the cache in bytes, divided by the number of ways. These competing memory references can cause excessive cache misses only if the number of outstanding memory references exceeds the number of ways in the working set:
  - On Pentium 4 and Intel Xeon processors with a CPUID signature of family encoding 15, model encoding of 0, 1, or 2; there will be an excess of first-level cache misses for more than 4 simultaneous competing memory references to addresses with 2-KByte modulus.
  - On Pentium 4 and Intel Xeon processors with a CPUID signature of family encoding 15, model encoding 3; there will be an excess of first-level cache misses for more than 8 simultaneous competing references to addresses that are apart by 2-KByte modulus.
  - On Intel Core 2 Duo, Intel Core Duo, Intel Core Solo, and Pentium M processors, there will be an excess of first-level cache misses for more than 8 simultaneous references to addresses that are apart by 4-KByte modulus.
- **L2 Set Conflicts** — Multiple references map to the same second-level cache set. The conflicting condition is also determined by the size of the cache or the number of ways:
  - On Pentium 4 and Intel Xeon processors, there will be an excess of second-level cache misses for more than 8 simultaneous competing references. The stride sizes that can cause capacity issues are 32 KBytes, 64 KBytes, or 128 KBytes, depending of the size of the second level cache.
  - On Pentium M processors, the stride sizes that can cause capacity issues are 128 KBytes or 256 KBytes, depending of the size of the second level cache. On Intel Core 2 Duo, Intel Core Duo, Intel Core Solo processors, stride size of 256 KBytes can cause capacity issue if the number of simultaneous accesses exceeded the way associativity of the L2 cache.

3.6.8.2 Aliasing Cases in the Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo and Intel® Core™ 2 Duo Processors

Pentium M, Intel Core Solo, Intel Core Duo and Intel Core 2 Duo processors have the following aliasing case:

- **Store forwarding** — If a store to an address is followed by a load from the same address, the load will not proceed until the store data is available. If a store is followed by a load and their addresses differ by a multiple of 4 KBytes, the load stalls until the store operation completes.

Assembly/Compiler Coding Rule 56. (H impact, M generality) *Avoid having a store followed by a non-dependent load with addresses that differ by a multiple of 4 KBytes. Also, lay out data or order computation to avoid having cache lines that have linear addresses that are a multiple of 64 KBytes apart in the same working set. Avoid having more than 4 cache lines that are some multiple of 2 KBytes apart in the same first-level cache working set, and avoid having more than 8 cache lines that are some multiple of 4 KBytes apart in the same first-level cache working set.*

When declaring multiple arrays that are referenced with the same index and are each a multiple of 64 KBytes (as can happen with STRUCT_OF_ARRAY data layouts), pad them to avoid declaring them contiguously. Padding can be accomplished by either intervening declarations of other variables or by artificially increasing the dimension.

User/Source Coding Rule 8. (H impact, ML generality) *Consider using a special memory allocation library with address offset capability to avoid aliasing.* One way to implement a memory allocator to avoid aliasing is to allocate more than enough space and pad. For example, allocate structures that are

68 KB instead of 64 KBytes to avoid the 64-KByte aliasing, or have the allocator pad and return random offsets that are a multiple of 128 Bytes (the size of a cache line).

User/Source Coding Rule 9. (M impact, M generality) *When padding variable declarations to avoid aliasing, the greatest benefit comes from avoiding aliasing on second-level cache lines, suggesting an offset of 128 bytes or more.*

4-KByte memory aliasing occurs when the code accesses two different memory locations with a 4-KByte offset between them. The 4-KByte aliasing situation can manifest in a memory copy routine where the addresses of the source buffer and destination buffer maintain a constant offset and the constant offset happens to be a multiple of the byte increment from one iteration to the next.

Example 3-50 shows a routine that copies 16 bytes of memory in each iteration of a loop. If the offsets (modular 4096) between source buffer (EAX) and destination buffer (EDX) differ by 16, 32, 48, 64, 80; loads have to wait until stores have been retired before they can continue. For example at offset 16, the load of the next iteration is 4-KByte aliased current iteration store, therefore the loop must wait until the store operation completes, making the entire loop serialized. The amount of time needed to wait decreases with larger offset until offset of 96 resolves the issue (as there is no pending stores by the time of the load with same address).

The Intel Core microarchitecture provides a performance monitoring event (see LOAD_BLOCK.OVERLAP_STORE in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*) that allows software tuning effort to detect the occurrence of aliasing conditions.

Example 3-50. Aliasing Between Loads and Stores Across Loop Iterations

```
LP:
        movaps xmm0, [eax+ecx]
        movaps [edx+ecx], xmm0
        add ecx, 16
        jnz lp
```

3.6.9 Mixing Code and Data

The aggressive prefetching and pre-decoding of instructions by Intel processors have two related effects:

- Self-modifying code works correctly, according to the Intel architecture processor requirements, but incurs a significant performance penalty. Avoid self-modifying code if possible.
- Placing writable data in the code segment might be impossible to distinguish from self-modifying code. Writable data in the code segment might suffer the same performance penalty as self-modifying code.

Assembly/Compiler Coding Rule 57. (M impact, L generality) *If (hopefully read-only) data must occur on the same page as code, avoid placing it immediately after an indirect jump. For example, follow an indirect jump with its mostly likely target, and place the data after an unconditional branch.*

Tuning Suggestion 1. *In rare cases, a performance problem may be caused by executing data on a code page as instructions. This is very likely to happen when execution is following an indirect branch that is not resident in the trace cache. If this is clearly causing a performance problem, try moving the data elsewhere, or inserting an illegal opcode or a* PAUSE *instruction immediately after the indirect branch. Note that the latter two alternatives may degrade performance in some circumstances.*

Assembly/Compiler Coding Rule 58. (H impact, L generality) *Always put code and data on separate pages. Avoid self-modifying code wherever possible. If code is to be modified, try to do it all at once and make sure the code that performs the modifications and the code being modified are on separate 4-KByte pages or on separate aligned 1-KByte subpages.*

3.6.9.1 Self-modifying Code

Self-modifying code (SMC) that ran correctly on Pentium III processors and prior implementations will run correctly on subsequent implementations. SMC and cross-modifying code (when multiple processors in a multiprocessor system are writing to a code page) should be avoided when high performance is desired.

Software should avoid writing to a code page in the same 1-KByte subpage that is being executed or fetching code in the same 2-KByte subpage of that is being written. In addition, sharing a page containing directly or speculatively executed code with another processor as a data page can trigger an SMC condition that causes the entire pipeline of the machine and the trace cache to be cleared. This is due to the self-modifying code condition.

Dynamic code need not cause the SMC condition if the code written fills up a data page before that page is accessed as code. Dynamically-modified code (for example, from target fix-ups) is likely to suffer from the SMC condition and should be avoided where possible. Avoid the condition by introducing indirect branches and using data tables on data pages (not code pages) using register-indirect calls.

3.6.9.2 Position Independent Code

Position independent code often needs to obtain the value of the instruction pointer. Example 3-51a shows one technique to put the value of IP into the ECX register by issuing a CALL without a matching RET. Example 3-51b shows an alternative technique to put the value of IP into the ECX register using a matched pair of CALL/RET.

Example 3-51. Instruction Pointer Query Techniques

```
a) Using call without return to obtain IP does not corrupt the RSB
        call _label; return address pushed is the IP of next instruction
_label:
        pop ECX; IP of this instruction is now put into ECX


b) Using matched call/ret pair

        call _lblcx;
        ... ; ECX now contains IP of this instruction
        ...
_lblcx
        mov ecx, [esp];
        ret
```

3.6.10 Write Combining

Write combining (WC) improves performance in two ways:

- On a write miss to the first-level cache, it allows multiple stores to the same cache line to occur before that cache line is read for ownership (RFO) from further out in the cache/memory hierarchy. Then the rest of line is read, and the bytes that have not been written are combined with the unmodified bytes in the returned line.

- Write combining allows multiple writes to be assembled and written further out in the cache hierarchy as a unit. This saves port and bus traffic. Saving traffic is particularly important for avoiding partial writes to uncached memory.

There are six write-combining buffers (on Pentium 4 and Intel Xeon processors with a CPUID signature of family encoding 15, model encoding 3; there are 8 write-combining buffers). Two of these buffers may be written out to higher cache levels and freed up for use on other write misses. Only four write-combining buffers are guaranteed to be available for simultaneous use. Write combining applies to memory type WC; it does not apply to memory type UC.

There are six write-combining buffers in each processor core in Intel Core Duo and Intel Core Solo processors. Processors based on Intel Core microarchitecture have eight write-combining buffers in each core. Starting with Intel microarchitecture code name Nehalem, there are 10 buffers available for write-combining.

Assembly/Compiler Coding Rule 59. (H impact, L generality) *If an inner loop writes to more than four arrays (four distinct cache lines), apply loop fission to break up the body of the loop such that only four arrays are being written to in each iteration of each of the resulting loops.*

Write combining buffers are used for stores of all memory types. They are particularly important for writes to uncached memory: writes to different parts of the same cache line can be grouped into a single, full-cache-line bus transaction instead of going across the bus (since they are not cached) as several partial writes. Avoiding partial writes can have a significant impact on bus bandwidth-bound graphics applications, where graphics buffers are in uncached memory. Separating writes to uncached memory and writes to writeback memory into separate phases can assure that the write combining buffers can fill before getting evicted by other write traffic. Eliminating partial write transactions has been found to have performance impact on the order of 20% for some applications. Because the cache lines are 64 bytes, a write to the bus for 63 bytes will result in 8 partial bus transactions.

When coding functions that execute simultaneously on two threads, reducing the number of writes that are allowed in an inner loop will help take full advantage of write-combining store buffers. For write-combining buffer recommendations for Hyper-Threading Technology, see Chapter 8, “Multicore and Hyper-Threading Technology.”

Store ordering and visibility are also important issues for write combining. When a write to a write-combining buffer for a previously-unwritten cache line occurs, there will be a read-for-ownership (RFO). If a subsequent write happens to another write-combining buffer, a separate RFO may be caused for that cache line. Subsequent writes to the first cache line and write-combining buffer will be delayed until the second RFO has been serviced to guarantee properly ordered visibility of the writes. If the memory type for the writes is write-combining, there will be no RFO since the line is not cached, and there is no such delay. For details on write-combining, see Chapter 7, “Optimizing Cache Usage.”

3.6.11 Locality Enhancement

Locality enhancement can reduce data traffic originating from an outer-level sub-system in the cache/memory hierarchy. This is to address the fact that the access-cost in terms of cycle-count from an outer level will be more expensive than from an inner level. Typically, the cycle-cost of accessing a given cache level (or memory system) varies across different microarchitectures, processor implementations, and platform components. It may be sufficient to recognize the relative data access cost trend by locality rather than to follow a large table of numeric values of cycle-costs, listed per locality, per processor/platform implementations, etc. The general trend is typically that access cost from an outer sub-system may be approximately 3-10X more expensive than accessing data from the immediate inner level in the cache/memory hierarchy, assuming similar degrees of data access parallelism.

Thus locality enhancement should start with characterizing the dominant data traffic locality. Section A, “Application Performance Tools,” describes some techniques that can be used to determine the dominant data traffic locality for any workload.

Even if cache miss rates of the last level cache may be low relative to the number of cache references, processors typically spend a sizable portion of their execution time waiting for cache misses to be serviced. Reducing cache misses by enhancing a program’s locality is a key optimization. This can take several forms:

- Blocking to iterate over a portion of an array that will fit in the cache (with the purpose that subsequent references to the data-block [or tile] will be cache hit references).
- Loop interchange to avoid crossing cache lines or page boundaries.
- Loop skewing to make accesses contiguous.

Locality enhancement to the last level cache can be accomplished with sequencing the data access pattern to take advantage of hardware prefetching. This can also take several forms:

- Transformation of a sparsely populated multi-dimensional array into a one-dimension array such that memory references occur in a sequential, small-stride pattern that is friendly to the hardware prefetch (see Section 2.3.5.4, "Data Prefetching").
- Optimal tile size and shape selection can further improve temporal data locality by increasing hit rates into the last level cache and reduce memory traffic resulting from the actions of hardware prefetching (see Section 7.6.11, "Hardware Prefetching and Cache Blocking Techniques").

It is important to avoid operations that work against locality-enhancing techniques. Using the lock prefix heavily can incur large delays when accessing memory, regardless of whether the data is in the cache or in system memory.

User/Source Coding Rule 10. (H impact, H generality) *Optimization techniques such as blocking, loop interchange, loop skewing, and packing are best done by the compiler. Optimize data structures either to fit in one-half of the first-level cache or in the second-level cache; turn on loop optimizations in the compiler to enhance locality for nested loops.*

Optimizing for one-half of the first-level cache will bring the greatest performance benefit in terms of cycle-cost per data access. If one-half of the first-level cache is too small to be practical, optimize for the second-level cache. Optimizing for a point in between (for example, for the entire first-level cache) will likely not bring a substantial improvement over optimizing for the second-level cache.

3.6.12 Minimizing Bus Latency

Each bus transaction includes the overhead of making requests and arbitrations. The average latency of bus read and bus write transactions will be longer if reads and writes alternate. Segmenting reads and writes into phases can reduce the average latency of bus transactions. This is because the number of incidences of successive transactions involving a read following a write, or a write following a read, are reduced.

User/Source Coding Rule 11. (M impact, ML generality) *If there is a blend of reads and writes on the bus, changing the code to separate these bus transactions into read phases and write phases can help performance.*

Note, however, that the order of read and write operations on the bus is not the same as it appears in the program.

Bus latency for fetching a cache line of data can vary as a function of the access stride of data references. In general, bus latency will increase in response to increasing values of the stride of successive cache misses. Independently, bus latency will also increase as a function of increasing bus queue depths (the number of outstanding bus requests of a given transaction type). The combination of these two trends can be highly non-linear, in that bus latency of large-stride, bandwidth-sensitive situations are such that effective throughput of the bus system for data-parallel accesses can be significantly less than the effective throughput of small-stride, bandwidth-sensitive situations.

To minimize the per-access cost of memory traffic or amortize raw memory latency effectively, software should control its cache miss pattern to favor higher concentration of smaller-stride cache misses.

User/Source Coding Rule 12. (H impact, H generality) *To achieve effective amortization of bus latency, software should favor data access patterns that result in higher concentrations of cache miss patterns, with cache miss strides that are significantly smaller than half the hardware prefetch trigger threshold.*

3.6.13 Non-Temporal Store Bus Traffic

Peak system bus bandwidth is shared by several types of bus activities, including reads (from memory), reads for ownership (of a cache line), and writes. The data transfer rate for bus write transactions is higher if 64 bytes are written out to the bus at a time.

Typically, bus writes to Writeback (WB) memory must share the system bus bandwidth with read-for-ownership (RFO) traffic. Non-temporal stores do not require RFO traffic; they do require care in managing the access patterns in order to ensure 64 bytes are evicted at once (rather than evicting several 8-byte chunks).

Although the data bandwidth of full 64-byte bus writes due to non-temporal stores is twice that of bus writes to WB memory, transferring 8-byte chunks wastes bus request bandwidth and delivers significantly lower data bandwidth. This difference is depicted in Examples 3-52 and 3-53.

Example 3-52. Using Non-temporal Stores and 64-byte Bus Write Transactions

```
#define STRIDESIZE 256
lea ecx, p64byte_Aligned
mov edx, ARRAY_LEN
xor eax, eax
slloop:
movntps XMMWORD ptr [ecx + eax], xmm0
movntps XMMWORD ptr [ecx + eax+16], xmm0
movntps XMMWORD ptr [ecx + eax+32], xmm0
movntps XMMWORD ptr [ecx + eax+48], xmm0
; 64 bytes is written in one bus transaction
add eax, STRIDESIZE
cmp eax, edx
jl slloop
```

Example 3-53. On-temporal Stores and Partial Bus Write Transactions

```
#define STRIDESIZE 256
Lea ecx, p64byte_Aligned
Mov edx, ARRAY_LEN
Xor eax, eax
slloop:
movntps XMMWORD ptr [ecx + eax], xmm0
movntps XMMWORD ptr [ecx + eax+16], xmm0
movntps XMMWORD ptr [ecx + eax+32], xmm0


; Storing 48 bytes results in 6 bus partial transactions
add eax, STRIDESIZE
cmp eax, edx
jl slloop
```

3.7 PREFETCHING

Recent Intel processor families employ several prefetching mechanisms to accelerate the movement of data or code and improve performance:

- Hardware instruction prefetcher.
- Software prefetch for data.

- Hardware prefetch for cache lines of data or instructions.

3.7.1 Hardware Instruction Fetching and Software Prefetching

Software prefetching requires a programmer to use PREFETCH hint instructions and anticipate some suitable timing and location of cache misses.

Software PREFETCH operations work the same way as do load from memory operations, with the following exceptions:

- Software PREFETCH instructions retire after virtual to physical address translation is completed.
- If an exception, such as page fault, is required to prefetch the data, then the software prefetch instruction retires without prefetching data.
- Avoid specifying a NULL address for software prefetches.

3.7.2 Hardware Prefetching for First-Level Data Cache

The hardware prefetching mechanism for L1 in Intel Core microarchitecture is discussed in Section 2.4.4.2.

Example 3-54 depicts a technique to trigger hardware prefetch. The code demonstrates traversing a linked list and performing some computational work on 2 members of each element that reside in 2 different cache lines. Each element is of size 192 bytes. The total size of all elements is larger than can be fitted in the L2 cache.

Example 3-54. Using DCU Hardware Prefetch

Original code	Modified sequence benefit from prefetch

mov ebx, DWORD PTR [First]
xor eax, eax
scan_list:
mov eax, [ebx+4]
mov ecx, 60
	mov ebx, DWORD PTR [First]
	xor eax, eax
	scan_list:
	mov eax, [ebx+4]
	mov eax, [ebx+4]
	mov eax, [ebx+4]
	mov ecx, 60

do_some_work_1:
add eax, eax
and eax, 6
sub ecx, 1
jnz do_some_work_1
mov eax, [ebx+64]
mov ecx, 30
do_some_work_2:
add eax, eax
and eax, 6
sub ecx, 1
jnz do_some_work_2
 	
mov ebx, [ebx]
test ebx, ebx
jnz scan_list
	do_some_work_1:
	add eax, eax
	and eax, 6
	sub ecx, 1
	jnz do_some_work_1
	mov eax, [ebx+64]
	mov ecx, 30
	do_some_work_2:
	add eax, eax
	and eax, 6
	sub ecx, 1
	jnz do_some_work_2
	
	mov ebx, [ebx]
	test ebx, ebx
	jnz scan_list

The additional instructions to load data from one member in the modified sequence can trigger the DCU hardware prefetch mechanisms to prefetch data in the next cache line, enabling the work on the second member to complete sooner.

Software can gain from the first-level data cache prefetchers in two cases:

- If data is not in the second-level cache, the first-level data cache prefetcher enables early trigger of the second-level cache prefetcher.
- If data is in the second-level cache and not in the first-level data cache, then the first-level data cache prefetcher triggers earlier data bring-up of sequential cache line to the first-level data cache.

There are situations that software should pay attention to a potential side effect of triggering unnecessary DCU hardware prefetches. If a large data structure with many members spanning many cache lines is accessed in ways that only a few of its members are actually referenced, but there are multiple pair accesses to the same cache line. The DCU hardware prefetcher can trigger fetching of cache lines that are not needed. In Example , references to the "Pts" array and "AltPts" will trigger DCU prefetch to fetch additional cache lines that won't be needed. If significant negative performance impact is detected due to DCU hardware prefetch on a portion of the code, software can try to reduce the size of that contemporaneous working set to be less than half of the L2 cache.

Example 3-55. Avoid Causing DCU Hardware Prefetch to Fetch Un-needed Lines

```
while ( CurrBond != NULL )
   {
   MyATOM  *a1 = CurrBond->At1 ;
   MyATOM  *a2 = CurrBond->At2 ;


   if ( a1->CurrStep <= a1->LastStep &&
     a2->CurrStep <= a2->LastStep
     )
       {
       a1->CurrStep++ ;
       a2->CurrStep++ ;

       double  ux = a1->Pts[0].x - a2->Pts[0].x ;
       double  uy = a1->Pts[0].y - a2->Pts[0].y ;
       double  uz = a1->Pts[0].z - a2->Pts[0].z ;
       a1->AuxPts[0].x += ux ;
       a1->AuxPts[0].y += uy ;
       a1->AuxPts[0].z += uz ;


       a2->AuxPts[0].x += ux ;
       a2->AuxPts[0].y += uy ;
       a2->AuxPts[0].z += uz ;
       } ;
   CurrBond = CurrBond->Next ;
   } ;
```

To fully benefit from these prefetchers, organize and access the data using one of the following methods:

Method 1:

- Organize the data so consecutive accesses can usually be found in the same 4-KByte page.
- Access the data in constant strides forward or backward IP Prefetcher.

Method 2:

- Organize the data in consecutive lines.
- Access the data in increasing addresses, in sequential cache lines.

Example demonstrates accesses to sequential cache lines that can benefit from the first-level cache prefetcher.

Example 3-56. Technique For Using L1 Hardware Prefetch

```
unsigned int *p1, j, a, b;
for (j = 0; j < num; j += 16)
{
a = p1[j];
b = p1[j+1];
// Use these two values
}
```

By elevating the load operations from memory to the beginning of each iteration, it is likely that a significant part of the latency of the pair cache line transfer from memory to the second-level cache will be in parallel with the transfer of the first cache line.

The IP prefetcher uses only the lower 8 bits of the address to distinguish a specific address. If the code size of a loop is bigger than 256 bytes, two loads may appear similar in the lowest 8 bits and the IP prefetcher will be restricted. Therefore, if you have a loop bigger than 256 bytes, make sure that no two loads have the same lowest 8 bits in order to use the IP prefetcher.

3.7.3 Hardware Prefetching for Second-Level Cache

The Intel Core microarchitecture contains two second-level cache prefetchers:

- **Streamer** — Loads data or instructions from memory to the second-level cache. To use the streamer, organize the data or instructions in blocks of 128 bytes, aligned on 128 bytes. The first access to one of the two cache lines in this block while it is in memory triggers the streamer to prefetch the pair line. To software, the L2 streamer's functionality is similar to the adjacent cache line prefetch mechanism found in processors based on Intel NetBurst microarchitecture.
- **Data prefetch logic (DPL)** — DPL and L2 Streamer are triggered only by writeback memory type. They prefetch only inside page boundary (4 KBytes). Both L2 prefetchers can be triggered by software prefetch instructions and by prefetch request from DCU prefetchers. DPL can also be triggered by read for ownership (RFO) operations. The L2 Streamer can also be triggered by DPL requests for L2 cache misses.

Software can gain from organizing data both according to the instruction pointer and according to line strides. For example, for matrix calculations, columns can be prefetched by IP-based prefetches, and rows can be prefetched by DPL and the L2 streamer.

3.7.4 Cacheability Instructions

SSE2 provides additional cacheability instructions that extend those provided in SSE. The new cacheability instructions include:

- New streaming store instructions.
- New cache line flush instruction.
- New memory fencing instructions.

For more information, see Chapter 7, "Optimizing Cache Usage."

3.7.5 REP Prefix and Data Movement

The REP prefix is commonly used with string move instructions for memory related library functions such as MEMCPY (using REP MOVSD) or MEMSET (using REP STOS). These STRING/MOV instructions with the REP prefixes are implemented in MS-ROM and have several implementation variants with different performance levels.

The specific variant of the implementation is chosen at execution time based on data layout, alignment and the counter (ECX) value. For example, MOVSB/STOSB with the REP prefix should be used with counter value less than or equal to three for best performance.

String MOVE/STORE instructions have multiple data granularities. For efficient data movement, larger data granularities are preferable. This means better efficiency can be achieved by decomposing an arbitrary counter value into a number of doublewords plus single byte moves with a count value less than or equal to 3.

Because software can use SIMD data movement instructions to move 16 bytes at a time, the following paragraphs discuss general guidelines for designing and implementing high-performance library functions such as MEMCPY(), MEMSET(), and MEMMOVE(). Four factors are to be considered:

- **Throughput per iteration** — If two pieces of code have approximately identical path lengths, efficiency favors choosing the instruction that moves larger pieces of data per iteration. Also, smaller code size per iteration will in general reduce overhead and improve throughput. Sometimes, this may involve a comparison of the relative overhead of an iterative loop structure versus using REP prefix for iteration.
- **Address alignment** — Data movement instructions with highest throughput usually have alignment restrictions, or they operate more efficiently if the destination address is aligned to its natural data size. Specifically, 16-byte moves need to ensure the destination address is aligned to 16-byte boundaries, and 8-bytes moves perform better if the destination address is aligned to 8-byte boundaries. Frequently, moving at doubleword granularity performs better with addresses that are 8-byte aligned.
- **REP string move vs. SIMD move** — Implementing general-purpose memory functions using SIMD extensions usually requires adding some prolog code to ensure the availability of SIMD instructions, preamble code to facilitate aligned data movement requirements at runtime. Throughput comparison must also take into consideration the overhead of the prolog when considering a REP string implementation versus a SIMD approach.
- **Cache eviction** — If the amount of data to be processed by a memory routine approaches half the size of the last level on-die cache, temporal locality of the cache may suffer. Using streaming store instructions (for example: MOVNTQ, MOVNTDQ) can minimize the effect of flushing the cache. The threshold to start using a streaming store depends on the size of the last level cache. Determine the size using the deterministic cache parameter leaf of CPUID.

  Techniques for using streaming stores for implementing a MEMSET()-type library must also consider that the application can benefit from this technique only if it has no immediate need to reference the target addresses. This assumption is easily upheld when testing a streaming-store implementation on a micro-benchmark configuration, but violated in a full-scale application situation.

When applying general heuristics to the design of general-purpose, high-performance library routines, the following guidelines can are useful when optimizing an arbitrary counter value N and address alignment. Different techniques may be necessary for optimal performance, depending on the magnitude of N:

- When N is less than some small count (where the small count threshold will vary between microarchitectures -- empirically, 8 may be a good value when optimizing for Intel NetBurst microarchitecture), each case can be coded directly without the overhead of a looping structure. For example, 11 bytes can be processed using two MOVSD instructions explicitly and a MOVSB with REP counter equaling 3.
- When N is not small but still less than some threshold value (which may vary for different microarchitectures, but can be determined empirically), an SIMD implementation using run-time CPUID and alignment prolog will likely deliver less throughput due to the overhead of the prolog. A REP string implementation should favor using a REP string of doublewords. To improve address alignment, a small piece of prolog code using MOVSB/STOSB with a count less than 4 can be used to peel off the non-aligned data moves before starting to use MOVSD/STOSD.

- When N is less than half the size of last level cache, throughput consideration may favor either:
  - — An approach using a REP string with the largest data granularity because a REP string has little overhead for loop iteration, and the branch misprediction overhead in the prolog/epilogue code to handle address alignment is amortized over many iterations.
  - — An iterative approach using the instruction with largest data granularity, where the overhead for SIMD feature detection, iteration overhead, and prolog/epilogue for alignment control can be minimized. The trade-off between these approaches may depend on the microarchitecture.

  An example of MEMSET() implemented using stosd for arbitrary counter value with the destination address aligned to doubleword boundary in 32-bit mode is shown in Example 3-57.
- When N is larger than half the size of the last level cache, using 16-byte granularity streaming stores with prolog/epilog for address alignment will likely be more efficient, if the destination addresses will not be referenced immediately afterwards.

Example 3-57. REP STOSD with Arbitrary Count Size and 4-Byte-Aligned Destination

A 'C' example of Memset()	Equivalent Implementation Using REP STOSD

void memset(void *dst,int c,size_t size)
{
char *d = (char *)dst;
size_t i;
for (i=0;i<size;i++)
    *d++ = (char)c;
}
	push edi
	movzx eax, byte ptr [esp+12]
	mov ecx, eax
	shl ecx, 8
	or ecx, eax
	mov ecx, eax
	shl ecx, 16
	or eax, ecx
	mov edi, [esp+8] ; 4-byte aligned
	mov ecx, [esp+16] ; byte count
	shr ecx, 2 ; do dword
	cmp ecx, 127
	jle _main
	test edi, 4
	jz _main
	stosd ;peel off one dword
	dec ecx
	_main: ; 8-byte aligned
	rep stosd
	mov ecx, [esp + 16]
	and ecx, 3 ; do count <= 3
	rep stosb ; optimal with <= 3
	pop edi
	ret

Memory routines in the runtime library generated by Intel compilers are optimized across a wide range of address alignments, counter values, and microarchitectures. In most cases, applications should take advantage of the default memory routines provided by Intel compilers.

In some situations, the byte count of the data is known by the context (as opposed to being known by a parameter passed from a call), and one can take a simpler approach than those required for a general-purpose library routine. For example, if the byte count is also small, using REP MOVSB/STOSB with a count less than four can ensure good address alignment and loop-unrolling to finish the remaining data; using MOVSD/STOSD can reduce the overhead associated with iteration.

Using a REP prefix with string move instructions can provide high performance in the situations described above. However, using a REP prefix with string scan instructions (SCASB, SCASW, SCASD, SCASQ) or compare instructions (CMPSB, CMPSW, SMPSD, SMPSQ) is not recommended for high performance. Consider using SIMD instructions instead.

3.7.6 Enhanced REP MOVSB and STOSB operation (ERMSB)

Beginning with processors based on Intel microarchitecture code name Ivy Bridge, REP string operation using MOVSB and STOSB can provide both flexible and high-performance REP string operations for software in common situations like memory copy and set operations. Processors that provide enhanced MOVSB/STOSB operations are enumerated by the CPUID feature flag: CPUID:(EAX=7H, ECX=0H):EBX.ERMSB[bit 9] = 1.

3.7.6.1 Memcpy Considerations

The interface for the standard library function memcpy introduces several factors (e.g. length, alignment of the source buffer and destination) that interact with microarchitecture to determine the performance characteristics of the implementation of the library function. Two of the common approaches to implement memcpy are driven from small code size vs. maximum throughput. The former generally uses REP MOVSD+B (see Section 3.7.5), while the latter uses SIMD instruction sets and has to deal with additional data alignment restrictions.

For processors supporting enhanced REP MOVSB/STOSB, implementing memcpy with REP MOVSB will provide even more compact benefits in code size and better throughput than using the combination of REP MOVSD+B. For processors based on Intel microarchitecture code name Ivy Bridge, implementing memcpy using ERMSB might not reach the same level of throughput as using 256-bit or 128-bit AVX alternatives, depending on length and alignment factors.

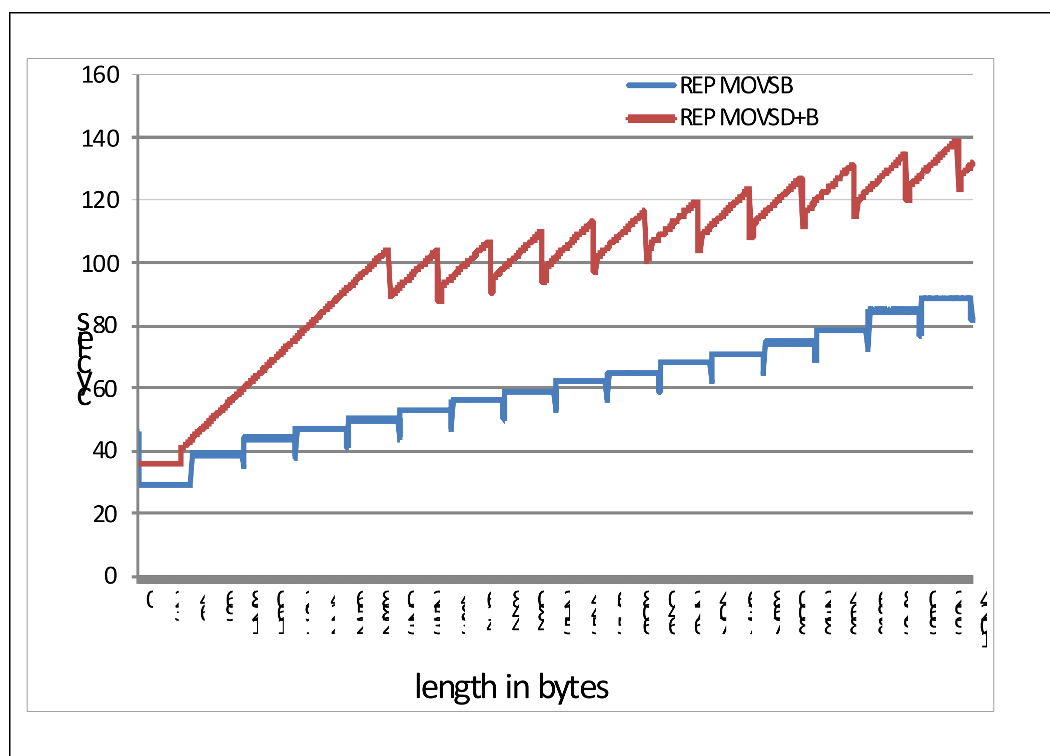


Figure 3-4. Memcpy Performance Comparison for Lengths up to 2KB

Figure 3-4 depicts the relative performance of memcpy implementation on a third-generation Intel Core processor using ERMSB versus REP MOVSD+B, for alignment conditions when both the source and destination addresses are aligned to a 16-Byte boundary and the source region does not overlap with the destination region. Using ERMSB always delivers better performance than using REP MOVSD+B. If the length is a multiple of 64, it can produce even higher performance. For example, copying 65-128 bytes takes 40 cycles, while copying 128 bytes needs only 35 cycles.

If an application wishes to bypass standard memcpy library implementation with its own custom implementation and have freedom to manage the buffer length allocation for both source and destination, it may be worthwhile to manipulate the lengths of its memory copy operation to be multiples of 64 to take advantage the code size and performance benefit of ERMSB.

The performance characteristic of implementing a general-purpose memcpy library function using a SIMD register is significantly more colorful than an equivalent implementation using a general-purpose register, depending on length, instruction set selection between SSE2, 128-bit AVX, 256-bit AVX, relative alignment of source/destination, and memory address alignment granularities/boundaries, etc.

Hence comparing performance characteristics between a memcpy using ERMSB versus a SIMD implementation is highly dependent on the particular SIMD implementation. The remainder of this section discusses the relative performance of memcpy using ERMSB versus unpublished, optimized 128-bit AVX implementation of memcpy to illustrate the hardware capability of Intel microarchitecture code name Ivy Bridge.

Table 3-4. Relative Performance of Memcpy() Using ERMSB Vs. 128-bit AVX

Range of Lengths (bytes)	<128	128 to 2048	2048 to 4096
Memcpy_ERMSB/Memcpy_AVX128	0x7X	1X	1.02X

Table 3-4 shows the relative performance of the Memcpy function implemented using enhanced REP MOVSB versus 128-bit AVX for several ranges of memcpy lengths, when both the source and destination addresses are 16-byte aligned and the source region and destination region do not overlap. For memcpy length less than 128 bytes, using ERMSB is slower than what's possible using 128-bit AVX, due to internal start-up overhead in the REP string.

For situations with address misalignment, memcpy performance will generally be reduced relative to the 16-byte alignment scenario (see Table 3-5).

.

Table 3-5. Effect of Address Misalignment on Memcpy() Performance

Address Misalignment	Performance Impact
Source Buffer	The impact on ERMSB implementation versus 128-bit AVX is similar.
Destination Buffer	The impact on ERMSB implementation can be 25% degradation, while 128-bit AVX implementation of memcpy may degrade only 5%, relative to 16-byte aligned scenario.

Memcpy() implemented with ERMSB can benefit further from the 256-bit SIMD integer data-path on the Haswell microarchitecture. see Section 11.16.3.

3.7.6.2 Memmove Considerations

When there is an overlap between the source and destination regions, software may need to use memmove instead of memcpy to ensure correctness. It is possible to use REP MOVSB in conjunction with the direction flag (DF) in a memmove() implementation to handle situations where the latter part of the source region overlaps with the beginning of the destination region. However, setting the DF to force REP MOVSB to copy bytes from high towards low addresses will experience significant performance degradation.

When using ERMSB to implement memmove function, one can detect the above situation and handle first the rear chunks in the source region that will be written to as part of the destination region, using REP MOVSB with the DF=0, to the non-overlapping region of the destination. After the overlapping chunks in the rear section are copied, the rest of the source region can be processed normally, also with DF=0.

3.7.6.3 Memset Considerations

The consideration of code size and throughput also applies for memset() implementations. For processors supporting ERMSB, using REP STOSB will again deliver more compact code size and significantly better performance than the combination of STOSD+B technique described in Section 3.7.5.

When the destination buffer is 16-byte aligned, memset() using ERMSB can perform better than SIMD approaches. When the destination buffer is misaligned, memset() performance using ERMSB can degrade about 20% relative to aligned case, for processors based on Intel microarchitecture code name Ivy Bridge. In contrast, SIMD implementation of memset() will experience smaller degradation when the destination is misaligned.

Memset() implemented with ERMSB can benefit further from the 256-bit data path on the Haswell microarchitecture. see Section 11.16.3.3.

3.8 FLOATING-POINT CONSIDERATIONS

When programming floating-point applications, it is best to start with a high-level programming language such as C, C++, or Fortran. Many compilers perform floating-point scheduling and optimization when it is possible. However in order to produce optimal code, the compiler may need some assistance.

3.8.1 Guidelines for Optimizing Floating-point Code

User/Source Coding Rule 13. (M impact, M generality) *Enable the compiler's use of SSE, SSE2 and more advanced SIMD instruction sets (e.g. AVX) with appropriate switches. Favor scalar SIMD code generation to replace x87 code generation.*

Follow this procedure to investigate the performance of your floating-point application:

- Understand how the compiler handles floating-point code.
- Look at the assembly dump and see what transforms are already performed on the program.
- Study the loop nests in the application that dominate the execution time.
- Determine why the compiler is not creating the fastest code.
- See if there is a dependence that can be resolved.
- Determine the problem area: bus bandwidth, cache locality, trace cache bandwidth, or instruction latency. Focus on optimizing the problem area. For example, adding PREFETCH instructions will not help if the bus is already saturated. If trace cache bandwidth is the problem, added prefetch µops may degrade performance.

Also, in general, follow the general coding recommendations discussed in this chapter, including:

- Blocking the cache.
- Using prefetch.
- Enabling vectorization.
- Unrolling loops.

User/Source Coding Rule 14. (H impact, ML generality) *Make sure your application stays in range to avoid denormal values, underflows.*

Out-of-range numbers cause very high overhead.

When converting floating-point values to 16-bit, 32-bit, or 64-bit integers using truncation, the instructions CVTTSS2SI and CVTTSD2SI are recommended over instructions that access x87 FPU stack. This avoids changing the rounding mode.

User/Source Coding Rule 15. (M impact, ML generality) *Usually, math libraries take advantage of the transcendental instructions (for example, FSIN) when evaluating elementary functions. If there is no critical need to evaluate the transcendental functions using the extended precision of 80 bits, applications should consider an alternate, software-based approach, such as a look-up-table-based algorithm using interpolation techniques. It is possible to improve transcendental performance with these techniques by choosing the desired numeric precision and the size of the look-up table, and by taking advantage of the parallelism of the SSE and the SSE2 instructions.*

3.8.2 Microarchitecture Specific Considerations

3.8.2.1 Long-Latency FP Instructions

In the Haswell microarchitecture, long-latency floating-point instructions for division, square root operations have continued the improvements from the Ivy Bridge microarchitecture with pipe-lined hardware implementation. These improvement will benefit existing code transparently.

3.8.2.2 Miscellaneous Instructions

In the Haswell microarchitecture, SIMD floating-point compare and set flag instructions (COMISD/SS, UCOMISD/SS) is a one micro-op implementation for the register/register flavor of these instructions. The latency is increased slightly to 3 cycles.

The ROUNDPD/SD instructions (both AVX and SSE4.1) are implemented as two micro-ops with latency increased to 6 cycles. This can have a measurable impact to math library functions using these instructions to polynomial evaluation of exponentiation.

3.8.3 Floating-point Modes and Exceptions

When working with floating-point numbers, high-speed microprocessors frequently must deal with situations that need special handling in hardware or code.

3.8.3.1 Floating-point Exceptions

The most frequent cause of performance degradation is the use of masked floating-point exception conditions such as:

- Arithmetic overflow.
- Arithmetic underflow.
- Denormalized operand.

Refer to Chapter 4 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for definitions of overflow, underflow and denormal exceptions.

Denormalized floating-point numbers impact performance in two ways:

- Directly when are used as operands.
- Indirectly when are produced as a result of an underflow situation.

If a floating-point application never underflows, the denormals can only come from floating-point constants.

User/Source Coding Rule 16. (H impact, ML generality) *Denormalized floating-point constants should be avoided as much as possible.*

Denormal and arithmetic underflow exceptions can occur during the execution of x87 instructions or SSE/SSE2/SSE3 instructions. Processors based on Intel NetBurst microarchitecture handle these exceptions more efficiently when executing SSE/SSE2/SSE3 instructions and when speed is more important than complying with the IEEE standard. The following paragraphs give recommendations on how to optimize your code to reduce performance degradations related to floating-point exceptions.

3.8.3.2 Dealing with floating-point exceptions in x87 FPU code

Every special situation listed in Section 3.8.3.1, "Floating-point Exceptions," is costly in terms of performance. For that reason, x87 FPU code should be written to avoid these situations.

There are basically three ways to reduce the impact of overflow/underflow situations with x87 FPU code:

- Choose floating-point data types that are large enough to accommodate results without generating arithmetic overflow and underflow exceptions.
- Scale the range of operands/results to reduce as much as possible the number of arithmetic overflow/underflow situations.
- Keep intermediate results on the x87 FPU register stack until the final results have been computed and stored in memory. Overflow or underflow is less likely to happen when intermediate results are kept in the x87 FPU stack (this is because data on the stack is stored in double extended-precision format and overflow/underflow conditions are detected accordingly).
- Denormalized floating-point constants (which are read-only, and hence never change) should be avoided and replaced, if possible, with zeros of the same sign.

3.8.3.3 Floating-point Exceptions in SSE/SSE2/SSE3 Code

Most special situations that involve masked floating-point exceptions are handled efficiently in hardware. When a masked overflow exception occurs while executing SSE/SSE2/SSE3 code, processor hardware can handles it without performance penalty.

Underflow exceptions and denormalized source operands are usually treated according to the IEEE 754 specification, but this can incur significant performance delay. If a programmer is willing to trade pure IEEE 754 compliance for speed, two non-IEEE 754 compliant modes are provided to speed situations where underflows and input are frequent: FTZ mode and DAZ mode.

When the FTZ mode is enabled, an underflow result is automatically converted to a zero with the correct sign. Although this behavior is not compliant with IEEE 754, it is provided for use in applications where performance is more important than IEEE 754 compliance. Since denormal results are not produced when the FTZ mode is enabled, the only denormal floating-point numbers that can be encountered in FTZ mode are the ones specified as constants (read only).

The DAZ mode is provided to handle denormal source operands efficiently when running a SIMD floating-point application. When the DAZ mode is enabled, input denormals are treated as zeros with the same sign. Enabling the DAZ mode is the way to deal with denormal floating-point constants when performance is the objective.

If departing from the IEEE 754 specification is acceptable and performance is critical, run SSE/SSE2/SSE3 applications with FTZ and DAZ modes enabled.

NOTE

The DAZ mode is available with both the SSE and SSE2 extensions, although the speed improvement expected from this mode is fully realized only in SSE code.

3.8.4 Floating-point Modes

For x87 code, using the FLDCW instruction to change floating modes can be an expensive operation in many cases.

Recent processor generations provide hardware optimization for FLDCW that allows programmers to alternate between two constant values efficiently. For the FLDCW optimization to be effective, the two constant FCW values are only allowed to differ on the following 5 bits in the FCW:

```
FCW[8-9]     ; Precision control
FCW[10-11]   ; Rounding control
FCW[12]      ; Infinity control
```

If programmers need to modify other bits (for example: mask bits) in the FCW, the FLDCW instruction is still an expensive operation.

In situations where an application cycles between three (or more) constant values, FLDCW optimization does not apply, and the performance degradation occurs for each FLDCW instruction.

One solution to this problem is to choose two constant FCW values, take advantage of the optimization of the FLDCW instruction to alternate between only these two constant FCW values, and devise some means to accomplish the task that requires the 3rd FCW value without actually changing the FCW to a third constant value. An alternative solution is to structure the code so that, for periods of time, the application alternates between only two constant FCW values. When the application later alternates between a pair of different FCW values, the performance degradation occurs only during the transition.

It is expected that SIMD applications are unlikely to alternate between FTZ and DAZ mode values. Consequently, the SIMD control word does not have the short latencies that the floating-point control register does. A read of the MXCSR register has a fairly long latency, and a write to the register is a serializing instruction.

There is no separate control word for single and double precision; both use the same modes. Notably, this applies to both FTZ and DAZ modes.

Assembly/Compiler Coding Rule 60. (H impact, M generality) *Minimize changes to bits 8-12 of the floating-point control word. Changes for more than two values (each value being a combination of the following bits: precision, rounding and infinity control, and the rest of bits in FCW) leads to delays that are on the order of the pipeline depth.*

3.8.4.1 Rounding Mode

Many libraries provide float-to-integer library routines that convert floating-point values to integer. Many of these libraries conform to ANSI C coding standards which state that the rounding mode should be truncation. With the Pentium 4 processor, one can use the CVTTSD2SI and CVTTSS2SI instructions to convert operands with truncation without ever needing to change rounding modes. The cost savings of using these instructions over the methods below is enough to justify using SSE and SSE2 wherever possible when truncation is involved.

For x87 floating-point, the FIST instruction uses the rounding mode represented in the floating-point control word (FCW). The rounding mode is generally “round to nearest”, so many compiler writers implement a change in the rounding mode in the processor in order to conform to the C and FORTRAN standards. This implementation requires changing the control word on the processor using the FLDCW instruction. For a change in the rounding, precision, and infinity bits, use the FSTCW instruction to store the floating-point control word. Then use the FLDCW instruction to change the rounding mode to truncation.

In a typical code sequence that changes the rounding mode in the FCW, a FSTCW instruction is usually followed by a load operation. The load operation from memory should be a 16-bit operand to prevent store-forwarding problem. If the load operation on the previously-stored FCW word involves either an 8-bit or a 32-bit operand, this will cause a store-forwarding problem due to mismatch of the size of the data between the store operation and the load operation.

To avoid store-forwarding problems, make sure that the write and read to the FCW are both 16-bit operations.

If there is more than one change to the rounding, precision, and infinity bits, and the rounding mode is not important to the result, use the algorithm in Example 3-58 to avoid synchronization issues, the overhead of the FLDCW instruction, and having to change the rounding mode. Note that the example suffers

from a store-forwarding problem which will lead to a performance penalty. However, its performance is still better than changing the rounding, precision, and infinity bits among more than two values.

Example 3-58. Algorithm to Avoid Changing Rounding Mode

```
_fto132proc
    lea     ecx, [esp-8]
    sub     esp, 16             ; Allocate frame
    and     ecx, -8             ; Align pointer on boundary of 8
    fld     st(0)               ; Duplicate FPU stack top

    fistp   qword ptr[ecx]
    fild    qword ptr[ecx]
    mov     edx, [ecx+4]        ; High DWORD of integer
    mov     eax, [ecx]          ; Low DWIRD of integer
    test    eax, eax
    je      integer_QnaN_or_zero
```

```
arg_is_not_integer_QnaN:
    fsubp   st(1), st           ; TOS=d-round(d), { st(1) = st(1)-st & pop ST}
    test    edx, edx            ; What's sign of integer
    jns     positive            ; Number is negative
    fstp    dword ptr[ecx]      ; Result of subtraction
    mov     ecx, [ecx]          ; DWORD of diff(single-precision)
    add     esp, 16
    xor     ecx, 80000000h
    add     ecx,7fffffffh       ; If diff<0 then decrement integer
    adc     eax,0               ; INC EAX (add CARRY flag)
    ret
positive:

    positive:
    fstp    dword ptr[ecx]      ; 17-18 result of subtraction
    mov     ecx, [ecx]          ; DWORD of diff(single precision)
    add     esp, 16
    add     ecx, 7fffffffh      ; If diff<0 then decrement integer
    sbb     eax, 0              ; DEC EAX (subtract CARRY flag)
    ret
integer_QnaN_or_zero:
    test    edx, 7fffffffh
    jnz     arg_is_not_integer_QnaN
    add  esp, 16
    ret
```

Assembly/Compiler Coding Rule 61. (H impact, L generality) *Minimize the number of changes to the rounding mode. Do not use changes in the rounding mode to implement the floor and ceiling functions if this involves a total of more than two values of the set of rounding, precision, and infinity bits.*

3.8.4.2 Precision

If single precision is adequate, use it instead of double precision. This is true because:

- Single precision operations allow the use of longer SIMD vectors, since more single precision data elements can fit in a register.
- If the precision control (PC) field in the x87 FPU control word is set to single precision, the floating-point divider can complete a single-precision computation much faster than either a double-precision

computation or an extended double-precision computation. If the PC field is set to double precision, this will enable those x87 FPU operations on double-precision data to complete faster than extended double-precision computation. These characteristics affect computations including floating-point divide and square root.

Assembly/Compiler Coding Rule 62. (H impact, L generality) *Minimize the number of changes to the precision mode.*

3.8.5 x87 vs. Scalar SIMD Floating-point Trade-offs

There are a number of differences between x87 floating-point code and scalar floating-point code (using SSE and SSE2). The following differences should drive decisions about which registers and instructions to use:

- When an input operand for a SIMD floating-point instruction contains values that are less than the representable range of the data type, a denormal exception occurs. This causes a significant performance penalty. An SIMD floating-point operation has a flush-to-zero mode in which the results will not underflow. Therefore subsequent computation will not face the performance penalty of handling denormal input operands. For example, in the case of 3D applications with low lighting levels, using flush-to-zero mode can improve performance by as much as 50% for applications with large numbers of underflows.
- Scalar floating-point SIMD instructions have lower latencies than equivalent x87 instructions. Scalar SIMD floating-point multiply instruction may be pipelined, while x87 multiply instruction is not.
- Although x87 supports transcendental instructions, software library implementation of transcendental function can be faster in many cases.
- x87 supports 80-bit precision, double extended floating-point. SSE support a maximum of 32-bit precision. SSE2 supports a maximum of 64-bit precision.
- Scalar floating-point registers may be accessed directly, avoiding FXCH and top-of-stack restrictions.
- The cost of converting from floating-point to integer with truncation is significantly lower with Streaming SIMD Extensions 2 and Streaming SIMD Extensions in the processors based on Intel NetBurst microarchitecture than with either changes to the rounding mode or the sequence prescribed in the Example 3-58.

Assembly/Compiler Coding Rule 63. (M impact, M generality) *Use Streaming SIMD Extensions 2 or Streaming SIMD Extensions unless you need an x87 feature. Most SSE2 arithmetic operations have shorter latency then their X87 counterpart and they eliminate the overhead associated with the management of the X87 register stack.*

3.8.5.1 Scalar SSE/SSE2

In code sequences that have conversions from floating-point to integer, divide single-precision instructions, or any precision change, x87 code generation from a compiler typically writes data to memory in single-precision and reads it again in order to reduce precision. Using SSE/SSE2 scalar code instead of x87 code can generate a large performance benefit using Intel NetBurst microarchitecture and a modest benefit on Intel Core Solo and Intel Core Duo processors.

Recommendation: Use the compiler switch to generate scalar floating-point code using XMM rather than x87 code.

When working with scalar SSE/SSE2 code, pay attention to the need for clearing the content of unused slots in an XMM register and the associated performance impact. For example, loading data from memory with MOVSS or MOVSD causes an extra micro-op for zeroing the upper part of the XMM register.

3.8.5.2 Transcendental Functions

If an application needs to emulate math functions in software for performance or other reasons (see Section 3.8.1, “Guidelines for Optimizing Floating-point Code”), it may be worthwhile to inline math

library calls because the CALL and the prologue/epilogue involved with such calls can significantly affect the latency of operations.

3.9 MAXIMIZING PCIE PERFORMANCE

PCIe performance can be dramatically impacted by the size and alignment of upstream reads and writes (read and write transactions issued from a PCIe agent to the host's memory). As a general rule, the best performance, in terms of both bandwidth and latency, is obtained by aligning the start addresses of upstream reads and writes on 64-byte boundaries and ensuring that the request size is a multiple of 64-bytes, with modest further increases in bandwidth when larger multiples (128, 192, 256 bytes) are employed. In particular, a partial write will cause a delay for the following request (read or write).

A second rule is to avoid multiple concurrently outstanding accesses to a single cache line. This can result in a conflict which in turn can cause serialization of accesses that would otherwise be pipelined, resulting in higher latency and/or lower bandwidth. Patterns that violate this rule include sequential accesses (reads or writes) that are not a multiple of 64-bytes, as well as explicit accesses to the same cache line address. Overlapping requests—those with different start addresses but with request lengths that result in overlap of the requests—can have the same effect. For example, a 96-byte read of address 0x00000200 followed by a 64-byte read of address 0x00000240 will cause a conflict—and a likely delay—for the second read.

Upstream writes that are a multiple of 64-byte but are non-aligned will have the performance of a series of partial and full sequential writes. For example, a write of length 128-byte to address 0x00000070 will perform similarly to 3 sequential writes of lengths 16, 64, and 48 to addresses 0x00000070, 0x00000080, and 0x00000100, respectively.

For PCIe cards implementing multi-function devices, such as dual or quad port network interface cards (NICs) or dual-GPU graphics cards, it is important to note that non-optimal behavior by one of those devices can impact the bandwidth and/or latency observed by the other devices on that card. With respect to the behavior described in this section, all traffic on a given PCIe port is treated as if it originated from a single device and function.

For the best PCIe bandwidth:

1. Align start addresses of upstream reads and writes on 64-byte boundaries.
2. Use read and write requests that are a multiple of 64-bytes.
3. Eliminate or avoid sequential and random partial line upstream writes.
4. Eliminate or avoid conflicting upstream reads, including sequential partial line reads.

Techniques for avoiding performance pitfalls include cache line aligning all descriptors and data buffers, padding descriptors that are written upstream to 64-byte alignment, buffering incoming data to achieve larger upstream write payloads, allocating data structures intended for sequential reading by the PCIe device in such a way as to enable use of (multiple of) 64-byte reads. The negative impact of unoptimized reads and writes depends on the specific workload and the microarchitecture on which the product is based.

CHAPTER 4
CODING FOR SIMD ARCHITECTURES

Processors based on Intel Core microarchitecture supports MMX, SSE, SSE2, SSE3, and SSSE3. Processors based on Enhanced Intel Core microarchitecture supports MMX, SSE, SSE2, SSE3, SSSE3 and SSE4.1. Processors based on Intel microarchitecture code name Nehalem supports MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2. Processors based on Intel microarchitecture code name Westmere supports MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2 and AESNI. Processors based on Intel microarchitecture code name Sandy Bridge supports MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AESNI, PCLMULQDQ and Intel AVX.

Intel Pentium 4, Intel Xeon and Pentium M processors include support for SSE2, SSE, and MMX technology. SSE3 were introduced with the Pentium 4 processor supporting Hyper-Threading Technology at 90 nm technology. Intel Core Solo and Intel Core Duo processors support SSE3/SSE2/SSE, and MMX.

Single-instruction, multiple-data (SIMD) technologies enable the development of advanced multimedia, signal processing, and modeling applications.

Single-instruction, multiple-data techniques can be applied to text/string processing, lexing and parser applications. This is covered in Chapter 10, "SSE4.2 and SIMD Programming For Text-Processing/Lexing/Parsing". Techniques for optimizing AESNI are discussed in Section 5.10.

To take advantage of the performance opportunities presented by these capabilities, do the following:

- Ensure that the processor supports MMX technology, SSE, SSE2, SSE3, SSSE3 and SSE4.1.
- Ensure that the operating system supports MMX technology and SSE (OS support for SSE2, SSE3 and SSSE3 is the same as OS support for SSE).
- Employ the optimization and scheduling strategies described in this book.
- Use stack and data alignment techniques to keep data properly aligned for efficient memory use.
- Utilize the cacheability instructions offered by SSE and SSE2, where appropriate.

4.1 CHECKING FOR PROCESSOR SUPPORT OF SIMD TECHNOLOGIES

This section shows how to check whether a processor supports MMX technology, SSE, SSE2, SSE3, SSSE3, and SSE4.1.

SIMD technology can be included in your application in three ways:

1. Check for the SIMD technology during installation. If the desired SIMD technology is available, the appropriate DLLs can be installed.
2. Check for the SIMD technology during program execution and install the proper DLLs at runtime. This is effective for programs that may be executed on different machines.
3. Create a "fat" binary that includes multiple versions of routines; versions that use SIMD technology and versions that do not. Check for SIMD technology during program execution and run the appropriate versions of the routines. This is especially effective for programs that may be executed on different machines.

4.1.1 Checking for MMX Technology Support

If MMX technology is available, then CPUID.01H:EDX[BIT 23] = 1. Use the code segment in Example 4-1 to test for MMX technology.

Example 4-1. Identification of MMX Technology with CPUID

```
...identify existence of cpuid instruction
...                         ;
...                         ; Identify signature is genuine Intel
...                         ;
mov eax, 1                  ; Request for feature flags
cpuid                       ; 0FH, 0A2H CPUID instruction
test edx, 00800000h         ; Is MMX technology bit (bit 23) in feature flags equal to 1
jnz      Found
```

For more information on CPUID see, *Intel® Processor Identification with CPUID Instruction*, order number 241618.

4.1.2 Checking for Streaming SIMD Extensions Support

Checking for processor support of Streaming SIMD Extensions (SSE) on your processor is similar to checking for MMX technology. However, operating system (OS) must provide support for SSE states save and restore on context switches to ensure consistent application behavior when using SSE instructions.

To check whether your system supports SSE, follow these steps:

1. Check that your processor supports the CPUID instruction.
2. Check the feature bits of CPUID for SSE existence.

Example 4-2 shows how to find the SSE feature bit (bit 25) in CPUID feature flags.

Example 4-2. Identification of SSE with CPUID

```
...Identify existence of cpuid instruction
                            ; Identify signature is genuine intel
mov eax, 1                  ; Request for feature flags
cpuid                       ; 0FH, 0A2H cpuid instruction
test EDX, 002000000h        ; Bit 25 in feature flags equal to 1
jnz           Found
```

4.1.3 Checking for Streaming SIMD Extensions 2 Support

Checking for support of SSE2 is like checking for SSE support. The OS requirements for SSE2 Support are the same as the OS requirements for SSE.

To check whether your system supports SSE2, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bits of CPUID for SSE2 technology existence.

Example 4-3 shows how to find the SSE2 feature bit (bit 26) in the CPUID feature flags.

Example 4-3. Identification of SSE2 with cpuid

```
    …identify existence of cpuid instruction
    …                          ; Identify signature is genuine intel
mov eax, 1                     ; Request for feature flags
cpuid                          ; 0FH, 0A2H CPUID instruction
test EDX, 004000000h           ; Bit 26 in feature flags equal to 1
jnz      Found
```

4.1.4 Checking for Streaming SIMD Extensions 3 Support

SSE3 includes 13 instructions, 11 of those are suited for SIMD or x87 style programming. Checking for support of SSE3 instructions is similar to checking for SSE support. The OS requirements for SSE3 Support are the same as the requirements for SSE.

To check whether your system supports the x87 and SIMD instructions of SSE3, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the ECX feature bit 0 of CPUID for SSE3 technology existence.

Example 4-4 shows how to find the SSE3 feature bit (bit 0 of ECX) in the CPUID feature flags.

Example 4-4. Identification of SSE3 with CPUID

```
    …identify existence of cpuid instruction
    …                       ; Identify signature is genuine intel
mov eax, 1                  ; Request for feature flags
cpuid                       ; 0FH, 0A2H CPUID instruction
test ECX, 000000001h        ; Bit 0 in feature flags equal to 1
jnz      Found
```

Software must check for support of MONITOR and MWAIT before attempting to use MONITOR and MWAIT.Detecting the availability of MONITOR and MWAIT can be done using a code sequence similar to Example 4-4. The availability of MONITOR and MWAIT is indicated by bit 3 of the returned value in ECX.

4.1.5 Checking for Supplemental Streaming SIMD Extensions 3 Support

Checking for support of SSSE3 is similar to checking for SSE support. The OS requirements for SSSE3 support are the same as the requirements for SSE.

To check whether your system supports SSSE3, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bits of CPUID for SSSE3 technology existence.

Example 4-5 shows how to find the SSSE3 feature bit in the CPUID feature flags.

Example 4-5. Identification of SSSE3 with cpuid

```
    …Identify existence of CPUID instruction
    …                       ; Identify signature is genuine intel
mov eax, 1                  ; Request for feature flags
cpuid                       ; 0FH, 0A2H CPUID instruction
test ECX, 000000200h        ; ECX bit 9
jnz      Found
```

4.1.6 Checking for SSE4.1 Support

Checking for support of SSE4.1 is similar to checking for SSE support. The OS requirements for SSE4.1 support are the same as the requirements for SSE.

To check whether your system supports SSE4.1, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bit of CPUID for SSE4.1.

Example 4-6 shows how to find the SSE4.1 feature bit in the CPUID feature flags.

Example 4-6. Identification of SSE4.1 with cpuid

```
    ...Identify existence of CPUID instruction
    ...                         ; Identify signature is genuine intel
mov eax, 1                      ; Request for feature flags
cpuid                           ; 0FH, 0A2H CPUID instruction
test ECX, 000080000h            ; ECX bit 19
jnz      Found
```

4.1.7 Checking for SSE4.2 Support

Checking for support of SSE4.2 is similar to checking for SSE support. The OS requirements for SSE4.2 support are the same as the requirements for SSE.

To check whether your system supports SSE4.2, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bit of CPUID for SSE4.2.

Example 4-7 shows how to find the SSE4.2 feature bit in the CPUID feature flags.

Example 4-7. Identification of SSE4.2 with cpuid

```
    ...Identify existence of CPUID instruction
    ...                         ; Identify signature is genuine intel
mov eax, 1                      ; Request for feature flags
cpuid                           ; 0FH, 0A2H CPUID instruction
test ECX, 000100000h            ; ECX bit 20
jnz      Found
```

4.1.8 DetectiON of PCLMULQDQ and AESNI Instructions

Before an application attempts to use the following AESNI instructions: AESDEC/AESDECLAST/AESENC/AESENCLAST/AESIMC/AESKEYGENASSIST, it must check that the processor supports the AESNI extensions. AESNI extensions is supported if CPUID.01H:ECX.AESNI[bit 25] = 1.

Prior to using PCLMULQDQ instruction, application must check if CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1.

Operating systems that support handling SSE state will also support applications that use AESNI extensions and PCLMULQDQ instruction. This is the same requirement for SSE2, SSE3, SSSE3, and SSE4.

Example 4-8. Detection of AESNI Instructions

```
    ...Identify existence of CPUID instruction
    ...                        ; Identify signature is genuine intel
mov eax, 1                     ; Request for feature flags
cpuid                          ; 0FH, 0A2H CPUID instruction
test ECX, 002000000h           ; ECX bit 25
jnz     Found
```

Example 4-9. Detection of PCLMULQDQ Instruction

```
    ...Identify existence of CPUID instruction
    ...                        ; Identify signature is genuine intel
mov eax, 1                     ; Request for feature flags
cpuid                          ; 0FH, 0A2H CPUID instruction
test ECX, 000000002h           ; ECX bit 1
jnz     Found
```

4.1.9 Detection of AVX Instructions

Intel AVX operates on the 256-bit YMM register state. Application detection of new instruction extensions operating on the YMM state follows the general procedural flow in Figure 4-1.

Prior to using AVX, the application must identify that the operating system supports the XGETBV instruction, the YMM register state, in addition to processor's support for YMM state management using XSAVE/XRSTOR and AVX instructions. The following simplified sequence accomplishes both and is strongly recommended.

1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use[1])

2) Issue XGETBV and verify that XFEATURE_ENABLED_MASK[2:1] = '11b' (XMM state and YMM state are enabled by OS).

3) Detect CPUID.1:ECX.AVX[bit 28] = 1 (AVX instructions supported).

Note: Step 3 can be done in any order relative to 1 and 2.

1. If CPUID.01H:ECX.OSXSAVE reports 1, it also indirectly implies the processor supports XSAVE, XRSTOR, XGETBV, processor extended state bit vector XFEATURE_ENALBED_MASK register. Thus an application may streamline the checking of CPUID feature flags for XSAVE and OSXSAVE. XSETBV is a privileged instruction.

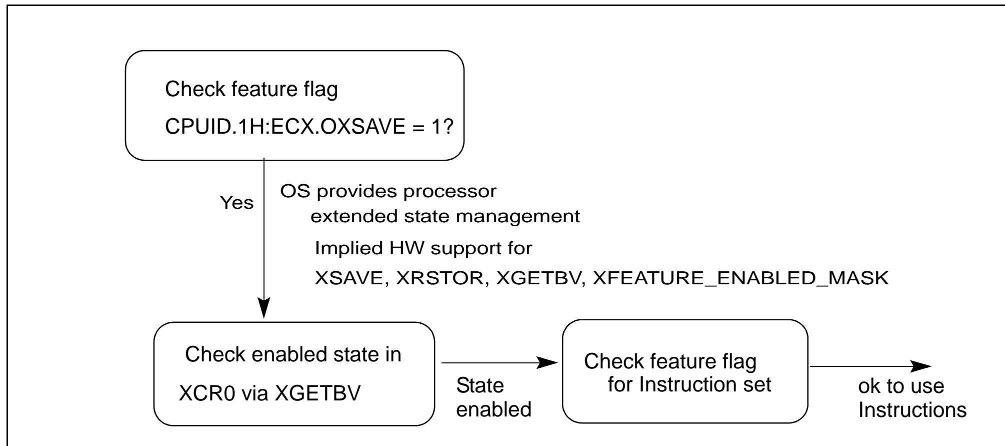


Figure 4-1. General Procedural Flow of Application Detection of AVX

The following pseudocode illustrates this recommended application AVX detection process:

Example 4-10. Detection of AVX Instruction

```
INT supports_AVX()
{    mov      eax, 1
     cpuid
     and      ecx, 018000000H
     cmp      ecx, 018000000H; check both OSXSAVE and AVX feature flags
      jne     not_supported
     ; processor supports AVX instructions and XGETBV is enabled by OS
     mov      ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
     XGETBV              ; result in EDX:EAX
     and      eax, 06H
     cmp      eax, 06H; check OS has enabled both XMM and YMM state support
     jne      not_supported
     mov      eax, 1
     jmp      done
NOT_SUPPORTED:
     mov      eax, 0
done:
```

Note: It is unwise for an application to rely exclusively on CPUID.1:ECX.AVX[bit 28] or at all on CPUID.1:ECX.XSAVE[bit 26]: These indicate hardware support but not operating system support. If YMM state management is not enabled by an operating systems, AVX instructions will #UD regardless of CPUID.1:ECX.AVX[bit 28]. "CPUID.1:ECX.XSAVE[bit 26] = 1" does not guarantee the OS actually uses the XSAVE process for state management.

4.1.10 Detection of VEX-Encoded AES and VPCLMULQDQ

VAESDEC/VAESDECLAST/VAESENC/VAESENCLAST/VAESIMC/VAESKEYGENASSIST instructions operate on YMM states. The detection sequence must combine checking for CPUID.1:ECX.AES[bit 25] = 1 and the sequence for detection application support for AVX.

Example 4-11. Detection of VEX-Encoded AESNI Instructions

```
INT supports_VAESNI()
{    mov      eax, 1
     cpuid
     and      ecx, 01A000000H
     cmp      ecx, 01A000000H; check OSXSAVE AVX and AESNI feature flags
      jne     not_supported
     ; processor supports AVX and VEX-encoded AESNI and XGETBV is enabled by OS
     mov      ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
     XGETBV            ; result in EDX:EAX
     and      eax, 06H
     cmp      eax, 06H; check OS has enabled both XMM and YMM state support
     jne      not_supported
     mov      eax, 1
     jmp      done
NOT_SUPPORTED:
     mov      eax, 0
done:
```

Similarly, the detection sequence for VPCLMULQDQ must combine checking for CPUID.1:ECX.PCLMULQDQ[bit 1] = 1 and the sequence for detection application support for AVX.

This is shown in the pseudocode:

Example 4-12. Detection of VEX-Encoded AESNI Instructions

```
INT supports_VPCLMULQDQ)
{    mov      eax, 1
     cpuid

     and      ecx, 018000002H
     cmp      ecx, 018000002H; check OSXSAVE AVX and PCLMULQDQ feature flags
      jne     not_supported
     ; processor supports AVX and VEX-encoded PCLMULQDQ and XGETBV is enabled by OS
     mov      ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
     XGETBV            ; result in EDX:EAX
     and      eax, 06H
     cmp      eax, 06H; check OS has enabled both XMM and YMM state support
     jne      not_supported
     mov      eax, 1
     jmp      done
NOT_SUPPORTED:
     mov      eax, 0
done:
```

4.1.11 Detection of F16C Instructions

Application using float 16 instruction must follow a detection sequence similar to AVX to ensure:

- The OS has enabled YMM state management support.

- The processor support AVX as indicated by the CPUID feature flag, i.e. CPUID.01H:ECX.AVX[bit 28] = 1.
- The processor support 16-bit floating-point conversion instructions via a CPUID feature flag (CPUID.01H:ECX.F16C[bit 29] = 1).

Application detection of Float-16 conversion instructions follow the general procedural flow in Figure 4-2.

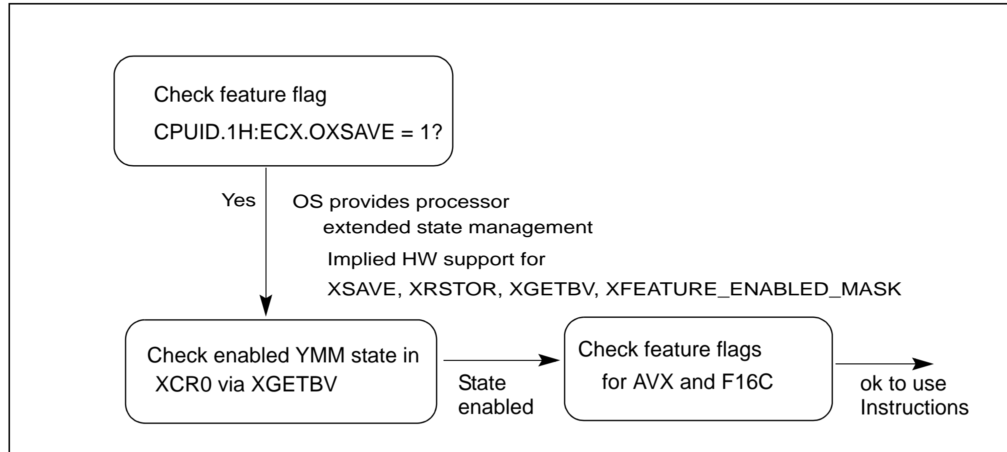


Figure 4-2. General Procedural Flow of Application Detection of Float-16

```
----------------------------------------------------------------------------------
INT supports_f16c()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 038000000H
        cmp ecx, 038000000H; check OSXSAVE, AVX, F16C feature flags
         jne not_supported
        ; processor supports AVX,F16C instructions and XGETBV is enabled by OS
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
}
-------------------------------------------------------------------------
```

4.1.12 Detection of FMA

Hardware support for FMA is indicated by CPUID.1:ECX.FMA[bit 12]=1.

Application Software must identify that hardware supports AVX, after that it must also detect support for FMA by CPUID.1:ECX.FMA[bit 12]. The recommended pseudocode sequence for detection of FMA is:

```
----------------------------------------------------------------------------------------
INT supports_fma()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 018001000H
        cmp ecx, 018001000H; check OSXSAVE, AVX, FMA feature flags
        jne not_supported
        ; processor supports AVX,FMA instructions and XGETBV is enabled by OS
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
}
-------------------------------------------------------------------------------
```

4.1.13 Detection of AVX2

Hardware support for AVX2 is indicated by CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]=1.

Application Software must identify that hardware supports AVX, after that it must also detect support for AVX2 by checking CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]. The recommended pseudocode sequence for detection of AVX2 is:

```
----------------------------------------------------------------------------------------
INT supports_avx2()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 018000000H
        cmp ecx, 018000000H; check both OSXSAVE and AVX feature flags
        jne not_supported
        ; processor supports AVX instructions and XGETBV is enabled by OS
        mov eax, 7
        mov ecx, 0
        cpuid
        and ebx, 20H
        cmp ebx, 20H; check AVX2 feature flags
        jne not_supported
```

```
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
}
-------------------------------------------------------------------------------
```

4.2 CONSIDERATIONS FOR CODE CONVERSION TO SIMD PROGRAMMING

The VTune Performance Enhancement Environment CD provides tools to aid in the evaluation and tuning. Before implementing them, you need answers to the following questions:

1. Will the current code benefit by using MMX technology, Streaming SIMD Extensions, Streaming SIMD Extensions 2, Streaming SIMD Extensions 3, or Supplemental Streaming SIMD Extensions 3?
2. Is this code integer or floating-point?
3. What integer word size or floating-point precision is needed?
4. What coding techniques should I use?
5. What guidelines do I need to follow?
6. How should I arrange and align the datatypes?

Figure 4-3 provides a flowchart for the process of converting code to MMX technology, SSE, SSE2, SSE3, or SSSE3.

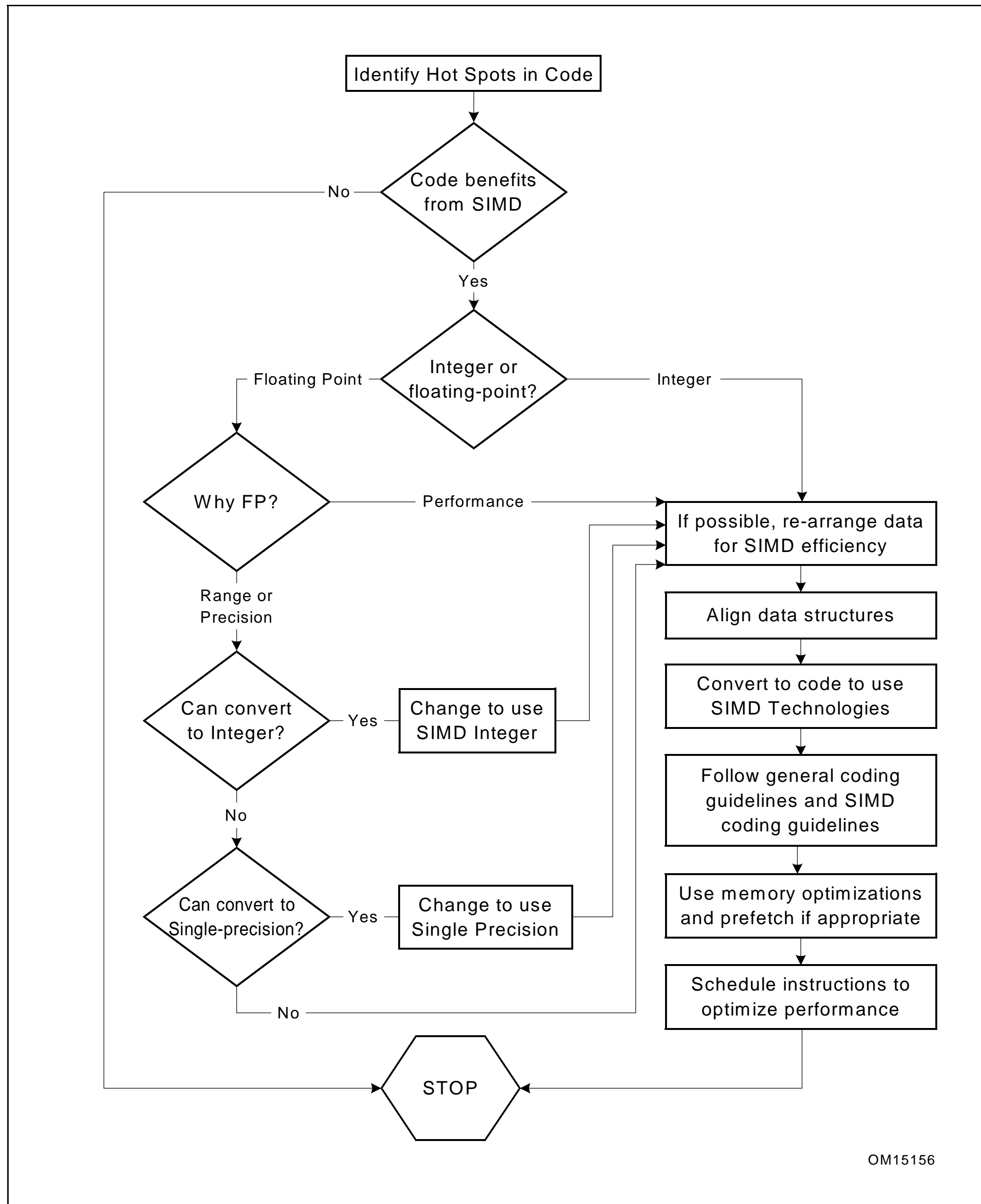


Figure 4-3. Converting to Streaming SIMD Extensions Chart

To use any of the SIMD technologies optimally, you must evaluate the following situations in your code:

- Fragments that are computationally intensive.
- Fragments that are executed often enough to have an impact on performance.
- Fragments that with little data-dependent control flow.
- Fragments that require floating-point computations.
- Fragments that can benefit from moving data 16 bytes at a time.
- Fragments of computation that can coded using fewer instructions.
- Fragments that require help in using the cache hierarchy efficiently.

4.2.1 Identifying Hot Spots

To optimize performance, use the VTune Performance Analyzer to find sections of code that occupy most of the computation time. Such sections are called the hotspots. See Appendix A, "Application Performance Tools."

The VTune analyzer provides a hotspots view of a specific module to help you identify sections in your code that take the most CPU time and that have potential performance problems. The hotspots view helps you identify sections in your code that take the most CPU time and that have potential performance problems.

The VTune analyzer enables you to change the view to show hotspots by memory location, functions, classes, or source files. You can double-click on a hotspot and open the source or assembly view for the hotspot and see more detailed information about the performance of each instruction in the hotspot.

The VTune analyzer offers focused analysis and performance data at all levels of your source code and can also provide advice at the assembly language level. The code coach analyzes and identifies opportunities for better performance of C/C++, Fortran and Java* programs, and suggests specific optimizations. Where appropriate, the coach displays pseudo-code to suggest the use of highly optimized intrinsics and functions in the Intel® Performance Library Suite. Because VTune analyzer is designed specifically for Intel architecture (IA)-based processors, including the Pentium 4 processor, it can offer detailed approaches to working with IA. See Appendix A.1.1, "Recommended Optimization Settings for Intel® 64 and IA-32 Processors," for details.

4.2.2 Determine If Code Benefits by Conversion to SIMD Execution

Identifying code that benefits by using SIMD technologies can be time-consuming and difficult. Likely candidates for conversion are applications that are highly computation intensive, such as the following:

- Speech compression algorithms and filters.
- Speech recognition algorithms.
- Video display and capture routines.
- Rendering routines.
- 3D graphics (geometry).
- Image and video processing algorithms.
- Spatial (3D) audio.
- Physical modeling (graphics, CAD).
- Workstation applications.
- Encryption algorithms.
- Complex arithmetics.

Generally, good candidate code is code that contains small-sized repetitive loops that operate on sequential arrays of integers of 8, 16 or 32 bits, single-precision 32-bit floating-point data, double precision 64-bit floating-point data (integer and floating-point data items should be sequential in memory). The repetitiveness of these loops incurs costly application processing time. However, these routines have potential for increased performance when you convert them to use one of the SIMD technologies.

Once you identify your opportunities for using a SIMD technology, you must evaluate what should be done to determine whether the current algorithm or a modified one will ensure the best performance.

4.3 CODING TECHNIQUES

The SIMD features of SSE3, SSE2, SSE, and MMX technology require new methods of coding algorithms. One of them is vectorization. Vectorization is the process of transforming sequentially-executing, or scalar, code into code that can execute in parallel, taking advantage of the SIMD architecture parallelism.

This section discusses the coding techniques available for an application to make use of the SIMD architecture.

To vectorize your code and thus take advantage of the SIMD architecture, do the following:

- Determine if the memory accesses have dependencies that would prevent parallel execution.
- "Strip-mine" the inner loop to reduce the iteration count by the length of the SIMD operations (for example, four for single-precision floating-point SIMD, eight for 16-bit integer SIMD on the XMM registers).
- Re-code the loop with the SIMD instructions.

Each of these actions is discussed in detail in the subsequent sections of this chapter. These sections also discuss enabling automatic vectorization using the Intel C++ Compiler.

4.3.1 Coding Methodologies

Software developers need to compare the performance improvement that can be obtained from assembly code versus the cost of those improvements. Programming directly in assembly language for a target platform may produce the required performance gain, however, assembly code is not portable between processor architectures and is expensive to write and maintain.

Performance objectives can be met by taking advantage of the different SIMD technologies using high-level languages as well as assembly. The new C/C++ language extensions designed specifically for SSSE3, SSE3, SSE2, SSE, and MMX technology help make this possible.

Figure 4-4 illustrates the trade-offs involved in the performance of hand-coded assembly versus the ease of programming and portability.

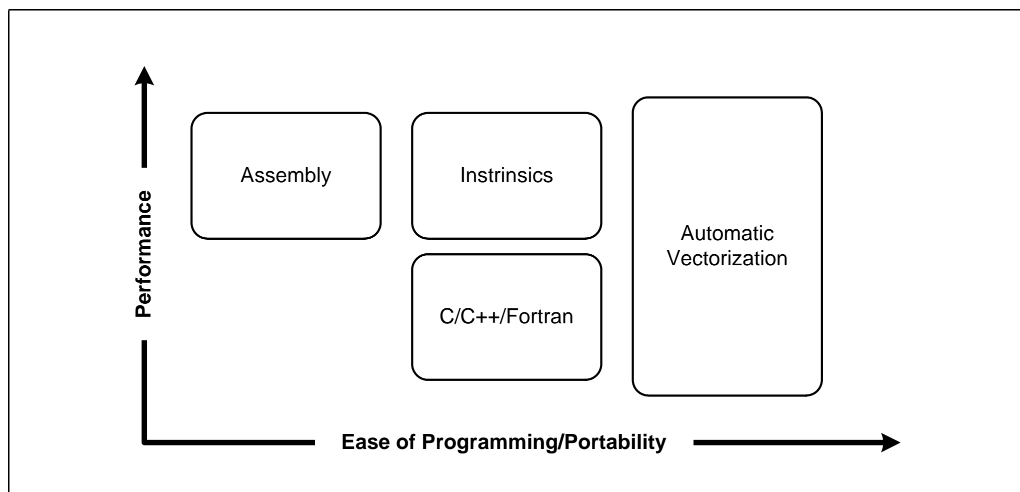


Figure 4-4. Hand-Coded Assembly and High-Level Compiler Performance Trade-offs

The examples that follow illustrate the use of coding adjustments to enable the algorithm to benefit from the SSE. The same techniques may be used for single-precision floating-point, double-precision floating-point, and integer data under SSSE3, SSE3, SSE2, SSE, and MMX technology.

As a basis for the usage model discussed in this section, consider a simple loop shown in Example 4-13.

Example 4-13. Simple Four-Iteration Loop

```
void add(float *a, float *b, float *c)
{
int i;
for (i = 0; i < 4; i++) {
   c[i] = a[i] + b[i];
 }
}
```

Note that the loop runs for only four iterations. This allows a simple replacement of the code with Streaming SIMD Extensions.

For the optimal use of the Streaming SIMD Extensions that need data alignment on the 16-byte boundary, all examples in this chapter assume that the arrays passed to the routine, *A*, *B*, *C*, are aligned to 16-byte boundaries by a calling routine. For the methods to ensure this alignment, please refer to the application notes for the Pentium 4 processor.

The sections that follow provide details on the coding methodologies: inlined assembly, intrinsics, C++ vector classes, and automatic vectorization.

4.3.1.1 Assembly

Key loops can be coded directly in assembly language using an assembler or by using inlined assembly (C-asm) in C/C++ code. The Intel compiler or assembler recognize the new instructions and registers, then directly generate the corresponding code. This model offers the opportunity for attaining greatest performance, but this performance is not portable across the different processor architectures.

Example 4-14 shows the Streaming SIMD Extensions inlined assembly encoding.

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
 __asm {
   mov    eax, a
   mov    edx, b
   mov    ecx, c
   movaps  xmm0, XMMWORD PTR [eax]
   addps   xmm0, XMMWORD PTR [edx]
   movaps  XMMWORD PTR [ecx], xmm0
 }
}
```

4.3.1.2 Intrinsics

Intrinsics provide the access to the ISA functionality using C/C++ style coding instead of assembly language. Intel has defined three sets of intrinsic functions that are implemented in the Intel C++ Compiler to support the MMX technology, Streaming SIMD Extensions and Streaming SIMD Extensions 2. Four new C data types, representing 64-bit and 128-bit objects are used as the operands of these intrinsic functions. __M64 is used for MMX integer SIMD, __M128 is used for single-precision floating-point SIMD, __M128I is used for Streaming SIMD Extensions 2 integer SIMD, and __M128D is used for double precision floating-point SIMD. These types enable the programmer to choose the implementation of an algorithm directly, while allowing the compiler to perform register allocation and instruction sched-

uling where possible. The intrinsics are portable among all Intel architecture-based processors supported by a compiler.

The use of intrinsics allows you to obtain performance close to the levels achievable with assembly. The cost of writing and maintaining programs with intrinsics is considerably less. For a detailed description of the intrinsics and their use, refer to the Intel C++ Compiler documentation.

Example 4-15 shows the loop from Example 4-13 using intrinsics.

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

The intrinsics map one-to-one with actual Streaming SIMD Extensions assembly code. The XMMINTRIN.H header file in which the prototypes for the intrinsics are defined is part of the Intel C++ Compiler included with the VTune Performance Enhancement Environment CD.

Intrinsics are also defined for the MMX technology ISA. These are based on the __m64 data type to represent the contents of an mm register. You can specify values in bytes, short integers, 32-bit values, or as a 64-bit object.

The intrinsic data types, however, are not a basic ANSI C data type, and therefore you must observe the following usage restrictions:

- Use intrinsic data types only on the left-hand side of an assignment as a return value or as a parameter. You cannot use it with other arithmetic expressions (for example, “+”, “>>”).
- Use intrinsic data type objects in aggregates, such as unions to access the byte elements and structures; the address of an __M64 object may be also used.
- Use intrinsic data type data only with the MMX technology intrinsics described in this guide.

For complete details of the hardware instructions, see the *Intel Architecture MMX Technology Programmer’s Reference Manual*. For a description of data types, see the *Intel® 64 and IA-32 Architectures Software Developer’s Manual.*

4.3.1.3 Classes

A set of C++ classes has been defined and available in Intel C++ Compiler to provide both a higher-level abstraction and more flexibility for programming with MMX technology, Streaming SIMD Extensions and Streaming SIMD Extensions 2. These classes provide an easy-to-use and flexible interface to the intrinsic functions, allowing developers to write more natural C++ code without worrying about which intrinsic or assembly language instruction to use for a given operation. Since the intrinsic functions underlie the implementation of these C++ classes, the performance of applications using this methodology can approach that of one using the intrinsics. Further details on the use of these classes can be found in the *Intel C++ Class Libraries for SIMD Operations User’s Guide*, order number 693500.

Example 4-16 shows the C++ code using a vector class library. The example assumes the arrays passed to the routine are already aligned to 16-byte boundaries.

Example 4-16. C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
        F32vec4 *av=(F32vec4 *) a;
        F32vec4 *bv=(F32vec4 *) b;
        F32vec4 *cv=(F32vec4 *) c;
                *cv=*av + *bv;
}
```

Here, fvec.h is the class definition file and F32vec4 is the class representing an array of four floats. The "+" and "=" operators are overloaded so that the actual Streaming SIMD Extensions implementation in the previous example is abstracted out, or hidden, from the developer. Note how much more this resembles the original code, allowing for simpler and faster programming.

Again, the example is assuming the arrays, passed to the routine, are already aligned to 16-byte boundary.

4.3.1.4 Automatic Vectorization

The Intel C++ Compiler provides an optimization mechanism by which loops, such as in Example 4-13 can be automatically vectorized, or converted into Streaming SIMD Extensions code. The compiler uses similar techniques to those used by a programmer to identify whether a loop is suitable for conversion to SIMD. This involves determining whether the following might prevent vectorization:

- The layout of the loop and the data structures used.
- Dependencies amongst the data accesses in each iteration and across iterations.

Once the compiler has made such a determination, it can generate vectorized code for the loop, allowing the application to use the SIMD instructions.

The caveat to this is that only certain types of loops can be automatically vectorized, and in most cases user interaction with the compiler is needed to fully enable this.

Example 4-17 shows the code for automatic vectorization for the simple four-iteration loop (from Example 4-13).

Example 4-17. Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
                float *restrict b,
                float *restrict c)
{
        int i;
        for (i = 0; i < 4; i++) {
                c[i] = a[i] + b[i];
        }
}
```

Compile this code using the -QAX and -QRESTRICT switches of the Intel C++ Compiler, version 4.0 or later.

The RESTRICT qualifier in the argument list is necessary to let the compiler know that there are no other aliases to the memory to which the pointers point. In other words, the pointer for which it is used,

provides the only means of accessing the memory in question in the scope in which the pointers live. Without the restrict qualifier, the compiler will still vectorize this loop using runtime data dependence testing, where the generated code dynamically selects between sequential or vector execution of the loop, based on overlap of the parameters (See documentation for the Intel C++ Compiler). The restrict keyword avoids the associated overhead altogether.

See Intel C++ Compiler documentation for details.

4.4 STACK AND DATA ALIGNMENT

To get the most performance out of code written for SIMD technologies data should be formatted in memory according to the guidelines described in this section. Assembly code with an unaligned accesses is a lot slower than an aligned access.

4.4.1 Alignment and Contiguity of Data Access Patterns

The 64-bit packed data types defined by MMX technology, and the 128-bit packed data types for Streaming SIMD Extensions and Streaming SIMD Extensions 2 create more potential for misaligned data accesses. The data access patterns of many algorithms are inherently misaligned when using MMX technology and Streaming SIMD Extensions. Several techniques for improving data access, such as padding, organizing data elements into arrays, etc. are described below. SSE3 provides a special-purpose instruction LDDQU that can avoid cache line splits is discussed in Section 5.7.1.1, “Supplemental Techniques for Avoiding Cache Line Splits.”

4.4.1.1 Using Padding to Align Data

However, when accessing SIMD data using SIMD operations, access to data can be improved simply by a change in the declaration. For example, consider a declaration of a structure, which represents a point in space plus an attribute.

```
typedef struct {short x,y,z; char a} Point;
Point pt[N];
```

Assume we will be performing a number of computations on X, Y, Z in three of the four elements of a SIMD word; see Section 4.5.1, “Data Structure Layout,” for an example. Even if the first element in array PT is aligned, the second element will start 7 bytes later and not be aligned (3 shorts at two bytes each plus a single byte = 7 bytes).

By adding the padding variable PAD, the structure is now 8 bytes, and if the first element is aligned to 8 bytes (64 bits), all following elements will also be aligned. The sample declaration follows:

```
typedef struct {short x,y,z; char a; char pad;} Point;
Point pt[N];
```

4.4.1.2 Using Arrays to Make Data Contiguous

In the following code,

```
for (i=0; i<N; i++) pt[i].y *= scale;
```

the second dimension Y needs to be multiplied by a scaling value. Here, the FOR loop accesses each Y dimension in the array PT thus disallowing the access to contiguous data. This can degrade the performance of the application by increasing cache misses, by poor utilization of each cache line that is fetched, and by increasing the chance for accesses which span multiple cache lines.

The following declaration allows you to vectorize the scaling operation and further improve the alignment of the data access patterns:

```
short ptx[N], pty[N], ptz[N];
for (i=0; i<N; i++) pty[i] *= scale;
```

With the SIMD technology, choice of data organization becomes more important and should be made carefully based on the operations that will be performed on the data. In some applications, traditional data arrangements may not lead to the maximum performance.

A simple example of this is an FIR filter. An FIR filter is effectively a vector dot product in the length of the number of coefficient taps.

Consider the following code:

```
(data [ j ] *coeff [0] + data [j+1]*coeff [1]+...+data [j+num of taps-1]*coeff [num of taps-1]),
```

If in the code above the filter operation of data element I is the vector dot product that begins at data element J, then the filter operation of data element I+1 begins at data element J+1.

Assuming you have a 64-bit aligned data vector and a 64-bit aligned coefficients vector, the filter operation on the first data element will be fully aligned. For the second data element, however, access to the data vector will be misaligned. For an example of how to avoid the misalignment problem in the FIR filter, refer to Intel application notes on Streaming SIMD Extensions and filters.

Duplication and padding of data structures can be used to avoid the problem of data accesses in algorithms which are inherently misaligned. Section 4.5.1, "Data Structure Layout," discusses trade-offs for organizing data structures.

NOTE

> The duplication and padding technique overcomes the misalignment problem, thus avoiding the expensive penalty for misaligned data access, at the cost of increasing the data size. When developing your code, you should consider this tradeoff and use the option which gives the best performance.

4.4.2 Stack Alignment For 128-bit SIMD Technologies

For best performance, the Streaming SIMD Extensions and Streaming SIMD Extensions 2 require their memory operands to be aligned to 16-byte boundaries. Unaligned data can cause significant performance penalties compared to aligned data. However, the existing software conventions for IA-32 (STDCALL, CDECL, FASTCALL) as implemented in most compilers, do not provide any mechanism for ensuring that certain local data and certain parameters are 16-byte aligned. Therefore, Intel has defined a new set of IA-32 software conventions for alignment to support the new __M128* datatypes (__M128, __M128D, and __M218I). These meet the following conditions:

- Functions that use Streaming SIMD Extensions or Streaming SIMD Extensions 2 data need to provide a 16-byte aligned stack frame.
- __M128* parameters need to be aligned to 16-byte boundaries, possibly creating "holes" (due to padding) in the argument block.

The new conventions presented in this section as implemented by the Intel C++ Compiler can be used as a guideline for an assembly language code as well. In many cases, this section assumes the use of the __M128* data types, as defined by the Intel C++ Compiler, which represents an array of four 32-bit floats.

4.4.3 Data Alignment for MMX Technology

Many compilers enable alignment of variables using controls. This aligns variable bit lengths to the appropriate boundaries. If some of the variables are not appropriately aligned as specified, you can align them using the C algorithm in Example 4-18.

Example 4-18. C Algorithm for 64-bit Data Alignment

```
/* Make newp a pointer to a 64-bit aligned array of NUM_ELEMENTS 64-bit elements. */
double *p, *newp;
p = (double*)malloc (sizeof(double)*(NUM_ELEMENTS+1));
newp = (p+7) & (~0x7);
```

The algorithm in Example 4-18 aligns an array of 64-bit elements on a 64-bit boundary. The constant of 7 is derived from one less than the number of bytes in a 64-bit element, or 8-1. Aligning data in this manner avoids the significant performance penalties that can occur when an access crosses a cache line boundary.

Another way to improve data alignment is to copy the data into locations that are aligned on 64-bit boundaries. When the data is accessed frequently, this can provide a significant performance improvement.

4.4.4 Data Alignment for 128-bit data

Data must be 16-byte aligned when loading to and storing from the 128-bit XMM registers used by SSE/SSE2/SSE3/SSSE3. This must be done to avoid severe performance penalties and, at worst, execution faults.

There are MOVE instructions (and intrinsics) that allow unaligned data to be copied to and out of XMM registers when not using aligned data, but such operations are much slower than aligned accesses. If data is not 16-byte-aligned and the programmer or the compiler does not detect this and uses the aligned instructions, a fault occurs. So keep data 16-byte-aligned. Such alignment also works for MMX technology code, even though MMX technology only requires 8-byte alignment.

The following describes alignment techniques for Pentium 4 processor as implemented with the Intel C++ Compiler.

4.4.4.1 Compiler-Supported Alignment

The Intel C++ Compiler provides the following methods to ensure that the data is aligned.

Alignment by F32vec4 or __m128 Data Types

When the compiler detects F32VEC4 or __M128 data declarations or parameters, it forces alignment of the object to a 16-byte boundary for both global and local data, as well as parameters. If the declaration is within a function, the compiler also aligns the function's stack frame to ensure that local data and parameters are 16-byte-aligned. For details on the stack frame layout that the compiler generates for both debug and optimized ("release"-mode) compilations, refer to Intel's compiler documentation.

__declspec(align(16)) specifications

These can be placed before data declarations to force 16-byte alignment. This is useful for local or global data declarations that are assigned to 128-bit data types. The syntax for it is

```
__declspec(align(integer-constant))
```

where the INTEGER-CONSTANT is an integral power of two but no greater than 32. For example, the following increases the alignment to 16-bytes:

```
__declspec(align(16)) float buffer[400];
```

The variable BUFFER could then be used as if it contained 100 objects of type __M128 or F32VEC4. In the code below, the construction of the F32VEC4 object, X, will occur with aligned data.

```
void foo() {
    F32vec4 x = *(__m128 *) buffer;
    ...
}
```

Without the declaration of __DECLSPEC(ALIGN(16)), a fault may occur.

Alignment by Using a UNION Structure

When feasible, a UNION can be used with 128-bit data types to allow the compiler to align the data structure by default. This is preferred to forcing alignment with __DECLSPEC(ALIGN(16)) because it exposes the true program intent to the compiler in that __M128 data is being used. For example:

```
union {
    float f[400];
    __m128 m[100];
} buffer;
```

Now, 16-byte alignment is used by default due to the __M128 type in the UNION; it is not necessary to use __DECLSPEC(ALIGN(16)) to force the result.

In C++ (but not in C) it is also possible to force the alignment of a CLASS/STRUCT/UNION type, as in the code that follows:

```
struct __declspec(align(16)) my_m128
{
     float f[4];
};
```

If the data in such a CLASS is going to be used with the Streaming SIMD Extensions or Streaming SIMD Extensions 2, it is preferable to use a UNION to make this explicit. In C++, an anonymous UNION can be used to make this more convenient:

```
class my_m128 {
    union {
        __m128 m;
        float f[4];
    };
};
```

Because the UNION is anonymous, the names, M and F, can be used as immediate member names of MY__M128. Note that __DECLSPEC(ALIGN) has no effect when applied to a CLASS, STRUCT, or UNION member in either C or C++.

Alignment by Using __m64 or DOUBLE Data

In some cases, the compiler aligns routines with __M64 or DOUBLE data to 16-bytes by default. The command-line switch, -QSFALIGN16, limits the compiler so that it only performs this alignment on routines that contain 128-bit data. The default behavior is to use -QSFALIGN8. This switch instructs the complier to align routines with 8- or 16-byte data types to 16 bytes.

For more, see the Intel C++ Compiler documentation.

4.5 IMPROVING MEMORY UTILIZATION

Memory performance can be improved by rearranging data and algorithms for SSE, SSE2, and MMX technology intrinsics. Methods for improving memory performance involve working with the following:

- Data structure layout.
- Strip-mining for vectorization and memory utilization.
- Loop-blocking.

Using the cacheability instructions, prefetch and streaming store, also greatly enhance memory utilization. See also: Chapter 7, “Optimizing Cache Usage.”

4.5.1 Data Structure Layout

For certain algorithms, like 3D transformations and lighting, there are two basic ways to arrange vertex data. The traditional method is the array of structures (AoS) arrangement, with a structure for each

vertex (Example 4-19). However this method does not take full advantage of SIMD technology capabilities.

Example 4-19. AoS Data Structure

```
typedef struct{
    float x,y,z;
    int a,b,c;
    . . .
} Vertex;
Vertex Vertices[NumOfVertices];
```

The best processing method for code using SIMD technology is to arrange the data in an array for each coordinate (Example 4-20). This data arrangement is called structure of arrays (SoA).

Example 4-20. SoA Data Structure

```
typedef struct{
    float x[NumOfVertices];
    float y[NumOfVertices];
    float z[NumOfVertices];
    int a[NumOfVertices];
    int b[NumOfVertices];
    int c[NumOfVertices];
    . . .
} VerticesList;
VerticesList Vertices;
```

There are two options for computing data in AoS format: perform operation on the data as it stands in AoS format, or re-arrange it (swizzle it) into SoA format dynamically. See Example 4-21 for code samples of each option based on a dot-product computation.

Example 4-21. AoS and SoA Code Samples

```
; The dot product of an array of vectors (Array) and a fixed vector (Fixed) is a
; common operation in 3D lighting operations, where Array = (x0,y0,z0),(x1,y1,z1),...
; and Fixed = (xF,yF,zF)
; A dot product is defined as the scalar quantity d0 = x0*xF + y0*yF + z0*zF.

;
; AoS code
; All values marked DC are "don't-care."

; In the AOS model, the vertices are stored in the xyz format
movaps  xmm0, Array        ; xmm0 = DC, x0, y0, z0
movaps  xmm1, Fixed        ; xmm1 = DC, xF, yF, zF
mulps   xmm0, xmm1         ; xmm0 = DC, x0*xF, y0*yF, z0*zF
movhlps xmm, xmm0          ; xmm = DC, DC, DC, x0*xF

addps   xmm1, xmm0         ; xmm0 = DC, DC, DC,
                           ; x0*xF+z0*zFmovaps  xmm2, xmm1
shufps  xmm2, xmm2,55h     ; xmm2 = DC, DC,    DC,    y0*yF
addps   xmm2, xmm1         ; xmm1 = DC, DC,    DC,
                           ; x0*xF+y0*yF+z0*zF
```

Example 4-21. AoS and SoA Code Samples (Contd.)

```
; SoA code
; X = x0,x1,x2,x3
; Y = y0,y1,y2,y3
; Z = z0,z1,z2,z3
; A = xF,xF,xF,xF
; B = yF,yF,yF,yF
; C = zF,zF,zF,zF

movaps xmm0, X          ; xmm0 = x0,x1,x2,x3
movaps xmm1, Y          ; xmm0 = y0,y1,y2,y3
movaps xmm2, Z          ; xmm0 = z0,z1,z2,z3
mulps  xmm0, A          ; xmm0 = x0*xF, x1*xF, x2*xF, x3*xF
mulps  xmm1, B          ; xmm1 = y0*yF, y1*yF, y2*yF, y3*xF
mulps  xmm2, C          ; xmm2 = z0*zF, z1*zF, z2*zF, z3*zF
addps  xmm0, xmm1
addps  xmm0, xmm2       ; xmm0 = (x0*xF+y0*yF+z0*zF), ...
```

Performing SIMD operations on the original AoS format can require more calculations and some operations do not take advantage of all SIMD elements available. Therefore, this option is generally less efficient.

The recommended way for computing data in AoS format is to swizzle each set of elements to SoA format before processing it using SIMD technologies. Swizzling can either be done dynamically during program execution or statically when the data structures are generated. See Chapter 5 and Chapter 6 for examples. Performing the swizzle dynamically is usually better than using AoS, but can be somewhat inefficient because there are extra instructions during computation. Performing the swizzle statically, when data structures are being laid out, is best as there is no runtime overhead.

As mentioned earlier, the SoA arrangement allows more efficient use of the parallelism of SIMD technologies because the data is ready for computation in a more optimal vertical manner: multiplying components X0,X1,X2,X3 by XF,XF,XF,XF using 4 SIMD execution slots to produce 4 unique results. In contrast, computing directly on AoS data can lead to horizontal operations that consume SIMD execution slots but produce only a single scalar result (as shown by the many "don't-care" (DC) slots in Example 4-21).

Use of the SoA format for data structures can lead to more efficient use of caches and bandwidth. When the elements of the structure are not accessed with equal frequency, such as when element x, y, z are accessed ten times more often than the other entries, then SoA saves memory and prevents fetching unnecessary data items a, b, and c.

Example 4-22. Hybrid SoA Data Structure

```
NumOfGroups = NumOfVertices/SIMDwidth
typedef struct{
    float x[SIMDwidth];
    float y[SIMDwidth];
    float z[SIMDwidth];

} VerticesCoordList;
typedef struct{
    int a[SIMDwidth];
    int b[SIMDwidth];
    int c[SIMDwidth];
    . . .

} VerticesColorList;
VerticesCoordList VerticesCoord[NumOfGroups];
VerticesColorList VerticesColor[NumOfGroups];
```

Note that SoA can have the disadvantage of requiring more independent memory stream references. A computation that uses arrays X, Y, and Z (see Example 4-20) would require three separate data streams. This can require the use of more prefetches, additional address generation calculations, as well as having a greater impact on DRAM page access efficiency.

There is an alternative: a hybrid SoA approach blends the two alternatives (see Example 4-22). In this case, only 2 separate address streams are generated and referenced: one contains XXXX, YYYY,ZZZZ, ZZZZ,... and the other AAAA, BBBB, CCCC, AAAA, DDDD,... . The approach prevents fetching unnecessary data, assuming the variables X, Y, Z are always used together; whereas the variables A, B, C would also be used together, but not at the same time as X, Y, Z.

The hybrid SoA approach ensures:

- Data is organized to enable more efficient vertical SIMD computation.
- Simpler/less address generation than AoS.
- Fewer streams, which reduces DRAM page misses.
- Use of fewer prefetches, due to fewer streams.
- Efficient cache line packing of data elements that are used concurrently.

With the advent of the SIMD technologies, the choice of data organization becomes more important and should be carefully based on the operations to be performed on the data. This will become increasingly important in the Pentium 4 processor and future processors. In some applications, traditional data arrangements may not lead to the maximum performance. Application developers are encouraged to explore different data arrangements and data segmentation policies for efficient computation. This may mean using a combination of AoS, SoA, and Hybrid SoA in a given application.

4.5.2 Strip-Mining

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as providing a means of improving memory performance. First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure by:

- Increasing the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- Reducing the number of iterations of the loop by a factor of the length of each “vector,” or number of operations being performed per SIMD operation. In the case of Streaming SIMD Extensions, this vector or strip-length is reduced by 4 times: four floating-point data items per single Streaming SIMD Extensions single-precision floating-point SIMD operation are processed. Consider Example 4-23.

Example 4-23. Pseudo-code Before Strip Mining

```
typedef struct _VERTEX {
        float x, y, z, nx, ny, nz, u, v;
} Vertex_rec;

main()
{
        Vertex_rec v[Num];
        ....
        for (i=0; i<Num; i++) {
          Transform(v[i]);
        }
```

Example 4-23. Pseudo-code Before Strip Mining (Contd.)

```
        for (i=0; i<Num; i++) {
          Lighting(v[i]);
        }
        ....
}
```

The main loop consists of two functions: transformation and lighting. For each object, the main loop calls a transformation routine to update some data, then calls the lighting routine to further work on the data. If the size of array V[NUM] is larger than the cache, then the coordinates for V[I] that were cached during TRANSFORM(V[I]) will be evicted from the cache by the time we do LIGHTING(V[I]). This means that V[I] will have to be fetched from main memory a second time, reducing performance.

In Example 4-24, the computation has been strip-mined to a size STRIP_SIZE. The value STRIP_SIZE is chosen such that STRIP_SIZE elements of array V[NUM] fit into the cache hierarchy. By doing this, a given element V[I] brought into the cache by TRANSFORM(V[I]) will still be in the cache when we perform LIGHTING(V[I]), and thus improve performance over the non-strip-mined code.

Example 4-24. Strip Mined Code

```
MAIN()
{
    Vertex_rec v[Num];
    ....
    for (i=0; i < Num; i+=strip_size) {
      FOR (J=I; J < MIN(NUM, I+STRIP_SIZE); J++) {
          TRANSFORM(V[J]);
     }
      FOR (J=I; J < MIN(NUM, I+STRIP_SIZE); J++) {
          LIGHTING(V[J]);
      }
    }
}
```

4.5.3 Loop Blocking

Loop blocking is another useful technique for memory performance optimization. The main purpose of loop blocking is also to eliminate as many cache misses as possible. This technique transforms the memory domain of a given problem into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, thereby maximizing data reuse. In fact, one can treat loop blocking as strip mining in two or more dimensions. Consider the code in Example 4-23 and access pattern in Figure 4-5. The two-dimensional array A is referenced in the J (column) direction and then referenced in the I (row) direction (column-major order); whereas array B is referenced in the opposite manner (row-major order). Assume the memory layout is in column-major order; therefore, the access strides of array A and B for the code in Example 4-25 would be 1 and MAX, respectively.

Example 4-25. Loop Blocking

```
A. Original Loop
float A[MAX, MAX], B[MAX, MAX]
for (i=0; i< MAX; i++) {
    for (j=0; j< MAX; j++) {
          A[i,j] = A[i,j] + B[j, i];
    }
}
```

Example 4-25. Loop Blocking (Contd.)

```
B. Transformed Loop after Blocking
float A[MAX, MAX], B[MAX, MAX];
for (i=0; i< MAX; i+=block_size) {
    for (j=0; j< MAX; j+=block_size) {
        for (ii=i; ii<i+block_size; ii++) {
            for (jj=j; jj<j+block_size; jj++) {
                A[ii,jj] = A[ii,jj] + B[jj, ii];
            }
        }
    }
}
```

For the first iteration of the inner loop, each access to array B will generate a cache miss. If the size of one row of array A, that is, A[2, 0:MAX-1], is large enough, by the time the second iteration starts, each access to array B will always generate a cache miss. For instance, on the first iteration, the cache line containing B[0, 0:7] will be brought in when B[0,0] is referenced because the float type variable is four bytes and each cache line is 32 bytes. Due to the limitation of cache capacity, this line will be evicted due to conflict misses before the inner loop reaches the end. For the next iteration of the outer loop, another cache miss will be generated while referencing B[0, 1]. In this manner, a cache miss occurs when each element of array B is referenced, that is, there is no data reuse in the cache at all for array B.

This situation can be avoided if the loop is blocked with respect to the cache size. In Figure 4-5, a BLOCK_SIZE is selected as the loop blocking factor. Suppose that BLOCK_SIZE is 8, then the blocked chunk of each array will be eight cache lines (32 bytes each). In the first iteration of the inner loop, A[0, 0:7] and B[0, 0:7] will be brought into the cache. B[0, 0:7] will be completely consumed by the first iteration of the outer loop. Consequently, B[0, 0:7] will only experience one cache miss after applying loop blocking optimization in lieu of eight misses for the original algorithm. As illustrated in Figure 4-5, arrays A and B are blocked into smaller rectangular chunks so that the total size of two blocked A and B chunks is smaller than the cache size. This allows maximum data reuse.

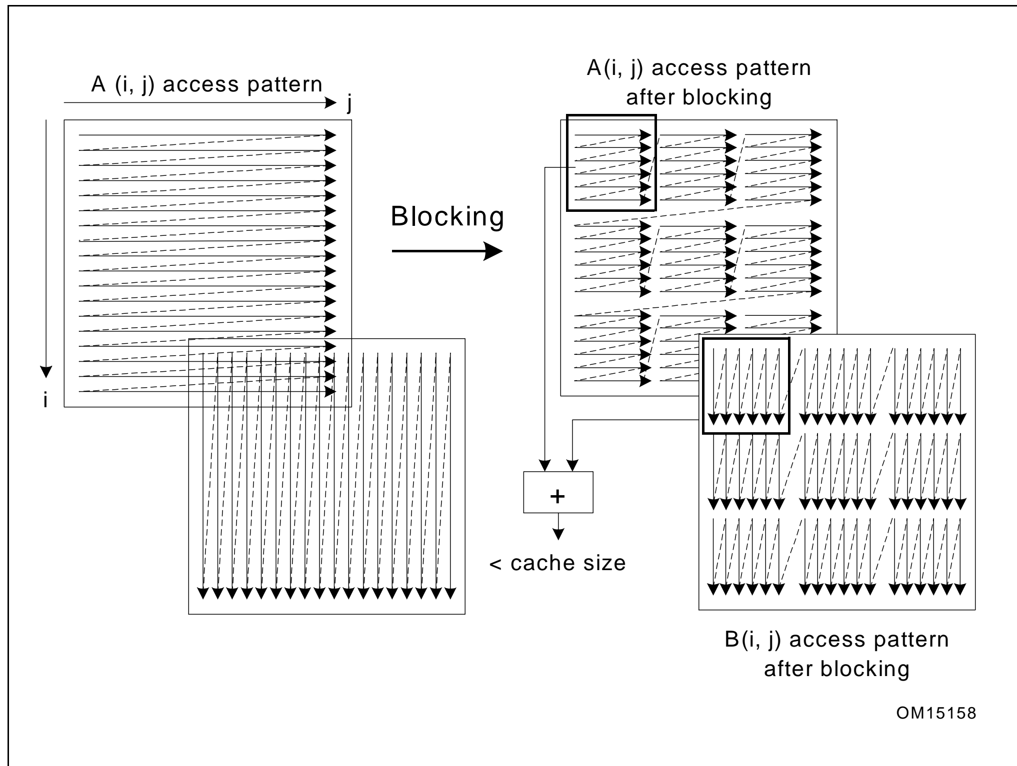


Figure 4-5. Loop Blocking Access Pattern

As one can see, all the redundant cache misses can be eliminated by applying this loop blocking technique. If MAX is huge, loop blocking can also help reduce the penalty from DTLB (data translation lookaside buffer) misses. In addition to improving the cache/memory performance, this optimization technique also saves external bus bandwidth.

4.6 INSTRUCTION SELECTION

The following section gives some guidelines for choosing instructions to complete a task.

One barrier to SIMD computation can be the existence of data-dependent branches. Conditional moves can be used to eliminate data-dependent branches. Conditional moves can be emulated in SIMD computation by using masked compares and logicals, as shown in Example 4-26. SSE4.1 provides packed blend instruction that can vectorize data-dependent branches in a loop.

Example 4-26. Emulation of Conditional Moves

```
High-level code:
__declspec(align(16)) short A[MAX_ELEMENT], B[MAX_ELEMENT], C[MAX_ELEMENT], D[MAX_ELEMENT],
E[MAX_ELEMENT];

for (i=0; i<MAX_ELEMENT; i++) {
    if (A[i] > B[i]) {
        C[i] = D[i];
    } else {
        C[i] = E[i];
    }
```

Example 4-26. Emulation of Conditional Moves (Contd.)

```
}
MMX assembly code processes 4 short values per iteration:
    xor         eax, eax

top_of_loop:
    movq        mm0, [A + eax]
    pcmpgtw xmm0, [B + eax]; Create compare mask
    movq        mm1, [D + eax]
    pand        mm1, mm0; Drop elements where A<B
    pandn       mm0, [E + eax] ; Drop elements where A>B

    por         mm0, mm1; Crete single word
    movq        [C + eax], mm0
    add         eax, 8
    cmp         eax, MAX_ELEMENT*2
    jle         top_of_loop

SSE4.1 assembly processes 8 short values per iteration:
    xor         eax, eax

top_of_loop:
    movdqq  xmm0, [A + eax]
    pcmpgtw xmm0, [B + eax]; Create compare mask
    movdqa  xmm1, [E + eax]
    pblendv xmm1, [D + eax], xmm0;
    movdqa  [C + eax], xmm1;
    add         eax, 16
    cmp         eax, MAX_ELEMENT*2
    jle         top_of_loop
```

If there are multiple consumers of an instance of a register, group the consumers together as closely as possible. However, the consumers should not be scheduled near the producer.

4.6.1 SIMD Optimizations and Microarchitectures

Pentium M, Intel Core Solo and Intel Core Duo processors have a different microarchitecture than Intel NetBurst microarchitecture. The following sub-section discusses optimizing SIMD code targeting Intel Core Solo and Intel Core Duo processors.

The register-register variant of the following instructions has improved performance on Intel Core Solo and Intel Core Duo processor relative to Pentium M processors. This is because the instructions consist of two micro-ops instead of three. Relevant instructions are: unpcklps, unpckhps, packsswb, packuswb, packssdw, pshufd, shuffps and shuffpd.

Recommendation: When targeting code generation for Intel Core Solo and Intel Core Duo processors, favor instructions consisting of two micro-ops over those with more than two micro-ops.

Intel Core microarchitecture generally executes SIMD instructions more efficiently than previous micro-architectures in terms of latency and throughput, most 128-bit SIMD operations have 1 cycle throughput (except shuffle, pack, unpack operations). Many of the restrictions specific to Intel Core Duo, Intel Core Solo processors (such as 128-bit SIMD operations having 2 cycle throughput at a minimum) do not apply to Intel Core microarchitecture. The same is true of Intel Core microarchitecture relative to Intel NetBurst microarchitectures.

Enhanced Intel Core microarchitecture provides dedicated 128-bit shuffler and radix-16 divider hard-ware. These capabilities and SSE4.1 instructions will make vectorization using 128-bit SIMD instructions even more efficient and effective.

Recommendation: With the proliferation of 128-bit SIMD hardware in Intel Core microarchitecture and Enhanced Intel Core microarchitecture, integer SIMD code written using MMX instructions should consider more efficient implementations using 128-bit SIMD instructions.

4.7 TUNING THE FINAL APPLICATION

The best way to tune your application once it is functioning correctly is to use a profiler that measures the application while it is running on a system. Intel VTune Amplifier XE can help you determine where to make changes in your application to improve performance. Using Intel VTune Amplifier XE can help you with various phases required for optimized performance. See Appendix A.3.1, "Intel® VTune™ Amplifier XE," for details. After every effort to optimize, you should check the performance gains to see where you are making your major optimization gains.

CHAPTER 5
OPTIMIZING FOR SIMD INTEGER APPLICATIONS

SIMD integer instructions provide performance improvements in applications that are integer-intensive and can take advantage of SIMD architecture.

Guidelines in this chapter for using SIMD integer instructions (in addition to those described in Chapter 3) may be used to develop fast and efficient code that scales across processor generations.

The collection of 64-bit and 128-bit SIMD integer instructions supported by MMX technology, SSE, SSE2, SSE3, SSSE3, SSE4.1, and PCMPEQQ in SSE4.2 are referred to as SIMD integer instructions.

Code sequences in this chapter demonstrates the use of basic 64-bit SIMD integer instructions and more efficient 128-bit SIMD integer instructions.

Processors based on Intel Core microarchitecture support MMX, SSE, SSE2, SSE3, and SSSE3. Processors based on Enhanced Intel Core microarchitecture support SSE4.1 and all previous generations of SIMD integer instructions. Processors based on Intel microarchitecture code name Nehalem supports MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2.

Single-instruction, multiple-data techniques can be applied to text/string processing, lexing and parser applications. SIMD programming in string/text processing and lexing applications often require sophisticated techniques beyond those commonly used in SIMD integer programming. This is covered in Chapter 10, "SSE4.2 and SIMD Programming For Text- Processing/Lexing/Parsing"

Execution of 128-bit SIMD integer instructions in Intel Core microarchitecture and Enhanced Intel Core microarchitecture are substantially more efficient than on previous microarchitectures. Thus newer SIMD capabilities introduced in SSE4.1 operate on 128-bit operands and do not introduce equivalent 64-bit SIMD capabilities. Conversion from 64-bit SIMD integer code to 128-bit SIMD integer code is highly recommended.

This chapter contains examples that will help you to get started with coding your application. The goal is to provide simple, low-level operations that are frequently used. The examples use a minimum number of instructions necessary to achieve best performance on the current generation of Intel 64 and IA-32 processors.

Each example includes a short description, sample code, and notes if necessary. These examples do not address scheduling as it is assumed the examples will be incorporated in longer code sequences.

For planning considerations of using the SIMD integer instructions, refer to Section 4.1.3.

5.1 GENERAL RULES ON SIMD INTEGER CODE

General rules and suggestions are:

- Do not intermix 64-bit SIMD integer instructions with x87 floating-point instructions. See Section 5.2, "Using SIMD Integer with x87 Floating-point." Note that all SIMD integer instructions can be intermixed without penalty.
- Favor 128-bit SIMD integer code over 64-bit SIMD integer code. On microarchitectures prior to Intel Core microarchitecture, most 128-bit SIMD instructions have two-cycle throughput restrictions due to the underlying 64-bit data path in the execution engine. Intel Core microarchitecture executes most SIMD instructions (except shuffle, pack, unpack operations) with one-cycle throughput and provides three ports to execute multiple SIMD instructions in parallel. Enhanced Intel Core microarchitecture speeds up 128-bit shuffle, pack, unpack operations with 1 cycle throughput.
- When writing SIMD code that works for both integer and floating-point data, use the subset of SIMD convert instructions or load/store instructions to ensure that the input operands in XMM registers contain data types that are properly defined to match the instruction.

Code sequences containing cross-typed usage produce the same result across different implementations but incur a significant performance penalty. Using SSE/SSE2/SSE3/SSSE3/SSE44.1 instructions to operate on type-mismatched SIMD data in the XMM register is strongly discouraged.

- Use the optimization rules and guidelines described in Chapter 3 and Chapter 4.
- Take advantage of hardware prefetcher where possible. Use the PREFETCH instruction only when data access patterns are irregular and prefetch distance can be pre-determined. See Chapter 7, "Optimizing Cache Usage."
- Emulate conditional moves by using blend, masked compares and logicals instead of using conditional branches.

5.2 USING SIMD INTEGER WITH X87 FLOATING-POINT

All 64-bit SIMD integer instructions use MMX registers, which share register state with the x87 floating-point stack. Because of this sharing, certain rules and considerations apply. Instructions using MMX registers cannot be freely intermixed with x87 floating-point registers. Take care when switching between 64-bit SIMD integer instructions and x87 floating-point instructions to ensure functional correctness. See Section 5.2.1.

Both Section 5.2.1 and Section 5.2.2 apply only to software that employs MMX instructions. As noted before, 128-bit SIMD integer instructions should be favored to replace MMX code and achieve higher performance. That also obviates the need to use EMMS, and the performance penalty of using EMMS when intermixing MMX and X87 instructions.

For performance considerations, there is no penalty of intermixing SIMD floating-point operations and 128-bit SIMD integer operations and x87 floating-point operations.

5.2.1 Using the EMMS Instruction

When generating 64-bit SIMD integer code, keep in mind that the eight MMX registers are aliased to x87 floating-point registers. Switching from MMX instructions to x87 floating-point instructions incurs a finite delay, so it is the best to minimize switching between these instruction types. But when switching, the EMMS instruction provides an efficient means to clear the x87 stack so that subsequent x87 code can operate properly.

As soon as an instruction makes reference to an MMX register, all valid bits in the x87 floating-point tag word are set, which implies that all x87 registers contain valid values. In order for software to operate correctly, the x87 floating-point stack should be emptied when starting a series of x87 floating-point calculations after operating on the MMX registers.

Using EMMS clears all valid bits, effectively emptying the x87 floating-point stack and making it ready for new x87 floating-point operations. The EMMS instruction ensures a clean transition between using operations on the MMX registers and using operations on the x87 floating-point stack. On the Pentium 4 processor, there is a finite overhead for using the EMMS instruction.

Failure to use the EMMS instruction (or the _MM_EMPTY() intrinsic) between operations on the MMX registers and x87 floating-point registers may lead to unexpected results.

NOTE

Failure to reset the tag word for FP instructions after using an MMX instruction can result in faulty execution or poor performance.

5.2.2 Guidelines for Using EMMS Instruction

When developing code with both x87 floating-point and 64-bit SIMD integer instructions, follow these steps:

1. Always call the EMMS instruction at the end of 64-bit SIMD integer code when the code transitions to x87 floating-point code.
2. Insert the EMMS instruction at the end of all 64-bit SIMD integer code segments to avoid an x87 floating-point stack overflow exception when an x87 floating-point instruction is executed.

When writing an application that uses both floating-point and 64-bit SIMD integer instructions, use the following guidelines to help you determine when to use EMMS:

- **If next instruction is x87 FP** — Use _MM_EMPTY() after a 64-bit SIMD integer instruction if the next instruction is an X87 FP instruction; for example, before doing calculations on floats, doubles or long doubles.
- **Don't empty when already empty** — If the next instruction uses an MMX register, _MM_EMPTY() incurs a cost with no benefit.
- **Group Instructions** — Try to partition regions that use X87 FP instructions from those that use 64-bit SIMD integer instructions. This eliminates the need for an EMMS instruction within the body of a critical loop.
- **Runtime initialization** — Use _MM_EMPTY() during runtime initialization of __M64 and X87 FP data types. This ensures resetting the register between data type transitions. See Example 5-1 for coding usage.

Example 5-1. Resetting Register Between __m64 and FP Data Types Code

Incorrect Usage	Correct Usage
__m64 x = _m_paddd(y, z); float f = init();	__m64 x = _m_paddd(y, z); float f = (_mm_empty(), init());

You must be aware that your code generates an MMX instruction, which uses MMX registers with the Intel C++ Compiler, in the following situations:

- when using a 64-bit SIMD integer intrinsic from MMX technology, SSE/SSE2/SSSE3
- when using a 64-bit SIMD integer instruction from MMX technology, SSE/SSE2/SSSE3 through inline assembly
- when referencing the __M64 data type variable

Additional information on the x87 floating-point programming model can be found in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. For more on EMMS, visit http://developer.intel.com.

5.3 DATA ALIGNMENT

Make sure that 64-bit SIMD integer data is 8-byte aligned and that 128-bit SIMD integer data is 16-byte aligned. Referencing unaligned 64-bit SIMD integer data can incur a performance penalty due to accesses that span 2 cache lines. Referencing unaligned 128-bit SIMD integer data results in an exception unless the MOVDQU (move double-quadword unaligned) instruction is used. Using the MOVDQU instruction on unaligned data can result in lower performance than using 16-byte aligned references. Refer to Section 4.4, "Stack and Data Alignment," for more information.

Loading 16 bytes of SIMD data efficiently requires data alignment on 16-byte boundaries. SSSE3 provides the PALIGNR instruction. It reduces overhead in situations that requires software to processing data elements from non-aligned address. The PALIGNR instruction is most valuable when loading or storing unaligned data with the address shifts by a few bytes. You can replace a set of unaligned loads with aligned loads followed by using PALIGNR instructions and simple register to register copies.

Using PALIGNRs to replace unaligned loads improves performance by eliminating cache line splits and other penalties. In routines like MEMCPY( ), PALIGNR can boost the performance of misaligned cases. Example 5-2 shows a situation that benefits by using PALIGNR.

Example 5-2. FIR Processing Example in C language Code

```
void FIR(float *in, float *out, float *coeff, int count)
{int i,j;
    for ( i=0; i<count - TAP; i++ )
    {     float sum = 0;
          for ( j=0; j<TAP; j++ )
          {     sum += in[j]*coeff[j]; }
          *out++ = sum;
          in++;
    }
}
```

Example 5-3 compares an optimal SSE2 sequence of the FIR loop and an equivalent SSSE3 implementation. Both implementations unroll 4 iteration of the FIR inner loop to enable SIMD coding techniques. The SSE2 code can not avoid experiencing cache line split once every four iterations. PALGNR allows the SSSE3 code to avoid the delays associated with cache line splits.

Example 5-3. SSE2 and SSSE3 Implementation of FIR Processing Code

Optimized for SSE2

```
pxor    xmm0, xmm0
xor     ecx, ecx
mov     eax, dword ptr[input]
mov     ebx, dword ptr[coeff4]

inner_loop:
movaps xmm1, xmmword ptr[eax+ecx]
mulps   xmm1, xmmword ptr[ebx+4*ecx]
addps   xmm0, xmm1

movups xmm1, xmmword ptr[eax+ecx+4]
mulps   xmm1, xmmword ptr[ebx+4*ecx+16]
addps   xmm0, xmm1

movups xmm1, xmmword ptr[eax+ecx+8]
mulps   xmm1, xmmword ptr[ebx+4*ecx+32]
addps   xmm0, xmm1

movups xmm1, xmmword ptr[eax+ecx+12]
mulps   xmm1, xmmword ptr[ebx+4*ecx+48]
addps   xmm0, xmm1

add     ecx, 16
cmp     ecx, 4*TAP
jl      inner_loop

mov     eax, dword ptr[output]
movaps xmmword ptr[eax], xmm0
```

Optimized for SSSE3

```
pxor xmm0, xmm0
xor     ecx, ecx
mov     eax, dword ptr[input]
mov     ebx, dword ptr[coeff4]

inner_loop:
movaps xmm1, xmmword ptr[eax+ecx]
movaps xmm3, xmm1
mulps   xmm1, xmmword ptr[ebx+4*ecx]
addps   xmm0, xmm1
movaps xmm2, xmmword ptr[eax+ecx+16]
movaps xmm1, xmm2
palignr xmm2, xmm3, 4
mulps   xmm2, xmmword ptr[ebx+4*ecx+16]
addps   xmm0, xmm2
movaps xmm2, xmm1
palignr xmm2, xmm3, 8
mulps   xmm2, xmmword ptr[ebx+4*ecx+32]
addps   xmm0, xmm2
movaps xmm2, xmm1
palignr xmm2, xmm3, 12
mulps   xmm2, xmmword ptr[ebx+4*ecx+48]
addps   xmm0, xmm2

add     ecx, 16
cmp     ecx, 4*TAP
jl      inner_loop

mov     eax, dword ptr[output]
movaps xmmword ptr[eax], xmm0
```

5.4 DATA MOVEMENT CODING TECHNIQUES

In general, better performance can be achieved if data is pre-arranged for SIMD computation (see Section 4.5, "Improving Memory Utilization"). This may not always be possible.

This section covers techniques for gathering and arranging data for more efficient SIMD computation.

5.4.1 Unsigned Unpack

MMX technology provides several instructions that are used to pack and unpack data in the MMX registers. SSE2 extends these instructions so that they operate on 128-bit source and destinations.

The unpack instructions can be used to zero-extend an unsigned number. Example 5-4 assumes the source is a packed-word (16-bit) data type.

Example 5-4. Zero Extend 16-bit Values into 32 Bits Using Unsigned Unpack Instructions Code

```
; Input:
;           XMM0        8 16-bit values in source
;           XMM7 0      a local variable can be used
;                       instead of the register XMM7 if
;                       desired.
; Output:
;           XMM0        four zero-extended 32-bit
;                       doublewords from four low-end
;                       words
;           XMM1        four zero-extended 32-bit
;                       doublewords from four high-end
;                       words

movdqa      xmm1, xmm0 ; copy source
punpcklwd   xmm0, xmm7 ; unpack the 4 low-end words
                       ; into 4 32-bit doubleword
punpckhwd   xmm1, xmm7 ; unpack the 4 high-end words
                       ; into 4 32-bit doublewords
```

5.4.2 Signed Unpack

Signed numbers should be sign-extended when unpacking values. This is similar to the zero-extend shown above, except that the PSRAD instruction (packed shift right arithmetic) is used to sign extend the values.

Example 5-5 assumes the source is a packed-word (16-bit) data type.

Example 5-5. Signed Unpack Code

```
 Input:
;           XMM0        source value
; Output:
;           XMM0        four sign-extended 32-bit doublewords
;                       from four low-end words
;           XMM1        four sign-extended 32-bit doublewords
;                       from four high-end words
;
```

Example 5-5. Signed Unpack Code (Contd.)

```
movdqa      xmm1, xmm0 ; copy source
punpcklwd   xmm0, xmm0 ; unpack four low end words of the source
                       ; into the upper 16 bits of each doubleword
                       ; in the destination
punpckhwd   xmm1, xmm1 ; unpack 4 high-end words of the source
                       ; into the upper 16 bits of each doubleword
                       ; in the destination

psrad       xmm0, 16   ; sign-extend the 4 low-end words of the source
                       ; into four 32-bit signed doublewords
psrad       xmm1, 16   ; sign-extend the 4 high-end words of the
                       ; source into four 32-bit signed doublewords
```

5.4.3 Interleaved Pack with Saturation

Pack instructions pack two values into a destination register in a predetermined order. PACKSSDW saturates two signed doublewords from a source operand and two signed doublewords from a destination operand into four signed words; and it packs the four signed words into a destination register. See Figure 5-1.

SSE2 extends PACKSSDW so that it saturates four signed doublewords from a source operand and four signed doublewords from a destination operand into eight signed words; the eight signed words are packed into the destination.

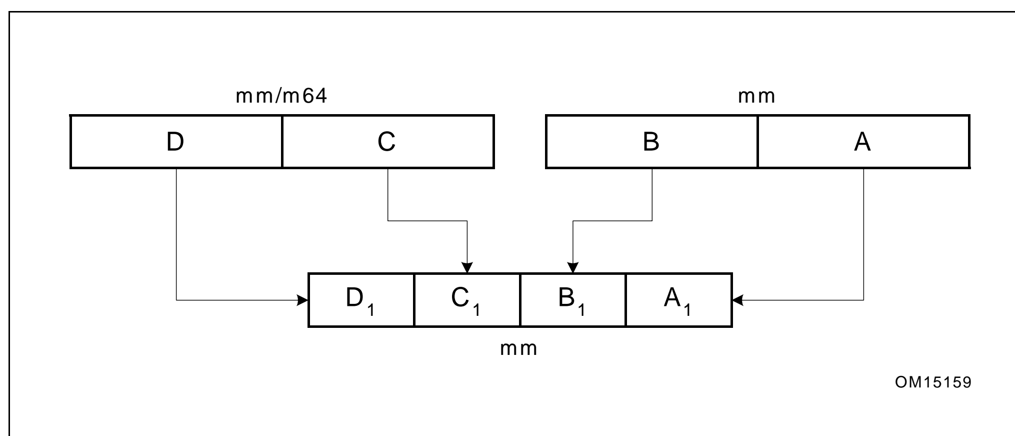


Figure 5-1. PACKSSDW mm, mm/mm64 Instruction

Figure 5-2 illustrates where two pairs of values are interleaved in a destination register; Example 5-6 shows MMX code that accomplishes the operation.

Two signed doublewords are used as source operands and the result is interleaved signed words. The sequence in Example 5-6 can be extended in SSE2 to interleave eight signed words using XMM registers.

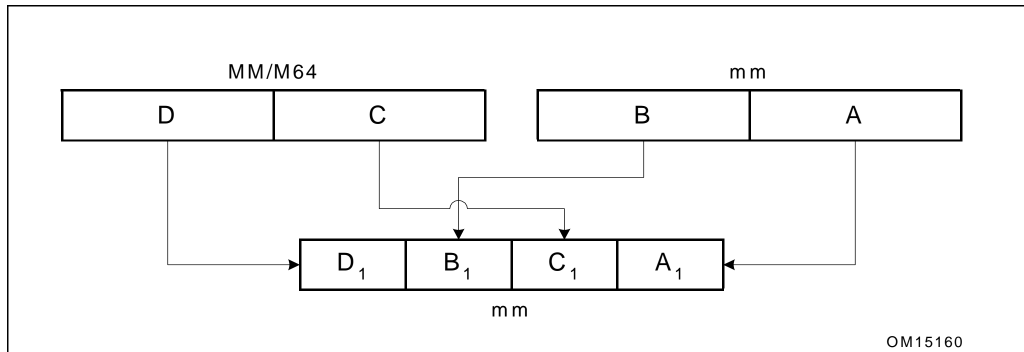


Figure 5-2. Interleaved Pack with Saturation

Example 5-6. Interleaved Pack with Saturation Code

```
; Input:
            MM0     signed source1 value
;           MM1     signed source2 value
; Output:
            MM0     the first and third words contain the
;                   signed-saturated doublewords from MM0,
;                   the second and fourth words contain
;                   signed-saturated doublewords from MM1
;
packssdw    mm0, mm0    ; pack and sign saturate
packssdw    mm1, mm1    ; pack and sign saturate
punpcklwd   mm0, mm1    ; interleave the low-end 16-bit
                        ; values of the operands
```

Pack instructions always assume that source operands are signed numbers. The result in the destination register is always defined by the pack instruction that performs the operation. For example, PACKSSDW packs each of two signed 32-bit values of two sources into four saturated 16-bit signed values in a destination register. PACKUSWB, on the other hand, packs the four signed 16-bit values of two sources into eight saturated eight-bit unsigned values in the destination.

5.4.4 Interleaved Pack without Saturation

Example 5-7 is similar to Example 5-6 except that the resulting words are not saturated. In addition, in order to protect against overflow, only the low order 16 bits of each doubleword are used. Again, Example 5-7 can be extended in SSE2 to accomplish interleaving eight words without saturation.

Example 5-7. Interleaved Pack without Saturation Code

```
; Input:
;       MM0         signed source value
;       MM1         signed source value

; Output:
;       MM0         the first and third words contain the
;                   low 16-bits of the doublewords in MM0,
;                   the second and fourth words contain the
;                   low 16-bits of the doublewords in MM1
```

Example 5-7. Interleaved Pack without Saturation Code (Contd.)

```
pslld   mm1, 16         ; shift the 16 LSB from each of the
                        ; doubleword values to the 16 MSB
                        ; position
pand    mm0, {0,ffff,0,ffff}
                        ; mask to zero the 16 MSB
                        ; of each doubleword value
por     mm0, mm1        ; merge the two operands
```

5.4.5 Non-Interleaved Unpack

Unpack instructions perform an interleave merge of the data elements of the destination and source operands into the destination register.

The following example merges the two operands into destination registers without interleaving. For example, take two adjacent elements of a packed-word data type in SOURCE1 and place this value in the low 32 bits of the results. Then take two adjacent elements of a packed-word data type in SOURCE2 and place this value in the high 32 bits of the results. One of the destination registers will have the combination illustrated in Figure 5-3.

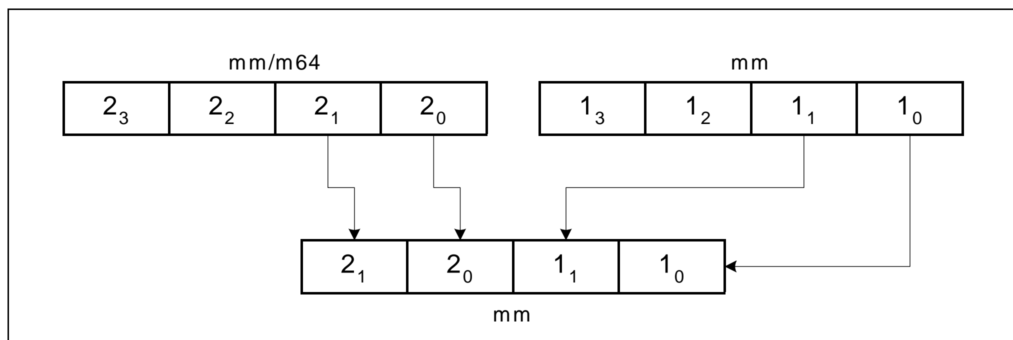


Figure 5-3. Result of Non-Interleaved Unpack Low in MM0

The other destination register will contain the opposite combination illustrated in Figure 5-4.

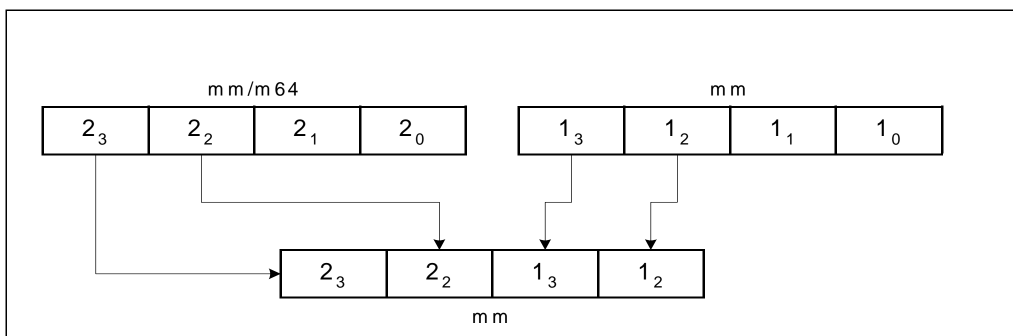


Figure 5-4. Result of Non-Interleaved Unpack High in MM1

Code in the Example 5-8 unpacks two packed-word sources in a non-interleaved way. The goal is to use the instruction which unpacks doublewords to a quadword, instead of using the instruction which unpacks words to doublewords.

Example 5-8. Unpacking Two Packed-word Sources in Non-interleaved Way Code

```
; Input:
;           MM0         packed-word source value
;           MM1         packed-word source value
; Output:
;           MM0         contains the two low-end words of the
;                       original sources, non-interleaved
;           MM2         contains the two high end words of the
;                       original sources, non-interleaved.
movq        mm2, mm0    ; copy source1
punpckldq   mm0, mm1    ; replace the two high-end words of MMO with
                        ; two low-end words of MM1;
                        ; leave the two low-end words of MM0 in place
punpckhdq   mm2, mm1    ; move two high-end words of MM2 to the two low-end
                        ; words of MM2; place the two high-end words of
                        ; MM1 in two high-end words of MM2
```

5.4.6 Extract Data Element

The PEXTRW instruction in SSE takes the word in the designated MMX register selected by the two least significant bits of the immediate value and moves it to the lower half of a 32-bit integer register. See Figure 5-5 and Example 5-9.

With SSE2, PEXTRW can extract a word from an XMM register to the lower 16 bits of an integer register. SSE4.1 provides extraction of a byte, word, dword and qword from an XMM register into either a memory location or integer register.

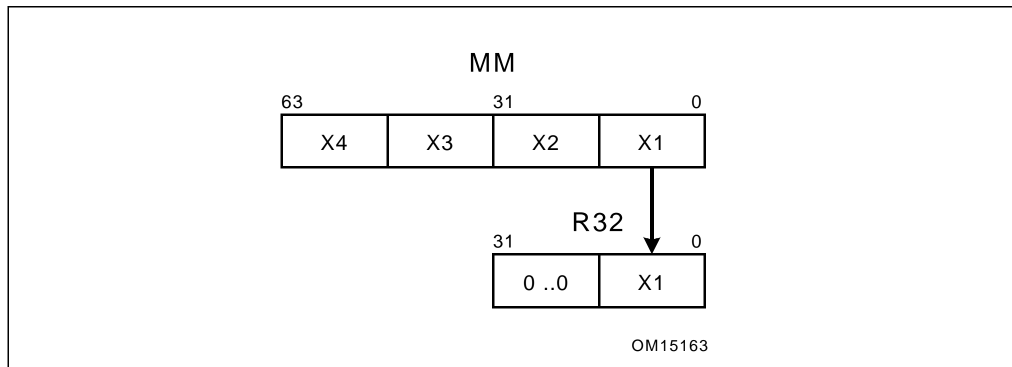


Figure 5-5. PEXTRW Instruction

Example 5-9. PEXTRW Instruction Code

```
; Input:
;         eax      source value
;                  immediate value:   "0"
; Output:
;         edx      32-bit integer register containing the extracted word in the
;                   low-order bits & the high-order bits zero-extended
movq    mm0, [eax]
pextrw  edx, mm0, 0
```

5.4.7 Insert Data Element

The PINSRW instruction in SSE loads a word from the lower half of a 32-bit integer register or from memory and inserts it in an MMX technology destination register at a position defined by the two least significant bits of the immediate constant. Insertion is done in such a way that three other words from the destination register are left untouched. See Figure 5-6 and Example 5-10.

With SSE2, PINSRW can insert a word from the lower 16 bits of an integer register or memory into an XMM register. SSE4.1 provides insertion of a byte, dword and qword from either a memory location or integer register into an XMM register.

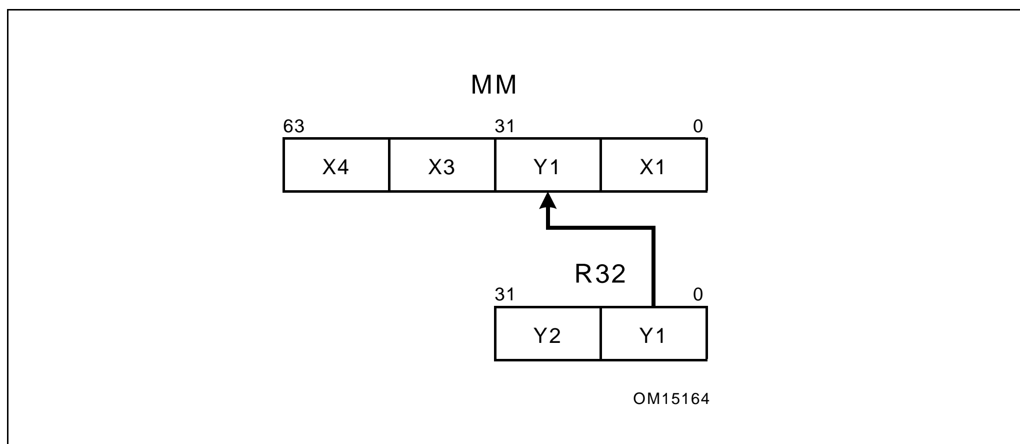


Figure 5-6. PINSRW Instruction

Example 5-10. PINSRW Instruction Code

```
; Input:
;         edx      pointer to source value
; Output:
;         mm0      register with new 16-bit value inserted
;
mov     eax, [edx]
pinsrw  mm0, eax, 1
```

If all of the operands in a register are being replaced by a series of PINSRW instructions, it can be useful to clear the content and break the dependence chain by either using the PXOR instruction or loading the register. See Example 5-11 and Section 3.5.1.8, "Clearing Registers and Dependency Breaking Idioms."

Example 5-11. Repeated PINSRW Instruction Code

```
; Input:
;           edx         pointer to structure containing source
;                       values at offsets: of +0, +10, +13, and +24
;                       immediate value: "1"
; Output:
;           MMX         register with new 16-bit value inserted
;
pxor        mm0, mm0    ; Breaks dependency on previous value of mm0
mov         eax, [edx]
pinsrw      mm0, eax, 0
mov         eax, [edx+10]
pinsrw      mm0, eax, 1
mov         eax, [edx+13]
pinsrw      mm0, eax, 2
mov         eax, [edx+24]
pinsrw      mm0, eax, 3
```

5.4.8 Non-Unit Stride Data Movement

SSE4.1 provides instructions to insert a data element from memory into an XMM register, and to extract.a data element from an XMM register into memory directly. Separate instructions are provided to handle floating-point data and integer byte, word, or dword. These instructions are suited for vectorizing code that loads/stores non-unit stride data from memory, see Example 5-12.

Example 5-12. Non-Unit Stride Load/Store Using SSE4.1 Instructions

/* Goal: Non-Unit Stride Load Dwords*/	/* Goal: Non-Unit Stride Store Dwords*/
movd xmm0, [addr] pinsrd xmm0, [addr + stride], 1 pinsrd xmm0, [addr + 2*stride], 2 pinsrd xmm0, [addr + 3*stride], 3	movd [addr], xmm0 pextrd [addr + stride], xmm0, 1 pextrd [addr + 2*stride], xmm0, 2 pextrd [addr + 3*stride], xmm0, 3

Example 5-13 provides two examples: using INSERTPS and PEXTRD to perform gather operations on floating-point data; using EXTRACTPS and PEXTRD to perform scatter operations on floating-point data.

Example 5-13. Scatter and Gather Operations Using SSE4.1 Instructions

/* Goal: Gather Operation*/	/* Goal: Scatter Operation*/
movd eax, xmm0 movss xmm1, [addr + 4*eax] pextrd eax, xmm0, 1 insertps xmm1, [addr + 4*eax], 1 pextrd eax, xmm0, 2 insertps xmm1, [addr + 4*eax], 2 pextrd eax, xmm0, 3 insertps xmm1, [addr + 4*eax], 3	movd eax, xmm0 movss [addr + 4*eax], xmm1 pextrd eax, xmm0, 1 extractps [addr + 4*eax], xmm1, 1 pextrd eax, xmm0, 2 extractps [addr + 4*eax], xmm1, 2 pextrd eax, xmm0, 3 extractps [addr + 4*eax], xmm1, 3

5.4.9 Move Byte Mask to Integer

The PMOVMSKB instruction returns a bit mask formed from the most significant bits of each byte of its source operand. When used with 64-bit MMX registers, this produces an 8-bit mask, zeroing out the upper 24 bits in the destination register. When used with 128-bit XMM registers, it produces a 16-bit mask, zeroing out the upper 16 bits in the destination register.

The 64-bit version of this instruction is shown in Figure 5-7 and Example 5-14.

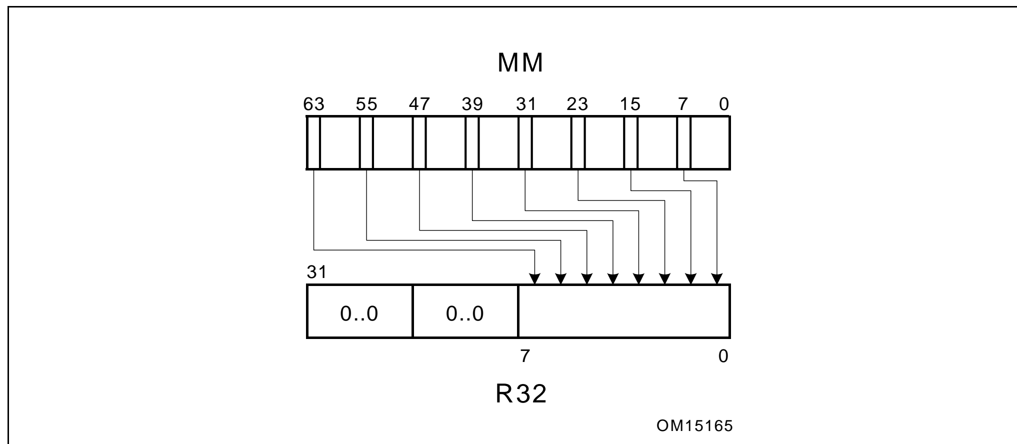


Figure 5-7. PMOVSMKB Instruction

Example 5-14. PMOVMSKB Instruction Code

```
; Input:
;           source value
; Output:
;           32-bit register containing the byte mask in the lower eight bits
;
movq     mm0, [edi]
pmovmskb eax, mm0
```

5.4.10 Packed Shuffle Word for 64-bit Registers

The PSHUFW instruction uses the immediate (IMM8) operand to select between the four words in either two MMX registers or one MMX register and a 64-bit memory location. SSE2 provides PSHUFLW to shuffle the lower four words into an XMM register. In addition to the equivalent to the PSHUFW, SSE2 also provides PSHUFHW to shuffle the higher four words. Furthermore, SSE2 offers PSHUFD to shuffle four dwords into an XMM register. All of these four PSHUF instructions use an immediate byte to encode the data path of individual words within the corresponding 8 bytes from source to destination, shown in Table 5-1.

Table 5-1. PSHUF Encoding

Bits	Words
1 - 0	0
3 - 2	1
5 - 4	2
7 - 6	3

5.4.11 Packed Shuffle Word for 128-bit Registers

The PSHUFLW/PSHUFHW instruction performs a full shuffle of any source word field within the low/high 64 bits to any result word field in the low/high 64 bits, using an 8-bit immediate operand; other high/low 64 bits are passed through from the source operand.

PSHUFD performs a full shuffle of any double-word field within the 128-bit source to any double-word field in the 128-bit result, using an 8-bit immediate operand.

No more than 3 instructions, using PSHUFLW/PSHUFHW/PSHUFD, are required to implement many common data shuffling operations. Broadcast, Swap, and Reverse are illustrated in Example 5-15 and Example 5-16.

Example 5-15. Broadcast a Word Across XMM, Using 2 SSE2 Instructions

```
/* Goal:  Broadcast the value from word 5 to all words */
/* Instruction       Result */
                     | 7| 6| 5| 4| 3| 2| 1| 0|

PSHUFHW (3,2,1,1)| 7| 6| 5| 5| 3| 2| 1| 0|

PSHUFD (2,2,2,2) | 5| 5| 5| 5| 5| 5| 5| 5|
```

Example 5-16. Swap/Reverse words in an XMM, Using 3 SSE2 Instructions

```
/* Goal:  Swap the values in word 6 and word 1 */
/* Instruction       Result */
                     | 7| 6| 5| 4| 3| 2| 1| 0|

PSHUFD (3,0,1,2) | 7| 6| 1| 0| 3| 2| 5| 4|

PSHUFHW (3,1,2,0)| 7| 1| 6| 0| 3| 2| 5| 4|

PSHUFD (3,0,1,2) | 7| 1| 5| 4| 3| 2| 6| 0|
```

```
/* Goal:  Reverse the order of the words */
/* Instruction       Result */
                     | 7| 6| 5| 4| 3| 2| 1| 0|

PSHUFLW (0,1,2,3)| 7| 6| 5| 4| 0| 1| 2| 3|

PSHUFHW (0,1,2,3)| 4| 5| 6| 7| 0| 1| 2| 3|

PSHUFD (1,0,3,2) | 0| 1| 2| 3| 4| 5| 6| 7|
```

5.4.12 Shuffle Bytes

SSSE3 provides PSHUFB; this instruction carries out byte manipulation within a 16 byte range. PSHUFB can replace up to 12 other instructions: including SHIFT, OR, AND and MOV.

Use PSHUFB if the alternative uses 5 or more instructions.

5.4.13 Conditional Data Movement

SSE4.1 provides two packed blend instructions on byte and word data elements in 128-bit operands. Packed blend instructions conditionally copies data elements from selected positions in the source to the corresponding data element using a mask specified by an immediate control byte or an implied XMM register (XMM0). The mask can be generated by a packed compare instruction for example. Thus packed blend instructions are most useful for vectorizing conditional flows within a loop and can be more efficient than inserting single element one at a time for some situations.

5.4.14 Unpacking/interleaving 64-bit Data in 128-bit Registers

The PUNPCKLQDQ/PUNPCHQDQ instructions interleave the low/high-order 64-bits of the source operand and the low/high-order 64-bits of the destination operand. It then writes the results to the destination register.

The high/low-order 64-bits of the source operands are ignored.

5.4.15 Data Movement

There are two additional instructions to enable data movement from 64-bit SIMD integer registers to 128-bit SIMD registers.

The MOVQ2DQ instruction moves the 64-bit integer data from an MMX register (source) to a 128-bit destination register. The high-order 64 bits of the destination register are zeroed-out.

The MOVDQ2Q instruction moves the low-order 64-bits of integer data from a 128-bit source register to an MMX register (destination).

5.4.16 Conversion Instructions

SSE provides Instructions to support 4-wide conversion of single-precision data to/from double-word integer data. Conversions between double-precision data to double-word integer data have been added in SSE2.

SSE4.1 provides 4 rounding instructions to convert floating-point values to integer values with rounding control specified in a more flexible manner and independent of the rounding control in MXCSR. The integer values produced by ROUNDxx instructions are maintained as floating-point data.

SSE4.1 also provides instructions to convert integer data from:

- Packed bytes to packed word/dword/qword format using either sign extension or zero extension.
- Packed words to packed dword/qword format using either sign extension or zero extension.
- Packed dword to packed qword format using either sign extension or zero extension.

5.5 GENERATING CONSTANTS

SIMD integer instruction sets do not have instructions that will load immediate constants to the SIMD registers.

The following code segments generate frequently used constants in the SIMD register. These examples can also be extended in SSE2 by substituting MMX with XMM registers. See Example 5-17.

Example 5-17. Generating Constants

```
pxor      mm0, mm0     ; generate a zero register in MM0
pcmpeq mm1, mm1        ; Generate all 1's in register MM1,
                       ; which is -1 in each of the packed
                       ; data type fields

pxor      mm0, mm0
pcmpeq mm1, mm1
psubb     mm0, mm1 [psubw  mm0, mm1] (psubd  mm0, mm1)
                       ; three instructions above generate
                       ; the constant 1 in every
                       ; packed-byte [or packed-word]
                       ; (or packed-dword) field

pcmpeq mm1, mm1
psrlw     mm1, 16-n(psrld  mm1, 32-n)
                       ; two instructions above generate
                       ; the signed constant 2^n–1 in every
                       ; packed-word (or packed-dword) field

pcmpeq mm1, mm1
psllw     mm1, n   (pslld mm1, n)
                       ; two instructions above generate
                       ; the signed constant -2n in every
                       ; packed-word (or packed-dword) field
```

NOTE

Because SIMD integer instruction sets do not support shift instructions for bytes, 2n–1 and -2n are relevant only for packed words and packed doublewords.

5.6 BUILDING BLOCKS

This section describes instructions and algorithms which implement common code building blocks.

5.6.1 Absolute Difference of Unsigned Numbers

Example 5-18 computes the absolute difference of two unsigned numbers. It assumes an unsigned packed-byte data type.

Here, we make use of the subtract instruction with unsigned saturation. This instruction receives UNSIGNED operands and subtracts them with UNSIGNED saturation. This support exists only for packed bytes and packed words, not for packed doublewords.

Example 5-18. Absolute Difference of Two Unsigned Numbers

```
; Input:
;          MM0 source operand
;          MM1 source operand
; Output:
;          MM0 absolute difference of the unsigned operands
```

Example 5-18. Absolute Difference of Two Unsigned Numbers (Contd.)

```
movq      mm2, mm0     ; make a copy of mm0
psubusbmm0, mm1        ; compute difference one way
psubusbmm1, mm2        ; compute difference the other way
por       mm0, mm1     ; OR them together
```

This example will not work if the operands are signed. Note that PSADBW may also be used in some situations. See Section 5.6.9 for details.

5.6.2 Absolute Difference of Signed Numbers

Example 5-19 computes the absolute difference of two signed numbers using SSSE3 instruction PABSW. This sequence is more efficient than using previous generation of SIMD instruction extensions.

Example 5-19. Absolute Difference of Signed Numbers

```
;Input:
;          XMM0 signed source operand
;          XMM1 signed source operand

;Output:
;          XMM1absolute difference of the unsigned operands

psubw   xmm0, xmm1 ; subtract words
pabsw   xmm1, xmm0 ; results in XMM1
```

5.6.3 Absolute Value

Example 5-20 show an MMX code sequence to compute |X|, where X is signed. This example assumes signed words to be the operands.

With SSSE3, this sequence of three instructions can be replaced by the PABSW instruction. Additionally, SSSE3 provides a 128-bit version using XMM registers and supports byte, word and doubleword granularity.

Example 5-20. Computing Absolute Value

```
; Input:
;          MM0          signed source operand
; Output:
;          MM1          ABS(MMO)
pxor      mm1, mm1     ; set mm1 to all zeros
psubw     mm1, mm0     ; make each mm1 word contain the
                       ; negative of each mm0 word
pmaxswmm1, mm0         ; mm1 will contain only the positive
                       ; (larger) values - the absolute value
```

NOTE

The absolute value of the most negative number (that is, 8000H for 16-bit) cannot be represented using positive numbers. This algorithm will return the original value for the absolute value (8000H).

5.6.4 Pixel Format Conversion

SSSE3 provides the PSHUFB instruction to carry out byte manipulation within a 16-byte range. PSHUFB can replace a set of up to 12 other instruction, including SHIFT, OR, AND and MOV.

Use PSHUFB if the alternative code uses 5 or more instructions. Example 5-21 shows the basic form of conversion of color pixel formats.

Example 5-21. Basic C Implementation of RGBA to BGRA Conversion

```
Standard C Code:
struct RGBA{BYTE r,g,b,a;};
struct BGRA{BYTE b,g,r,a;};

void BGRA_RGBA_Convert(BGRA *source, RGBA *dest, int num_pixels)
{
    for(int i = 0; i < num_pixels; i++){
        dest[i].r = source[i].r;
        dest[i].g = source[i].g;
        dest[i].b = source[i].b;
        dest[i].a = source[i].a;
    }
}
```

Example 5-22 and Example 5-23 show SSE2 code and SSSE3 code for pixel format conversion. In the SSSE3 example, PSHUFB replaces six SSE2 instructions.

Example 5-22. Color Pixel Format Conversion Using SSE2

```
; Optimized for SSE2
mov    esi, src
    mov    edi, dest
    mov  ecx, iterations
    movdqa xmm0, ag_mask //{0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff}
    movdqa xmm5, rb_mask //{ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0}
    mov  eax, remainder
convert16Pixs: // 16 pixels, 64 byte per iteration
    movdqa xmm1, [esi] // xmm1 = [r3g3b3a3,r2g2b2a2,r1g1b1a1,r0g0b0a0]
    movdqa xmm2, xmm1
    movdqa xmm7, xmm1      //xmm7 abgr
    psrld  xmm2, 16        //xmm2 00ab
    pslld  xmm1, 16        //xmm1 gr00

    por    xmm1, xmm2      //xmm1 grab
    pand   xmm7, xmm0      //xmm7 a0g0
    pand   xmm1, xmm5      //xmm1 0r0b
    por    xmm1, xmm7      //xmm1 argb
    movdqa [edi], xmm1
```

Example 5-22. Color Pixel Format Conversion Using SSE2 (Contd.)

```
    //repeats for another 3*16 bytes
    ...

    add   esi, 64
    add   edi, 64
    sub   ecx, 1
    jnz   convert16Pixs
```

Example 5-23. Color Pixel Format Conversion Using SSSE3

```
; Optimized for SSSE3

mov     esi, src
    mov     edi, dest
    mov   ecx, iterations
    movdqa  xmm0, _shufb
// xmm0 = [15,12,13,14,11,8,9,10,7,4,5,6,3,0,1,2]
    mov        eax, remainder

convert16Pixs: // 16 pixels, 64 byte per iteration
    movdqa xmm1, [esi]
// xmm1 = [r3g3b3a3,r2g2b2a2,r1g1b1a1,r0g0b0a0]
    movdqa xmm2, [esi+16]
    pshufb xmm1, xmm0
// xmm1 = [b3g3r3a3,b2g2r2a2,b1g1r1a1,b0g0r0a0]
    movdqa [edi], xmm1

    //repeats for another 3*16 bytes

    ...

    add        esi, 64
    add        edi, 64
    sub        ecx, 1
    jnz        convert16Pixs
```

5.6.5 Endian Conversion

The PSHUFB instruction can also be used to reverse byte ordering within a doubleword. It is more efficient than traditional techniques, such as BSWAP.

Example 5-24 (a) shows the traditional technique using four BSWAP instructions to reverse the bytes within a DWORD. Each BSWAP requires executing two micro-ops. In addition, the code requires 4 loads and 4 stores for processing 4 DWORDs of data.

Example 5-24 (b) shows an SSSE3 implementation of endian conversion using PSHUFB. The reversing of four DWORDs requires one load, one store, and PSHUFB.

On Intel Core microarchitecture, reversing 4 DWORDs using PSHUFB can be approximately twice as fast as using BSWAP.

Example 5-24. Big-Endian to Little-Endian Conversion

;;(a) Using BSWAP	;; (b) Using PSHUFB

<pre>lea eax, src
    lea ecx, dst
    mov edx, elCount
start:
    mov edi, [eax]
    mov esi, [eax+4]
    bswap edi
    mov ebx, [eax+8]
 	
    bswap esi
    mov ebp, [eax+12]
    mov [ecx], edi
    mov [ecx+4], esi
    bswap ebx
    mov [ecx+8], ebx
    bswap ebp
    mov [ecx+12], ebp
 	
    add eax, 16
    add ecx, 16
    sub edx, 4
    jnz start</pre>
	<pre>__declspec(align(16)) BYTE bswapMASK[16] =
	{3,2,1,0, 7,6,5,4, 11,10,9,8, 15,14,13,12};
	    lea eax, src
	    lea ecx, dst
	    mov edx, elCount
	    movaps xmm7, bswapMASK
	start:
	    movdqa xmm0, [eax]
	
	    pshufb xmm0, xmm7
	    movdqa [ecx], xmm0
	    add eax, 16
	    add ecx, 16
	    sub edx, 4
	    jnz start</pre>

5.6.6 Clipping to an Arbitrary Range [High, Low]

This section explains how to clip a values to a range [HIGH, LOW]. Specifically, if the value is less than LOW or greater than HIGH, then clip to LOW or HIGH, respectively. This technique uses the packed-add and packed-subtract instructions with saturation (signed or unsigned), which means that this technique can only be used on packed-byte and packed-word data types.

The examples in this section use the constants PACKED_MAX and PACKED_MIN and show operations on word values. For simplicity, we use the following constants (corresponding constants are used in case the operation is done on byte values):

```
PACKED_MAX equals 0X7FFF7FFF7FFF7FFF
PACKED_MIN equals 0X8000800080008000
PACKED_LOW contains the value LOW in all four words of the packed-words data type
PACKED_HIGH contains the value HIGH in all four words of the packed-words data type
PACKED_USMAX all values equal 1
HIGH_US adds the HIGH value to all data elements (4 words) of PACKED_MIN
LOW_US adds the LOW value to all data elements (4 words) of PACKED_MIN
```

5.6.6.1 Highly Efficient Clipping

For clipping signed words to an arbitrary range, the PMAXSW and PMINSW instructions may be used. For clipping unsigned bytes to an arbitrary range, the PMAXUB and PMINUB instructions may be used.

Example 5-25 shows how to clip signed words to an arbitrary range; the code for clipping unsigned bytes is similar.

Example 5-25. Clipping to a Signed Range of Words [High, Low]

```
; Input:
;       MM0     signed source operands
; Output:
;       MM0     signed words clipped to the signed
;               range [high, low]
pminsw mm0, packed_high
pmaxswmm0, packed_low
```

With SSE4.1, Example 5-25 can be easily extended to clip signed bytes, unsigned words, signed and unsigned dwords.

Example 5-26. Clipping to an Arbitrary Signed Range [High, Low]

```
; Input:
;       MM0                 signed source operands
; Output:
;       MM1                 signed operands clipped to the unsigned
;                           range [high, low]
paddw   mm0, packed_min     ; add with no saturation
                            ; 0x8000 to convert to unsigned
padduswmm0, (packed_usmax - high_us)
                            ; in effect this clips to high
psubuswmm0, (packed_usmax - high_us + low_us)
                            ; in effect this clips to low
paddw   mm0, packed_low     ; undo the previous two offsets
```

The code above converts values to unsigned numbers first and then clips them to an unsigned range. The last instruction converts the data back to signed data and places the data within the signed range.

Conversion to unsigned data is required for correct results when (High - Low) < 0X8000. If (High - Low) >= 0X8000, simplify the algorithm as in Example 5-27.

Example 5-27. Simplified Clipping to an Arbitrary Signed Range

```
; Input:      MM0       signed source operands
; Output:     MM1       signed operands clipped to the unsigned
;                       range [high, low]
paddssw     mm0, (packed_max - packed_high)
                        ; in effect this clips to high
psubssw     mm0, (packed_usmax - packed_high + packed_low)
                        ; clips to low
paddw       mm0, low    ; undo the previous two offsets
```

This algorithm saves a cycle when it is known that (High - Low) >= 0x8000. The three-instruction algorithm does not work when (High - Low) < 0x8000 because 0xffff minus any number < 0x8000 will yield a number greater in magnitude than 0x8000 (which is a negative number).

When the second instruction, psubssw MM0, (0xffff - High + Low) in the three-step algorithm (Example 5-27) is executed, a negative number is subtracted. The result of this subtraction causes the values in MM0 to be increased instead of decreased, as should be the case, and an incorrect answer is generated.

5.6.6.2 Clipping to an Arbitrary Unsigned Range [High, Low]

Example 5-28 clips an unsigned value to the unsigned range [High, Low]. If the value is less than low or greater than high, then clip to low or high, respectively. This technique uses the packed-add and packed-subtract instructions with unsigned saturation, thus the technique can only be used on packed-bytes and packed-words data types.

Figure 5-28 illustrates operation on word values.

Example 5-28. Clipping to an Arbitrary Unsigned Range [High, Low]

```
; Input:
;           MM0     unsigned source operands
; Output:
;           MM1     unsigned operands clipped to the unsigned
;                   range [HIGH, LOW]
paddusw     mm0, 0xffff - high
                    ; in effect this clips to high
psubusw     mm0, (0xffff - high + low)
                    ; in effect this clips to low
paddw       mm0, low
                    ; undo the previous two offsets
```

5.6.7 Packed Max/Min of Byte, Word and Dword

The PMAXSW instruction returns the maximum between four signed words in either of two SIMD registers, or one SIMD register and a memory location.

The PMINSW instruction returns the minimum between the four signed words in either of two SIMD registers, or one SIMD register and a memory location.

The PMAXUB instruction returns the maximum between the eight unsigned bytes in either of two SIMD registers, or one SIMD register and a memory location.

The PMINUB instruction returns the minimum between the eight unsigned bytes in either of two SIMD registers, or one SIMD register and a memory location.

SSE2 extended PMAXSW/PMAXUB/PMINSW/PMINUB to 128-bit operations. SSE4.1 adds 128-bit operations for signed bytes, unsigned word, signed and unsigned dword.

5.6.8 Packed Multiply Integers

The PMULHUW/PMULHW instruction multiplies the unsigned/signed words in the destination operand with the unsigned/signed words in the source operand. The high-order 16 bits of the 32-bit intermediate results are written to the destination operand. The PMULLW instruction multiplies the signed words in the destination operand with the signed words in the source operand. The low-order 16 bits of the 32-bit intermediate results are written to the destination operand.

SSE2 extended PMULHUW/PMULHW/PMULLW to 128-bit operations and adds PMULUDQ.

The PMULUDQ instruction performs an unsigned multiply on the lower pair of double-word operands within 64-bit chunks from the two sources; the full 64-bit result from each multiplication is returned to the destination register.

This instruction is added in both a 64-bit and 128-bit version; the latter performs 2 independent operations, on the low and high halves of a 128-bit register.

SSE4.1 adds 128-bit operations of PMULDQ and PMULLD. The PMULLD instruction multiplies the signed dwords in the destination operand with the signed dwords in the source operand. The low-order 32 bits of the 64-bit intermediate results are written to the destination operand. The PMULDQ instruction multiplies the two low-order, signed dwords in the destination operand with the two low-order, signed dwords in the source operand and stores two 64-bit results in the destination operand.

5.6.9 Packed Sum of Absolute Differences

The PSADBW instruction computes the absolute value of the difference of unsigned bytes for either two SIMD registers, or one SIMD register and a memory location. The differences of 8 pairs of unsigned bytes are then summed to produce a word result in the lower 16-bit field, and the upper three words are set to zero. With SSE2, PSADBW is extended to compute two word results.

The subtraction operation presented above is an absolute difference. That is, T = ABS(X-Y). Byte values are stored in temporary space, all values are summed together, and the result is written to the lower word of the destination register.

Motion estimation involves searching reference frames for best matches. Sum absolute difference (SAD) on two blocks of pixels is a common ingredient in video processing algorithms to locate matching blocks of pixels. PSADBW can be used as building blocks for finding best matches by way of calculating SAD results on 4x4, 8x4, 8x8 blocks of pixels.

5.6.10 MPSADBW and PHMINPOSUW

The MPSADBW instruction in SSE4.1 performs eight SAD operations. Each SAD operation produces a word result from 4 pairs of unsigned bytes. With 8 SAD result in an XMM register, PHMINPOSUM can help search for the best match between eight 4x4 pixel blocks.

For motion estimation algorithms, MPSADBW is likely to improve over PSADBW in several ways:

- Simplified data movement to construct packed data format for SAD computation on pixel blocks.
- Higher throughput in terms of SAD results per iteration (less iteration required per frame).
- MPSADBW results are amenable to efficient search using PHMINPOSUW.

Examples of MPSADBW vs. PSADBW for 4x4 and 8x8 block search can be found in the white paper listed in the reference section of Chapter 1.

5.6.11 Packed Average (Byte/Word)

The PAVGB and PAVGW instructions add the unsigned data elements of the source operand to the unsigned data elements of the destination register, along with a carry-in. The results of the addition are then independently shifted to the right by one bit position. The high order bits of each element are filled with the carry bits of the corresponding sum.

The destination operand is an SIMD register. The source operand can either be an SIMD register or a memory operand.

The PAVGB instruction operates on packed unsigned bytes and the PAVGW instruction operates on packed unsigned words.

5.6.12 Complex Multiply by a Constant

Complex multiplication is an operation which requires four multiplications and two additions. This is exactly how the PMADDWD instruction operates. In order to use this instruction, you need to format the data into multiple 16-bit values. The real and imaginary components should be 16-bits each. Consider Example 5-29, which assumes that the 64-bit MMX registers are being used:

- Let the input data be DR and DI, where DR is real component of the data and DI is imaginary component of the data.
- Format the constant complex coefficients in memory as four 16-bit values [CR -CI CI CR]. Remember to load the values into the MMX register using MOVQ.
- The real component of the complex product is PR = DR*CR - DI*CI and the imaginary component of the complex product is PI = DR*CI + DI*CR.

- The output is a packed doubleword. If needed, a pack instruction can be used to convert the result to 16-bit (thereby matching the format of the input).

Example 5-29. Complex Multiply by a Constant

```
; Input:
;           MM0         complex value, Dr, Di
;           MM1         constant complex coefficient in the form
;                       [Cr -Ci Ci Cr]
; Output:
;           MM0         two 32-bit dwords containing [Pr Pi]
;
punpckldq   mm0, mm0    ; makes [dr di dr di]
pmaddwd     mm0, mm1    ; done, the result is
                        ; [(Dr*Cr-Di*Ci)(Dr*Ci+Di*Cr)]
```

5.6.13 Packed 64-bit Add/Subtract

The PADDQ/PSUBQ instructions add/subtract quad-word operands within each 64-bit chunk from the two sources; the 64-bit result from each computation is written to the destination register. Like the integer ADD/SUB instruction, PADDQ/PSUBQ can operate on either unsigned or signed (two's complement notation) integer operands.

When an individual result is too large to be represented in 64-bits, the lower 64-bits of the result are written to the destination operand and therefore the result wraps around. These instructions are added in both a 64-bit and 128-bit version; the latter performs 2 independent operations, on the low and high halves of a 128-bit register.

5.6.14 128-bit Shifts

The PSLLDQ/PSRLDQ instructions shift the first operand to the left/right by the number of bytes specified by the immediate operand. The empty low/high-order bytes are cleared (set to zero).

If the value specified by the immediate operand is greater than 15, then the destination is set to all zeros.

5.6.15 PTEST and Conditional Branch

SSE4.1 offers PTEST instruction that can be used in vectorizing loops with conditional branches. PTEST is an 128-bit version of the general-purpose instruction TEST. The ZF or CF field of the EFLAGS register are modified as a result of PTEST.

Example 5-30(a) depicts a loop that requires a conditional branch to handle the special case of divide-by-zero. In order to vectorize such loop, any iteration that may encounter divide-by-zero must be treated outside the vectorizable iterations.

Example 5-30. Using PTEST to Separate Vectorizable and non-Vectorizable Loop Iterations

<table>
<tr>
<td>

```
(a) /* Loops requiring infrequent exception
handling*/
float a[CNT];
unsigned int i;

    for (i=0;i<CNT;i++)
    {
        if (a[i] != 0.0)
        {     a[i] = 1.0f/a[i];
        }
        else
        {     call DivException();
        }
    }
```

</td>
<td>

```
(b) /* PTEST enables early out to handle infrequent, non-vectorizable
portion*/
    xor       eax,eax
    movaps    xmm7, [all_ones]
    xorps     xmm6, xmm6
lp:
    movaps    xmm0, a[eax]
    cmpeqps xmm6, xmm0 ; convert each non-zero to ones
    ptest     xmm6, xmm7
    jnc   zero_present; carry will be set if all 4 were non-zero
    movaps    xmm1,[_1_0f_]
    divps     xmm1, xmm0
    movaps    a[eax], xmm1
    add       eax, 16
    cmp       eax, CNT
    jnz       lp
    jmp       end
zero_present:
// execute one by one, call
// exception when value is zero
```

</td>
</tr>
</table>

Example 5-30(b) shows an assembly sequence that uses PTEST to cause an early-out branch whenever any one of the four floating-point values in xmm0 is zero. The fall-through path enables the rest of the floating-point calculations to be vectorized because none of the four values are zero.

5.6.16 Vectorization of Heterogeneous Computations across Loop Iterations

Vectorization techniques on un-rolled loops generally rely on repetitive, homogeneous operations between each loop iteration. Using SSE4.1's PTEST and variable blend instructions, vectorization of heterogeneous operations across loop iterations may be possible.

Example 5-31(a) depicts a simple heterogeneous loop. The heterogeneous operation and conditional branch makes simple loop-unrolling technique infeasible for vectorization.

Example 5-31. Using PTEST and Variable BLEND to Vectorize Heterogeneous Loops

<table>
<tr>
<td>

```
(a) /* Loops with heterogeneous operation
across iterations*/
float a[CNT];
unsigned int i;


for (i=0;i<CNT;i++)
{
    if (a[i] > b[i])
    { a[i] += b[i]; }
    else
    { a[i] -= b[i]; }
}
```

</td>
<td>

```
(b) /* Vectorize Condition Flow with PTEST, BLENDVPS*/
    xor       eax,eax
lp:
    movaps    xmm0, a[eax]
    movaps    xmm1, b[eax]
    movaps    xmm2, xmm0
    // compare a and b values
    cmpgtps  xmm0, xmm1
    // xmm3 - will hold -b
    movaps    xmm3, [SIGN_BIT_MASK]
    xorps     xmm3, xmm1
```

</td>
</tr>
</table>

Example 5-31. Using PTEST and Variable BLEND to Vectorize Heterogeneous Loops (Contd.)

	// select values for the add operation,
	// true condition produce a+b, false will become a+(-b)
	// blend mask is xmm0
	blendvps xmm1,xmm3, xmm0
	addps xmm2, xmm1
	movaps a[eax], xmm2
	add eax, 16
	cmp eax, CNT
	jnz lp

Example 5-31(b) depicts an assembly sequence that uses BLENDVPS and PTEST to vectorize the handling of heterogeneous computations occurring across four consecutive loop iterations.

5.6.17 Vectorization of Control Flows in Nested Loops

The PTEST and BLENDVPx instructions can be used as building blocks to vectorize more complex control-flow statements, where each control flow statement is creating a "working" mask used as a predicate of which the conditional code under the mask will operate.

The Mandelbrot-set map evaluation is useful to illustrate a situation with more complex control flows in nested loops. The Mandelbrot-set is a set of height values mapped to a 2-D grid. The height value is the number of Mandelbrot iterations (defined over the complex number space as $I_n = I_{n-1}^2 + I_0$) needed to get $|I_n| > 2$. It is common to limit the map generation by setting some maximum threshold value of the height, all other points are assigned with a height equal to the threshold. Example 5-32 shows an example of Mandelbrot map evaluation implemented in C.

Example 5-32. Baseline C Code for Mandelbrot Set Map Evaluation

```
#define DIMX (64)
#define DIMY (64)
#define X_STEP (0.5f/DIMX)
#define Y_STEP (0.4f/(DIMY/2))
int map[DIMX][DIMY];

void mandelbrot_C()
{   int i,j;
    float x,y;
    for (i=0,x=-1.8f;i<DIMX;i++,x+=X_STEP)
    {
        for (j=0,y=-0.2f;j<DIMY/2;j++,y+=Y_STEP)
        {float sx,sy;
            int iter = 0;
            sx = x;
            sy = y;
```

Example 5-32. Baseline C Code for Mandelbrot Set Map Evaluation (Contd.)

```
                while (iter < 256)
                {     if (sx*sx + sy*sy >= 4.0f)     break;
                      float old_sx = sx;
                      sx = x + sx*sx - sy*sy;
                      sy = y + 2*old_sx*sy;
                      iter++;
                }
                map[i][j] = iter;
          }
     }
}
```

Example 5-33 shows a vectorized implementation of Mandelbrot map evaluation. Vectorization is not done on the inner most loop, because the presence of the break statement implies the iteration count will vary from one pixel to the next. The vectorized version take into account the parallel nature of 2-D, vectorize over four iterations of Y values of 4 consecutive pixels, and conditionally handles three scenarios:

- In the inner most iteration, when all 4 pixels do not reach break condition, vectorize 4 pixels.
- When one or more pixels reached break condition, use blend intrinsics to accumulate the complex height vector for the remaining pixels not reaching the break condition and continue the inner iteration of the complex height vector.
- When all four pixels reached break condition, exit the inner loop.

Example 5-33. Vectorized Mandelbrot Set Map Evaluation Using SSE4.1 Intrinsics

```
__declspec(align(16)) float _INIT_Y_4[4] = {0,Y_STEP,2*Y_STEP,3*Y_STEP};
F32vec4 _F_STEP_Y(4*Y_STEP);
I32vec4 _I_ONE_ = _mm_set1_epi32(1);
F32vec4 _F_FOUR_(4.0f);
F32vec4 _F_TWO_(2.0f);;

void mandelbrot_C()
{    int i,j;
     F32vec4 x,y;

     for (i = 0, x = F32vec4(-1.8f); i < DIMX; i ++, x += F32vec4(X_STEP))
     {
          for (j = DIMY/2, y = F32vec4(-0.2f) +
                *(F32vec4*)_INIT_Y_4; j < DIMY; j += 4, y += _F_STEP_Y)
          {     F32vec4 sx,sy;
                I32vec4 iter = _mm_setzero_si128();
                int scalar_iter = 0;
                sx = x;
                sy = y;
```

Example 5-33. Vectorized Mandelbrot Set Map Evaluation Using SSE4.1 Intrinsics (Contd.)

```
            while (scalar_iter < 256)
            {    int mask = 0;
                 F32vec4 old_sx = sx;
                 __m128 vmask = _mm_cmpnlt_ps(sx*sx + sy*sy,_F_FOUR_);
                 // if all data points in our vector are hitting the "exit" condition,
                 // the vectorized loop can exit
                 if (_mm_test_all_ones(_mm_castps_si128(vmask)))
                      break;
                                    (continue)
// if non of the data points are out, we don't need the extra code which blends the results
                 if (_mm_test_all_zeros(_mm_castps_si128(vmask),
                      _mm_castps_si128(vmask)))
                 {    sx = x + sx*sx - sy*sy;
                      sy = y + _F_TWO_*old_sx*sy;
                      iter += _I_ONE_;
                 }
                 else
                 {
// Blended flavour of the code, this code blends values from previous iteration with the values
// from current iteration. Only values which did not hit the "exit" condition are being stored;
// values which are already "out" are maintaining their value
                      sx = _mm_blendv_ps(x + sx*sx - sy*sy,sx,vmask);
                      sy = _mm_blendv_ps(y + _F_TWO_*old_sx*sy,sy,vmask);
                      iter = I32vec4(_mm_blendv_epi8(iter + _I_ONE_,
                           iter,_mm_castps_si128(vmask)));
                 }
                 scalar_iter++;
            }
            _mm_storeu_si128((__m128i*)&map[i][j],iter);
        }
    }
}
```

5.7 MEMORY OPTIMIZATIONS

You can improve memory access using the following techniques:

- Avoiding partial memory accesses.
- Increasing the bandwidth of memory fills and video fills.
- Prefetching data with Streaming SIMD Extensions. See Chapter 7, "Optimizing Cache Usage."

MMX registers and XMM registers allow you to move large quantities of data without stalling the processor. Instead of loading single array values that are 8, 16, or 32 bits long, consider loading the values in a single quadword or double quadword and then incrementing the structure or array pointer accordingly.

Any data that will be manipulated by SIMD integer instructions should be loaded using either:

- An SIMD integer instruction that loads a 64-bit or 128-bit operand (for example: MOVQ MM0, M64).
- The register-memory form of any SIMD integer instruction that operates on a quadword or double quadword memory operand (for example, PMADDW MM0, M64).

All SIMD data should be stored using an SIMD integer instruction that stores a 64-bit or 128-bit operand (for example: MOVQ M64, MM0).

The goal of the above recommendations is twofold. First, the loading and storing of SIMD data is more efficient using the larger block sizes. Second, following the above recommendations helps to avoid mixing of 8-, 16-, or 32-bit load and store operations with SIMD integer technology load and store operations to the same SIMD data.

This prevents situations in which small loads follow large stores to the same area of memory, or large loads follow small stores to the same area of memory. The Pentium II, Pentium III, and Pentium 4 processors may stall in such situations. See Chapter 3 for details.

5.7.1 Partial Memory Accesses

Consider a case with a large load after a series of small stores to the same area of memory (beginning at memory address MEM). The large load stalls in the case shown in Example 5-34.

Example 5-34. A Large Load after a Series of Small Stores (Penalty)

```
mov     mem, eax        ; store dword to address "mem"
mov     mem + 4, ebx    ; store dword to address "mem + 4"
    :
    :
movq    mm0, mem        ; load qword at address "mem", stalls
```

MOVQ must wait for the stores to write memory before it can access all data it requires. This stall can also occur with other data types (for example, when bytes or words are stored and then words or doublewords are read from the same area of memory). When you change the code sequence as shown in Example 5-35, the processor can access the data without delay.

Example 5-35. Accessing Data Without Delay

```
movd    mm1, ebx        ; build data into a qword first
                        ; before storing it to memory
movd    mm2, eax
psllq   mm1, 32

por     mm1, mm2
movq    mem, mm1        ; store SIMD variable to "mem" as
                        ; a qword
    :
    :
movq    mm0, mem        ; load qword SIMD "mem", no stall
```

Consider a case with a series of small loads after a large store to the same area of memory (beginning at memory address MEM), as shown in Example 5-36. Most of the small loads stall because they are not aligned with the store. See Section 3.6.5, "Store Forwarding," for details.

Example 5-36. A Series of Small Loads After a Large Store

```
movq    mem, mm0        ; store qword to address "mem"
    :
    :
mov     bx, mem + 2     ; load word at "mem + 2" stalls
mov     cx, mem + 4     ; load word at "mem + 4" stalls
```

The word loads must wait for the quadword store to write to memory before they can access the data they require. This stall can also occur with other data types (for example: when doublewords or words are stored and then words or bytes are read from the same area of memory).

When you change the code sequence as shown in Example 5-37, the processor can access the data without delay.

Example 5-37. Eliminating Delay for a Series of Small Loads after a Large Store

```
movq    mem, mm0     ; store qword to address "mem"
      :
      :

movq    mm1, mem     ; load qword at address "mem"
movd    eax, mm1     ; transfer "mem + 2" to eax from
                     ; MMX register, not memory

psrlq   mm1, 32
shr     eax, 16
movd    ebx, mm1     ; transfer "mem + 4" to bx from
                     ; MMX register, not memory
and     ebx, 0ffffh
```

These transformations, in general, increase the number of instructions required to perform the desired operation. For Pentium II, Pentium III, and Pentium 4 processors, the benefit of avoiding forwarding problems outweighs the performance penalty due to the increased number of instructions.

5.7.1.1 Supplemental Techniques for Avoiding Cache Line Splits

Video processing applications sometimes cannot avoid loading data from memory addresses that are not aligned to 16-byte boundaries. An example of this situation is when each line in a video frame is averaged by shifting horizontally half a pixel.

Example shows a common operation in video processing that loads data from memory address not aligned to a 16-byte boundary. As video processing traverses each line in the video frame, it experiences a cache line split for each 64 byte chunk loaded from memory.

Example 5-38. An Example of Video Processing with Cache Line Splits

```
// Average half-pels horizontally (on // the "x" axis),
// from one reference frame only.

nextLinesLoop:
movdqu xmm0, XMMWORD PTR [edx]  // may not be 16B aligned
movdqu xmm0, XMMWORD PTR [edx+1]
movdqu xmm1, XMMWORD PTR [edx+eax]
movdqu xmm1, XMMWORD PTR [edx+eax+1]

pavgbxmm0, xmm1
pavgbxmm2, xmm3
movdqaXMMWORD PTR [ecx], xmm0
movdqaXMMWORD PTR [ecx+eax], xmm2
// (repeat ...)
```

SSE3 provides an instruction LDDQU for loading from memory address that are not 16-byte aligned. LDDQU is a special 128-bit unaligned load designed to avoid cache line splits. If the address of the load is aligned on a 16-byte boundary, LDQQU loads the 16 bytes requested. If the address of the load is not aligned on a 16-byte boundary, LDDQU loads a 32-byte block starting at the 16-byte aligned address immediately below the address of the load request. It then provides the requested 16 bytes. If the

address is aligned on a 16-byte boundary, the effective number of memory requests is implementation dependent (one, or more).

LDDQU is designed for programming usage of loading data from memory without storing modified data back to the same address. Thus, the usage of LDDQU should be restricted to situations where no store-to-load forwarding is expected. For situations where store-to-load forwarding is expected, use regular store/load pairs (either aligned or unaligned based on the alignment of the data accessed).

Example 5-39. Video Processing Using LDDQU to Avoid Cache Line Splits

```
// Average half-pels horizontally (on // the "x" axis),
// from one reference frame only.
nextLinesLoop:
lddqu xmm0, XMMWORD PTR [edx]  // may not be 16B aligned
lddqu xmm0, XMMWORD PTR [edx+1]
lddqu xmm1, XMMWORD PTR [edx+eax]
lddqu xmm1, XMMWORD PTR [edx+eax+1]
pavgbxmm0, xmm1
pavgbxmm2, xmm3
movdqaXMMWORD PTR [ecx], xmm0  //results stored elsewhere
movdqaXMMWORD PTR [ecx+eax], xmm2
// (repeat ...)
```

5.7.2 Increasing Bandwidth of Memory Fills and Video Fills

It is beneficial to understand how memory is accessed and filled. A memory-to-memory fill (for example a memory-to-video fill) is defined as a 64-byte (cache line) load from memory which is immediately stored back to memory (such as a video frame buffer).

The following are guidelines for obtaining higher bandwidth and shorter latencies for sequential memory fills (video fills). These recommendations are relevant for all Intel architecture processors with MMX technology and refer to cases in which the loads and stores do not hit in the first- or second-level cache.

5.7.2.1 Increasing Memory Bandwidth Using the MOVDQ Instruction

Loading any size data operand will cause an entire cache line to be loaded into the cache hierarchy. Thus, any size load looks more or less the same from a memory bandwidth perspective. However, using many smaller loads consumes more microarchitectural resources than fewer larger stores. Consuming too many resources can cause the processor to stall and reduce the bandwidth that the processor can request of the memory subsystem.

Using MOVDQ to store the data back to UC memory (or WC memory in some cases) instead of using 32-bit stores (for example, MOVD) will reduce by three-quarters the number of stores per memory fill cycle. As a result, using the MOVDQ in memory fill cycles can achieve significantly higher effective bandwidth than using MOVD.

5.7.2.2 Increasing Memory Bandwidth by Loading and Storing to and from the Same DRAM Page

DRAM is divided into pages, which are not the same as operating system (OS) pages. The size of a DRAM page is a function of the total size of the DRAM and the organization of the DRAM. Page sizes of several Kilobytes are common. Like OS pages, DRAM pages are constructed of sequential addresses. Sequential memory accesses to the same DRAM page have shorter latencies than sequential accesses to different DRAM pages.

In many systems the latency for a page miss (that is, an access to a different page instead of the page previously accessed) can be twice as large as the latency of a memory page hit (access to the same page

as the previous access). Therefore, if the loads and stores of the memory fill cycle are to the same DRAM page, a significant increase in the bandwidth of the memory fill cycles can be achieved.

5.7.2.3 Increasing UC and WC Store Bandwidth by Using Aligned Stores

Using aligned stores to fill UC or WC memory will yield higher bandwidth than using unaligned stores. If a UC store or some WC stores cross a cache line boundary, a single store will result in two transaction on the bus, reducing the efficiency of the bus transactions. By aligning the stores to the size of the stores, you eliminate the possibility of crossing a cache line boundary, and the stores will not be split into separate transactions.

5.7.3 Reverse Memory Copy

Copying blocks of memory from a source location to a destination location in reverse order presents a challenge for software to make the most out of the machines capabilities while avoiding microarchitectural hazards. The basic, un-optimized C code is shown in Example 5-40.

The simple C code in Example 5-40 is sub-optimal, because it loads and stores one byte at a time (even in situations that hardware prefetcher might have brought data in from system memory to cache).

Example 5-40. Un-optimized Reverse Memory Copy in C

```
unsigned char* src;
unsigned char* dst;
while (len > 0)
{
*dst-- = *src++;
--len;
}
```

Using MOVDQA or MOVDQU, software can load and store up to 16 bytes at a time but must either ensure 16 byte alignment requirement (if using MOVDQA) or minimize the delays MOVDQU may encounter if data span across cache line boundary.

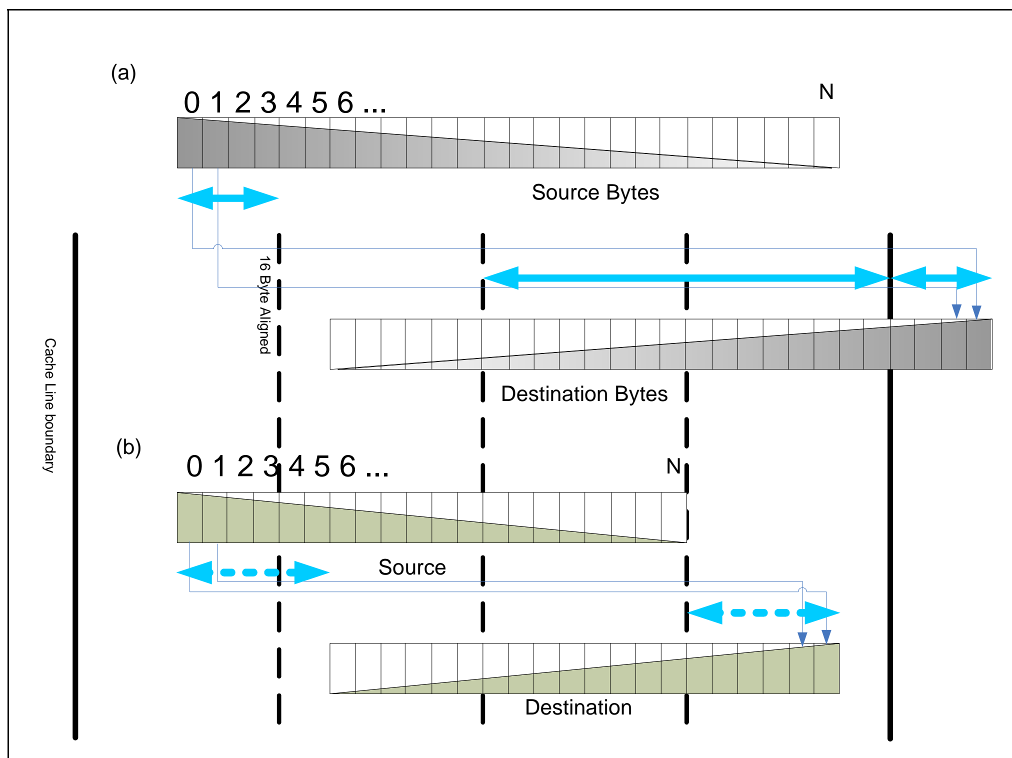


Figure 5-8. Data Alignment of Loads and Stores in Reverse Memory Copy

Given the general problem of arbitrary byte count to copy, arbitrary offsets of leading source byte and destination bytes, address alignment relative to 16 byte and cache line boundaries, these alignment situations can be a bit complicated. Figure 5-8 (a) and (b) depict the alignment situations of reverse memory copy of N bytes.

The general guidelines for dealing with unaligned loads and stores are (in order of importance):

- Avoid stores that span cache line boundaries.
- Minimize the number of loads that span cacheline boundaries.
- Favor 16-byte aligned loads and stores over unaligned versions.

In Figure 5-8 (a), the guidelines above can be applied to the reverse memory copy problem as follows:

1. Peel off several leading destination bytes until it aligns on 16 Byte boundary, then the ensuing destination bytes can be written to using MOVAPS until the remaining byte count falls below 16 bytes.
2. After the leading source bytes have been peeled (corresponding to step 1 above), the source alignment in Figure 5-8 (a) allows loading 16 bytes at a time using MOVAPS until the remaining byte count falls below 16 bytes.

Switching the byte ordering of each 16 bytes of data can be accomplished by a 16-byte mask with PSHUFB. The pertinent code sequence is shown in Example 5-41.

Example 5-41. Using PSHUFB to Reverse Byte Ordering 16 Bytes at a Time

```
__declspec(align(16)) static const unsigned char BswapMask[16] = {15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0};
        mov esi, src
        mov edi, dst
        mov ecx, len
        movaps xmm7, BswapMask
start:
        movdqa xmm0, [esi]
        pshufb xmm0, xmm7
        movdqa [edi-16], xmm0
sub edi, 16
        add esi, 16
        sub ecx, 16
        cmp ecx, 32
        jae start
        //handle left-overs
```

In Figure 5-8 (b), we also start with peeling the destination bytes:

1. Peel off several leading destination bytes until it aligns on 16 Byte boundary, then the ensuing destination bytes can be written to using MOVAPS until the remaining byte count falls below 16 bytes. However, the remaining source bytes are not aligned on 16 byte boundaries, replacing MOVDQA with MOVDQU for loads will inevitably run into cache line splits.
2. To achieve higher data throughput than loading unaligned bytes with MOVDQU, the 16 bytes of data targeted to each of 16 bytes of aligned destination addresses can be assembled using two aligned loads. This technique is illustrated in Figure 5-9.

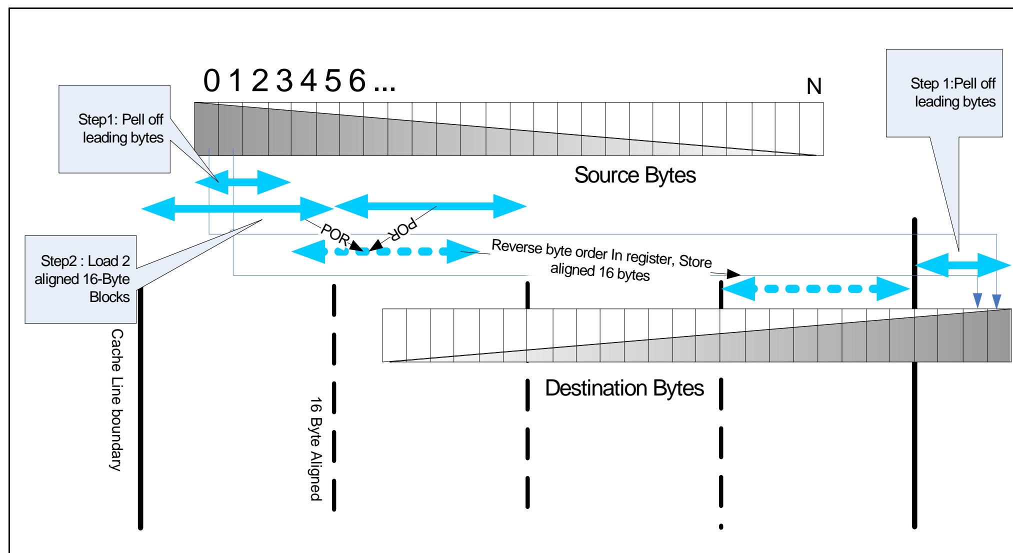


Figure 5-9. A Technique to Avoid Cacheline Split Loads in Reverse Memory Copy Using Two Aligned Loads

5.8 CONVERTING FROM 64-BIT TO 128-BIT SIMD INTEGERS

SSE2 defines a superset of 128-bit integer instructions currently available in MMX technology; the operation of the extended instructions remains. The superset simply operates on data that is twice as wide. This simplifies porting of 64-bit integer applications. However, there are few considerations:

- Computation instructions which use a memory operand that may not be aligned to a 16-byte boundary must be replaced with an unaligned 128-bit load (MOVDQU) followed by the same computation operation that uses instead register operands.

  Use of 128-bit integer computation instructions with memory operands that are not 16-byte aligned will result in a #GP. Unaligned 128-bit loads and stores are not as efficient as corresponding aligned versions; this fact can reduce the performance gains when using the 128-bit SIMD integer extensions.
- General guidelines on the alignment of memory operands are:
  - The greatest performance gains can be achieved when all memory streams are 16-byte aligned.
  - Reasonable performance gains are possible if roughly half of all memory streams are 16-byte aligned and the other half are not.
  - Little or no performance gain may result if all memory streams are not aligned to 16-bytes. In this case, use of the 64-bit SIMD integer instructions may be preferable.
- Loop counters need to be updated because each 128-bit integer instruction operates on twice the amount of data as its 64-bit integer counterpart.
- Extension of the PSHUFW instruction (shuffle word across 64-bit integer operand) across a full 128-bit operand is emulated by a combination of the following instructions: PSHUFHW, PSHUFLW, and PSHUFD.
- Use of the 64-bit shift by bit instructions (PSRLQ, PSLLQ) are extended to 128 bits by:
  - Use of PSRLQ and PSLLQ, along with masking logic operations.
  - A Code sequence rewritten to use the PSRLDQ and PSLLDQ instructions (shift double quad-word operand by bytes).

5.8.1 SIMD Optimizations and Microarchitectures

Pentium M, Intel Core Solo and Intel Core Duo processors have a different microarchitecture than Intel NetBurst microarchitecture. The following sections discuss optimizing SIMD code that targets Intel Core Solo and Intel Core Duo processors.

On Intel Core Solo and Intel Core Duo processors, LDDQU behaves identically to movdqu by loading 16 bytes of data irrespective of address alignment.

5.8.1.1 Packed SSE2 Integer versus MMX Instructions

In general, 128-bit SIMD integer instructions should be favored over 64-bit MMX instructions on Intel Core Solo and Intel Core Duo processors. This is because:

- Improved decoder bandwidth and more efficient micro-op flows relative to the Pentium M processor.
- Wider width of the XMM registers can benefit code that is limited by either decoder bandwidth or execution latency. XMM registers can provide twice the space to store data for in-flight execution. Wider XMM registers can facilitate loop-unrolling or in reducing loop overhead by halving the number of loop iterations.

In microarchitectures prior to Intel Core microarchitecture, execution throughput of 128-bit SIMD integration operations is basically the same as 64-bit MMX operations. Some shuffle/unpack/shift operations do not benefit from the front end improvements. The net impact of using 128-bit SIMD integer instruction on Intel Core Solo and Intel Core Duo processors is likely to be slightly positive overall, but there may be a few situations where their use will generate an unfavorable performance impact.

Intel Core microarchitecture generally executes 128-bit SIMD instructions more efficiently than previous microarchitectures in terms of latency and throughput, many of the limitations specific to Intel Core Duo, Intel Core Solo processors do not apply. The same is true of Intel Core microarchitecture relative to Intel NetBurst microarchitectures.

Enhanced Intel Core microarchitecture provides even more powerful 128-bit SIMD execution capabilities and more comprehensive sets of SIMD instruction extensions than Intel Core microarchitecture. The integer SIMD instructions offered by SSE4.1 operates on 128-bit XMM register only. All of these highly encourages software to favor 128-bit vectorizable code to take advantage of processors based on Enhanced Intel Core microarchitecture and Intel Core microarchitecture.

5.8.1.2 Work-around for False Dependency Issue

In processor based on Intel microarchitecture code name Nehalem, using PMOVSX and PMOVZX instructions to combine data type conversion and data movement in the same instruction will create a false-dependency due to hardware causes. A simple work-around to avoid the false dependency issue is to use PMOVSX, PMOVZX instruction solely for data type conversion and issue separate instruction to move data to destination or from origin.

Example 5-42. PMOVSX/PMOVZX Work-around to Avoid False Dependency

```
#issuing the instruction below will create a false dependency on xmm0
      pmovzxbd xmm0, dword ptr [eax]
// the above instruction may be blocked if xmm0 are updated by other instructions in flight
................................................................

#Alternate solution to avoid false dependency
      movd xmm0, dword ptr [eax] ; OOO hardware can hoist loads to hide latency
      pmovsxbd xmm0, xmm0
```

5.9 TUNING PARTIALLY VECTORIZABLE CODE

Some loop structured code are more difficult to vectorize than others. Example 5-43 depicts a loop carrying out table look-up operation and some arithmetic computation.

Example 5-43. Table Look-up Operations in C Code

```
// pIn1        integer input arrays.
// pOut        integer output array.
// count       size of array.
// LookUpTable    integer values.
TABLE_SIZE        size of the look-up table.
for (unsigned i=0; i < count; i++)
{    pOut[i] =
     ( ( LookUpTable[pIn1[i] % TABLE_SIZE] + pIn1[i] + 17 ) | 17
     ) % 256;
}
```

Although some of the arithmetic computations and input/output to data array in each iteration can be easily vectorizable, but the table look-up via an index array is not. This creates different approaches to

tuning. A compiler can take a scalar approach to execute each iteration sequentially. Hand-tuning of such loops may use a couple of different techniques to handle the non-vectorizable table look-up operation. One vectorization technique is to load the input data for four iteration at once, then use SSE2 instruction to shift out individual index out of an XMM register to carry out table look-up sequentially. The shift technique is depicted by Example 5-44. Another technique is to use PEXTRD in SSE4.1 to extract the index from an XMM directly and then carry out table look-up sequentially. The PEXTRD technique is depicted by Example 5-45.

Example 5-44. Shift Techniques on Non-Vectorizable Table Look-up

```
int modulo[4] = {256-1, 256-1, 256-1, 256-1};
int c[4] = {17, 17, 17, 17};
    mov       esi, pIn1
    mov       ebx, pOut
    mov       ecx, count
    mov       edx, pLookUpTablePTR
    movaps    xmm6, modulo
    movaps    xmm5, c
lloop:
// vectorizable multiple consecutive data accesses
    movaps        xmm4, [esi]          // read 4 indices from pIn1
    movaps        xmm7, xmm4
    pand          xmm7, tableSize
//Table look-up is not vectorizable, shift out one data element to look up table one by one
    movd          eax, xmm7            // get first index
    movd          xmm0, word ptr[edx + eax*4]
    psrldq        xmm7, 4
    movd          eax, xmm7            // get 2nd index
    movd          xmm1, word ptr[edx + eax*4]
    psrldq        xmm7, 4
    movd          eax, xmm7            // get 3rdindex
    movd          xmm2, word ptr[edx + eax*4]
    psrldq        xmm7, 4
    movd          eax, xmm7            // get fourth index
    movd          xmm3, word ptr[edx + eax*4]
//end of scalar part
//packing
    movlhps       xmm1,xmm3
    psllq         xmm1,32
    movlhps       xmm0,xmm2
    orps          xmm0,xmm1
//end of packing
                                                    (continue)

//Vectorizable computation operations
    paddd         xmm0, xmm4 //+pIn1
    paddd         xmm0, xmm5 // +17
    por           xmm0, xmm5
    andps         xmm0, xmm6 //mod
    movaps        [ebx], xmm0
//end of vectorizable operation
```

Example 5-44. Shift Techniques on Non-Vectorizable Table Look-up (Contd.)

```
    add         ebx, 16
    add         esi, 16
    add         edi, 16
    sub         ecx, 1
    test        ecx, ecx
    jne lloop
```

Example 5-45. PEXTRD Techniques on Non-Vectorizable Table Look-up

```
int modulo[4] = {256-1, 256-1, 256-1, 256-1};
int c[4] = {17, 17, 17, 17};
    mov     esi, pIn1
    mov     ebx, pOut
    mov     ecx, count
    mov     edx, pLookUpTablePTR
    movaps  xmm6, modulo
    movaps  xmm5, c
lloop:
// vectorizable multiple consecutive data accesses
    movaps      xmm4, [esi]         // read 4 indices from pIn1
    movaps      xmm7, xmm4
    pand        xmm7, tableSize
//Table look-up is not vectorizable, extract one data element to look up table one by one
    movd        eax, xmm7           // get first index
    mov         eax, [edx + eax*4]
    movd        xmm0, eax

                                                        (continue)

    pextrd      eax, xmm7, 1        // extract 2nd index
    mov         eax, [edx + eax*4]
    pinsrd      xmm0, eax, 1
    pextrd      eax, xmm7, 2        // extract 2nd index
    mov         eax, [edx + eax*4]
    pinsrd      xmm0, eax, 2
    pextrd      eax, xmm7, 3        // extract 2nd index
    mov         eax, [edx + eax*4]
    pinsrd      xmm0, eax, 2
//end of scalar part
//packing not needed
//Vectorizable operations
    paddd       xmm0, xmm4 //+pIn1
    paddd       xmm0, xmm5 // +17
    por         xmm0, xmm5
    andps       xmm0, xmm6 //mod
    movaps      [ebx], xmm0
```

Example 5-45. PEXTRD Techniques on Non-Vectorizable Table Look-up (Contd.)

```
add         ebx, 16
add         esi, 16
add         edi, 16
sub         ecx, 1
test        ecx, ecx
jne lloop
```

The effectiveness of these two hand-tuning techniques on partially vectorizable code depends on the relative cost of transforming data layout format using various forms of pack and unpack instructions.

The shift technique requires additional instructions to pack scalar table values into an XMM to transition into vectorized arithmetic computations. The net performance gain or loss of this technique will vary with the characteristics of different microarchitectures. The alternate PEXTRD technique uses less instruction to extract each index, does not require extraneous packing of scalar data into packed SIMD data format to begin vectorized arithmetic computation.

5.10 PARALLEL MODE AES ENCRYPTION AND DECRYPTION

To deliver optimal encryption and decryption throughput using AESNI, software can optimize by re-ordering the computations and working on multiple blocks in parallel. This can speed up encryption (and decryption) in parallel modes of operation such as ECB, CTR, and CBC-Decrypt (comparing to CBC-Encrypt which is serial mode of operation). See details in Recommendation for Block Cipher Modes of Operation?. The Related Documentation section provides a pointer to this document.

In Intel microarchitecture code name Sandy Bridge, the AES round instructions (AESENC / AESECNLAST / AESDEC / AESDECLAST) have a throughput of one cycle and latency of eight cycles. This allows independent AES instructions for multiple blocks to be dispatched every cycle, if data can be provided sufficiently fast. Compared to the prior Intel microarchitecture code name Westmere, where these instructions have throughput of two cycles and a latency of six cycles, the AES encryption/decryption throughput can be significantly increased, for parallel modes of operation.

To achieve optimal parallel operation with multiple blocks, write the AES software sequences in a way that it computes one AES round on multiple blocks, using one Round Key, and then it continues to compute the subsequent round for multiple blocks, using another Round Key.

For such software optimization, you need to define the number of blocks that are processed in parallel. In Intel microarchitecture code name Sandy Bridge, the optimal parallelization parameter is eight blocks, compared to four blocks on prior microarchitecture.

5.10.1 AES Counter Mode of Operation

Example 5-46 is an example of a function that implements the Counter Mode (CTR mode) of operations while operating on eight blocks in parallel. The following pseudo-code encrypts n data blocks of 16 byte each (PT[i]):

Example 5-46. Pseudo-Code Flow of AES Counter Mode Operation

```
CTRBLK := NONCE || IV || ONE
FOR i := 1 to n-1 DO
    CT[i] := PT[i] XOR AES(CTRBLK)
    CTRBLK := CTRBLK + 1) % 256;
END
CT[n] := PT[n] XOR TRUNC(AES(CTRBLK)) CTRBLK := NONCE || IV || ONE
FOR i := 1 to n-1 DO
    CT[i] := PT[i] XOR AES(CTRBLK)// CT [i] is the i-th ciphetext block
    CTRBLK := CTRBLK + 1
END
CT[n]:= PT[n] XOR TRUNC(AES(CTRBLK))
```

Example 5-47 in the following pages show the assembly implementation of the above code, optimized for Intel microarchitecture code name Sandy Bridge.

Example 5-47. AES128-CTR Implementation with Eight Block in Parallel

```
/*****************************************************************************/
/* This function encrypts an input buffer using AES in CTR mode          */
/* The parameters:                                                 */
/*   const unsigned char *in - pointer to the palintext for encryption or   */
/* ciphertextfor decryption                */
/*   unsigned char *out - pointer to the buffer where the encrypted/decrypted*/
/*                    data will be stored                          */
/*   const unsigned char ivec[8] - 8 bytes of the initialization vector     */
/*   const unsigned char nonce[4] - 4 bytes of the nonce                  */
/*   const unsigned long length - the length of the input in bytes         */
/*   int number_of_rounds - number of AES round. 10 = AES128, 12 = AES192, 14 = AES256 */
/*   unsigned char *key_schedule - pointer to the AES key schedule         */
/*****************************************************************************/
//void AES_128_CTR_encrypt_parallelize_8_blocks_unrolled (
//                              const unsigned char *in,
//                              unsigned char *out,
//                              const unsigned char ivec[8],
//                              const unsigned char nonce[4],
//                              const unsigned long length,
//                              unsigned char *key_schedule)
.align   16,0x90
.align   16
ONE:              .quad 0x00000000,0x00000001
.align   16
FOUR:             .quad 0x00000004,0x00000004
.align   16
EIGHT:            .quad 0x00000008,0x00000008
                                                (continue)
```

Example 5-47. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
.align   16
TWO_N_ONE:      .quad 0x00000002,0x00000001
.align   16
TWO_N_TWO:      .quad 0x00000002,0x00000002
.align   16
LOAD_HIGH_BROADCAST_AND_BSWAP:      .byte 15,14,13,12,11,10,9,8
                                    .byte 15,14,13,12,11,10,9,8
align   16
BSWAP_EPI_64:             .byte 7,6,5,4,3,2,1,0
                          .byte 15,14,13,12,11,10,9,8


.globl AES_CTR_encrypt

AES_CTR_encrypt:
# parameter 1: %rdi         # parameter 2: %rsi
# parameter 3: %rdx         # parameter 4: %rcx
# parameter 5: %r8          # parameter 6: %r9
# parameter 7: 8 + %rsp
movq     %r8, %r10
     movl     8(%rsp), %r12d
     shrq     $4, %r8
     shlq     $60, %r10
     je       NO_PARTS
     addq     $1, %r8
NO_PARTS:
     movq     %r8, %r10
     shlq     $61, %r10
     shrq     $61, %r10

     pinsrq   $1, (%rdx), %xmm0
     pinsrd   $1, (%rcx), %xmm0
     psrldq   $4, %xmm0
     movdqa   %xmm0, %xmm4
     pshufb   (LOAD_HIGH_BROADCAST_AND_BSWAP), %xmm4
     paddq    (TWO_N_ONE), %xmm4
     movdqa   %xmm4, %xmm1
     paddq    (TWO_N_TWO), %xmm4
     movdqa   %xmm4, %xmm2
     paddq    (TWO_N_TWO), %xmm4
     movdqa   %xmm4, %xmm3
     paddq    (TWO_N_TWO), %xmm4
     pshufb   (BSWAP_EPI_64), %xmm1
     pshufb   (BSWAP_EPI_64), %xmm2
     pshufb   (BSWAP_EPI_64), %xmm3
     pshufb   (BSWAP_EPI_64), %xmm4

shrq     $3, %r8
     je       REMAINDER
     subq     $128, %rsi
     subq     $128, %rdi
```

(continue)

Example 5-47. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
LOOP:
    addq    $128, %rsi
    addq    $128, %rdi

    movdqa   %xmm0, %xmm7
    movdqa   %xmm0, %xmm8
    movdqa   %xmm0, %xmm9
    movdqa   %xmm0, %xmm10
    movdqa   %xmm0, %xmm11
    movdqa   %xmm0, %xmm12
    movdqa   %xmm0, %xmm13
    movdqa   %xmm0, %xmm14

    shufpd   $2, %xmm1, %xmm7
    shufpd   $0, %xmm1, %xmm8
    shufpd   $2, %xmm2, %xmm9
    shufpd   $0, %xmm2, %xmm10
    shufpd   $2, %xmm3, %xmm11
    shufpd   $0, %xmm3, %xmm12
    shufpd   $2, %xmm4, %xmm13
    shufpd   $0, %xmm4, %xmm14

    pshufb   (BSWAP_EPI_64), %xmm1
    pshufb   (BSWAP_EPI_64), %xmm2
    pshufb   (BSWAP_EPI_64), %xmm3
    pshufb   (BSWAP_EPI_64), %xmm4

    movdqa   (%r9), %xmm5
    movdqa   16(%r9), %xmm6

    paddq    (EIGHT), %xmm1
    paddq    (EIGHT), %xmm2
    paddq    (EIGHT), %xmm3
    paddq    (EIGHT), %xmm4

    pxor     %xmm5, %xmm7
    pxor     %xmm5, %xmm8
    pxor     %xmm5, %xmm9
    pxor     %xmm5, %xmm10

    pxor     %xmm5, %xmm11
    pxor     %xmm5, %xmm12
    pxor     %xmm5, %xmm13
    pxor     %xmm5, %xmm14

    pshufb   (BSWAP_EPI_64), %xmm1
    pshufb   (BSWAP_EPI_64), %xmm2
    pshufb   (BSWAP_EPI_64), %xmm3
    pshufb   (BSWAP_EPI_64), %xmm4
```

(continue)

Example 5-47. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
aesenc   %xmm6, %xmm7
aesenc   %xmm6, %xmm8
aesenc   %xmm6, %xmm9
aesenc   %xmm6, %xmm10
aesenc   %xmm6, %xmm11
aesenc   %xmm6, %xmm12
aesenc   %xmm6, %xmm13
aesenc   %xmm6, %xmm14

movdqa   32(%r9), %xmm5
movdqa   48(%r9), %xmm6

aesenc   %xmm5, %xmm7
aesenc   %xmm5, %xmm8
aesenc   %xmm5, %xmm9
aesenc   %xmm5, %xmm10
aesenc   %xmm5, %xmm11
aesenc   %xmm5, %xmm12
aesenc   %xmm5, %xmm13
aesenc   %xmm5, %xmm14

aesenc   %xmm6, %xmm7
aesenc   %xmm6, %xmm8
aesenc   %xmm6, %xmm9
aesenc   %xmm6, %xmm10
aesenc   %xmm6, %xmm11
aesenc   %xmm6, %xmm12
aesenc   %xmm6, %xmm13
aesenc   %xmm6, %xmm14

movdqa   64(%r9), %xmm5
movdqa   80(%r9), %xmm6

aesenc   %xmm5, %xmm7
aesenc   %xmm5, %xmm8
aesenc   %xmm5, %xmm9
aesenc   %xmm5, %xmm10
aesenc   %xmm5, %xmm11
aesenc   %xmm5, %xmm12
aesenc   %xmm5, %xmm13
aesenc   %xmm5, %xmm14

aesenc   %xmm6, %xmm7
aesenc   %xmm6, %xmm8
aesenc   %xmm6, %xmm9
aesenc   %xmm6, %xmm10
aesenc   %xmm6, %xmm11
aesenc   %xmm6, %xmm12
aesenc   %xmm6, %xmm13
aesenc   %xmm6, %xmm14
```

(continue)

Example 5-47. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
movdqa   96(%r9), %xmm5
movdqa   112(%r9), %xmm6

aesenc   %xmm5, %xmm7
aesenc   %xmm5, %xmm8
aesenc   %xmm5, %xmm9
aesenc   %xmm5, %xmm10
aesenc   %xmm5, %xmm11
aesenc   %xmm5, %xmm12
aesenc   %xmm5, %xmm13
aesenc   %xmm5, %xmm14

aesenc   %xmm6, %xmm7
aesenc   %xmm6, %xmm8
aesenc   %xmm6, %xmm9
aesenc   %xmm6, %xmm10
aesenc   %xmm6, %xmm11
aesenc   %xmm6, %xmm12
aesenc   %xmm6, %xmm13
aesenc   %xmm6, %xmm14


movdqa   128(%r9), %xmm5
movdqa   144(%r9), %xmm6
movdqa   160(%r9), %xmm15
cmp      $12, %r12d

aesenc   %xmm5, %xmm7
aesenc   %xmm5, %xmm8
aesenc   %xmm5, %xmm9
aesenc   %xmm5, %xmm10
aesenc   %xmm5, %xmm11
aesenc   %xmm5, %xmm12
aesenc   %xmm5, %xmm13
aesenc   %xmm5, %xmm14

aesenc   %xmm6, %xmm7
aesenc   %xmm6, %xmm8
aesenc   %xmm6, %xmm9
aesenc   %xmm6, %xmm10
aesenc   %xmm6, %xmm11
aesenc   %xmm6, %xmm12
aesenc   %xmm6, %xmm13
aesenc   %xmm6, %xmm14

jb       LAST
```

(continue)

Example 5-47. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
    movdqa   160(%r9), %xmm5
    movdqa   176(%r9), %xmm6
    movdqa   192(%r9), %xmm15
    cmp      $14, %r12d

    aesenc   %xmm5, %xmm7
    aesenc   %xmm5, %xmm8
    aesenc   %xmm5, %xmm9
    aesenc   %xmm5, %xmm10
    aesenc   %xmm5, %xmm11
    aesenc   %xmm5, %xmm12
    aesenc   %xmm5, %xmm13
    aesenc   %xmm5, %xmm14


    aesenc   %xmm6, %xmm7
    aesenc   %xmm6, %xmm8
    aesenc   %xmm6, %xmm9
    aesenc   %xmm6, %xmm10
    aesenc   %xmm6, %xmm11
    aesenc   %xmm6, %xmm12
    aesenc   %xmm6, %xmm13
    aesenc   %xmm6, %xmm14

    jb       LAST

    movdqa   192(%r9), %xmm5
    movdqa   208(%r9), %xmm6
    movdqa   224(%r9), %xmm15

    aesenc   %xmm5, %xmm7
    aesenc   %xmm5, %xmm8
    aesenc   %xmm5, %xmm9
    aesenc   %xmm5, %xmm10
    aesenc   %xmm5, %xmm11
    aesenc   %xmm5, %xmm12
    aesenc   %xmm5, %xmm13
    aesenc   %xmm5, %xmm14

    aesenc   %xmm6, %xmm7
    aesenc   %xmm6, %xmm8
    aesenc   %xmm6, %xmm9
    aesenc   %xmm6, %xmm10
    aesenc   %xmm6, %xmm11
    aesenc   %xmm6, %xmm12
    aesenc   %xmm6, %xmm13
    aesenc   %xmm6, %xmm14
LAST:
```

(continue)

Example 5-47. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
    aesenclast %xmm15, %xmm7
    aesenclast %xmm15, %xmm8
    aesenclast %xmm15, %xmm9
    aesenclast %xmm15, %xmm10
    aesenclast %xmm15, %xmm11
    aesenclast %xmm15, %xmm12

    aesenclast %xmm15, %xmm13
    aesenclast %xmm15, %xmm14

    pxor   (%rdi), %xmm7
    pxor   16(%rdi), %xmm8
    pxor   32(%rdi), %xmm9
    pxor   48(%rdi), %xmm10
    pxor   64(%rdi), %xmm11
    pxor   80(%rdi), %xmm12
    pxor   96(%rdi), %xmm13
    pxor   112(%rdi), %xmm14

    dec %r8

    movdqu  %xmm7, (%rsi)
    movdqu  %xmm8, 16(%rsi)
    movdqu  %xmm9, 32(%rsi)
    movdqu  %xmm10, 48(%rsi)
    movdqu  %xmm11, 64(%rsi)
    movdqu  %xmm12, 80(%rsi)
    movdqu  %xmm13, 96(%rsi)
    movdqu  %xmm14, 112(%rsi)
    jne LOOP

    addq  $128,%rsi
    addq  $128,%rdi

REMAINDER:
    cmp $0, %r10
    je END
    shufpd  $2, %xmm1, %xmm0
IN_LOOP:
    movdqa %xmm0, %xmm11
    pshufb (BSWAP_EPI_64), %xmm0
    pxor   (%r9), %xmm11
    paddq  (ONE), %xmm0
    aesenc 16(%r9), %xmm11
    aesenc 32(%r9), %xmm11
    pshufb (BSWAP_EPI_64), %xmm0
```

(continue)

Example 5-47. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
    aesenc 48(%r9), %xmm11
    aesenc 64(%r9), %xmm11
    aesenc 80(%r9), %xmm11
    aesenc 96(%r9), %xmm11
    aesenc 112(%r9), %xmm11
    aesenc 128(%r9), %xmm11
    aesenc 144(%r9), %xmm11
    movdqa 160(%r9), %xmm2
    cmp   $12, %r12d
    jb    IN_LAST
    aesenc 160(%r9), %xmm11
    aesenc 176(%r9), %xmm11
    movdqa 192(%r9), %xmm2
    cmp    $14, %r12d
    jb    IN_LAST
    aesenc 192(%r9), %xmm11
    aesenc 208(%r9), %xmm11
    movdqa 224(%r9), %xmm2
IN_LAST:
    aesenclast %xmm2, %xmm11
    pxor      (%rdi) ,%xmm11
    movdqu    %xmm11, (%rsi)
    addq      $16,%rdi
    addq      $16,%rsi
    dec       %r10
    jne       IN_LOOP
END:
    ret
```

5.10.2 AES Key Expansion Alternative

In Intel microarchitecture code name Sandy Bridge, the throughput of AESKEYGENASSIST is two cycles with higher latency than the AESENC/AESDEC instructions. Software may consider implementing the AES key expansion by using the AESENCLAST instruction with the second operand (i.e., the round key) being the RCON value, duplicated four times in the register. The AESENCLAST instruction performs the SubBytes step and the xor-with-RCON step, while the ROTWORD step can be done using a PSHUFB instruction. Following are code examples of AES128 key expansion using either method.

Example 5-48. AES128 Key Expansion

// Use AESKENYGENASSIST	// Use AESENCLAST
.align 16,0x90	mask:
.globl AES_128_Key_Expansion	.long 0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d
AES_128_Key_Expansion:	con1:
# parameter 1: %rdi	.long 1,1,1,1
# parameter 2: %rsi	con2:
movl $10, 240(%rsi)	.long 0x1b,0x1b,0x1b,0x1b
movdqu (%rdi), %xmm1	.align 16,0x90
movdqa %xmm1, (%rsi)	.globl AES_128_Key_Expansion
(continue)	(continue)

Example 5-48. AES128 Key Expansion (Contd.)

```
    aeskeygenassist $1, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 16(%rsi)
  aeskeygenassist $2, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 32(%rsi)
  aeskeygenassist $4, %xmm1, %xmm2

ASSISTS:
    call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 48(%rsi)
  aeskeygenassist $8, %xmm1, %xmm2
    call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 64(%rsi)
  aeskeygenassist $16, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 80(%rsi)
  aeskeygenassist $32, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 96(%rsi)
  aeskeygenassist $64, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 112(%rsi)
  aeskeygenassist $0x80, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 128(%rsi)
  aeskeygenassist $0x1b, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 144(%rsi)
  aeskeygenassist $0x36, %xmm1, %xmm2
  call PREPARE_ROUNDKEY_128
  movdqa %xmm1, 160(%rsi)
  ret

PREPARE_ROUNDKEY_128:
  pshufd $255, %xmm2, %xmm2
  movdqa %xmm1, %xmm3
  pslldq $4, %xmm3
  pxor %xmm3, %xmm1
  pslldq $4, %xmm3
  pxor %xmm3, %xmm1
  pslldq $4, %xmm3
  pxor %xmm3, %xmm1
  pxor %xmm2, %xmm1
  ret
```

```
AES_128_Key_Expansion:
# parameter 1: %rdi
# parameter 2: %rsi
    movdqu   (%rdi), %xmm1
    movdqa   %xmm1, (%rsi)
    movdqa %xmm1, %xmm2
    movdqa (con1), %xmm0
    movdqa (mask), %xmm15
    mov $8, %ax
LOOP1:
    add $16, %rsi
    dec %ax
    pshufb %xmm15,%xmm2
    aesenclast %xmm0, %xmm2
    pslld $1, %xmm0
    movdqa %xmm1, %xmm3
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pxor %xmm2, %xmm1
   movdqa %xmm1, (%rsi)
    movdqa %xmm1, %xmm2
    jne LOOP1
   movdqa (con2), %xmm0
    pshufb %xmm15,%xmm2
    aesenclast %xmm0, %xmm2
    pslld $1, %xmm0
    movdqa %xmm1, %xmm3
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pslldq $4, %xmm3
   pxor %xmm3, %xmm1
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pxor %xmm2, %xmm1
    movdqa %xmm1, 16(%rsi)
    movdqa %xmm1, %xmm2
    pshufb %xmm15,%xmm2
    aesenclast %xmm0, %xmm2
    movdqa %xmm1, %xmm3
   pslldq $4, %xmm3
    pxor %xmm3, %xmm1
    pslldq $4, %xmm3
    pxor %xmm3, %xmm1
```

(continue)

Example 5-48. AES128 Key Expansion (Contd.)

	pslldq $4, %xmm3 pxor %xmm3, %xmm1 pxor %xmm2, %xmm1 movdqa %xmm1, 32(%rsi) movdqa %xmm1, %xmm2 ret

5.10.3 Enhancement in Intel Microarchitecture Code Name Haswell

5.10.3.1 AES and Multi-Buffer Cryptographic Throughput

The AESINC/AESINCLAST, AESDEC/AESDECLAST instructions in Intel microarchitecture code name Haswell have slightly improvement latency and are one micro-op. These improvements are expected to benefit AES algorithms operating in parallel modes (e.g. CBC decryption) and multiple-buffer implementations of AES algorithms. Several white papers provide more details and examples of using AESNI. See the following links:

- http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set.
- http://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library/.
- http://software.intel.com/file/26898.

5.10.3.2 PCLMULQDQ Improvement

The latency of PCLMULQDQ in Intel microarchitecture code name Haswell is reduced from 14 to 7 cycles, and throughput improved from once every 8 cycle to every other cycle, when compared to prior generations. This will speed up CRC calculations for generic polynomials. Details and examples can be found at:

- http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/fast-crc-computation-paper.html.
- http://download.intel.com/embedded/processor/whitepaper/327889.pdf.
- http://www.intel.com/Assets/PDF/manual/323640.pdf.

AES-GCM implemented using PCLMULQDQ can be found in OpenSSL project at:

- https://www-ssl.intel.com/content/www/us/en/intelligent-systems/wireless-infrastructure/aes-ipsec-performance-linux-paper.html.

5.11 LIGHT-WEIGHT DECOMPRESSION AND DATABASE PROCESSING

Traditionally, database storage requires high-compression ratio means to preserve the finite disk I/O bandwidth limitations. In row-optimized database architecture, the primary limitation on database processing performance often correlates to the hardware constraints of the storage I/O bandwidth, the locality issues of data records from rows in large tables that must be decompressed from its storage format. Many recent database innovations are centered around columnar database architecture, where storage format is optimized for query operations to fetch data in a sequential manner.

Some of the recent advances in columnar database (also known as in-memory database) are light-weight compression/decompression techniques and vectorized query operation primitives using SSE4.2 and other SIMD instructions. When a database engine combines those processing techniques with a column-optimized storage system using solid state drives, query performance increase of several fold has been reported[1]. This section discusses the usage of SIMD instructions for light-weight compression/decompression in columnar databases.

1. See published TPC-H non-clustered performance results at www.tpc.org

The optimal objective for light-weight compression/decompression is to deliver high throughput at reasonably low CPU utilization, such that the finite total compute bandwidth can be divided more favorably between query processing and decompression to achieve maximal query throughput. SSE4.2 can raise the compute bandwidth for some query operations to a significantly higher level (see Section 10.3.3), compared to query primitives implemented using general-purpose-register instructions. This also places higher demand on the streaming data feed of decompressed columnar data.

5.11.1 Reduced Dynamic Range Datasets

One of the more successful approaches to compress/decompress columnar data in high-speed is based on the idea that an ensemble of integral values in a sequential data stream of fixed-size storage width can be represented more compactly if the dynamic range of that ensemble is reduced by way of partitioning, offset from a common reference value, and additional techniques[2,3].

For example, a column that stores 5-digit ZIPCODE as 32-bit integers only requires a dynamic range of 17 bits. The unique primary keys in a 2 billion row table can be reduced through partitioning of sequential blocks of 2^N entries to store the offset in the block header and reducing the storage size of each 32-bit integer as N bits.

5.11.2 Compression and Decompression Using SIMD Instructions

To illustrate the usage of SIMD instructions for reduced-dynamic-range compression/decompression, and compressed data elements are not byte-aligned, we consider an array of 32-bit integers whose dynamic range only requires 5 bits per value.

To pack a stream of 32-bit integer values into consecutive 5-bit buckets, the SIMD technique illustrated in Example 5-49 consists of the following phases:

- Dword-to-byte packing and byte-array sequencing: The stream of dword elements is reduced to byte streams with each iteration handling 32 elements. The two resulting 16-byte vectors are sequenced to enable 4-way bit-stitching using PSLLD and PSRLD instructions.

Example 5-49. Compress 32-bit Integers into 5-bit Buckets

```
;
static __declspec(align(16)) short mask_dw_5b[16] = // 5-bit mask for 4 way bit-packing via dword
{0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0}; // packed shift
static __declspec(align(16)) short sprdb_0_5_10_15[8] = // shuffle control to re-arrange
{ 0xff00, 0xffff, 0x04ff, 0xffff, 0xffff, 0xff08, 0xffff, 0x0cff}; // bytes 0, 4, 8, 12 to gap positions at 0, 5, 10, 15

void RDRpack32x4_sse(int *src, int cnt, char * out)

int i, j;
__m128i a0, a1, a2, a3, c0, c1, b0, b1, b2, b3, bb;
__m128i msk4 ;
__m128i sprd4 = _mm_loadu_si128( (__m128i*) &sprdb_0_5_10_15[0]);
    switch( bucket_width) {
    case 5:j= 0;
                    (continue)
```

2. "SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units", T. Willhalm, et. al., Proceedings of the VLDB Endowment, Vol. 2, #1, August 2009.

3. "Super-Scalar RAM-CPU Cache Compression," M. Zukowski, et, al, Data Engineering, International Conference, vol. 0, no. 0, pp. 59, 2006.

Example 5-49. Compress 32-bit Integers into 5-bit Buckets (Contd.)

```
    msk4 = _mm_loadu_si128( (__m128i*) &mask_dw_5b[0]);
   // process 32 elements in each iteration
    for (i = 0; i < cnt; i+= 32) {
        b0 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i]), _mm_loadu_si128( (__m128i*) &src[i+4]));
        b1 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i+8]), _mm_loadu_si128( (__m128i*) &src[i+12]));
        b2 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i+16]), _mm_loadu_si128( (__m128i*) 
&src[i+20]));
        b3 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i+24]), _mm_loadu_si128( (__m128i*) 
&src[i+28]));
        c0 = _mm_packus_epi16( _mm_unpacklo_epi64(b0, b1), _mm_unpacklo_epi64(b2, b3));
        // c0 contains bytes: 0-3, 8-11, 16-19, 24-27  elements
        c1 = _mm_packus_epi16( _mm_unpackhi_epi64(b0, b1), _mm_unpackhi_epi64(b2, b3));
        // c1 contains bytes: 4-7, 12-15, 20-23, 28-31


        b0 = _mm_and_si128( c0, msk4);              // keep lowest 5 bits in each way/dword
        b1 = _mm_and_si128( _mm_srli_epi32(c0, 3), _mm_slli_epi32(msk4, 5));
        b0 = _mm_or_si128( b0, b1);                         // add next 5 bits to each way/dword
        b1 = _mm_and_si128( _mm_srli_epi32(c0, 6), _mm_slli_epi32(msk4, 10));
        b0 = _mm_or_si128( b0, b1);
        b1 = _mm_and_si128( _mm_srli_epi32(c0, 9), _mm_slli_epi32(msk4, 15));
        b0 = _mm_or_si128( b0, b1);
        b1 = _mm_and_si128( _mm_slli_epi32(c1, 20), _mm_slli_epi32(msk4, 20));
        b0 = _mm_or_si128( b0, b1);
        b1 = _mm_and_si128( _mm_slli_epi32(c1, 17), _mm_slli_epi32(msk4, 25));
        b0 = _mm_or_si128( b0, b1);
        b1 = _mm_and_si128( _mm_slli_epi32(c1, 14), _mm_slli_epi32(msk4, 30));
        b0 = _mm_or_si128( b0, b1);      // add next 2 bits from each dword channel, xmm full
        *(int*)&out[j] = _mm_cvtsi128_si32( b0);// the first dword is compressed and ready
        // re-distribute the remaining 3 dword and add gap bytes to store remained bits
        b0 = _mm_shuffle_epi8(b0, gap4x3);
        b1 = _mm_and_si128( _mm_srli_epi32(c1, 18), _mm_srli_epi32(msk4, 2)); // do 4-way packing of the next 3 bits
        b2 = _mm_and_si128( _mm_srli_epi32(c1, 21), _mm_slli_epi32(msk4, 3));
        b1 = _mm_or_si128( b1, b2); //5th byte compressed at bytes 0, 4, 8, 12
        // shuffle the fifth byte result to byte offsets of 0, 5, 10, 15
        b0 = _mm_or_si128( b0, _mm_shuffle_epi8(b1, sprd4));
        _mm_storeu_si128( (__m128i *) &out[j+4] , b0);
        j += bucket_width*4;
    }
    // handle remainder if cnt is not multiples of 32
    break;
   }
}
```

- Four-way bit stitching: In each way (dword) of the destination, 5 bits are packed consecutively from the corresponding byte element that contains 5 non-zero bit patterns. Since each dword destination will be completely filled up by the contents of 7 consecutive elements, the remaining three bits of the 7th element and the 8th element are done separately in a similar 4-way stitching operation but require the assistance of shuffle operations.

Example 5-50 shows the reverse operation of decompressing consecutively packed 5-bit buckets into 32-bit data elements.

Example 5-50. Decompression of a Stream of 5-bit Integers into 32-bit Elements

```
;
static __declspec(align(16)) short mask_dw_5b[16] = // 5-bit mask for 4 way bit-packing via dword
{0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0}; // packed shift
static __declspec(align(16)) short pack_dw_4x3[8] = // pack 3 dwords 1-4, 6-9, 11-14
{ 0xffff, 0xffff, 0x0201, 0x0403, 0x0706, 0x0908, 0xc0b, 0x0e0d}; // to vacate bytes 0-3
static __declspec(align(16)) short packb_0_5_10_15[8] = // shuffle control to re-arrange bytes
{ 0xffff, 0x0ff, 0xffff, 0x5ff, 0xffff, 0xaff, 0xffff, 0x0fff}; // 0, 5, 10, 15 to gap positions at 3, 7, 11, 15

void RDRunpack32x4_sse(char *src, int cnt, int * out)
{int i, j;
__m128i a0, a1, a2, a3, c0, c1, b0, b1, b2, b3, bb, d0, d1, d2, d3;
__m128i msk4 ;
__m128i pck4 = _mm_loadu_si128( (__m128i*) &packb_0_5_10_15[0]);
__m128i pckdw3 = _mm_loadu_si128( (__m128i*) &pack_dw_4x3[0]);

    switch( bucket_width) {
    case 5:j= 0;
     msk4 = _mm_loadu_si128( (__m128i*) &mask_dw_5b[0]);
     for (i = 0; i < cnt; i+= 32) {
        a1 = _mm_loadu_si128( (__m128i*) &src[j +4]);
       // pick up bytes 4, 9, 14, 19 and shuffle into offset 3, 7, 11, 15
        c0 = _mm_shuffle_epi8(a1, pck4);
        b1 = _mm_and_si128( _mm_srli_epi32(c0, 3), _mm_slli_epi32(msk4, 24));
       // put 3 unaligned dword 1-4, 6-9, 11-14 to vacate bytes 0-3
        a1 = _mm_shuffle_epi8(a1, pckdw3);
        b0 = _mm_and_si128( _mm_srli_epi32(c0, 6), _mm_slli_epi32(msk4, 16));
        a0 = _mm_cvtsi32_si128( *(int *)&src[j ]);
        b1 = _mm_or_si128( b0, b1); // finished decompress source bytes 4, 9, 14, 19
        a0 = _mm_or_si128( a0, a1); // bytes 0-16 contain compressed bits
        b0 = _mm_and_si128( _mm_srli_epi32(a0, 14), _mm_slli_epi32(msk4, 16));
        b1 = _mm_or_si128( b0, b1);
        b0 = _mm_and_si128( _mm_srli_epi32(a0, 17), _mm_slli_epi32(msk4, 8));
        b1 = _mm_or_si128( b0, b1);
        b0 = _mm_and_si128( _mm_srli_epi32(a0, 20), msk4);
        b1 = _mm_or_si128( b0, b1);// b1 now full with decompressed 4-7,12-15,20-23,28-31
        _mm_storeu_si128( (__m128i *) &out[i+4] , _mm_cvtepu8_epi32(b1));
        b0 = _mm_and_si128( _mm_slli_epi32(a0, 9), _mm_slli_epi32(msk4, 24));
        c0 = _mm_and_si128( _mm_slli_epi32(a0, 6), _mm_slli_epi32(msk4, 16));
        b0 = _mm_or_si128( b0, c0);
        _mm_storeu_si128( (__m128i *) &out[i+12] , _mm_cvtepu8_epi32(_mm_srli_si128(b1, 4)));
        c0 = _mm_and_si128( _mm_slli_epi32(a0, 3), _mm_slli_epi32(msk4, 8));
        _mm_storeu_si128( (__m128i *) &out[i+20] , _mm_cvtepu8_epi32(_mm_srli_si128(b1, 8)));
        b0 = _mm_or_si128( b0, c0);
        _mm_storeu_si128( (__m128i *) &out[i+28] , _mm_cvtepu8_epi32(_mm_srli_si128(b1, 12)));
        c0 = _mm_and_si128( a0, msk4);
        b0 = _mm_or_si128( b0, c0);// b0 now full with decompressed 0-3,8-11,16-19,24-27
```

Example 5-50. Decompression of a Stream of 5-bit Integers into 32-bit Elements (Contd.)

```
            _mm_storeu_si128( (__m128i *) &out[i] , _mm_cvtepu8_epi32(b0));
            _mm_storeu_si128( (__m128i *) &out[i+8] , _mm_cvtepu8_epi32(_mm_srli_si128(b0, 4)));
            _mm_storeu_si128( (__m128i *) &out[i+16] , _mm_cvtepu8_epi32(_mm_srli_si128(b0, 8)));
            _mm_storeu_si128( (__m128i *) &out[i+24] , _mm_cvtepu8_epi32(_mm_srli_si128(b0, 12)));
           j += g_bwidth*4;
      }
      break;
     }
}
```

Compression/decompression of integers for dynamic range that are non-power-of-2 can generally use similar mask/packed shift/stitch technique with additional adaptation of the horizontal rearrangement of partially stitched vectors. The increase in throughput relative to using general-purpose scalar instructions will depend on implementation and bucket width.

When compiled with the "/O2" option on an Intel Compiler, the compression throughput can reach 6 Bytes/cycle on Intel microarchitecture code name Sandy Bridge, and the throughput varies little for working set sizes due to the streaming data access pattern and the effectiveness of hardware prefetchers. The decompression throughput of the above example is more than 5 Bytes/cycle at full utilization, allowing a database query engine to partition CPU utilization effectively to allocate a small fraction for on-the-fly decompression to feed vectorized query computation.

The decompression throughput increase using a SIMD light-weight compression technique offers database architects new degrees of freedom to relocate critical performance bottlenecks from a lower-throughput technology (disk I/O, DRAM) to a faster pipeline.

CHAPTER 6
OPTIMIZING FOR SIMD FLOATING-POINT APPLICATIONS

This chapter discusses rules for optimizing for the single-instruction, multiple-data (SIMD) floating-point instructions available in SSE, SSE2 SSE3, and SSE4.1. The chapter also provides examples that illustrate the optimization techniques for single-precision and double-precision SIMD floating-point applications.

6.1 GENERAL RULES FOR SIMD FLOATING-POINT CODE

The rules and suggestions in this section help optimize floating-point code containing SIMD floating-point instructions. Generally, it is important to understand and balance port utilization to create efficient SIMD floating-point code. Basic rules and suggestions include the following:

- Follow all guidelines in Chapter 3 and Chapter 4.
- Mask exceptions to achieve higher performance. When exceptions are unmasked, software performance is slower.
- Utilize the flush-to-zero and denormals-are-zero modes for higher performance to avoid the penalty of dealing with denormals and underflows.
- Use the reciprocal instructions followed by iteration for increased accuracy. These instructions yield reduced accuracy but execute much faster. Note the following:
  - If reduced accuracy is acceptable, use them with no iteration.
  - If near full accuracy is needed, use a Newton-Raphson iteration.
  - If full accuracy is needed, then use divide and square root which provide more accuracy, but slow down performance.

6.2 PLANNING CONSIDERATIONS

Whether adapting an existing application or creating a new one, using SIMD floating-point instructions to achieve optimum performance gain requires programmers to consider several issues. In general, when choosing candidates for optimization, look for code segments that are computationally intensive and floating-point intensive. Also consider efficient use of the cache architecture.

The sections that follow answer the questions that should be raised before implementation:

- Can data layout be arranged to increase parallelism or cache utilization?
- Which part of the code benefits from SIMD floating-point instructions?
- Is the current algorithm the most appropriate for SIMD floating-point instructions?
- Is the code floating-point intensive?
- Do either single-precision floating-point or double-precision floating-point computations provide enough range and precision?
- Does the result of computation affected by enabling flush-to-zero or denormals-to-zero modes?
- Is the data arranged for efficient utilization of the SIMD floating-point registers?
- Is this application targeted for processors without SIMD floating-point instructions?

See also: Section 4.2, “Considerations for Code Conversion to SIMD Programming.”

6.3 USING SIMD FLOATING-POINT WITH X87 FLOATING-POINT

Because the XMM registers used for SIMD floating-point computations are separate registers and are not mapped to the existing x87 floating-point stack, SIMD floating-point code can be mixed with x87 floating-point or 64-bit SIMD integer code.

With Intel Core microarchitecture, 128-bit SIMD integer instructions provides substantially higher efficiency than 64-bit SIMD integer instructions. Software should favor using SIMD floating-point and integer SIMD instructions with XMM registers where possible.

6.4 SCALAR FLOATING-POINT CODE

There are SIMD floating-point instructions that operate only on the lowest order element in the SIMD register. These instructions are known as scalar instructions. They allow the XMM registers to be used for general-purpose floating-point computations.

In terms of performance, scalar floating-point code can be equivalent to or exceed x87 floating-point code and has the following advantages:

- SIMD floating-point code uses a flat register model, whereas x87 floating-point code uses a stack model. Using scalar floating-point code eliminates the need to use FXCH instructions. These have performance limits on the Intel Pentium 4 processor.
- Mixing with MMX technology code without penalty.
- Flush-to-zero mode.
- Shorter latencies than x87 floating-point.

When using scalar floating-point instructions, it is not necessary to ensure that the data appears in vector form. However, the optimizations regarding alignment, scheduling, instruction selection, and other optimizations covered in Chapter 3 and Chapter 4 should be observed.

6.5 DATA ALIGNMENT

SIMD floating-point data is 16-byte aligned. Referencing unaligned 128-bit SIMD floating-point data will result in an exception unless MOVUPS or MOVUPD (move unaligned packed single or unaligned packed double) is used. The unaligned instructions used on aligned or unaligned data will also suffer a performance penalty relative to aligned accesses.

See also: Section 4.4, “Stack and Data Alignment.”

6.5.1 Data Arrangement

Because SSE and SSE2 incorporate SIMD architecture, arranging data to fully use the SIMD registers produces optimum performance. This implies contiguous data for processing, which leads to fewer cache misses. Correct data arrangement can potentially quadruple data throughput when using SSE or double throughput when using SSE2. Performance gains can occur because four data elements can be loaded with 128-bit load instructions into XMM registers using SSE (MOVAPS). Similarly, two data elements can loaded with 128-bit load instructions into XMM registers using SSE2 (MOVAPD).

Refer to the Section 4.4, “Stack and Data Alignment,” for data arrangement recommendations. Duplicating and padding techniques overcome misalignment problems that occur in some data structures and arrangements. This increases the data space but avoids penalties for misaligned data access.

For some applications (for example: 3D geometry), traditional data arrangement requires some changes to fully utilize the SIMD registers and parallel techniques. Traditionally, the data layout has been an array of structures (AoS). To fully utilize the SIMD registers in such applications, a new data layout has been proposed — a structure of arrays (SoA) resulting in more optimized performance.

6.5.1.1 Vertical versus Horizontal Computation

The majority of the floating-point arithmetic instructions in SSE/SSE2 provide greater performance gain on vertical data processing for parallel data elements. This means each element of the destination is the result of an arithmetic operation performed from the source elements in the same vertical position (Figure 6-1).

To supplement these homogeneous arithmetic operations on parallel data elements, SSE and SSE2 provides data movement instructions (e.g., SHUFPS, UNPCKLPS, UNPCKHPS, MOVLHPS, MOVHLPS, etc.) that facilitate moving data elements
horizontally.

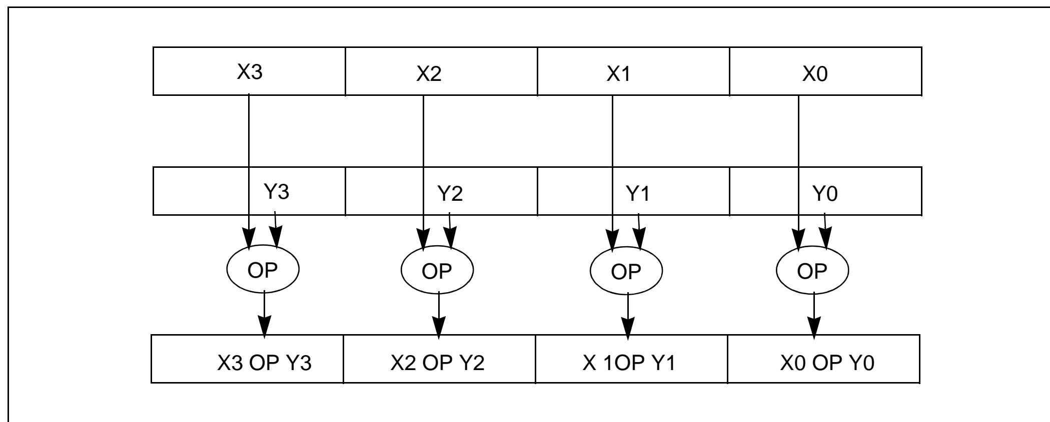


Figure 6-1. Homogeneous Operation on Parallel Data Elements

The organization of structured data have a significant impact on SIMD programming efficiency and performance. This can be illustrated using two common type of data structure organizations:

- Array of Structure: This refers to the arrangement of an array of data structures. Within the data structure, each member is a scalar. This is shown in Figure 6-2. Typically, a repetitive sequence of computation is applied to each element of an array, i.e. a data structure. Computational sequence for the scalar members of the structure is likely to be non-homogeneous within each iteration. AoS is generally associated with a horizontal computation model.

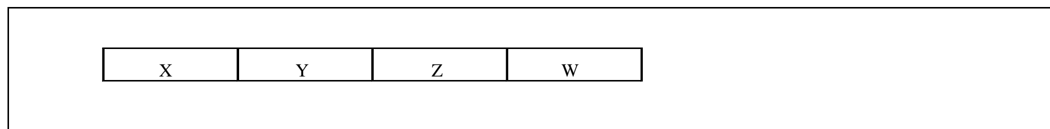


Figure 6-2. Horizontal Computation Model

- Structure of Array: Here, each member of the data structure is an array. Each element of the array is a scalar. This is shown Table 6-1. Repetitive computational sequence is applied to scalar elements and homogeneous operation can be easily achieved across consecutive iterations within the same structural member. Consequently, SoA is generally amenable to the vertical computation model.

Table 6-1. SoA Form of Representing Vertices Data

Vx array	X1	X2	X3	X4	.....	Xn
Vy array	Y1	Y2	Y3	Y4	.....	Yn
Vz array	Z1	Z2	Z3	Y4	.....	Zn
Vw array	W1	W2	W3	W4	.....	Wn

Using SIMD instructions with vertical computation on SOA arrangement can achieve higher efficiency and performance than AOS and horizontal computation. This can be seen with dot-product operation on vectors. The dot product operation on SoA arrangement is shown in Figure 6-3.

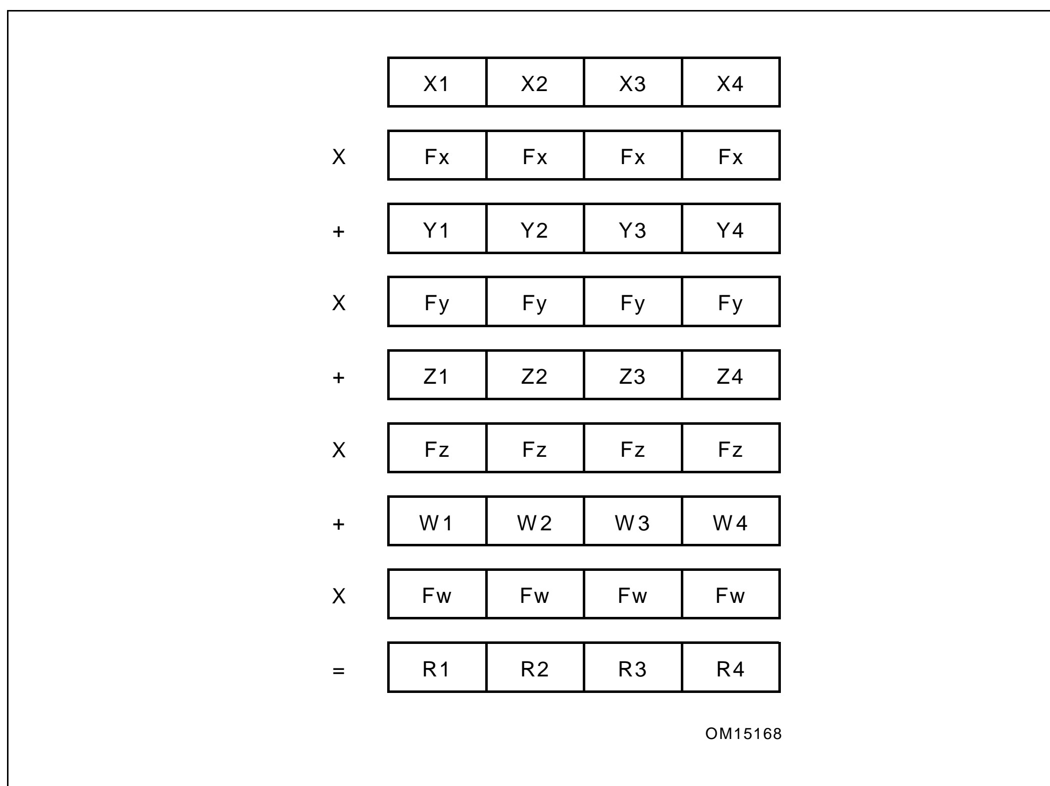


Figure 6-3. Dot Product Operation

Example 6-1 shows how one result would be computed for seven instructions if the data were organized as AoS and using SSE alone: four results would require 28 instructions.

Example 6-1. Pseudocode for Horizontal (xyz, AoS) Computation

```
mulps    ; x*x', y*y', z*z'
movaps   ; reg->reg move, since next steps overwrite
shufps   ; get b,a,d,c from a,b,c,d
addps    ; get a+b,a+b,c+d,c+d
movaps   ; reg->reg move
shufps   ; get c+d,c+d,a+b,a+b from prior addps
addps    ; get a+b+c+d,a+b+c+d,a+b+c+d,a+b+c+d
```

Now consider the case when the data is organized as SoA. Example 6-2 demonstrates how four results are computed for five instructions.

Example 6-2. Pseudocode for Vertical (xxxx, yyyy, zzzz, SoA) Computation

```
mulps   ; x*x' for all 4 x-components of 4 vertices
mulps   ; y*y' for all 4 y-components of 4 vertices
mulps   ; z*z' for all 4 z-components of 4 vertices
addps   ; x*x' + y*y'
addps   ; x*x'+y*y'+z*z'
```

For the most efficient use of the four component-wide registers, reorganizing the data into the SoA format yields increased throughput and hence much better performance for the instructions used.

As seen from this simple example, vertical computation can yield 100% use of the available SIMD registers to produce four results. (The results may vary for other situations.) If the data structures are represented in a format that is not "friendly" to vertical computation, it can be rearranged "on the fly" to facilitate better utilization of the SIMD registers. This operation is referred to as "swizzling" operation and the reverse operation is referred to as "deswizzling."

6.5.1.2 Data Swizzling

Swizzling data from SoA to AoS format can apply to a number of application domains, including 3D geometry, video and imaging. Two different swizzling techniques can be adapted to handle floating-point and integer data. Example 6-3 illustrates a swizzle function that uses SHUFPS, MOVLHPS, MOVHLPS instructions.

Example 6-3. Swizzling Data Using SHUFPS, MOVLHPS, MOVHLPS

```
typedef struct _VERTEX_AOS {
    float x, y, z, color;
} Vertex_aos;                           // AoS structure declaration
typedef struct _VERTEX_SOA {
    float x[4], float y[4], float z[4];
    float color[4];
} Vertex_soa;                           // SoA structure declaration
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
// in mem: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
// SWIZZLE XYZW --> XXXX
asm {
        mov  ebx, in                    // get structure addresses
        mov  edx, out
        movaps  xmm1, [ebx ]        // x4 x3 x2 x1
        movaps  xmm2, [ebx + 16] // y4 y3 y2 y1
        movaps  xmm3, [ebx + 32] // z4 z3 z2 z1
        movaps  xmm4, [ebx + 48] // w4 w3 w2 w1
        movaps  xmm7, xmm4 // xmm7= w4 z4 y4 x4
        movhlps  xmm7, xmm3 // xmm7= w4 z4 w3 z3
        movaps  xmm6, xmm2 // xmm6= w2 z2 y2 x2
        movlhps  xmm3, xmm4 // xmm3= y4 x4 y3 x3
        movhlps  xmm2, xmm1 // xmm2= w2 z2 w1 z1
        movlhps  xmm1, xmm6 // xmm1= y2 x2 y1 x1
```

Example 6-3. Swizzling Data (Contd.)Using SHUFPS, MOVLHPS, MOVHLPS (Contd.)

```
        movaps   xmm6, xmm2// xmm6= w2 z2 w1 z1
        movaps   xmm5, xmm1// xmm5= y2 x2 y1 x1
        shufps   xmm2, xmm7, 0xDD // xmm2= w4 w3 w2 w1 => v4
        shufps   xmm1, xmm3, 0x88 // xmm1= x4 x3 x2 x1 => v1
        shufps   xmm5, xmm3, 0xDD // xmm5= y4 y3 y2 y1 => v2
        shufps   xmm6, xmm7, 0x88 // xmm6= z4 z3 z2 z1 => v3

        movaps   [edx], xmm1                    // store X
        movaps   [edx+16], xmm5                 // store Y
        movaps   [edx+32], xmm6                 // store Z
        movaps   [edx+48], xmm2                 // store W
    }
}
```

Example 6-4 shows a similar data-swizzling algorithm using SIMD instructions in the integer domain.

Example 6-4. Swizzling Data Using UNPCKxxx Instructions

```
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
// in mem: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
// SWIZZLE XYZW --> XXXX
asm {
        mov  ebx, in                    // get structure addresses
        mov  edx, out

        movdqa       xmm1, [ebx + 0*16]     //w0 z0 y0 x0
        movdqa       xmm2, [ebx + 1*16]     //w1 z1 y1 x1
        movdqa        xmm3, [ebx + 2*16]     //w2 z2 y2 x2
        movdqa       xmm4, [ebx + 3*16]     //w3 z3 y3 x3
        movdqa       xmm5, xmm1
        punpckldq    xmm1, xmm2          // y1 y0 x1 x0
        punpckhdq    xmm5, xmm2          // w1 w0 z1 z0
        movdqa       xmm2, xmm3
        punpckldq    xmm3, xmm4          // y3 y2 x3 x2
        punpckldq    xmm2, xmm4          // w3 w2 z3 z2
        movdqa       xmm4, xmm1
        punpcklqdq   xmm1, xmm3         // x3 x2 x1 x0
        punpckhqdq   xmm4, xmm3         // y3 y2 y1 y0
        movdqa       xmm3, xmm5
        punpcklqdq   xmm5, xmm2         // z3 z2 z1 z0
        punpckhqdq   xmm3, xmm2         // w3 w2 w1 w0

        movdqa       [edx+0*16], xmm1   //x3 x2 x1 x0
        movdqa       [edx+1*16], xmm4   //y3 y2 y1 y0
        movdqa       [edx+2*16], xmm5   //z3 z2 z1 z0
        movdqa       [edx+3*16], xmm3   //w3 w2 w1 w0
}
```

The technique in Example 6-3 (loading 16 bytes, using SHUFPS and copying halves of XMM registers) is preferable over an alternate approach of loading halves of each vector using MOVLPS/MOVHPS on newer microarchitectures. This is because loading 8 bytes using MOVLPS/MOVHPS can create code dependency and reduce the throughput of the execution engine.

The performance considerations of Example 6-3 and Example 6-4 often depends on the characteristics of each microarchitecture. For example, in Intel Core microarchitecture, executing a SHUFPS tend to be slower than a PUNPCKxxx instruction. In Enhanced Intel Core microarchitecture, SHUFPS and PUNPCKxxx instruction all executes with 1 cycle throughput due to the 128-bit shuffle execution unit. Then the next important consideration is that there is only one port that can execute PUNPCKxxx vs. MOVLHPS/MOVHLPS can execute on multiple ports. The performance of both techniques improves on Intel Core microarchitecture over previous microarchitectures due to 3 ports for executing SIMD instructions. Both techniques improves further on Enhanced Intel Core microarchitecture due to the 128-bit shuffle unit.

6.5.1.3 Data Deswizzling

In the deswizzle operation, we want to arrange the SoA format back into AoS format so the XXXX, YYYY, ZZZZ are rearranged and stored in memory as XYZ. Example 6-5 illustrates one deswizzle function for floating-point data.

Example 6-5. Deswizzling Single-Precision SIMD Data

```
void deswizzle_asm(Vertex_soa *in, Vertex_aos *out)
{
  __asm {
    mov         ecx, in                          // load structure addresses
    mov         edx, out
    movaps      xmm0, [ecx ]      //x3 x2 x1 x0
    movaps      xmm1, [ecx + 16]      //y3 y2 y1 y0
    movaps      xmm2, [ecx + 32]      //z3 z2 z1 z0
    movaps      xmm3, [ecx + 48]      //w3 w2 w1 w0
    movaps      xmm5, xmm0
    movaps      xmm7, xmm2
    unpcklps    xmm0, xmm1       // y1 x1 y0 x0
    unpcklps    xmm2, xmm3       // w1 z1 w0 z0
    movdqa      xmm4, xmm0
    movlhps     xmm0, xmm2       // w0 z0 y0 x0
    movhlps     xmm4, xmm2       // w1 z1 y1 x1
    unpckhps    xmm5, xmm1       // y3 x3 y2 x2
    unpckhps    xmm7, xmm3       // w3 z3 w2 z2
    movdqa      xmm6, xmm5
    movlhps     xmm5, xmm7       // w2 z2 y2 x2
    movhlps     xmm6, xmm7       // w3 z3 y3 x3
    movaps      [edx+0*16], xmm0   //w0 z0 y0 x0
    movaps      [edx+1*16], xmm4   //w1 z1 y1 x1
    movaps      [edx+2*16], xmm5   //w2 z2 y2 x2
    movaps      [edx+3*16], xmm6   //w3 z3 y3 x3
  }
}
```

Example 6-6 shows a similar deswizzle function using SIMD integer instructions. Both of these techniques demonstrate loading 16 bytes and performing horizontal data movement in registers. This approach is likely to be more efficient than alternative techniques of storing 8-byte halves of XMM registers using MOVLPS and MOVHPS.

Example 6-6. Deswizzling Data Using SIMD Integer Instructions

```
void deswizzle_rgb(Vertex_soa *in, Vertex_aos *out)
{
//---deswizzle rgb---
// assume: xmm1=rrrr, xmm2=gggg, xmm3=bbbb, xmm4=aaaa
__asm {
    mov         ecx, in                     // load structure addresses
    mov         edx, out
    movdqa      xmm0, [ecx]                 // load r4 r3 r2 r1 => xmm1
    movdqa      xmm1, [ecx+16]              // load g4 g3 g2 g1 => xmm2
    movdqa      xmm2, [ecx+32]              // load b4 b3 b2 b1 => xmm3
    movdqa      xmm3, [ecx+48]              // load a4 a3 a2 a1 => xmm4
// Start deswizzling here
    movdqa      xmm5, xmm0
    movdqa      xmm7, xmm2
    punpckldq   xmm0, xmm1      // g2 r2 g1 r1
    punpckldq   xmm2, xmm3      // a2 b2 a1 b1
    movdqa      xmm4, xmm0
    punpcklqdq  xmm0, xmm2      // a1 b1 g1 r1 => v1
    punpckhqdq  xmm4, xmm2      // a2 b2 g2 r2 => v2
    punpckhdq   xmm5, xmm1      // g4 r4 g3 r3
    punpckhdq   xmm7, xmm3      // a4 b4 a3 b3
    movdqa      xmm6, xmm5
    punpcklqdq  xmm5, xmm7      // a3 b3 g3 r3 => v3
    punpckhqdq  xmm6, xmm7      // a4 b4 g4 r4 => v4
    movdqa      [edx], xmm0                 // v1
    movdqa      [edx+16], xmm4              // v2
    movdqa      [edx+32], xmm5              // v3
    movdqa      [edx+48], xmm6              // v4
// DESWIZZLING ENDS HERE
    }
}
```

6.5.1.4 Horizontal ADD Using SSE

Although vertical computations generally make use of SIMD performance better than horizontal computations, in some cases, code must use a horizontal operation.

MOVLHPS/MOVHLPS and shuffle can be used to sum data horizontally. For example, starting with four 128-bit registers, to sum up each register horizontally while having the final results in one register, use the MOVLHPS/MOVHLPS to align the upper and lower parts of each register. This allows you to use a vertical add. With the resulting partial horizontal summation, full summation follows easily.

Figure 6-4 presents a horizontal add using MOVHLPS/MOVLHPS. Example 6-7 and Example 6-8 provide the code for this operation.

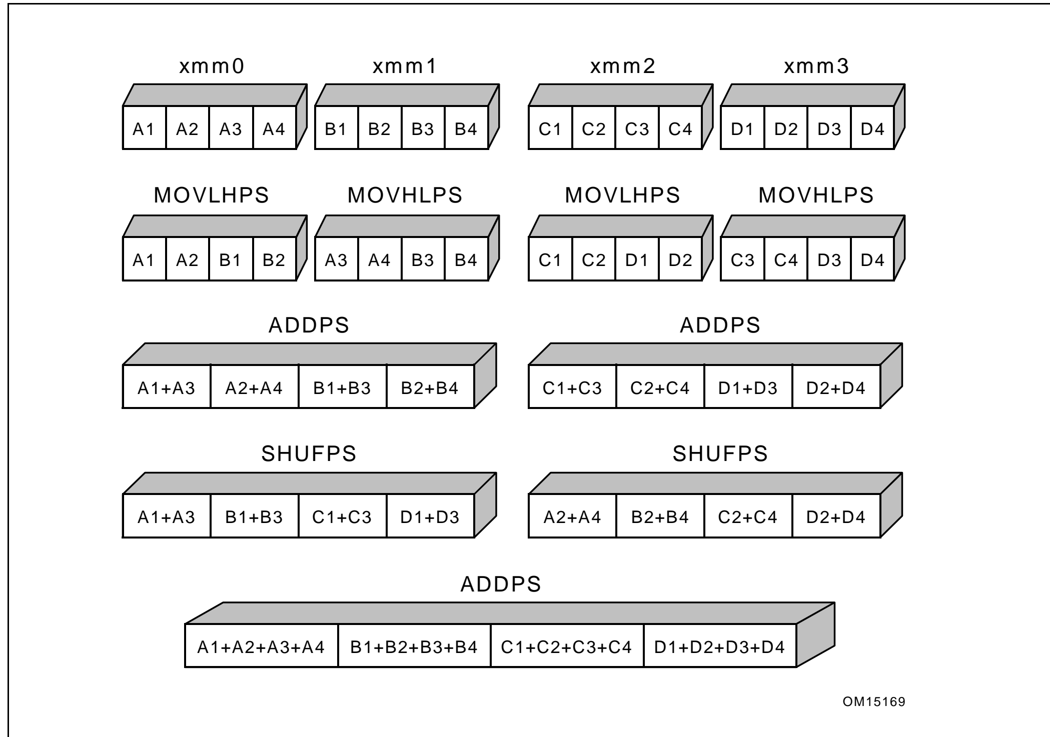


Figure 6-4. Horizontal Add Using MOVHLPS/MOVLHPS

Example 6-7. Horizontal Add Using MOVHLPS/MOVLHPS

```
void horiz_add(Vertex_soa *in, float *out) {
 __asm {
    mov      ecx, in                    // load structure addresses
    mov      edx, out
    movaps   xmm0, [ecx]                // load A1 A2 A3 A4 => xmm0
    movaps   xmm1, [ecx+16]             // load B1 B2 B3 B4 => xmm1
    movaps   xmm2, [ecx+32]             // load C1 C2 C3 C4 => xmm2
    movaps   xmm3, [ecx+48]             // load D1 D2 D3 D4 => xmm3
 // START HORIZONTAL ADD
    movaps  xmm5, xmm0                  // xmm5= A1,A2,A3,A4
    movlhps xmm5, xmm1                  // xmm5= A1,A2,B1,B2
    movhlps xmm1, xmm0                  // xmm1= A3,A4,B3,B4
    addps   xmm5, xmm1                  // xmm5= A1+A3,A2+A4,B1+B3,B2+B4
    movaps  xmm4, xmm2
    movlhps xmm2, xmm3                  // xmm2= C1,C2,D1,D2
    movhlps xmm3, xmm4                  // xmm3= C3,C4,D3,D4
    addps   xmm3, xmm2                  // xmm3= C1+C3,C2+C4,D1+D3,D2+D4
    movaps  xmm6, xmm3                  // xmm6= C1+C3,C2+C4,D1+D3,D2+D4
    shufps  xmm3, xmm5, 0xDD
                                        //xmm6=A1+A3,B1+B3,C1+C3,D1+D3
    shufps xmm5, xmm6, 0x88
                                        // xmm5= A2+A4,B2+B4,C2+C4,D2+D4
    addps  xmm6, xmm5                   // xmm6= D,C,B,A
```

Example 6-7. Horizontal Add Using MOVHLPS/MOVLHPS (Contd.)

```
// END HORIZONTAL ADD
   movaps [edx], xmm6
  }
}
```

Example 6-8. Horizontal Add Using Intrinsics with MOVHLPS/MOVLHPS

```
void horiz_add_intrin(Vertex_soa *in, float *out)
{
  __m128 v, v2, v3, v4;
  __m128 tmm0,tmm1,tmm2,tmm3,tmm4,tmm5,tmm6;
                                          // Temporary variables
      tmm0 = _mm_load_ps(in->x);          // tmm0 = A1 A2 A3 A4
     tmm1 = _mm_load_ps(in->y);           // tmm1 = B1 B2 B3 B4
     tmm2 = _mm_load_ps(in->z);           // tmm2 = C1 C2 C3 C4
     tmm3 = _mm_load_ps(in->w);           // tmm3 = D1 D2 D3 D4
     tmm5 = tmm0;                         // tmm0 = A1 A2 A3 A4
     tmm5 = _mm_movelh_ps(tmm5, tmm1);    // tmm5 = A1 A2 B1 B2
     tmm1 = _mm_movehl_ps(tmm1, tmm0);    // tmm1 = A3 A4 B3 B4
     tmm5 = _mm_add_ps(tmm5, tmm1);       // tmm5 = A1+A3 A2+A4 B1+B3 B2+B4
     tmm4 = tmm2;
     tmm2 = _mm_movelh_ps(tmm2, tmm3);    // tmm2 = C1 C2 D1 D2
     tmm3 = _mm_movehl_ps(tmm3, tmm4);    // tmm3 = C3 C4 D3 D4
     tmm3 = _mm_add_ps(tmm3, tmm2);       // tmm3 = C1+C3 C2+C4 D1+D3 D2+D4
     tmm6 = tmm3;                         // tmm6 = C1+C3 C2+C4 D1+D3 D2+D4
     tmm6 = _mm_shuffle_ps(tmm3, tmm5, 0xDD);
                                          // tmm6 = A1+A3 B1+B3 C1+C3 D1+D3
     tmm5 = _mm_shuffle_ps(tmm5, tmm6, 0x88);
                                          // tmm5 = A2+A4 B2+B4 C2+C4 D2+D4
     tmm6 = _mm_add_ps(tmm6, tmm5);
                                          // tmm6 = A1+A2+A3+A4 B1+B2+B3+B4
                                          // C1+C2+C3+C4 D1+D2+D3+D4
  _mm_store_ps(out, tmm6);
}
```

6.5.2 Use of CVTTPS2PI/CVTTSS2SI Instructions

The CVTTPS2PI and CVTTSS2SI instructions encode the truncate/chop rounding mode implicitly in the instruction. They take precedence over the rounding mode specified in the MXCSR register. This behavior can eliminate the need to change the rounding mode from round-nearest, to truncate/chop, and then back to round-nearest to resume computation.

Avoid frequent changes to the MXCSR register since there is a penalty associated with writing this register. Typically, when using CVTTPS2P/CVTTSS2SI, rounding control in MXCSR can always be set to round-nearest.

6.5.3 Flush-to-Zero and Denormals-are-Zero Modes

The flush-to-zero (FTZ) and denormals-are-zero (DAZ) modes are not compatible with the IEEE Standard 754. They are provided to improve performance for applications where underflow is common and where the generation of a denormalized result is not necessary.

See also: Section 3.8.3, "Floating-point Modes and Exceptions."

6.6 SIMD OPTIMIZATIONS AND MICROARCHITECTURES

Pentium M, Intel Core Solo and Intel Core Duo processors have a different microarchitecture than Intel NetBurst microarchitecture. Intel Core microarchitecture offers significantly more efficient SIMD floating-point capability than previous microarchitectures. In addition, instruction latency and throughput of SSE3 instructions are significantly improved in Intel Core microarchitecture over previous microarchitectures.

6.6.1 SIMD Floating-point Programming Using SSE3

SSE3 enhances SSE and SSE2 with nine instructions targeted for SIMD floating-point programming. In contrast to many SSE/SSE2 instructions offering homogeneous arithmetic operations on parallel data elements and favoring the vertical computation model, SSE3 offers instructions that performs asymmetric arithmetic operation and arithmetic operation on horizontal data elements.

ADDSUBPS and ADDSUBPD are two instructions with asymmetric arithmetic processing capability (see Figure 6-5). HADDPS, HADDPD, HSUBPS and HSUBPD offers horizontal arithmetic processing capability (see Figure 6-6). In addition: MOVSLDUP, MOVSHDUP and MOVDDUP load data from memory (or XMM register) and replicate data elements at once.

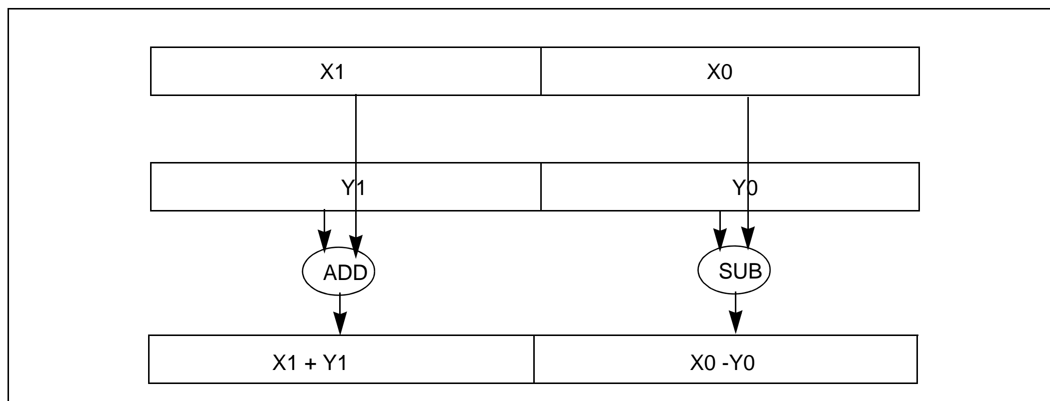


Figure 6-5. Asymmetric Arithmetic Operation of the SSE3 Instruction

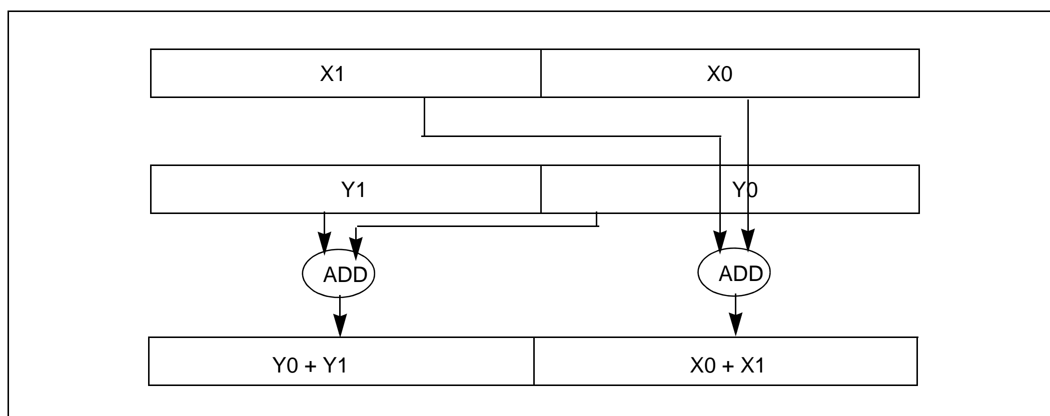


Figure 6-6. Horizontal Arithmetic Operation of the SSE3 Instruction HADDPD

6.6.1.1 SSE3 and Complex Arithmetics

The flexibility of SSE3 in dealing with AOS-type of data structure can be demonstrated by the example of multiplication and division of complex numbers. For example, a complex number can be stored in a structure consisting of its real and imaginary part. This naturally leads to the use of an array of structure. Example 6-9 demonstrates using SSE3 instructions to perform multiplications of single-precision complex numbers. Example 6-10 demonstrates using SSE3 instructions to perform division of complex numbers.

Example 6-9. Multiplication of Two Pair of Single-precision Complex Number

```
// Multiplication of  (ak + i bk ) * (ck + i dk )
// a + i b can be stored as a data structure
movsldup xmm0, Src1; load real parts into the destination,
                        ; a1, a1, a0, a0

movaps  xmm1, src2; load the 2nd pair of complex  values,
                        ; i.e. d1, c1, d0, c0
mulps   xmm0, xmm1; temporary results, a1d1, a1c1, a0d0,
                        ; a0c0

shufps  xmm1, xmm1, b1; reorder the real and imaginary
                        ; parts, c1, d1, c0, d0
movshdup xmm2, Src1; load the imaginary parts into the
                        ; destination, b1, b1, b0, b0

mulps   xmm2, xmm1; temporary results, b1c1, b1d1, b0c0,
                        ; b0d0
addsubps xmm0, xmm2; b1c1+a1d1, a1c1 -b1d1, b0c0+a0d0,
                        ; a0c0-b0d0
```

Example 6-10. Division of Two Pair of Single-precision Complex Numbers

```
// Division of (ak + i bk ) / (ck + i dk )
movshdup xmm0, Src1; load imaginary parts into the
                        ; destination, b1, b1, b0, b0
movaps   xmm1, src2; load the 2nd pair of complex values,
                        ; i.e. d1, c1, d0, c0
mulps    xmm0, xmm1; temporary results, b1d1, b1c1, b0d0,
                        ; b0c0

shufps   xmm1, xmm1, b1; reorder the real and imaginary
                        ; parts, c1, d1, c0, d0
movsldup xmm2, Src1; load the real parts into the
                        ; destination, a1, a1, a0, a0

mulps    xmm2, xmm1; temp results, a1c1, a1d1, a0c0, a0d0
addsubps xmm0, xmm2; a1c1+b1d1, b1c1-a1d1, a0c0+b0d0,
                        ; b0c0-a0d0

mulps    xmm1, xmm1 ; c1c1, d1d1, c0c0, d0d0
movps    xmm2, xmm1; c1c1, d1d1, c0c0, d0d0
shufps   xmm2, xmm2, b1; d1d1, c1c1, d0d0, c0c0
addps    xmm2, xmm1; c1c1+d1d1, c1c1+d1d1, c0c0+d0d0,
                        ; c0c0+d0d0
```

Example 6-10. Division of Two Pair of Single-precision Complex Numbers (Contd.)

```
divps    xmm0, xmm2
shufps   xmm0, xmm0, b1  ; (b1c1-a1d1)/(c1c1+d1d1),
                         ; (a1c1+b1d1)/(c1c1+d1d1),
                         ; (b0c0-a0d0)/( c0c0+d0d0),
                         ; (a0c0+b0d0)/( c0c0+d0d0)
```

In both examples, the complex numbers are store in arrays of structures. MOVSLDUP, MOVSHDUP and the asymmetric ADDSUBPS allow performing complex arithmetics on two pair of single-precision complex number simultaneously and without any unnecessary swizzling between data elements.

Due to microarchitectural differences, software should implement multiplication of complex double-precision numbers using SSE3 instructions on processors based on Intel Core microarchitecture. In Intel Core Duo and Intel Core Solo processors, software should use scalar SSE2 instructions to implement double-precision complex multiplication. This is because the data path between SIMD execution units is 128 bits in Intel Core microarchitecture, and only 64 bits in previous microarchitectures. Processors based on the Enhanced Intel Core microarchitecture generally executes SSE3 instruction more efficiently than previous microarchitectures, they also have a 128-bit shuffle unit that will benefit complex arithmetic operations further than Intel Core microarchitecture did.

Example 6-11 shows two equivalent implementations of double-precision complex multiply of two pair of complex numbers using vector SSE2 versus SSE3 instructions. Example 6-12 shows the equivalent scalar SSE2 implementation.

Example 6-11. Double-Precision Complex Multiplication of Two Pairs

SSE2 Vector Implementation	SSE3 Vector Implementation

movapd xmm0, [eax] ;y x
movapd xmm1, [eax+16] ;w z
unpcklpd xmm1, xmm1 ;z z
movapd xmm2, [eax+16] ;w z
unpckhpd xmm2, xmm2 ;w w
mulpd xmm1, xmm0 ;z*y z*x
mulpd xmm2, xmm0 ;w*y w*x
xorpd xmm2, xmm7 ;-w*y +w*x
shufpd xmm2, xmm2,1 ;w*x -w*y
addpd xmm2, xmm1 ;z*y+w*x z*x-w*y
movapd [ecx], xmm2
	movapd xmm0, [eax] ;y x
	movapd xmm1, [eax+16] ;z z
	movapd xmm2, xmm1
	unpcklpd xmm1, xmm1
	unpckhpd xmm2, xmm2
	mulpd xmm1, xmm0 ;z*y z*x
	mulpd xmm2, xmm0 ;w*y w*x
	shufpd xmm2, xmm2, 1 ;w*x w*y
	addsubpd xmm1, xmm2 ;w*x+z*y z*x-w*y
	movapd [ecx], xmm1

Example 6-12. Double-Precision Complex Multiplication Using Scalar SSE2

```
movsd   xmm0, [eax]        ;x
movsd   xmm5, [eax+8]      ;y
movsd   xmm1, [eax+16]     ;z
movsd   xmm2, [eax+24]     ;w

movsd   xmm3, xmm1     ;z
movsd   xmm4, xmm2     ;w
mulsd   xmm1, xmm0     ;z*x
mulsd   xmm2, xmm0     ;w*x
mulsd   xmm3, xmm5     ;z*y
```

Example 6-12. Double-Precision Complex Multiplication Using Scalar SSE2 (Contd.)

```
mulsd   xmm4, xmm5    ;w*y
subsd   xmm1, xmm4    ;z*x - w*y
addsd   xmm3, xmm2    ;z*y + w*x
movsd   [ecx], xmm1
movsd   [ecx+8], xmm3
```

6.6.1.2 Packed Floating-Point Performance in Intel Core Duo Processor

Most packed SIMD floating-point code will speed up on Intel Core Solo processors relative to Pentium M processors. This is due to improvement in decoding packed SIMD instructions.

The improvement of packed floating-point performance on the Intel Core Solo processor over Pentium M processor depends on several factors. Generally, code that is decoder-bound and/or has a mixture of integer and packed floating-point instructions can expect significant gain. Code that is limited by execution latency and has a "cycles per instructions" ratio greater than one will not benefit from decoder improvement.

When targeting complex arithmetics on Intel Core Solo and Intel Core Duo processors, using single-precision SSE3 instructions can deliver higher performance than alternatives. On the other hand, tasks requiring double-precision complex arithmetics may perform better using scalar SSE2 instructions on Intel Core Solo and Intel Core Duo processors. This is because scalar SSE2 instructions can be dispatched through two ports and executed using two separate floating-point units.

Packed horizontal SSE3 instructions (HADDPS and HSUBPS) can simplify the code sequence for some tasks. However, these instruction consist of more than five micro-ops on Intel Core Solo and Intel Core Duo processors. Care must be taken to ensure the latency and decoding penalty of the horizontal instruction does not offset any algorithmic benefits.

6.6.2 Dot Product and Horizontal SIMD Instructions

Sometimes the AOS type of data organization are more natural in many algebraic formula, one common example is the dot product operation. Dot product operation can be implemented using SSE/SSE2 instruction sets. SSE3 added a few horizontal add/subtract instructions for applications that rely on the horizontal computation model. SSE4.1 provides additional enhancement with instructions that are capable of directly evaluating dot product operations of vectors of 2, 3 or 4 components.

Example 6-13. Dot Product of Vector Length 4 Using SSE/SSE2

Using SSE/SSE2 to compute one dot product

```
movaps  xmm0, [eax] // a4, a3, a2, a1
mulps  xmm0, [eax+16] // a4*b4, a3*b3, a2*b2, a1*b1
movhlps xmm1, xmm0 // X, X,  a4*b4, a3*b3, upper half not needed
addps  xmm0, xmm1 // X, X, a2*b2+a4*b4, a1*b1+a3*b3,
pshufd  xmm1, xmm0, 1 // X, X, X, a2*b2+a4*b4
addss   xmm0, xmm1 // a1*b1+a3*b3+a2*b2+a4*b4
movss   [ecx], xmm0
```

Example 6-14. Dot Product of Vector Length 4 Using SSE3

Using SSE3 to compute one dot product
movaps xmm0, [eax] mulps xmm0, [eax+16] // a4*b4, a3*b3, a2*b2, a1*b1 haddps xmm0, xmm0 // a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1 movaps xmm1, xmm0 // a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1 psrlq xmm0, 32 // 0, a4*b4+a3*b3, 0, a4*b4+a3*b3 addss xmm0, xmm1 // -, -, -, a1*b1+a3*b3+a2*b2+a4*b4 movss [eax], xmm0

Example 6-15. Dot Product of Vector Length 4 Using SSE4.1

Using SSE4.1 to compute one dot product
movaps xmm0, [eax] dpps xmm0, [eax+16], 0xf1 // 0, 0, 0, a1*b1+a3*b3+a2*b2+a4*b4 movss [eax], xmm0

Example 6-13, Example 6-14, and Example 6-15 compare the basic code sequence to compute one dot-product result for a pair of vectors.

The selection of an optimal sequence in conjunction with an application's memory access patterns may favor different approaches. For example, if each dot product result is immediately consumed by additional computational sequences, it may be more optimal to compare the relative speed of these different approaches. If dot products can be computed for an array of vectors and kept in the cache for subsequent computations, then more optimal choice may depend on the relative throughput of the sequence of instructions.

In Intel Core microarchitecture, Example 6-14 has higher throughput than Example 6-13. Due to the relatively longer latency of HADDPS, the speed of Example 6-14 is slightly slower than Example 6-13.

In Enhanced Intel Core microarchitecture, Example 6-15 is faster in both speed and throughput than Example 6-13 and Example 6-14. Although the latency of DPPS is also relatively long, it is compensated by the reduction of number of instructions in Example 6-15 to do the same amount of work.

Unrolling can further improve the throughput of each of three dot product implementations. Example 6-16 shows two unrolled versions using the basic SSE2 and SSE3 sequences. The SSE4.1 version can also be unrolled and using INSERTPS to pack 4 dot-product results.

Example 6-16. Unrolled Implementation of Four Dot Products

SSE2 Implementation	SSE3 Implementation

movaps xmm0, [eax]
mulps xmm0, [eax+16]
;w0*w1 z0*z1 y0*y1 x0*x1
movaps xmm2, [eax+32]
mulps xmm2, [eax+16+32]
;w2*w3 z2*z3 y2*y3 x2*x3
movaps xmm3, [eax+64]
mulps xmm3, [eax+16+64]
;w4*w5 z4*z5 y4*y5 x4*x5
movaps xmm4, [eax+96]
mulps xmm4, [eax+16+96]
;w6*w7 z6*z7 y6*y7 x6*x7
	movaps xmm0, [eax]
	mulps xmm0, [eax+16]
	movaps xmm1, [eax+32]
	mulps xmm1, [eax+16+32]
	movaps xmm2, [eax+64]
	mulps xmm2, [eax+16+64]
	movaps xmm3, [eax+96]
	mulps xmm3, [eax+16+96]
	haddps xmm0, xmm1
	haddps xmm2, xmm3
	haddps xmm0, xmm2
	movaps [ecx], xmm0

Example 6-16. Unrolled Implementation of Four Dot Products (Contd.)

SSE2 Implementation	SSE3 Implementation

movaps xmm1, xmm0
unpcklps xmm0, xmm2
; y2*y3 y0*y1 x2*x3 x0*x1
unpckhps xmm1, xmm2
; w2*w3 w0*w1 z2*z3 z0*z1
movaps xmm5, xmm3
unpcklps xmm3, xmm4
; y6*y7 y4*y5 x6*x7 x4*x5
unpckhps xmm5, xmm4
; w6*w7 w4*w5 z6*z7 z4*z5
 	
addps xmm0, xmm1
addps xmm5, xmm3
movaps xmm1, xmm5
movhlps xmm1, xmm0
movlhps xmm0, xmm5
addps xmm0, xmm1
movaps [ecx], xmm0	

6.6.3 Vector Normalization

Normalizing vectors is a common operation in many floating-point applications. Example 6-17 shows an example in C of normalizing an array of (x, y, z) vectors.

Example 6-17. Normalization of an Array of Vectors

```
for (i=0;i<CNT;i++)
{ float size = nodes[i].vec.dot();
    if (size != 0.0)
    { size = 1.0f/sqrtf(size); }
    else
    { size = 0.0; }
    nodes[i].vec.x *= size;
    nodes[i].vec.y *= size;
    nodes[i].vec.z *= size;
}
```

Example 6-18 shows an assembly sequence that normalizes the x, y, z components of a vector.

Example 6-18. Normalize (x, y, z) Components of an Array of Vectors Using SSE2

```
Vec3 *p = &nodes[i].vec;
__asm
{    mov      eax, p
     xorps    xmm2, xmm2
     movups   xmm1, [eax] // loads the (x, y, z) of input vector plus x of next vector
     movaps   xmm7, xmm1 // save a copy of data from memory (to restore the unnormalized value)
     movaps   xmm5, _mask // mask to select (x, y, z) values from an xmm register to normalize
     andps    xmm1, xmm5 // mask 1st 3 elements
     movaps   xmm6, xmm1 // save a copy of (x, y, z) to compute normalized vector later
     mulps    xmm1,xmm1 // 0, z*z, y*y, x*x
     pshufd   xmm3, xmm1, 0x1b // x*x, y*y, z*z, 0
     addps    xmm1, xmm3 // x*x, z*z+y*y, z*z+y*y, x*x
     pshufd   xmm3, xmm1, 0x41 // z*z+y*y, x*x, x*x, z*z+y*y
     addps    xmm1, xmm3 // x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z
     comisd   xmm1, xmm2 // compare size to 0
     jz zero
     movaps   xmm3, xmm4// preloaded unitary vector (1.0, 1.0, 1.0, 1.0)
     sqrtps   xmm1, xmm1
     divps    xmm3, xmm1
     jmp      store
zero:
     movaps   xmm3, xmm2
store:

     mulps    xmm3, xmm6 //normalize the vector in the lower 3 elements
     andnps   xmm5, xmm7 // mask off the lower 3 elements to keep the un-normalized value
     orps     xmm3, xmm5 // order the un-normalized component after the normalized vector
     movaps   [eax ], xmm3 // writes normalized x, y, z; followed by unmodified value
```

Example 6-19 shows an assembly sequence using SSE4.1 to normalizes the x, y, z components of a vector.

Example 6-19. Normalize (x, y, z) Components of an Array of Vectors Using SSE4.1

```
Vec3 *p = &nodes[i].vec;
__asm
{   mov      eax, p
    xorps    xmm2, xmm2
    movups   xmm1, [eax] // loads the (x, y, z) of input vector plus x of next vector
    movaps   xmm7, xmm1 // save a copy of data from memory
    dpps     xmm1, xmm1, 0x7f // x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z
    comisd   xmm1, xmm2 // compare size to 0
    jz zero
    movaps   xmm3, xmm4// preloaded unitary vector (1.0, 1.0, 1.0, 1.0)
    sqrtps   xmm1, xmm1
    divps    xmm3, xmm1
    jmp      store
zero:
    movaps   xmm3, xmm2
store:
    mulps    xmm3, xmm6 //normalize the vector in the lower 3 elements
    blendps  xmm3, xmm7, 0x8 // copy the un-normalized component next to the normalized vector
    movaps   [eax ], xmm3
```

In Example 6-18 and Example 6-19, the throughput of these instruction sequences are basically limited by the long-latency instructions of DIVPS and SQRTPS. In Example 6-19, the use of DPPS replaces eight SSE2 instructions to evaluate and broadcast the dot-product result to four elements of an XMM register. This could result in improvement of the relative speed of Example 6-19 over Example 6-18.

6.6.4 Using Horizontal SIMD Instruction Sets and Data Layout

SSE and SSE2 provide packed add/subtract, multiply/divide instructions that are ideal for situations that can take advantage of vertical computation model, such as SOA data layout. SSE3 and SSE4.1 added horizontal SIMD instructions including horizontal add/subtract, dot-product operations. These more recent SIMD extensions provide tools to solve problems involving data layouts or operations that do not conform to the vertical SIMD computation model.

In this section, we consider a vector-matrix multiplication problem and discuss the relevant factors for choosing various horizontal SIMD instructions.

Example 6-20 shows the vector-matrix data layout in AOS, where the input and out vectors are stored as an array of structure.

Example 6-20. Data Organization in Memory for AOS Vector-Matrix Multiplication

```
Matrix M4x4 (pMat):   M00 M01 M02 M03
                      M10 M11 M12 M13
                      M20 M21 M22 M23
                      M30 M31 M32 M33
4 input vertices V4x1 (pVert):  V0x V0y V0z V0w
                                V1x V1y V1z V1w
                                V2x V2y V2z V2w
                                V3x V3y V3z V3w
Output vertices O4x1 (pOutVert): O0x O0y O0z O0w
                                 O1x O1y O1z O1w
                                 O2x O2y O2z O2w
                                 O3x O3y O3z O3w
```

Example 6-21 shows an example using HADDPS and MULPS to perform vector-matrix multiplication with data layout in AOS. After three HADDPS completing the summations of each output vector component, the output components are arranged in AOS.

Example 6-21. AOS Vector-Matrix Multiplication with HADDPS

```
    mov     eax, pMat
    mov     ebx, pVert
    mov     ecx, pOutVert
    xor                 edx, edx
    movaps  xmm5,[eax+16] // load row M1?
    movaps  xmm6,[eax+2*16] // load row M2?
    movaps  xmm7,[eax+3*16] // load row M3?
lloop:
    movaps  xmm4, [ebx + edx] // load input vector
    movaps  xmm0, xmm4
    mulps   xmm0, [eax] // m03*vw, m02*vz, m01*vy, m00*vx,
    movaps  xmm1, xmm4
    mulps   xmm1, xmm5 // m13*vw, m12*vz, m11*vy, m10*vx,

    movaps  xmm2, xmm4
    mulps   xmm2, xmm6// m23*vw, m22*vz, m21*vy, m20*vx
    movaps  xmm3, xmm4
    mulps   xmm3, xmm7 // m33*vw, m32*vz, m31*vy, m30*vx,
    haddps  xmm0, xmm1
    haddps  xmm2, xmm3
    haddps  xmm0, xmm2
    movaps  [ecx + edx], xmm0// store a vector of length 4
    add     edx, 16
    cmp     edx, top
    jb lloop
```

Example 6-22 shows an example using DPPS to perform vector-matrix multiplication in AOS.

Example 6-22. AOS Vector-Matrix Multiplication with DPPS

```
    mov         eax, pMat
    mov     ebx, pVert
    mov     ecx, pOutVert
    xor             edx, edx
    movaps  xmm5,[eax+16] // load row M1?
    movaps  xmm6,[eax+2*16] // load row M2?
    movaps  xmm7,[eax+3*16] // load row M3?
lloop:
    movaps  xmm4, [ebx + edx] // load input vector
    movaps  xmm0, xmm4
    dpps    xmm0, [eax], 0xf1// calculate dot product of length 4, store to lowest dword
    movaps  xmm1, xmm4
    dpps    xmm1, xmm5, 0xf1
    movaps  xmm2, xmm4
    dpps    xmm2, xmm6, 0xf1
    movaps  xmm3, xmm4
    dpps    xmm3, xmm7, 0xf1
    movss   [ecx + edx + 0*4], xmm0// store one element of vector length 4
    movss   [ecx + edx + 1*4], xmm1
    movss   [ecx + edx + 2*4], xmm2
    movss   [ecx + edx + 3*4], xmm3
    add     edx, 16
    cmp     edx, top
    jb      lloop
```

Example 6-21 and Example 6-22 both work with AOS data layout using different horizontal processing techniques provided by SSE3 and SSE4.1. The effectiveness of either techniques will vary, depending on the degree of exposures of long-latency instruction in the inner loop, the overhead/efficiency of data movement, and the latency of HADDPS vs. DPPS.

On processors that support both HADDPS and DPPS, the choice between either technique may depend on application-specific considerations. If the output vectors are written back to memory directly in a batch situation, Example 6-21 may be preferable over Example 6-22, because the latency of DPPS is long and storing each output vector component individually is less than ideal for storing an array of vectors.

There may be partially-vectorizable situations that the individual output vector component is consumed immediately by other non-vectorizable computations. Then, using DPPS producing individual component may be more suitable than dispersing the packed output vector produced by three HADDPS as in Example 6-21.

6.6.4.1 SOA and Vector Matrix Multiplication

If the native data layout of a problem conforms to SOA, then vector-matrix multiply can be coded using MULPS, ADDPS without using the longer-latency horizontal arithmetic instructions, or packing scalar components into packed format (Example 6-22). To achieve higher throughput with SOA data layout, there are either pre-requisite data preparation or swizzling/deswizzling on-the-fly that must be comprehended. For example, an SOA data layout for vector-matrix multiplication is shown in Example 6-23.

Each matrix element is replicated four times to minimize data movement overhead for producing packed results.

Example 6-23. Data Organization in Memory for SOA Vector-Matrix Multiplication

```
Matrix M16x4 (pMat):
        M00 M00 M00 M00 M01 M01 M01 M01 M02 M02 M02 M02 M03 M03 M03 M03
        M10 M10 M10 M10 M11 M11 M11 M11 M12 M12 M12 M12 M13 M13 M13 M13
        M20 M20 M20 M20 M21 M21 M21 M21 M22 M22 M22 M22 M23 M23 M23 M23
        M30 M30 M30 M30 M31 M31 M31 M31 M32 M32 M32 M32 M33 M33 M33 M33
4 input vertices V4x1 (pVert):   V0x V1x V2x V3x
                                 V0y V1y V2y V3y
                                 V0z V1z V2z V3z
                                 V0w V1w V2w V3w
Ouput vertices O4x1 (pOutVert): O0x O1x O2x O3x
                                 O0y O1y O2y O3y
                                 O0z O1z O2z O3z
                                 O0w O1w O2w O3w
```

The corresponding vector-matrix multiply example in SOA (unrolled for four iteration of vectors) is shown in Example 6-24.

Example 6-24. Vector-Matrix Multiplication with Native SOA Data Layout

```
    mov     ebx, pVert
    mov     ecx, pOutVert
    xor     edx, edx
    movaps  xmm5,[eax+16] // load row M1?
    movaps  xmm6,[eax+2*16] // load row M2?
    movaps  xmm7,[eax+3*16] // load row M3?
lloop_vert:
    mov     eax, pMat
    xor     edi, edi
    movaps  xmm0, [ebx ] // load V3x, V2x, V1x, V0x
    movaps  xmm1, [ebx ] // load V3y, V2y, V1y, V0y
    movaps  xmm2, [ebx ] // load V3z, V2z, V1z, V0z
    movaps  xmm3, [ebx ] // load V3w, V2w, V1w, V0w
loop_mat:
    movaps  xmm4, [eax] // m00, m00, m00, m00,
    mulps   xmm4, xmm0 // m00*V3x, m00*V2x, m00*V1x, m00*V0x,
    movaps  xmm4, [eax + 16] // m01, m01, m01, m01,
    mulps   xmm5, xmm1 // m01*V3y, m01*V2y, m01*V1y, m01*V0y,
    addps   xmm4, xmm5
    movaps  xmm5, [eax + 32] // m02, m02, m02, m02,
    mulps   xmm5, xmm2 // m02*V3z, m02*V2z, m02*V1z, m02*V0z,
    addps   xmm4, xmm5
    movaps  xmm5, [eax + 48] // m03, m03, m03, m03,
    mulps   xmm5, xmm3 // m03*V3w, m03*V2w, m03*V1w, m03*V0w,
    addps   xmm4, xmm5
    movaps  [ecx + edx], xmm4
    add     eax, 64
    add     edx, 16
    add     edi, 1
    cmp     edi, 4
    jb lloop_mat
    add     ebx, 64
    cmp     edx, top
    jb lloop_vert
```

CHAPTER 7
OPTIMIZING CACHE USAGE

Over the past decade, processor speed has increased. Memory access speed has increased at a slower pace. The resulting disparity has made it important to tune applications in one of two ways: either (a) a majority of data accesses are fulfilled from processor caches, or (b) effectively masking memory latency to utilize peak memory bandwidth as much as possible.

Hardware prefetching mechanisms are enhancements in microarchitecture to facilitate the latter aspect, and will be most effective when combined with software tuning. The performance of most applications can be considerably improved if the data required can be fetched from the processor caches or if memory traffic can take advantage of hardware prefetching effectively.

Standard techniques to bring data into the processor before it is needed involve additional programming which can be difficult to implement and may require special steps to prevent performance degradation. Streaming SIMD Extensions addressed this issue by providing various prefetch instructions.

Streaming SIMD Extensions introduced the various non-temporal store instructions. SSE2 extends this support to new data types and also introduce non-temporal store support for the 32-bit integer registers.

This chapter focuses on:

- Hardware Prefetch Mechanism, Software Prefetch and Cacheability Instructions — Discusses micro-architectural feature and instructions that allow you to affect data caching in an application.
- Memory Optimization Using Hardware Prefetching, Software Prefetch and Cacheability Instructions — Discusses techniques for implementing memory optimizations using the above instructions.

NOTE

In a number of cases presented, the prefetching and cache utilization described are specific to current implementations of Intel NetBurst microarchitecture but are largely applicable for the future processors.

- Using deterministic cache parameters to manage cache hierarchy.

7.1 GENERAL PREFETCH CODING GUIDELINES

The following guidelines will help you to reduce memory traffic and utilize peak memory system bandwidth more effectively when large amounts of data movement must originate from the memory system:

- Take advantage of the hardware prefetcher's ability to prefetch data that are accessed in linear patterns, in either a forward or backward direction.
- Take advantage of the hardware prefetcher's ability to prefetch data that are accessed in a regular pattern with access strides that are substantially smaller than half of the trigger distance of the hardware prefetch.
- Use a current-generation compiler, such as the Intel C++ Compiler that supports C++ language-level features for Streaming SIMD Extensions. Streaming SIMD Extensions and MMX technology instructions provide intrinsics that allow you to optimize cache utilization. Examples of Intel compiler intrinsics include: _mm_prefetch, _mm_stream and _mm_load, _mm_sfence. For details, refer to Intel C++ Compiler User's Guide documentation.
- Facilitate compiler optimization by:
  - Minimize use of global variables and pointers.
  - Minimize use of complex control flow.
  - Use the const modifier, avoid register modifier.
  - Choose data types carefully (see below) and avoid type casting.

- Use cache blocking techniques (for example, strip mining) as follows:
  - Improve cache hit rate by using cache blocking techniques such as strip-mining (one dimensional arrays) or loop blocking (two dimensional arrays).
  - Explore using hardware prefetching mechanism if your data access pattern has sufficient regularity to allow alternate sequencing of data accesses (for example: tiling) for improved spatial locality. Otherwise use PREFETCHNTA.
- Balance single-pass versus multi-pass execution:
  - Single-pass, or unlayered execution passes a single data element through an entire computation pipeline.
  - Multi-pass, or layered execution performs a single stage of the pipeline on a batch of data elements before passing the entire batch on to the next stage.
  - If your algorithm is single-pass use PREFETCHNTA. If your algorithm is multi-pass use PREFETCHT0.
- Resolve memory bank conflict issues. Minimize memory bank conflicts by applying array grouping to group contiguously used data together or by allocating data within 4-KByte memory pages.
- Resolve cache management issues. Minimize the disturbance of temporal data held within processor's caches by using streaming store instructions.
- Optimize software prefetch scheduling distance:
  - Far ahead enough to allow interim computations to overlap memory access time.
  - Near enough that prefetched data is not replaced from the data cache.
- Use software prefetch concatenation. Arrange prefetches to avoid unnecessary prefetches at the end of an inner loop and to prefetch the first few iterations of the inner loop inside the next outer loop.
- Minimize the number of software prefetches. Prefetch instructions are not completely free in terms of bus cycles, machine cycles and resources; excessive usage of prefetches can adversely impact application performance.
- Interleave prefetches with computation instructions. For best performance, software prefetch instructions must be interspersed with computational instructions in the instruction sequence (rather than clustered together).

7.2 HARDWARE PREFETCHING OF DATA

Pentium M, Intel Core Solo, and Intel Core Duo processors and processors based on Intel Core microarchitecture and Intel NetBurst microarchitecture provide hardware data prefetch mechanisms which monitor application data access patterns and prefetches data automatically. This behavior is automatic and does not require programmer intervention.

For processors based on Intel NetBurst microarchitecture, characteristics of the hardware data prefetcher are:

1. It requires two successive cache misses in the last level cache to trigger the mechanism; these two cache misses must satisfy the condition that strides of the cache misses are less than the trigger distance of the hardware prefetch mechanism.
2. Attempts to stay 256 bytes ahead of current data access locations.
3. Follows only one stream per 4-KByte page (load or store).
4. Can prefetch up to 8 simultaneous, independent streams from eight different 4-KByte regions
5. Does not prefetch across 4-KByte boundary. This is independent of paging modes.
6. Fetches data into second/third-level cache.
7. Does not prefetch UC or WC memory types.

8. Follows load and store streams. Issues Read For Ownership (RFO) transactions for store streams and Data Reads for load streams.

Other than items 2 and 4 discussed above, most other characteristics also apply to Pentium M, Intel Core Solo and Intel Core Duo processors. The hardware prefetcher implemented in the Pentium M processor fetches data to a second level cache. It can track 12 independent streams in the forward direction and 4 independent streams in the backward direction. The hardware prefetcher of Intel Core Solo processor can track 16 forward streams and 4 backward streams. On the Intel Core Duo processor, the hardware prefetcher in each core fetches data independently.

Hardware prefetch mechanisms of processors based on Intel Core microarchitecture are discussed in Section 3.7.2 and Section 3.7.3. Despite differences in hardware implementation technique, the overall benefit of hardware prefetching to software are similar between Intel Core microarchitecture and prior microarchitectures.

7.3 PREFETCH AND CACHEABILITY INSTRUCTIONS

The PREFETCH instruction, inserted by the programmers or compilers, accesses a minimum of two cache lines of data on the Pentium 4 processor prior to the data actually being needed (one cache line of data on the Pentium M processor). This hides the latency for data access in the time required to process data already resident in the cache.

Many algorithms can provide information in advance about the data that is to be required. In cases where memory accesses are in long, regular data patterns; the automatic hardware prefetcher should be favored over software prefetches.

The cacheability control instructions allow you to control data caching strategy in order to increase cache efficiency and minimize cache pollution.

Data reference patterns can be classified as follows:

- Temporal — Data will be used again soon.
- Spatial — Data will be used in adjacent locations (for example, on the same cache line).
- Non-temporal — Data which is referenced once and not reused in the immediate future (for example, for some multimedia data types, as the vertex buffer in a 3D graphics application).

These data characteristics are used in the discussions that follow.

7.4 PREFETCH

This section discusses the mechanics of the software PREFETCH instructions. In general, software prefetch instructions should be used to supplement the practice of tuning an access pattern to suit the automatic hardware prefetch mechanism.

7.4.1 Software Data Prefetch

The PREFETCH instruction can hide the latency of data access in performance-critical sections of application code by allowing data to be fetched in advance of actual usage. PREFETCH instructions do not change the user-visible semantics of a program, although they may impact program performance. PREFETCH merely provides a hint to the hardware and generally does not generate exceptions or faults.

PREFETCH loads either non-temporal data or temporal data in the specified cache level. This data access type and the cache level are specified as a hint. Depending on the implementation, the instruction fetches 32 or more aligned bytes (including the specified address byte) into the instruction-specified cache levels.

PREFETCH is implementation-specific; applications need to be tuned to each implementation to maximize performance.

NOTE

Using the PREFETCH instruction is recommended only if data does not fit in cache. Use of software prefetch should be limited to memory addresses that are managed or owned within the application context. Prefetching to addresses that are not mapped to physical pages can experience non-deterministic performance penalty. For example specifying a NULL pointer (0L) as address for a prefetch can cause long delays.

PREFETCH provides a hint to the hardware; it does not generate exceptions or faults except for a few special cases (see Section 7.4.3, “Prefetch and Load Instructions”). However, excessive use of PREFETCH instructions may waste memory bandwidth and result in a performance penalty due to resource constraints.

Nevertheless, PREFETCH can lessen the overhead of memory transactions by preventing cache pollution and by using caches and memory efficiently. This is particularly important for applications that share critical system resources, such as the memory bus. See an example in Section 7.7.2.1, “Video Encoder.”

PREFETCH is mainly designed to improve application performance by hiding memory latency in the background. If segments of an application access data in a predictable manner (for example, using arrays with known strides), they are good candidates for using PREFETCH to improve performance.

Use the PREFETCH instructions in:

- Predictable memory access patterns.
- Time-consuming innermost loops.
- Locations where the execution pipeline may stall if data is not available.

7.4.2 Prefetch Instructions – Pentium® 4 Processor Implementation

Streaming SIMD Extensions include four PREFETCH instructions variants, one non-temporal and three temporal. They correspond to two types of operations, temporal and non-temporal.

NOTE

At the time of PREFETCH, if data is already found in a cache level that is closer to the processor than the cache level specified by the instruction, no data movement occurs.

The non-temporal instruction is:

- PREFETCHNTA— Fetch the data into the second-level cache, minimizing cache pollution.

Temporal instructions are:

- PREFETCHT0 — Fetch the data into all cache levels; that is, to the second-level cache for the Pentium 4 processor.
- PREFETCHT1 — This instruction is identical to PREFETCHT0.
- PREFETCHT2 — This instruction is identical to PREFETCHT0.

7.4.3 Prefetch and Load Instructions

The Pentium 4 processor has a decoupled execution and memory architecture that allows instructions to be executed independently with memory accesses (if there are no data and resource dependencies). Programs or compilers can use dummy load instructions to imitate PREFETCH functionality; but preloading is not completely equivalent to using PREFETCH instructions. PREFETCH provides greater performance than preloading.

Currently, PREFETCH provides greater performance than preloading because:

- Has no destination register, it only updates cache lines.
- Does not stall the normal instruction retirement.
- Does not affect the functional behavior of the program.

- Has no cache line split accesses.
- Does not cause exceptions except when the LOCK prefix is used. The LOCK prefix is not a valid prefix for use with PREFETCH.
- Does not complete its own execution if that would cause a fault.

Currently, the advantage of PREFETCH over preloading instructions are processor-specific. This may change in the future.

There are cases where a PREFETCH will not perform the data prefetch. These include:

- PREFETCH causes a DTLB (Data Translation Lookaside Buffer) miss. This applies to Pentium 4 processors with CPUID signature corresponding to family 15, model 0, 1, or 2. PREFETCH resolves DTLB misses and fetches data on Pentium 4 processors with CPUID signature corresponding to family 15, model 3.
- An access to the specified address that causes a fault/exception.
- If the memory subsystem runs out of request buffers between the first-level cache and the second-level cache.
- PREFETCH targets an uncacheable memory region (for example, USWC and UC).
- The LOCK prefix is used. This causes an invalid opcode exception.

7.5 CACHEABILITY CONTROL

This section covers the mechanics of cacheability control instructions.

7.5.1 The Non-temporal Store Instructions

This section describes the behavior of streaming stores and reiterates some of the information presented in the previous section.

In Streaming SIMD Extensions, the MOVNTPS, MOVNTPD, MOVNTQ, MOVNTDQ, MOVNTI, MASKMOVQ and MASKMOVDQU instructions are streaming, non-temporal stores. With regard to memory characteristics and ordering, they are similar to the Write-Combining (WC) memory type:

- Write combining — Successive writes to the same cache line are combined.
- Write collapsing — Successive writes to the same byte(s) result in only the last write being visible.
- Weakly ordered — No ordering is preserved between WC stores or between WC stores and other loads or stores.
- Uncacheable and not write-allocating — Stored data is written around the cache and will not generate a read-for-ownership bus request for the corresponding cache line.

7.5.1.1 Fencing

Because streaming stores are weakly ordered, a fencing operation is required to ensure that the stored data is flushed from the processor to memory. Failure to use an appropriate fence may result in data being “trapped” within the processor and will prevent visibility of this data by other processors or system agents.

WC stores require software to ensure coherence of data by performing the fencing operation. See Section 7.5.5, “FENCE Instructions.”

7.5.1.2 Streaming Non-temporal Stores

Streaming stores can improve performance by:

- Increasing store bandwidth if the 64 bytes that fit within a cache line are written consecutively (since they do not require read-for-ownership bus requests and 64 bytes are combined into a single bus write transaction).
- Reducing disturbance of frequently used cached (temporal) data (since they write around the processor caches).

Streaming stores allow cross-aliasing of memory types for a given memory region. For instance, a region may be mapped as write-back (WB) using page attribute tables (PAT) or memory type range registers (MTRRs) and yet is written using a streaming store.

7.5.1.3 Memory Type and Non-temporal Stores

Memory type can take precedence over a non-temporal hint, leading to the following considerations:

- If the programmer specifies a non-temporal store to strongly-ordered uncacheable memory (for example, Uncacheable (UC) or Write-Protect (WP) memory types), then the store behaves like an uncacheable store. The non-temporal hint is ignored and the memory type for the region is retained.
- If the programmer specifies the weakly-ordered uncacheable memory type of Write-Combining (WC), then the non-temporal store and the region have the same semantics and there is no conflict.
- If the programmer specifies a non-temporal store to cacheable memory (for example, Write-Back (WB) or Write-Through (WT) memory types), two cases may result:
  - CASE 1 — If the data is present in the cache hierarchy, the instruction will ensure consistency. A particular processor may choose different ways to implement this. The following approaches are probable: (a) updating data in-place in the cache hierarchy while preserving the memory type semantics assigned to that region or (b) evicting the data from the caches and writing the new non-temporal data to memory (with WC semantics).

    The approaches (separate or combined) can be different for future processors. Pentium 4, Intel Core Solo and Intel Core Duo processors implement the latter policy (of evicting data from all processor caches). The Pentium M processor implements a combination of both approaches.

    If the streaming store hits a line that is present in the first-level cache, the store data is combined in place within the first-level cache. If the streaming store hits a line present in the second-level, the line and stored data is flushed from the second-level to system memory.
  - CASE 2 — If the data is not present in the cache hierarchy and the destination region is mapped as WB or WT; the transaction will be weakly ordered and is subject to all WC memory semantics. This non-temporal store will not write-allocate. Different implementations may choose to collapse and combine such stores.

7.5.1.4 Write-Combining

Generally, WC semantics require software to ensure coherence with respect to other processors and other system agents (such as graphics cards). Appropriate use of synchronization and a fencing operation must be performed for producer-consumer usage models (see Section 7.5.5, "FENCE Instructions"). Fencing ensures that all system agents have global visibility of the stored data. For instance, failure to fence may result in a written cache line staying within a processor, and the line would not be visible to other agents.

For processors which implement non-temporal stores by updating data in-place that already resides in the cache hierarchy, the destination region should also be mapped as WC. Otherwise, if mapped as WB or WT, there is a potential for speculative processor reads to bring the data into the caches. In such a case, non-temporal stores would then update in place and data would not be flushed from the processor by a subsequent fencing operation.

The memory type visible on the bus in the presence of memory type aliasing is implementation-specific. As one example, the memory type written to the bus may reflect the memory type for the first store to the line, as seen in program order. Other alternatives are possible. This behavior should be considered reserved and dependence on the behavior of any particular implementation risks future incompatibility.

7.5.2 Streaming Store Usage Models

The two primary usage domains for streaming store are coherent requests and non-coherent requests.

7.5.2.1 Coherent Requests

Coherent requests are normal loads and stores to system memory, which may also hit cache lines present in another processor in a multiprocessor environment. With coherent requests, a streaming store can be used in the same way as a regular store that has been mapped with a WC memory type (PAT or MTRR). An SFENCE instruction must be used within a producer-consumer usage model in order to ensure coherency and visibility of data between processors.

Within a single-processor system, the CPU can also re-read the same memory location and be assured of coherence (that is, a single, consistent view of this memory location). The same is true for a multiprocessor (MP) system, assuming an accepted MP software producer-consumer synchronization policy is employed.

7.5.2.2 Non-coherent requests

Non-coherent requests arise from an I/O device, such as an AGP graphics card, that reads or writes system memory using non-coherent requests, which are not reflected on the processor bus and thus will not query the processor's caches. An SFENCE instruction must be used within a producer-consumer usage model in order to ensure coherency and visibility of data between processors. In this case, if the processor is writing data to the I/O device, a streaming store can be used with a processor with any behavior of Case 1 (Section 7.5.1.3) only if the region has also been mapped with a WC memory type (PAT, MTRR).

NOTE

> Failure to map the region as WC may allow the line to be speculatively read into the processor caches (via the wrong path of a mispredicted branch).

In case the region is not mapped as WC, the streaming might update in-place in the cache and a subsequent SFENCE would not result in the data being written to system memory. Explicitly mapping the region as WC in this case ensures that any data read from this region will not be placed in the processor's caches. A read of this memory location by a non-coherent I/O device would return incorrect/out-of-date results.

For a processor which solely implements Case 2 (Section 7.5.1.3), a streaming store can be used in this non-coherent domain without requiring the memory region to also be mapped as WB, since any cached data will be flushed to memory by the streaming store.

7.5.3 Streaming Store Instruction Descriptions

MOVNTQ/MOVNTDQ (non-temporal store of packed integer in an MMX technology or Streaming SIMD Extensions register) store data from a register to memory. They are implicitly weakly-ordered, do no write-allocate, and so minimize cache pollution.

MOVNTPS (non-temporal store of packed single precision floating-point) is similar to MOVNTQ. It stores data from a Streaming SIMD Extensions register to memory in 16-byte granularity. Unlike MOVNTQ, the memory address must be aligned to a 16-byte boundary or a general protection exception will occur. The instruction is implicitly weakly-ordered, does not write-allocate, and thus minimizes cache pollution.

MASKMOVQ/MASKMOVDQU (non-temporal byte mask store of packed integer in an MMX technology or Streaming SIMD Extensions register) store data from a register to the location specified by the EDI register. The most significant bit in each byte of the second mask register is used to selectively write the data of the first register on a per-byte basis. The instructions are implicitly weakly-ordered (that is, successive stores may not write memory in original program-order), do not write-allocate, and thus minimize cache pollution.

7.5.4 The Streaming Load Instruction

SSE4.1 introduces the MOVNTDQA instruction. MOVNTDQA loads 16 bytes from memory using a non-temporal hint if the memory source is WC type. For WC memory type, the non-temporal hint may be implemented by loading into a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Subsequent MOVNTDQA reads to unread portions of the buffered WC data will cause 16 bytes of data transferred from the temporary internal buffer to an XMM register if data is available.

If used appropriately, MOVNTDQA can help software achieve significantly higher throughput when loading data in WC memory region into the processor than other means.

Chapter 1 provides a reference to an application note on using MOVNTDQA. Additional information and requirements to use MOVNTDQA appropriately can be found in Chapter 12, "Programming with SSE3, SSSE3 and SSE4" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, and the instruction reference pages of MOVNTDQA in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

7.5.5 FENCE Instructions

The following fence instructions are available: SFENCE, lFENCE, and MFENCE.

7.5.5.1 SFENCE Instruction

The SFENCE (STORE FENCE) instruction makes it possible for every STORE instruction that precedes an SFENCE in program order to be globally visible before any STORE that follows the SFENCE. SFENCE provides an efficient way of ensuring ordering between routines that produce weakly-ordered results.

The use of weakly-ordered memory types can be important under certain data sharing relationships (such as a producer-consumer relationship). Using weakly-ordered memory can make assembling the data more efficient, but care must be taken to ensure that the consumer obtains the data that the producer intended to see.

Some common usage models may be affected by weakly-ordered stores. Examples are:

- Library functions, which use weakly-ordered memory to write results.
- Compiler-generated code, which also benefits from writing weakly-ordered results.
- Hand-crafted code.

The degree to which a consumer of data knows that the data is weakly-ordered can vary for different cases. As a result, SFENCE should be used to ensure ordering between routines that produce weakly-ordered data and routines that consume this data.

7.5.5.2 LFENCE Instruction

The LFENCE (LOAD FENCE) instruction makes it possible for every LOAD instruction that precedes the LFENCE instruction in program order to be globally visible before any LOAD instruction that follows the LFENCE.

The LFENCE instruction provides a means of segregating LOAD instructions from other LOADs.

7.5.5.3 MFENCE Instruction

The MFENCE (MEMORY FENCE) instruction makes it possible for every LOAD/STORE instruction preceding MFENCE in program order to be globally visible before any LOAD/STORE following MFENCE. MFENCE provides a means of segregating certain memory instructions from other memory references.

The use of a LFENCE and SFENCE is not equivalent to the use of a MFENCE since the load and store fences are not ordered with respect to each other. In other words, the load fence can be executed before prior stores and the store fence can be executed before prior loads.

MFENCE should be used whenever the cache line flush instruction (CLFLUSH) is used to ensure that speculative memory references generated by the processor do not interfere with the flush. See Section 7.5.6, "CLFLUSH Instruction."

7.5.6 CLFLUSH Instruction

The CLFLUSH instruction invalidates the cache line associated with the linear address that contain the byte address of the memory location, from all levels of the processor cache hierarchy (data and instruction). This invalidation is broadcast throughout the coherence domain. If, at any level of the cache hierarchy, a line is inconsistent with memory (dirty), it is written to memory before invalidation. Other characteristics include:

- The data size affected is the cache coherency size, which is enumerated by the CPUID instruction. It is typically 64 bytes.
- The memory attribute of the page containing the affected line has no effect on the behavior of this instruction.
- The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load.

Executions of the CLFLUSH instruction are ordered with respect to each other and with respect to writes, locked read-modify-write instructions, fence instructions, and executions of CLFLUSHOPT to the same cache line[1]. They are not ordered with respect to executions of CLFLUSHOPT to different cache lines. For updated memory order details of CLFLUSH and other memory traffic, please refer to the CLFLUSH reference pages in Chapter 3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, and the "Memory Ordering" section in Chapter 8 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

As an example, consider a video usage model where a video capture device is using non-coherent accesses to write a capture stream directly to system memory. Since these non-coherent writes are not broadcast on the processor bus, they will not flush copies of the same locations that reside in the processor caches. As a result, before the processor re-reads the capture buffer, it should use CLFLUSH to ensure that stale, cached copies of the capture buffer are flushed from the processor caches.

Example 7-1 provides pseudo-code for CLFLUSH usage.

Example 7-1. Pseudo-code Using CLFLUSH

```
while (!buffer_ready} {}
sfence
    for(i=0;i<num_cachelines;i+=cacheline_size) {
    clflush (char *)((unsigned int)buffer + i)
}
    prefnta buffer[0];
    VAR = buffer[0];
```

The throughput characteristics of using CLFLUSH to flush cache lines can vary significantly depending on several factors. In general using CLFLUSH back-to-back to flush a large number of cache lines will experience larger cost per cache line than flushing a moderately-sized buffer (e.g. less than 4KB); the reduction of CLFLUSH throughput can be an order of magnitude. Flushing cache lines in modified state are more costly than flushing cache lines in non-modified states.

1. Memory order recommendation of CLFLUSH in previous manuals had required software to add MFENCE after CLFLUSH. MFENCE is not required following CLFLUSH as all processors implementing the CLFLUSH instruction also order it relative to the other operations enumerated above.

7.5.7 CLFLUSHOPT Instruction

The CLFLUSHOPT instruction is first introduced in the 6th Generation Intel Core Processors. Similar to CLFLUSH, CLFLUSHOPT invalidates the cache line associated with the linear address that contain the byte address of the memory location, in all levels of the processor cache hierarchy (data and instruction).

Executions of the CLFLUSHOPT instruction are ordered with respect to locked read-modify-write instructions, fence instructions, and writes to the cache line being invalidated. (They are also ordered with respect to executions of CLFLUSH and CLFLUSHOPT to the same cache line.) They are not ordered with respect to writes to cache lines other than the one being invalidated. (They are also not ordered with respect to executions of CLFLUSH and CLFLUSHOPT to different cache lines.) Software can insert an SFENCE instruction between CFLUSHOPT and a store to another cache line with which the CLFLUSHOPT should be ordered.

In general, CLFLUSHOPT throughput is higher than that of CLFLUSH, because CLFLUSHOPT orders itself with respect to a smaller set of memory traffic as described above and in Section 7.5.6. The throughput of CLFLUSHOPT will also vary. When using CLFLUSHOPT, flushing modified cache lines will experience a higher cost than flushing cache lines in non-modified states. CLFLUSHOPT will provide a performance benefit over CLFLUSH for cache lines in any coherence states. CLFLUSHOPT is more suitable to flush large buffers (e.g. greater than many KBytes), compared to CLFLUSH. In single-threaded applications, flushing buffers using CLFLUSHOPT may be up to 9X better than using CLFLUSH with Skylake microarchitecture.

Figure 7-1 shows the comparison of the performance characteristics of executing CLFLUSHOPT versus CLFLUSH for buffers of various sizes.

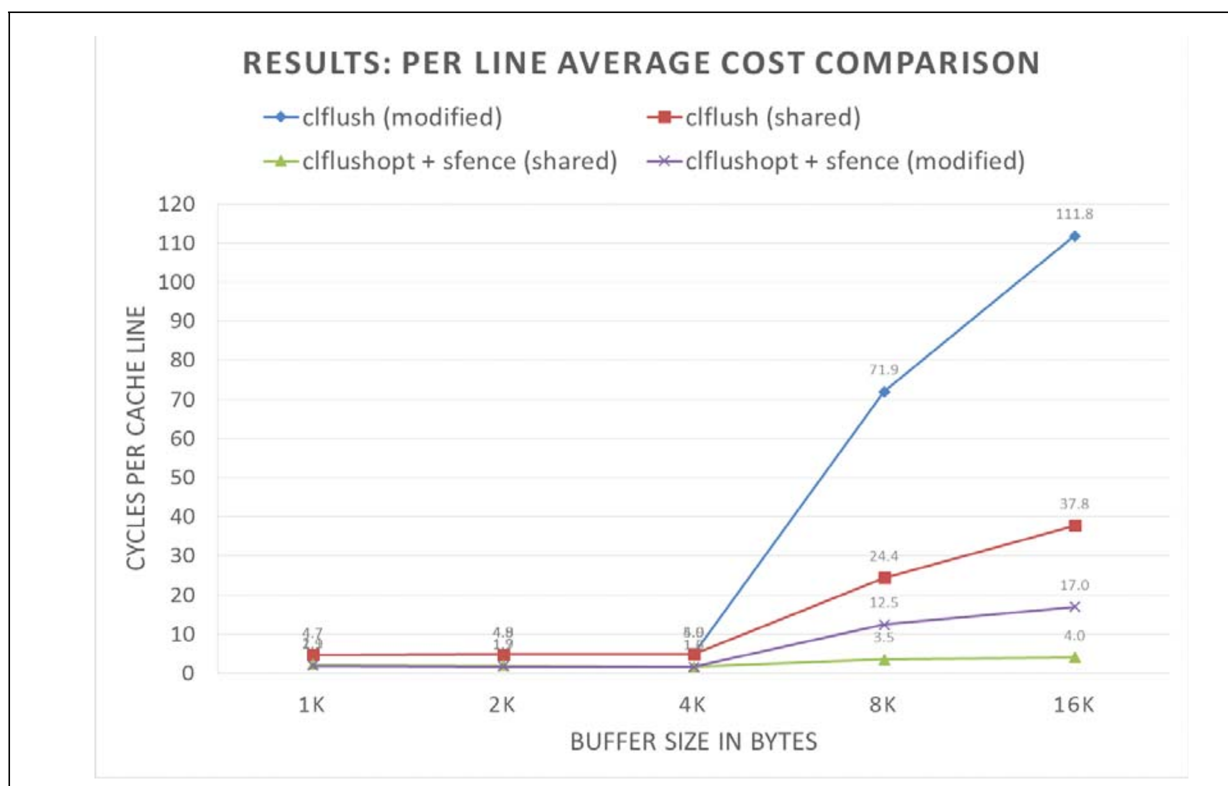


Figure 7-1. CLFLUSHOPT versus CLFLUSH In SkyLake Microarchitecture

User/Source Coding Rule 17. *If CLFLUSHOPT is available, use CLFLUSHOPT over CLFLUSH and use SFENCE to guard CLFLUSHOPT to ensure write order is globally observed. If CLUSHOPT is not available, consider flushing large buffers with CLFLUSH in smaller chunks of less than 4KB.*

Example 7-2 gives equivalent assembly sequences of flushing cache lines using CLFLUSH or CLFLUSHOPT. The corresponding sequence in C are:

CLFLUSH:

For (i = 0; i < iSizeOfBufferToFlush; i += CACHE_LINE_SIZE) _mm_clflush(&pBufferToFlush[i]);

CLFLUSHOPT:

_mm_sfence();

For (i = 0; i < iSizeOfBufferToFlush; i += CACHE_LINE_SIZE) _mm_clflushopt(&pBufferToFlush[i]);

_mm_sfence();

Example 7-2. Flushing Cache Lines Using CLFLUSH or CLFLUSHOPT

CLFLUSH no longer requires mfence	CLFLUSHOPT w/ SFENCE

xor rcx, rcx
mov r9, pBufferToFlush
mov rsi, iSizeOfBufferToFlush
;; mfence - obsolete
loop:
clflush [r9+rcx]
add rcx, 0x40
cmp rcx, rsi
jl loop
;; mfence - obsolete
	xor rcx, rcx
	mov r9, pBufferToFlush
	mov rsi, iSizeOfBufferToFlush
	sfence
	loop:
	clflushopt [r9+rcx]
	add rcx, 0x40
	cmp rcx, rsi
	jl loop
	sfence

* If imposing memory ordering rules is important for the application then executing CLFLUSHOPT instructions should be guarded with SFENCE instructions to guarantee order of memory writes. As per the figure above, such solution still performs better than using the CLFLUSH instruction, and its performance is identical to CLFLUSHOPT from 2048 byte buffers and bigger.	

7.6 MEMORY OPTIMIZATION USING PREFETCH

Recent generations of Intel processors have two mechanisms for data prefetch: software-controlled prefetch and an automatic hardware prefetch.

7.6.1 Software-Controlled Prefetch

The software-controlled prefetch is enabled using the four PREFETCH instructions introduced with Streaming SIMD Extensions instructions. These instructions are hints to bring a cache line of data in to various levels and modes in the cache hierarchy. The software-controlled prefetch is not intended for prefetching code. Using it can incur significant penalties on a multiprocessor system when code is shared.

Software prefetching has the following characteristics:

- Can handle irregular access patterns which do not trigger the hardware prefetcher.
- Can use less bus bandwidth than hardware prefetching; see below.
- Software prefetches must be added to new code, and do not benefit existing applications.

7.6.2 Hardware Prefetch

Automatic hardware prefetch can bring cache lines into the unified last-level cache based on prior data misses. It will attempt to prefetch two cache lines ahead of the prefetch stream. Characteristics of the hardware prefetcher are:

- It requires some regularity in the data access patterns.
  - If a data access pattern has constant stride, hardware prefetching is effective if the access stride is less than half of the trigger distance of hardware prefetcher.
  - If the access stride is not constant, the automatic hardware prefetcher can mask memory latency if the strides of two successive cache misses are less than the trigger threshold distance (small-stride memory traffic).
  - The automatic hardware prefetcher is most effective if the strides of two successive cache misses remain less than the trigger threshold distance and close to 64 bytes.
- There is a start-up penalty before the prefetcher triggers and there may be fetches an array finishes. For short arrays, overhead can reduce effectiveness.
  - The hardware prefetcher requires a couple misses before it starts operating.
  - Hardware prefetching generates a request for data beyond the end of an array, which is not be utilized. This behavior wastes bus bandwidth. In addition this behavior results in a start-up penalty when fetching the beginning of the next array. Software prefetching may recognize and handle these cases.
- It will not prefetch across a 4-KByte page boundary. A program has to initiate demand loads for the new page before the hardware prefetcher starts prefetching from the new page.
- The hardware prefetcher may consume extra system bandwidth if the application's memory traffic has significant portions with strides of cache misses greater than the trigger distance threshold of hardware prefetch (large-stride memory traffic).
- The effectiveness with existing applications depends on the proportions of small-stride versus large-stride accesses in the application's memory traffic. An application with a preponderance of small-stride memory traffic with good temporal locality will benefit greatly from the automatic hardware prefetcher.
- In some situations, memory traffic consisting of a preponderance of large-stride cache misses can be transformed by re-arrangement of data access sequences to alter the concentration of small-stride cache misses at the expense of large-stride cache misses to take advantage of the automatic hardware prefetcher.

7.6.3 Example of Effective Latency Reduction with Hardware Prefetch

Consider the situation that an array is populated with data corresponding to a constant-access-stride, circular pointer chasing sequence (see Example 7-3). The potential of employing the automatic hardware prefetching mechanism to reduce the effective latency of fetching a cache line from memory can be illustrated by varying the access stride between 64 bytes and the trigger threshold distance of hardware prefetch when populating the array for circular pointer chasing.

Example 7-3. Populating an Array for Circular Pointer Chasing with Constant Stride

```
register char ** p;
char  *next;        // Populating pArray for circular pointer
                    //  chasing with constant access stride
                    // p = (char **) *p; loads a value pointing to next load
    p = (char **)&pArray;
```

Example 7-3. Populating an Array for Circular Pointer Chasing with Constant Stride (Contd.)

```
for ( i = 0; i < aperture; i += stride) {
    p = (char **)&pArray[i];
    if (i + stride >= g_array_aperture) {
        next = &pArray[0 ];
        }

    else {
        next = &pArray[i + stride];
        }
 *p = next; // populate the address of the next node
}
```

The effective latency reduction for several microarchitecture implementations is shown in Figure 7-2. For a constant-stride access pattern, the benefit of the automatic hardware prefetcher begins at half the trigger threshold distance and reaches maximum benefit when the cache-miss stride is 64 bytes.

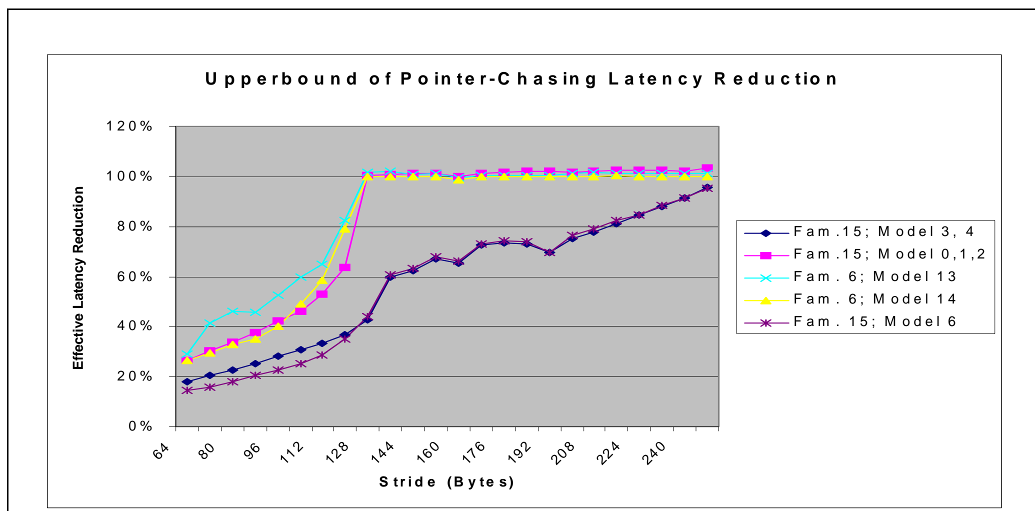


Figure 7-2. Effective Latency Reduction as a Function of Access Stride

7.6.4 Example of Latency Hiding with S/W Prefetch Instruction

Achieving the highest level of memory optimization using PREFETCH instructions requires an understanding of the architecture of a given machine. This section translates the key architectural implications into several simple guidelines for programmers to use.

Figure 7-3 and Figure 7-4 show two scenarios of a simplified 3D geometry pipeline as an example. A 3D-geometry pipeline typically fetches one vertex record at a time and then performs transformation and lighting functions on it. Both figures show two separate pipelines, an execution pipeline, and a memory pipeline (front-side bus).

Since the Pentium 4 processor (similar to the Pentium II and Pentium III processors) completely decouples the functionality of execution and memory access, the two pipelines can function concurrently. Figure 7-3 shows “bubbles” in both the execution and memory pipelines. When loads are issued for accessing vertex data, the execution units sit idle and wait until data is returned. On the other hand, the memory bus sits idle while the execution units are processing vertices. This scenario severely decreases the advantage of having a decoupled architecture.

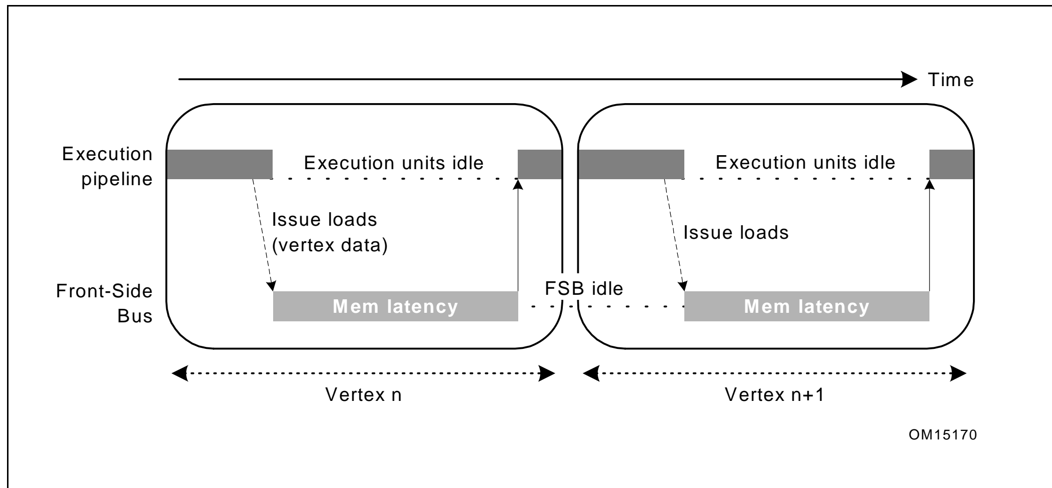


Figure 7-3. Memory Access Latency and Execution Without Prefetch

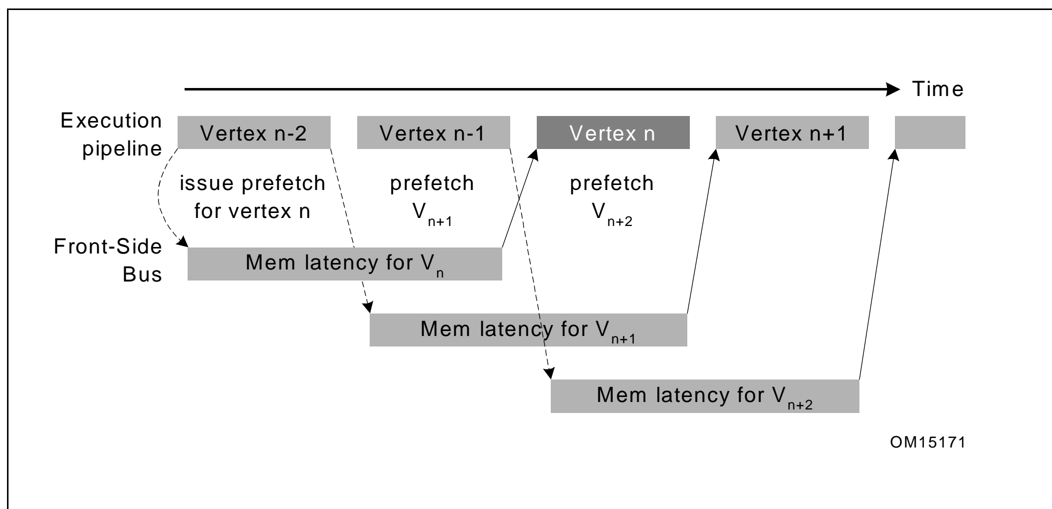


Figure 7-4. Memory Access Latency and Execution With Prefetch

The performance loss caused by poor utilization of resources can be completely eliminated by correctly scheduling the PREFETCH instructions. As shown in Figure 7-4, prefetch instructions are issued two vertex iterations ahead. This assumes that only one vertex gets processed in one iteration and a new data cache line is needed for each iteration. As a result, when iteration n, vertex V_n, is being processed; the requested data is already brought into cache. In the meantime, the front-side bus is transferring the data needed for iteration n+1, vertex V_{n+1}. Because there is no dependence between V_{n+1} data and the execution of V_n, the latency for data access of V_{n+1} can be entirely hidden behind the execution of V_n. Under such circumstances, no “bubbles” are present in the pipelines and thus the best possible performance can be achieved.

Prefetching is useful for inner loops that have heavy computations, or are close to the boundary between being compute-bound and memory-bandwidth-bound. It is probably not very useful for loops which are predominately memory bandwidth-bound.

When data is already located in the first level cache, prefetching can be useless and could even slow down the performance because the extra µops either back up waiting for outstanding memory accesses or may be dropped altogether. This behavior is platform-specific and may change in the future.

7.6.5 Software Prefetching Usage Checklist

The following checklist covers issues that need to be addressed and/or resolved to use the software PREFETCH instruction properly:

- Determine software prefetch scheduling distance.
- Use software prefetch concatenation.
- Minimize the number of software prefetches.
- Mix software prefetch with computation instructions.
- Use cache blocking techniques (for example, strip mining).
- Balance single-pass versus multi-pass execution.
- Resolve memory bank conflict issues.
- Resolve cache management issues.

Subsequent sections discuss the above items.

7.6.6 Software Prefetch Scheduling Distance

Determining the ideal prefetch placement in the code depends on many architectural parameters, including: the amount of memory to be prefetched, cache lookup latency, system memory latency, and estimate of computation cycle. The ideal distance for prefetching data is processor- and platform-dependent. If the distance is too short, the prefetch will not hide the latency of the fetch behind computation. If the prefetch is too far ahead, prefetched data may be flushed out of the cache by the time it is required.

Since prefetch distance is not a well-defined metric, for this discussion, we define a new term, prefetch scheduling distance (PSD), which is represented by the number of iterations. For large loops, prefetch scheduling distance can be set to 1 (that is, schedule prefetch instructions one iteration ahead). For small loop bodies (that is, loop iterations with little computation), the prefetch scheduling distance must be more than one iteration.

A simplified equation to compute PSD is deduced from the mathematical model. For a simplified equation, complete mathematical model, and methodology of prefetch distance determination, see Appendix D, "Summary of Rules and Suggestions."

Example 7-4 illustrates the use of a prefetch within the loop body. The prefetch scheduling distance is set to 3, ESI is effectively the pointer to a line, EDX is the address of the data being referenced and XMM1-XMM4 are the data used in computation. Example 7-5 uses two independent cache lines of data per iteration. The PSD would need to be increased/decreased if more/less than two cache lines are used per iteration.

Example 7-4. Prefetch Scheduling Distance

```
top_loop:
    prefetchnta [edx + esi + 128*3]
    prefetchnta [edx*4 + esi + 128*3]
    . . . . .
```

Example 7-4. Prefetch Scheduling Distance (Contd.)

```
     movaps  xmm1, [edx + esi]
     movaps  xmm2, [edx*4 + esi]
     movaps  xmm3, [edx + esi + 16]
     movaps  xmm4, [edx*4 + esi + 16]
     .....
     .....

     add     esi, 128
     cmp     esi, ecx
     jl      top_loop
```

7.6.7 Software Prefetch Concatenation

Maximum performance can be achieved when the execution pipeline is at maximum throughput, without incurring any memory latency penalties. This can be achieved by prefetching data to be used in successive iterations in a loop. De-pipelining memory generates bubbles in the execution pipeline.

To explain this performance issue, a 3D geometry pipeline that processes 3D vertices in strip format is used as an example. A strip contains a list of vertices whose predefined vertex order forms contiguous triangles. It can be easily observed that the memory pipe is de-pipelined on the strip boundary due to ineffective prefetch arrangement. The execution pipeline is stalled for the first two iterations for each strip. As a result, the average latency for completing an iteration will be 165 (FIX) clocks. See Appendix D, "Summary of Rules and Suggestions", for a detailed description.

This memory de-pipelining creates inefficiency in both the memory pipeline and execution pipeline. This de-pipelining effect can be removed by applying a technique called prefetch concatenation. With this technique, the memory access and execution can be fully pipelined and fully utilized.

For nested loops, memory de-pipelining could occur during the interval between the last iteration of an inner loop and the next iteration of its associated outer loop. Without paying special attention to prefetch insertion, loads from the first iteration of an inner loop can miss the cache and stall the execution pipeline waiting for data returned, thus degrading the performance.

In Example 7-5, the cache line containing A[II][0] is not prefetched at all and always misses the cache. This assumes that no array A[][] footprint resides in the cache. The penalty of memory de-pipelining stalls can be amortized across the inner loop iterations. However, it may become very harmful when the inner loop is short. In addition, the last prefetch in the last PSD iterations are wasted and consume machine resources. Prefetch concatenation is introduced here in order to eliminate the performance issue of memory de-pipelining.

Example 7-5. Using Prefetch Concatenation

```
for (ii = 0; ii < 100; ii++) {
  for (jj = 0; jj < 32; jj+=8) {
       prefetch a[ii][jj+8]
       computation a[ii][jj]
  }
}
```

Prefetch concatenation can bridge the execution pipeline bubbles between the boundary of an inner loop and its associated outer loop. Simply by unrolling the last iteration out of the inner loop and specifying

the effective prefetch address for data used in the following iteration, the performance loss of memory de-pipelining can be completely removed. Example 7-6 gives the rewritten code.

Example 7-6. Concatenation and Unrolling the Last Iteration of Inner Loop

```
for (ii = 0; ii < 100; ii++) {
  for (jj = 0; jj < 24; jj+=8) { /* N-1 iterations */
       prefetch a[ii][jj+8]
       computation a[ii][jj]
  }
  prefetch a[ii+1][0]
  computation a[ii][jj]/* Last iteration */
}
```

This code segment for data prefetching is improved and only the first iteration of the outer loop suffers any memory access latency penalty, assuming the computation time is larger than the memory latency. Inserting a prefetch of the first data element needed prior to entering the nested loop computation would eliminate or reduce the start-up penalty for the very first iteration of the outer loop. This uncomplicated high-level code optimization can improve memory performance significantly.

7.6.8 Minimize Number of Software Prefetches

Prefetch instructions are not completely free in terms of bus cycles, machine cycles and resources, even though they require minimal clock and memory bandwidth.

Excessive prefetching may lead to performance penalties because of issue penalties in the front end of the machine and/or resource contention in the memory sub-system. This effect may be severe in cases where the target loops are small and/or cases where the target loop is issue-bound.

One approach to solve the excessive prefetching issue is to unroll and/or software-pipeline loops to reduce the number of prefetches required. Figure 7-5 presents a code example which implements prefetch and unrolls the loop to remove the redundant prefetch instructions whose prefetch addresses hit the previously issued prefetch instructions. In this particular example, unrolling the original loop once saves six prefetch instructions and nine instructions for conditional jumps in every other iteration.

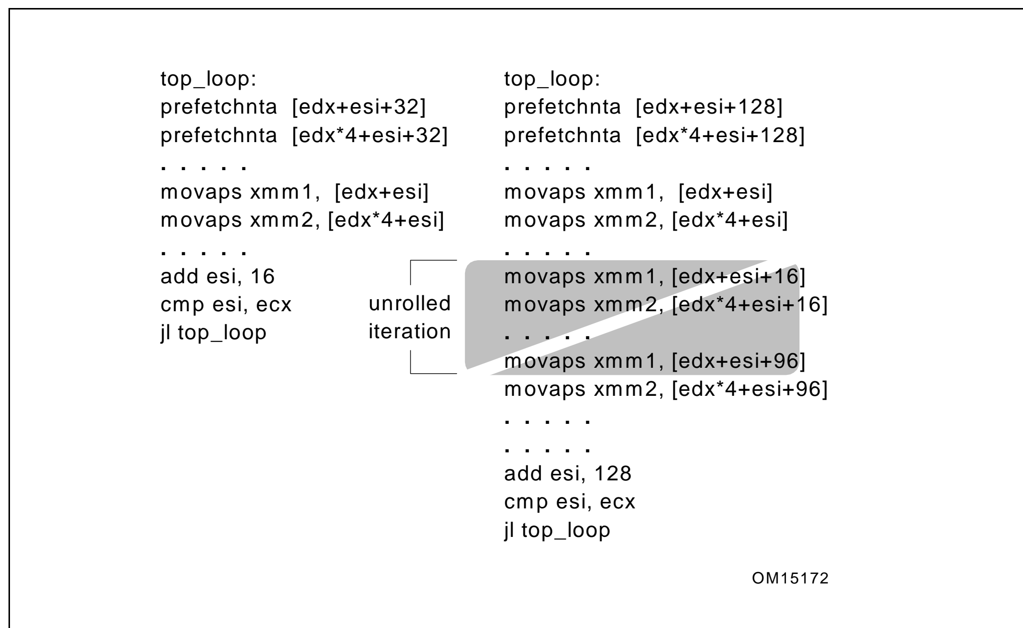


Figure 7-5. Prefetch and Loop Unrolling

Figure 7-6 demonstrates the effectiveness of software prefetches in latency hiding.

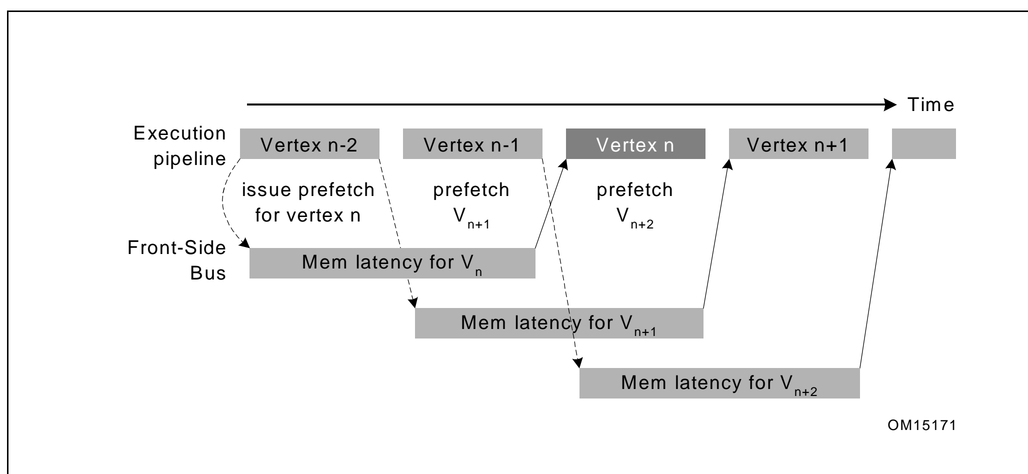


Figure 7-6. Memory Access Latency and Execution With Prefetch

The X axis in Figure 7-6 indicates the number of computation clocks per loop (each iteration is independent). The Y axis indicates the execution time measured in clocks per loop. The secondary Y axis indicates the percentage of bus bandwidth utilization. The tests vary by the following parameters:

- Number of load/store streams — Each load and store stream accesses one 128-byte cache line each per iteration.
- Amount of computation per loop — This is varied by increasing the number of dependent arithmetic operations executed.
- Number of the software prefetches per loop — For example, one every 16 bytes, 32 bytes, 64 bytes, 128 bytes.

As expected, the leftmost portion of each of the graphs in Figure 7-6 shows that when there is not enough computation to overlap the latency of memory access, prefetch does not help and that the execution is essentially memory-bound. The graphs also illustrate that redundant prefetches do not increase performance.

7.6.9 Mix Software Prefetch with Computation Instructions

It may seem convenient to cluster all of PREFETCH instructions at the beginning of a loop body or before a loop, but this can lead to severe performance degradation. In order to achieve the best possible performance, PREFETCH instructions must be interspersed with other computational instructions in the instruction sequence rather than clustered together. If possible, they should also be placed apart from loads. This improves the instruction level parallelism and reduces the potential instruction resource stalls. In addition, this mixing reduces the pressure on the memory access resources and in turn reduces the possibility of the prefetch retiring without fetching data.

Figure 7-7 illustrates distributing PREFETCH instructions. A simple and useful heuristic of prefetch spreading for a Pentium 4 processor is to insert a PREFETCH instruction every 20 to 25 clocks. Rearranging PREFETCH instructions could yield a noticeable speedup for the code which stresses the cache resource.

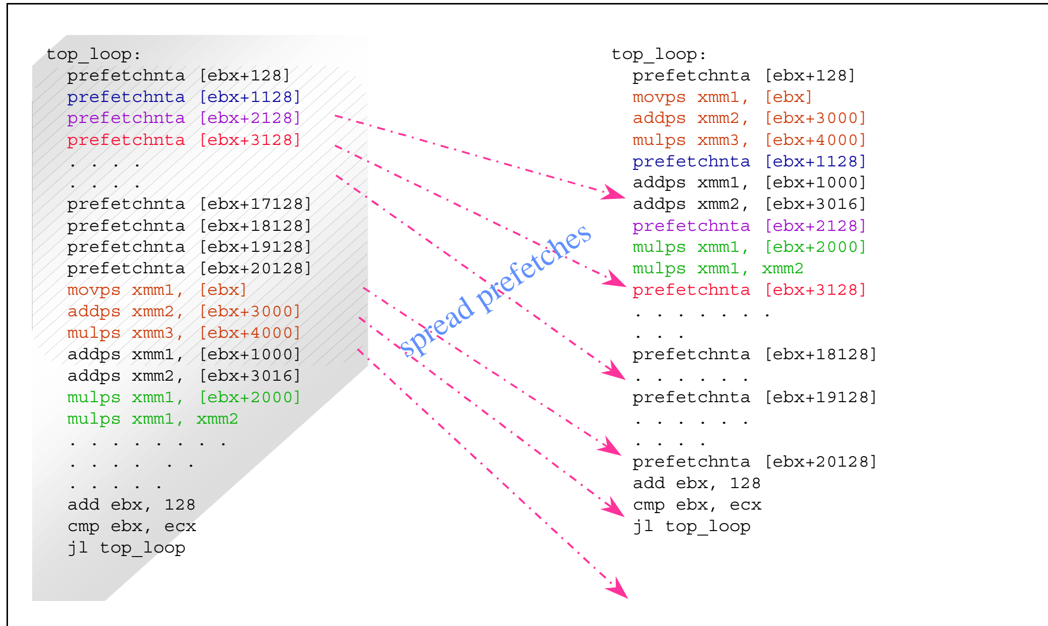


Figure 7-7. Spread Prefetch Instructions

NOTE

To avoid instruction execution stalls due to the over-utilization of the resource, PREFETCH instructions must be interspersed with computational instructions

7.6.10 Software Prefetch and Cache Blocking Techniques

Cache blocking techniques (such as strip-mining) are used to improve temporal locality and the cache hit rate. Strip-mining is one-dimensional temporal locality optimization for memory. When two-dimensional arrays are used in programs, loop blocking technique (similar to strip-mining but in two dimensions) can be applied for a better memory performance.

If an application uses a large data set that can be reused across multiple passes of a loop, it will benefit from strip mining. Data sets larger than the cache will be processed in groups small enough to fit into cache. This allows temporal data to reside in the cache longer, reducing bus traffic.

Data set size and temporal locality (data characteristics) fundamentally affect how PREFETCH instructions are applied to strip-mined code. Figure 7-8 shows two simplified scenarios for temporally-adjacent data and temporally-non-adjacent data.

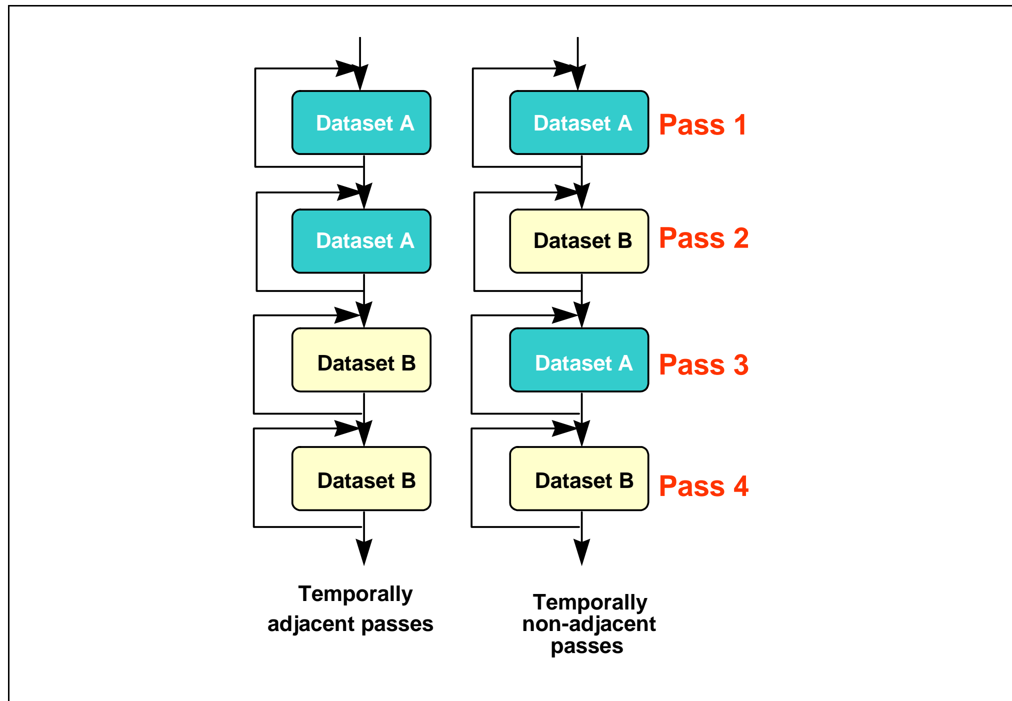


Figure 7-8. Cache Blocking – Temporally Adjacent and Non-adjacent Passes

In the temporally-adjacent scenario, subsequent passes use the same data and find it already in second-level cache. Prefetch issues aside, this is the preferred situation. In the temporally non-adjacent scenario, data used in pass *m* is displaced by pass *(m+1)*, requiring data *re*-fetch into the first level cache and perhaps the second level cache if a later pass reuses the data. If both data sets fit into the second-level cache, load operations in passes 3 and 4 become less expensive.

Figure 7-9 shows how prefetch instructions and strip-mining can be applied to increase performance in both of these scenarios.

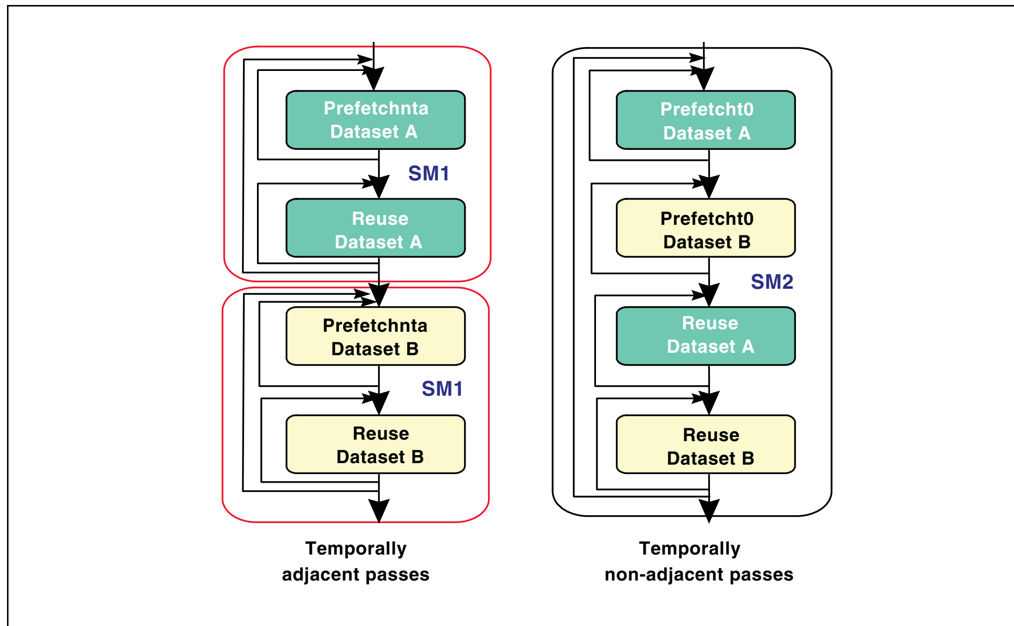


Figure 7-9. Examples of Prefetch and Strip-mining for Temporally Adjacent and Non-Adjacent Passes Loops

For Pentium 4 processors, the left scenario shows a graphical implementation of using PREFETCHNTA to prefetch data into selected ways of the second-level cache only (SM1 denotes strip mine one way of second-level), minimizing second-level cache pollution. Use PREFETCHNTA if the data is only touched once during the entire execution pass in order to minimize cache pollution in the higher level caches. This provides instant availability, assuming the prefetch was issued far ahead enough, when the read access is issued.

In scenario to the right (see Figure 7-9), keeping the data in one way of the second-level cache does not improve cache locality. Therefore, use PREFETCHT0 to prefetch the data. This amortizes the latency of the memory references in passes 1 and 2, and keeps a copy of the data in second-level cache, which reduces memory traffic and latencies for passes 3 and 4. To further reduce the latency, it might be worth considering extra PREFETCHNTA instructions prior to the memory references in passes 3 and 4.

In Example 7-7, consider the data access patterns of a 3D geometry engine first without strip-mining and then incorporating strip-mining. Note that 4-wide SIMD instructions of Pentium III processor can process 4 vertices per every iteration.

Without strip-mining, all the x,y,z coordinates for the four vertices must be re-fetched from memory in the second pass, that is, the lighting loop. This causes under-utilization of cache lines fetched during transformation loop as well as bandwidth wasted in the lighting loop.

Example 7-7. Data Access of a 3D Geometry Engine without Strip-mining

```
while (nvtx < MAX_NUM_VTX) {
    prefetchnta vertexi data           // v =[x,y,z,nx,ny,nz,tu,tv]
    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
TRANSFORMATION code                    // use only x,y,z,tu,tv of a vertex
    nvtx+=4
```

Example 7-7. Data Access of a 3D Geometry Engine without Strip-mining (Contd.)

```
}
while (nvtx < MAX_NUM_VTX) {
    prefetchnta vertexi data            // v =[x,y,z,nx,ny,nz,tu,tv]
                                        // x,y,z fetched again
    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
    compute the light vectors           // use only x,y,z
    LOCAL LIGHTING code                 // use only nx,ny,nz
    nvtx+=4
}
```

Now consider the code in Example 7-8 where strip-mining has been incorporated into the loops.

Example 7-8. Data Access of a 3D Geometry Engine with Strip-mining

```
while (nstrip < NUM_STRIP) {
/* Strip-mine the loop to fit data into one way of the second-level
  cache */
 while (nvtx < MAX_NUM_VTX_PER_STRIP) {
    prefetchnta vertex_i data              // v=[x,y,z,nx,ny,nz,tu,tv]
    prefetchnta vertex_i+1 data
    prefetchnta vertex_i+2 data
    prefetchnta vertex_i+3 data
    TRANSFORMATION code
         nvtx+=4
}
while (nvtx < MAX_NUM_VTX_PER_STRIP) {
   /* x y z coordinates are in the second-level cache, no prefetch is
       required */
compute the light vectors
POINT LIGHTING code
   nvtx+=4
 }
}
```

With strip-mining, all vertex data can be kept in the cache (for example, one way of second-level cache) during the strip-mined transformation loop and reused in the lighting loop. Keeping data in the cache reduces both bus traffic and the number of prefetches used.

Table 7-1 summarizes the steps of the basic usage model that incorporates only software prefetch with strip-mining. The steps are:

- Do strip-mining: partition loops so that the dataset fits into second-level cache.
- Use PREFETCHNTA if the data is only used once or the dataset fits into 32 KBytes (one way of second-level cache). Use PREFETCHT0 if the dataset exceeds 32 KBytes.

The above steps are platform-specific and provide an implementation example. The variables NUM_STRIP and MAX_NUM_VX_PER_STRIP can be heuristically determined for peak performance for specific application on a specific platform.

Table 7-1. Software Prefetching Considerations into Strip-mining Code

Read-Once Array References	Read-Multiple-Times Array References	
	Adjacent Passes	**Non-Adjacent Passes**
Prefetchnta	Prefetch0, SM1	Prefetch0, SM1 (2nd Level Pollution)
Evict one way; Minimize pollution	Pay memory access cost for the first pass of each array; Amortize the first pass with subsequent passes	Pay memory access cost for the first pass of every strip; Amortize the first pass with subsequent passes

7.6.11 Hardware Prefetching and Cache Blocking Techniques

Tuning data access patterns for the automatic hardware prefetch mechanism can minimize the memory access costs of the first-pass of the read-multiple-times and some of the read-once memory references. An example of the situations of read-once memory references can be illustrated with a matrix or image transpose, reading from a column-first orientation and writing to a row-first orientation, or vice versa.

Example 7-9 shows a nested loop of data movement that represents a typical matrix/image transpose problem. If the dimension of the array are large, not only the footprint of the dataset will exceed the last level cache but cache misses will occur at large strides. If the dimensions happen to be powers of 2, aliasing condition due to finite number of way-associativity (see "Capacity Limits and Aliasing in Caches" in Chapter) will exacerbate the likelihood of cache evictions.

Example 7-9. Using HW Prefetch to Improve Read-Once Memory Traffic

```
a)  Un-optimized image transpose
// dest and src represent two-dimensional arrays
for( i = 0;i < NUMCOLS; i ++) {
    // inner loop reads single column
    for( j = 0; j < NUMROWS ; j ++) {
        // Each read reference causes large-stride cache miss
        dest[i*NUMROWS +j] = src[j*NUMROWS + i];
    }
}
b)
// tilewidth = L2SizeInBytes/2/TileHeight/Sizeof(element)
for( i = 0; i < NUMCOLS; i += tilewidth) {
 for( j = 0; j < NUMROWS ; j ++) {
    // access multiple elements in the same row in the inner loop
    // access pattern friendly to hw prefetch and improves hit rate
    for( k = 0; k < tilewidth; k ++)
    dest[j+ (i+k)* NUMROWS] = src[i+k+ j* NUMROWS];
  }
}
```

Example 7-9 (b) shows applying the techniques of tiling with optimal selection of tile size and tile width to take advantage of hardware prefetch. With tiling, one can choose the size of two tiles to fit in the last level cache. Maximizing the width of each tile for memory read references enables the hardware prefetcher to initiate bus requests to read some cache lines before the code actually reference the linear addresses.

7.6.12 Single-pass versus Multi-pass Execution

An algorithm can use single- or multi-pass execution defined as follows:

- Single-pass, or unlayered execution passes a single data element through an entire computation pipeline.
- Multi-pass, or layered execution performs a single stage of the pipeline on a batch of data elements, before passing the batch on to the next stage.

A characteristic feature of both single-pass and multi-pass execution is that a specific trade-off exists depending on an algorithm's implementation and use of a single-pass or multiple-pass execution. See Figure 7-10.

Multi-pass execution is often easier to use when implementing a general purpose API, where the choice of code paths that can be taken depends on the specific combination of features selected by the application (for example, for 3D graphics, this might include the type of vertex primitives used and the number and type of light sources).

With such a broad range of permutations possible, a single-pass approach would be complicated, in terms of code size and validation. In such cases, each possible permutation would require a separate code sequence. For example, an object with features A, B, C, D can have a subset of features enabled, say, A, B, D. This stage would use one code path; another combination of enabled features would have a different code path. It makes more sense to perform each pipeline stage as a separate pass, with conditional clauses to select different features that are implemented within each stage. By using strip-mining, the number of vertices processed by each stage (for example, the batch size) can be selected to ensure that the batch stays within the processor caches through all passes. An intermediate cached buffer is used to pass the batch of vertices from one stage or pass to the next one.

Single-pass execution can be better suited to applications which limit the number of features that may be used at a given time. A single-pass approach can reduce the amount of data copying that can occur with a multi-pass engine. See Figure 7-10.

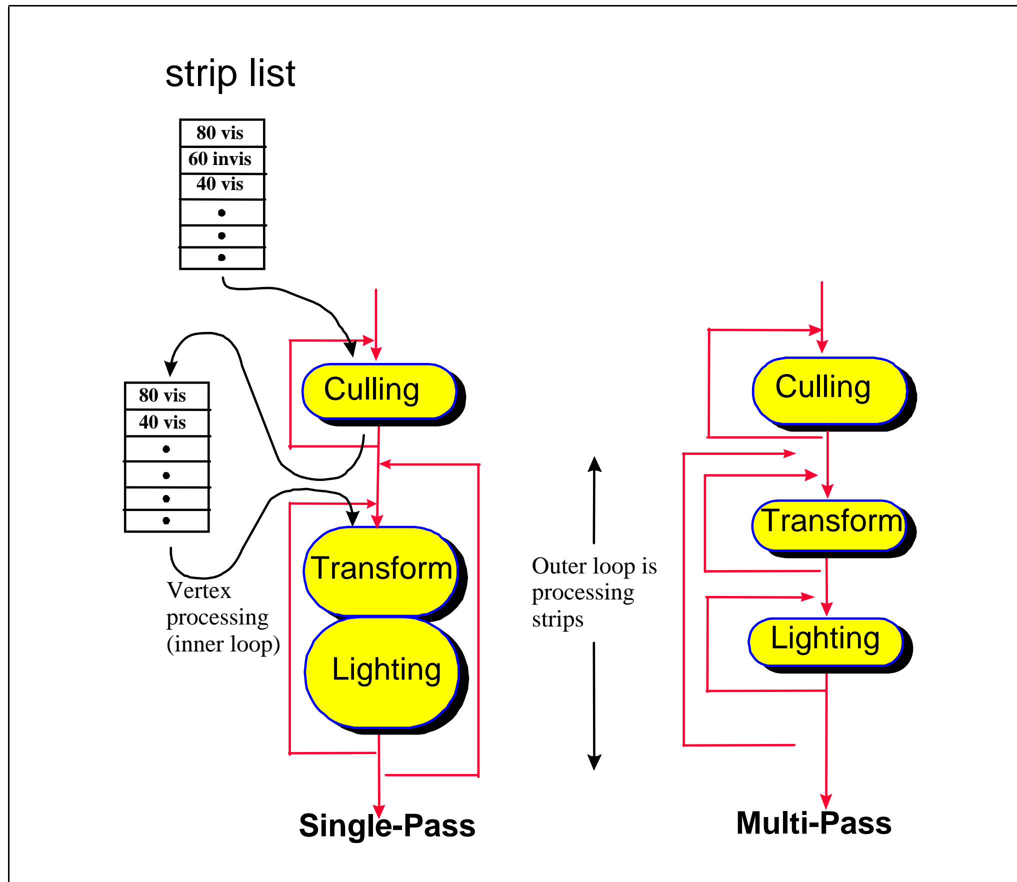


Figure 7-10. Single-Pass Vs. Multi-Pass 3D Geometry Engines

The choice of single-pass or multi-pass can have a number of performance implications. For instance, in a multi-pass pipeline, stages that are limited by bandwidth (either input or output) will reflect more of this performance limitation in overall execution time. In contrast, for a single-pass approach, bandwidth-limitations can be distributed/amortized across other computation-intensive stages. Also, the choice of which prefetch hints to use are also impacted by whether a single-pass or multi-pass approach is used.

7.7 MEMORY OPTIMIZATION USING NON-TEMPORAL STORES

Non-temporal stores can also be used to manage data retention in the cache. Uses for non-temporal stores include:

- To combine many writes without disturbing the cache hierarchy.
- To manage which data structures remain in the cache and which are transient.

Detailed implementations of these usage models are covered in the following sections.

7.7.1 Non-temporal Stores and Software Write-Combining

Use non-temporal stores in the cases when the data to be stored is:

- Write-once (non-temporal).
- Too large and thus cause cache thrashing.

Non-temporal stores do not invoke a cache line allocation, which means they are not write-allocate. As a result, caches are not polluted and no dirty writeback is generated to compete with useful data bandwidth. Without using non-temporal stores, bus bandwidth will suffer when caches start to be thrashed because of dirty writebacks.

In Streaming SIMD Extensions implementation, when non-temporal stores are written into writeback or write-combining memory regions, these stores are weakly-ordered and will be combined internally inside the processor's write-combining buffer and be written out to memory as a line burst transaction. To achieve the best possible performance, it is recommended to align data along the cache line boundary and write them consecutively in a cache line size while using non-temporal stores. If the consecutive writes are prohibitive due to programming constraints, then software write-combining (SWWC) buffers can be used to enable line burst transaction.

You can declare small SWWC buffers (a cache line for each buffer) in your application to enable explicit write-combining operations. Instead of writing to non-temporal memory space immediately, the program writes data into SWWC buffers and combines them inside these buffers. The program only writes a SWWC buffer out using non-temporal stores when the buffer is filled up, that is, a cache line (128 bytes for the Pentium 4 processor). Although the SWWC method requires explicit instructions for performing temporary writes and reads, this ensures that the transaction on the front-side bus causes line transaction rather than several partial transactions. Application performance gains considerably from implementing this technique. These SWWC buffers can be maintained in the second-level and re-used throughout the program.

7.7.2 Cache Management

Streaming instructions (PREFETCH and STORE) can be used to manage data and minimize disturbance of temporal data held within the processor's caches.

In addition, the Pentium 4 processor takes advantage of Intel C ++ Compiler support for C ++ language-level features for the Streaming SIMD Extensions. Streaming SIMD Extensions and MMX technology instructions provide intrinsics that allow you to optimize cache utilization. Examples of such Intel compiler intrinsics are _MM_PREFETCH, _MM_STREAM, _MM_LOAD, _MM_SFENCE. For detail, refer to the Intel C ++ Compiler User's Guide documentation.

The following examples of using prefetching instructions in the operation of video encoder and decoder as well as in simple 8-byte memory copy, illustrate performance gain from using the prefetching instructions for efficient cache management.

7.7.2.1 Video Encoder

In a video encoder, some of the data used during the encoding process is kept in the processor's second-level cache. This is done to minimize the number of reference streams that must be re-read from system memory. To ensure that other writes do not disturb the data in the second-level cache, streaming stores (MOVNTQ) are used to write around all processor caches.

The prefetching cache management implemented for the video encoder reduces the memory traffic. The second-level cache pollution reduction is ensured by preventing single-use video frame data from entering the second-level cache. Using a non-temporal PREFETCH (PREFETCHNTA) instruction brings data into only one way of the second-level cache, thus reducing pollution of the second-level cache.

If the data brought directly to second-level cache is not re-used, then there is a performance gain from the non-temporal prefetch over a temporal prefetch. The encoder uses non-temporal prefetches to avoid pollution of the second-level cache, increasing the number of second-level cache hits and decreasing the number of polluting write-backs to memory. The performance gain results from the more efficient use of the second-level cache, not only from the prefetch itself.

7.7.2.2 Video Decoder

In the video decoder example, completed frame data is written to local memory of the graphics card, which is mapped to WC (Write-combining) memory type. A copy of reference data is stored to the WB

memory at a later time by the processor in order to generate future data. The assumption is that the size of the reference data is too large to fit in the processor's caches. A streaming store is used to write the data around the cache, to avoid displaying other temporal data held in the caches. Later, the processor re-reads the data using PREFETCHNTA, which ensures maximum bandwidth, yet minimizes disturbance of other cached temporal data by using the non-temporal (NTA) version of prefetch.

7.7.2.3 Conclusions from Video Encoder and Decoder Implementation

These two examples indicate that by using an appropriate combination of non-temporal prefetches and non-temporal stores, an application can be designed to lessen the overhead of memory transactions by preventing second-level cache pollution, keeping useful data in the second-level cache and reducing costly write-back transactions. Even if an application does not gain performance significantly from having data ready from prefetches, it can improve from more efficient use of the second-level cache and memory. Such design reduces the encoder's demand for such critical resource as the memory bus. This makes the system more balanced, resulting in higher performance.

7.7.2.4 Optimizing Memory Copy Routines

Creating memory copy routines for large amounts of data is a common task in software optimization. Example 7-10 presents a basic algorithm for a the simple memory copy.

Example 7-10. Basic Algorithm of a Simple Memory Copy

```
#define N 512000
double a[N], b[N];
for (i = 0; i < N; i++) {
        b[i] = a[i];
}
```

This task can be optimized using various coding techniques. One technique uses software prefetch and streaming store instructions. It is discussed in the following paragraph and a code example shown in Example 7-11.

The memory copy algorithm can be optimized using the Streaming SIMD Extensions with these considerations:

- Alignment of data.
- Proper layout of pages in memory.
- Cache size.
- Interaction of the transaction lookaside buffer (TLB) with memory accesses.
- Combining prefetch and streaming-store instructions.

The guidelines discussed in this chapter come into play in this simple example. TLB priming is required for the Pentium 4 processor just as it is for the Pentium III processor, since software prefetch instructions will not initiate page table walks on either processor.

Example 7-11. A Memory Copy Routine Using Software Prefetch

```
#define PAGESIZE 4096;
#define NUMPERPAGE 512              // # of elements to fit a page

double a[N], b[N], temp;
for (kk=0; kk<N; kk+=NUMPERPAGE) {
    temp = a[kk+NUMPERPAGE];        // TLB priming
    // use block size = page size,
    // prefetch entire block, one cache line per loop
    for (j=kk+16; j<kk+NUMPERPAGE; j+=16) {
      _mm_prefetch((char*)&a[j], _MM_HINT_NTA);
  }
// copy 128 byte per loop
      for (j=kk; j<kk+NUMPERPAGE; j+=16) {
    _mm_stream_ps((float*)&b[j],
                         _mm_load_ps((float*)&a[j]));
            _mm_stream_ps((float*)&b[j+2],
                         _mm_load_ps((float*)&a[j+2]));
            _mm_stream_ps((float*)&b[j+4],
                         _mm_load_ps((float*)&a[j+4]));
            _mm_stream_ps((float*)&b[j+6],
                         _mm_load_ps((float*)&a[j+6]));
            _mm_stream_ps((float*)&b[j+8],
                         _mm_load_ps((float*)&a[j+8]));
            _mm_stream_ps((float*)&b[j+10],
                         _mm_load_ps((float*)&a[j+10]));
            _mm_stream_ps((float*)&b[j+12],
                         _mm_load_ps((float*)&a[j+12]));
            _mm_stream_ps((float*)&b[j+14],
                         _mm_load_ps((float*)&a[j+14]));
      }  // finished copying one block
}           // finished copying N elements
_mm_sfence();
```

7.7.2.5 TLB Priming

The TLB is a fast memory buffer that is used to improve performance of the translation of a virtual memory address to a physical memory address by providing fast access to page table entries. If memory pages are accessed and the page table entry is not resident in the TLB, a TLB miss results and the page table must be read from memory.

The TLB miss results in a performance degradation since another memory access must be performed (assuming that the translation is not already present in the processor caches) to update the TLB. The TLB can be preloaded with the page table entry for the next desired page by accessing (or touching) an address in that page. This is similar to prefetch, but instead of a data cache line the page table entry is being loaded in advance of its use. This helps to ensure that the page table entry is resident in the TLB and that the prefetch happens as requested subsequently.

7.7.2.6 Using the 8-byte Streaming Stores and Software Prefetch

Example 7-11 presents the copy algorithm that uses second level cache. The algorithm performs the following steps:

1. Uses blocking technique to transfer 8-byte data from memory into second-level cache using the _MM_PREFETCH intrinsic, 128 bytes at a time to fill a block. The size of a block should be less than one half of the size of the second-level cache, but large enough to amortize the cost of the loop.
2. Loads the data into an XMM register using the _MM_LOAD_PS intrinsic.
3. Transfers the 8-byte data to a different memory location via the _MM_STREAM intrinsics, bypassing the cache. For this operation, it is important to ensure that the page table entry prefetched for the memory is preloaded in the TLB.

In Example 7-11, eight _MM_LOAD_PS and _MM_STREAM_PS intrinsics are used so that all of the data prefetched (a 128-byte cache line) is written back. The prefetch and streaming-stores are executed in separate loops to minimize the number of transitions between reading and writing data. This significantly improves the bandwidth of the memory accesses.

The TEMP = A[KK+CACHESIZE] instruction is used to ensure the page table entry for array, and A is entered in the TLB prior to prefetching. This is essentially a prefetch itself, as a cache line is filled from that memory location with this instruction. Hence, the prefetching starts from KK+4 in this loop.

This example assumes that the destination of the copy is not temporally adjacent to the code. If the copied data is destined to be reused in the near future, then the streaming store instructions should be replaced with regular 128 bit stores (_MM_STORE_PS). This is required because the implementation of streaming stores on Pentium 4 processor writes data directly to memory, maintaining cache coherency.

7.7.2.7 Using 16-byte Streaming Stores and Hardware Prefetch

An alternate technique for optimizing a large region memory copy is to take advantage of hardware prefetcher, 16-byte streaming stores, and apply a segmented approach to separate bus read and write transactions. See Section 3.6.12, “Minimizing Bus Latency.”

The technique employs two stages. In the first stage, a block of data is read from memory to the cache sub-system. In the second stage, cached data are written to their destination using streaming stores.

Example 7-12. Memory Copy Using Hardware Prefetch and Bus Segmentation

```
void block_prefetch(void *dst,void *src)
{ _asm {
        mov edi,dst
        mov esi,src
        mov edx,SIZE
        align 16
    main_loop:
        xor ecx,ecx
        align 16
}

    prefetch_loop:
        movaps xmm0, [esi+ecx]
        movaps xmm0, [esi+ecx+64]
        add ecx,128
        cmp ecx,BLOCK_SIZE
        jne prefetch_loop
        xor ecx,ecx
        align 16
        cpy_loop:
```

Example 7-12. Memory Copy Using Hardware Prefetch and Bus Segmentation (Contd.)

```
        movdqa xmm0,[esi+ecx]
        movdqa xmm1,[esi+ecx+16]
        movdqa xmm2,[esi+ecx+32]
        movdqa xmm3,[esi+ecx+48]
        movdqa xmm4,[esi+ecx+64]
        movdqa xmm5,[esi+ecx+16+64]
        movdqa xmm6,[esi+ecx+32+64]
        movdqa xmm7,[esi+ecx+48+64]
        movntdq [edi+ecx],xmm0
        movntdq [edi+ecx+16],xmm1
        movntdq [edi+ecx+32],xmm2

        movntdq [edi+ecx+48],xmm3
        movntdq [edi+ecx+64],xmm4
        movntdq [edi+ecx+80],xmm5
        movntdq [edi+ecx+96],xmm6
        movntdq [edi+ecx+112],xmm7
        add ecx,128
        cmp ecx,BLOCK_SIZE
        jne cpy_loop

        add esi,ecx
        add edi,ecx
        sub edx,ecx
        jnz main_loop
        sfence
    }
}
```

7.7.2.8 Performance Comparisons of Memory Copy Routines

The throughput of a large-region, memory copy routine depends on several factors:

- Coding techniques that implements the memory copy task.
- Characteristics of the system bus (speed, peak bandwidth, overhead in read/write transaction protocols).
- Microarchitecture of the processor.

A comparison of the two coding techniques discussed above and two un-optimized techniques is shown in Table 7-2.

Table 7-2. Relative Performance of Memory Copy Routines

Processor, CPUID Signature and FSB Speed	Byte Sequential	DWORD Sequential	SW prefetch + 8 byte streaming store	4KB-Block HW prefetch + 16 byte streaming stores
Pentium M processor, 0x6Dn, 400	1.3X	1.2X	1.6X	2.5X
Intel Core Solo and Intel Core Duo processors, 0x6En, 667	3.3X	3.5X	2.1X	4.7X
Pentium D processor, 0xF4n, 800	3.4X	3.3X	4.9X	5.7X

The baseline for performance comparison is the throughput (bytes/sec) of 8-MByte region memory copy on a first-generation Pentium M processor (CPUID signature 0x69n) with a 400-MHz system bus using byte-sequential technique similar to that shown in Example 7-10. The degree of improvement relative to the performance baseline for some recent processors and platforms with higher system bus speed using different coding techniques are compared.

The second coding technique moves data at 4-Byte granularity using REP string instruction. The third column compares the performance of the coding technique listed in Example 7-11. The fourth column of performance compares the throughput of fetching 4-KBytes of data at a time (using hardware prefetch to aggregate bus read transactions) and writing to memory via 16-Byte streaming stores.

Increases in bus speed is the primary contributor to throughput improvements. The technique shown in Example 7-12 will likely take advantage of the faster bus speed in the platform more efficiently. Additionally, increasing the block size to multiples of 4-KBytes while keeping the total working set within the second-level cache can improve the throughput slightly.

The relative performance figure shown in Table 7-2 is representative of clean microarchitectural conditions within a processor (e.g. looping s simple sequence of code many times). The net benefit of integrating a specific memory copy routine into an application (full-featured applications tend to create many complicated micro-architectural conditions) will vary for each application.

7.7.3 Deterministic Cache Parameters

If CPUID supports the deterministic parameter leaf, software can use the leaf to query each level of the cache hierarchy. Enumeration of each cache level is by specifying an index value (starting form 0) in the ECX register (see "CPUID-CPU Identification" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

The list of parameters is shown in Table 7-3.

Table 7-3. Deterministic Cache Parameters Leaf

Bit Location	Name	Meaning
EAX[4:0]	Cache Type	0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved
EAX[7:5]	Cache Level	Starts at 1
EAX[8]	Self Initializing cache level	1: does not need SW initialization
EAX[9]	Fully Associative cache	1: Yes
EAX[13:10]	Reserved	
EAX[25:14]	Maximum number of logical processors sharing this cache	Plus encoding
EAX[31:26]	Maximum number of cores in a package	Plus 1 encoding
EBX[11:0]	System Coherency Line Size (L)	Plus 1 encoding (Bytes)
EBX[21:12]	Physical Line partitions (P)	Plus 1 encoding
EBX[31:22]	Ways of associativity (W)	Plus 1 encoding
ECX[31:0]	Number of Sets (S)	Plus 1 encoding
EDX	Reserved	
CPUID leaves > 3 < 80000000 are only visible when IA32_CR_MISC_ENABLES.BOOT_NT4 (bit 22) is clear (Default).		

The deterministic cache parameter leaf provides a means to implement software with a degree of forward compatibility with respect to enumerating cache parameters. Deterministic cache parameters can be used in several situations, including:

- Determine the size of a cache level.
- Adapt cache blocking parameters to different sharing topology of a cache-level across Hyper-Threading Technology, multicore and single-core processors.
- Determine multithreading resource topology in an MP system (See Chapter 8, "Multiple-Processor Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).
- Determine cache hierarchy topology in a platform using multicore processors (See topology enumeration white paper and reference code listed at the end of CHAPTER 1).
- Manage threads and processor affinities.
- Determine prefetch stride.

The size of a given level of cache is given by:

(# of Ways) * (Partitions) * (Line_size) * (Sets) = (EBX[31:22] + 1) * (EBX[21:12] + 1) * (EBX[11:0] + 1) * (ECX + 1)

7.7.3.1 Cache Sharing Using Deterministic Cache Parameters

Improving cache locality is an important part of software optimization. For example, a cache blocking algorithm can be designed to optimize block size at runtime for single-processor implementations and a variety of multiprocessor execution environments (including processors supporting HT Technology, or multicore processors).

The basic technique is to place an upper limit of the blocksize to be less than the size of the target cache level divided by the number of logical processors serviced by the target level of cache. This technique is applicable to multithreaded application programming. The technique can also benefit single-threaded applications that are part of a multi-tasking workloads.

7.7.3.2 Cache Sharing in Single-Core or Multicore

Deterministic cache parameters are useful for managing shared cache hierarchy in multithreaded applications for more sophisticated situations. A given cache level may be shared by logical processors in a processor or it may be implemented to be shared by logical processors in a physical processor package.

Using the deterministic cache parameter leaf and initial APIC_ID associated with each logical processor in the platform, software can extract information on the number and the topological relationship of logical processors sharing a cache level.

7.7.3.3 Determine Prefetch Stride

The prefetch stride (see description of CPUID.01H.EBX) provides the length of the region that the processor will prefetch with the PREFETCHh instructions (PREFETCHT0, PREFETCHT1, PREFETCHT2 and PREFETCHNTA). Software will use the length as the stride when prefetching into a particular level of the cache hierarchy as identified by the instruction used. The prefetch size is relevant for cache types of Data Cache (1) and Unified Cache (3); it should be ignored for other cache types. Software should not assume that the coherency line size is the prefetch stride.

If the prefetch stride field is zero, then software should assume a default size of 64 bytes is the prefetch stride. Software should use the following algorithm to determine what prefetch size to use depending on whether the deterministic cache parameter mechanism is supported or the legacy mechanism:

- If a processor supports the deterministic cache parameters and provides a non-zero prefetch size, then that prefetch size is used.
- If a processor supports the deterministic cache parameters and does not provides a prefetch size then default size for each level of the cache hierarchy is 64 bytes.

- If a processor does not support the deterministic cache parameters but provides a legacy prefetch size descriptor (0xF0 - 64 byte, 0xF1 - 128 byte) will be the prefetch size for all levels of the cache hierarchy.
- If a processor does not support the deterministic cache parameters and does not provide a legacy prefetch size descriptor, then 32-bytes is the default size for all levels of the cache hierarchy.

CHAPTER 8
MULTICORE AND HYPER-THREADING TECHNOLOGY

This chapter describes software optimization techniques for multithreaded applications running in an environment using either multiprocessor (MP) systems or processors with hardware-based multithreading support. Multiprocessor systems are systems with two or more sockets, each mated with a physical processor package. Intel 64 and IA-32 processors that provide hardware multithreading support include dual-core processors, quad-core processors and processors supporting HT Technology[1].

Computational throughput in a multithreading environment can increase as more hardware resources are added to take advantage of thread-level or task-level parallelism. Hardware resources can be added in the form of more than one physical-processor, processor-core-per-package, and/or logical-processor-per-core. Therefore, there are some aspects of multithreading optimization that apply across MP, multicore, and HT Technology. There are also some specific microarchitectural resources that may be implemented differently in different hardware multithreading configurations (for example: execution resources are not shared across different cores but shared by two logical processors in the same core if HT Technology is enabled). This chapter covers guidelines that apply to these situations.

This chapter covers:

- Performance characteristics and usage models.
- Programming models for multithreaded applications.
- Software optimization techniques in five specific areas.

8.1 PERFORMANCE AND USAGE MODELS

The performance gains of using multiple processors, multicore processors or HT Technology are greatly affected by the usage model and the amount of parallelism in the control flow of the workload. Two common usage models are:

- Multithreaded applications.
- Multitasking using single-threaded applications.

8.1.1 Multithreading

When an application employs multithreading to exploit task-level parallelism in a workload, the control flow of the multi-threaded software can be divided into two parts: parallel tasks and sequential tasks.

Amdahl's law describes an application's performance gain as it relates to the degree of parallelism in the control flow. It is a useful guide for selecting the code modules, functions, or instruction sequences that are most likely to realize the most gains from transforming sequential tasks and control flows into parallel code to take advantage multithreading hardware support.

Figure 8-1 illustrates how performance gains can be realized for any workload according to Amdahl's law. The bar in Figure 8-1 represents an individual task unit or the collective workload of an entire application.

1. The presence of hardware multithreading support in Intel 64 and IA-32 processors can be detected by checking the feature flag CPUID .01H:EDX[28]. A return value of in bit 28 indicates that at least one form of hardware multithreading is present in the physical processor package. The number of logical processors present in each package can also be obtained from CPUID. The application must check how many logical processors are enabled and made available to application at runtime by making the appropriate operating system calls. See the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* for information.

In general, the speed-up of running multiple threads on an MP systems with *N* physical processors, over single-threaded execution, can be expressed as:

$$\text{RelativeResponse} = \frac{\text{Tsequential}}{\text{Tparallel}} = \left(1 - P + \frac{P}{N} + O\right)$$

where *P* is the fraction of workload that can be parallelized, and *O* represents the overhead of multi-threading and may vary between different operating systems. In this case, performance gain is the inverse of the relative response.

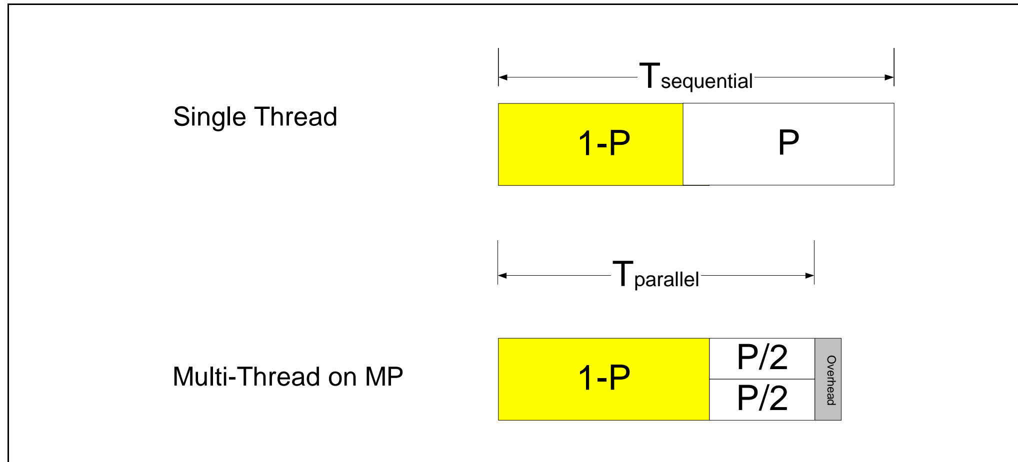


Figure 8-1. Amdahl's Law and MP Speed-up

When optimizing application performance in a multithreaded environment, control flow parallelism is likely to have the largest impact on performance scaling with respect to the number of physical processors and to the number of logical processors per physical processor.

If the control flow of a multi-threaded application contains a workload in which only 50% can be executed in parallel, the maximum performance gain using two physical processors is only 33%, compared to using a single processor. Using four processors can deliver no more than a 60% speed-up over a single processor. Thus, it is critical to maximize the portion of control flow that can take advantage of parallelism. Improper implementation of thread synchronization can significantly increase the proportion of serial control flow and further reduce the application's performance scaling.

In addition to maximizing the parallelism of control flows, interaction between threads in the form of thread synchronization and imbalance of task scheduling can also impact overall processor scaling significantly.

Excessive cache misses are one cause of poor performance scaling. In a multithreaded execution environment, they can occur from:

- Aliased stack accesses by different threads in the same process.
- Thread contentions resulting in cache line evictions.
- False-sharing of cache lines between different processors.

Techniques that address each of these situations (and many other areas) are described in sections in this chapter.

8.1.2 Multitasking Environment

Hardware multithreading capabilities in Intel 64 and IA-32 processors can exploit task-level parallelism when a workload consists of several single-threaded applications and these applications are scheduled to run concurrently under an MP-aware operating system. In this environment, hardware multithreading capabilities can deliver higher throughput for the workload, although the relative performance of a single

task (in terms of time of completion relative to the same task when in a single-threaded environment) will vary, depending on how much shared execution resources and memory are utilized.

For development purposes, several popular operating systems (for example Microsoft Windows* XP Professional and Home, Linux* distributions using kernel 2.4.19 or later[2]) include OS kernel code that can manage the task scheduling and the balancing of shared execution resources within each physical processor to maximize the throughput.

Because applications run independently under a multitasking environment, thread synchronization issues are less likely to limit the scaling of throughput. This is because the control flow of the workload is likely to be 100% parallel[3] (if no inter-processor communication is taking place and if there are no system bus constraints).

With a multitasking workload, however, bus activities and cache access patterns are likely to affect the scaling of the throughput. Running two copies of the same application or same suite of applications in a lock-step can expose an artifact in performance measuring methodology. This is because an access pattern to the first level data cache can lead to excessive cache misses and produce skewed performance results. Fix this problem by:

- Including a per-instance offset at the start-up of an application.
- Introducing heterogeneity in the workload by using different datasets with each instance of the application.
- Randomizing the sequence of start-up of applications when running multiple copies of the same suite.

When two applications are employed as part of a multitasking workload, there is little synchronization overhead between these two processes. It is also important to ensure each application has minimal synchronization overhead within itself.

An application that uses lengthy spin loops for intra-process synchronization is less likely to benefit from HT Technology in a multitasking workload. This is because critical resources will be consumed by the long spin loops.

8.2 PROGRAMMING MODELS AND MULTITHREADING

Parallelism is the most important concept in designing a multithreaded application and realizing optimal performance scaling with multiple processors. An optimized multithreaded application is characterized by large degrees of parallelism or minimal dependencies in the following areas:

- Workload.
- Thread interaction.
- Hardware utilization.

The key to maximizing workload parallelism is to identify multiple tasks that have minimal inter-dependencies within an application and to create separate threads for parallel execution of those tasks.

Concurrent execution of independent threads is the essence of deploying a multithreaded application on a multiprocessing system. Managing the interaction between threads to minimize the cost of thread synchronization is also critical to achieving optimal performance scaling with multiple processors.

Efficient use of hardware resources between concurrent threads requires optimization techniques in specific areas to prevent contentions of hardware resources. Coding techniques for optimizing thread synchronization and managing other hardware resources are discussed in subsequent sections.

Parallel programming models are discussed next.

---

2. This code is included in Red Hat* Linux Enterprise AS 2.1.
3. A software tool that attempts to measure the throughput of a multitasking workload is likely to introduce control flows that are not parallel. Thread synchronization issues must be considered as an integral part of its performance measuring methodology.

8.2.1 Parallel Programming Models

Two common programming models for transforming independent task requirements into application threads are:

- Domain decomposition.
- Functional decomposition.

8.2.1.1 Domain Decomposition

Usually large compute-intensive tasks use data sets that can be divided into a number of small subsets, each having a large degree of computational independence. Examples include:

- Computation of a discrete cosine transformation (DCT) on two-dimensional data by dividing the two-dimensional data into several subsets and creating threads to compute the transform on each subset.
- Matrix multiplication; here, threads can be created to handle the multiplication of half of matrix with the multiplier matrix.

Domain Decomposition is a programming model based on creating identical or similar threads to process smaller pieces of data independently. This model can take advantage of duplicated execution resources present in a traditional multiprocessor system. It can also take advantage of shared execution resources between two logical processors in HT Technology. This is because a data domain thread typically consumes only a fraction of the available on-chip execution resources.

Section 8.3.4, "Key Practices of Execution Resource Optimization," discusses additional guidelines that can help data domain threads use shared execution resources cooperatively and avoid the pitfalls creating contentions of hardware resources between two threads.

8.2.2 Functional Decomposition

Applications usually process a wide variety of tasks with diverse functions and many unrelated data sets. For example, a video codec needs several different processing functions. These include DCT, motion estimation and color conversion. Using a functional threading model, applications can program separate threads to do motion estimation, color conversion, and other functional tasks.

Functional decomposition will achieve more flexible thread-level parallelism if it is less dependent on the duplication of hardware resources. For example, a thread executing a sorting algorithm and a thread executing a matrix multiplication routine are not likely to require the same execution unit at the same time. A design recognizing this could advantage of traditional multiprocessor systems as well as multiprocessor systems using processors supporting HT Technology.

8.2.3 Specialized Programming Models

Intel Core Duo processor and processors based on Intel Core microarchitecture offer a second-level cache shared by two processor cores in the same physical package. This provides opportunities for two application threads to access some application data while minimizing the overhead of bus traffic.

Multi-threaded applications may need to employ specialized programming models to take advantage of this type of hardware feature. One such scenario is referred to as producer-consumer. In this scenario, one thread writes data into some destination (hopefully in the second-level cache) and another thread executing on the other core in the same physical package subsequently reads data produced by the first thread.

The basic approach for implementing a producer-consumer model is to create two threads; one thread is the producer and the other is the consumer. Typically, the producer and consumer take turns to work on a buffer and inform each other when they are ready to exchange buffers. In a producer-consumer model, there is some thread synchronization overhead when buffers are exchanged between the producer and consumer. To achieve optimal scaling with the number of cores, the synchronization overhead must be kept low. This can be done by ensuring the producer and consumer threads have comparable time constants for completing each incremental task prior to exchanging buffers.

Example 8-1 illustrates the coding structure of single-threaded execution of a sequence of task units, where each task unit (either the producer or consumer) executes serially (shown in Figure 8-2). In the equivalent scenario under multi-threaded execution, each producer-consumer pair is wrapped as a thread function and two threads can be scheduled on available processor resources simultaneously.

Example 8-1. Serial Execution of Producer and Consumer Work Items

```
for (i = 0; i < number_of_iterations; i++) {
    producer (i, buff);  // pass buffer index and buffer address
    consumer (i, buff);
}(
```

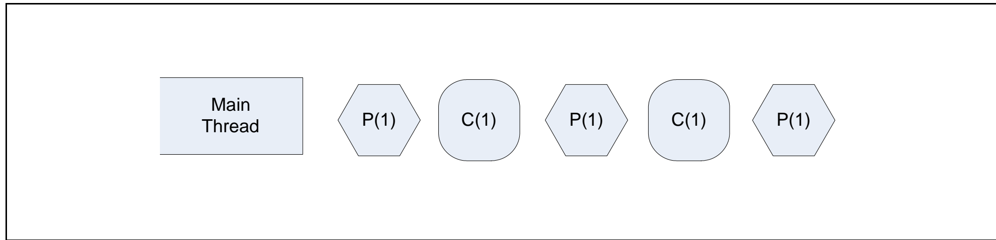


Figure 8-2. Single-threaded Execution of Producer-consumer Threading Model

8.2.3.1 Producer-Consumer Threading Models

Figure 8-3 illustrates the basic scheme of interaction between a pair of producer and consumer threads. The horizontal direction represents time. Each block represents a task unit, processing the buffer assigned to a thread.

The gap between each task represents synchronization overhead. The decimal number in the parenthesis represents a buffer index. On an Intel Core Duo processor, the producer thread can store data in the second-level cache to allow the consumer thread to continue work requiring minimal bus traffic.

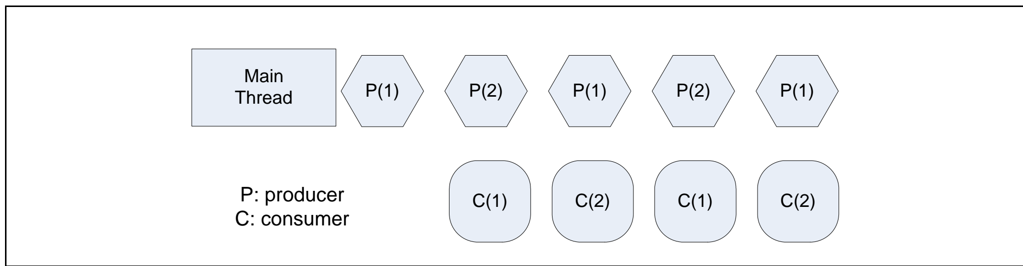


Figure 8-3. Execution of Producer-consumer Threading Model on a Multicore Processor

The basic structure to implement the producer and consumer thread functions with synchronization to communicate buffer index is shown in Example 8-2.

Example 8-2. Basic Structure of Implementing Producer Consumer Threads

```
(a) Basic structure of a producer thread function
void producer_thread()
{       int iter_num = workamount - 1; // make local copy
        int mode1 = 1; // track usage of two buffers via 0 and 1
        produce(buffs[0],count);  // placeholder function
        while (iter_num--) {

                Signal(&signal1,1);  // tell the other thread to commence
                produce(buffs[mode1],count); // placeholder function
                WaitForSignal(&end1);
                mode1 = 1 - mode1;  // switch to the other buffer
        }

}
b) Basic structure of a consumer thread
void consumer_thread()
{       int mode2 = 0;  // first iteration start with buffer 0, than alternate
        int iter_num = workamount - 1;
        while (iter_num--) {

                WaitForSignal(&signal1);
                consume(buffs[mode2],count); // placeholder function
                Signal(&end1,1);
                mode2 = 1 - mode2;
        }
        consume(buffs[mode2],count);
}
```

It is possible to structure the producer-consumer model in an interlaced manner such that it can minimize bus traffic and be effective on multicore processors without shared second-level cache.

In this interlaced variation of the producer-consumer model, each scheduling quanta of an application thread comprises of a producer task and a consumer task. Two identical threads are created to execute in parallel. During each scheduling quanta of a thread, the producer task starts first and the consumer task follows after the completion of the producer task; both tasks work on the same buffer. As each task completes, one thread signals to the other thread notifying its corresponding task to use its designated buffer. Thus, the producer and consumer tasks execute in parallel in two threads. As long as the data generated by the producer reside in either the first or second level cache of the same core, the consumer can access them without incurring bus traffic. The scheduling of the interlaced producer-consumer model is shown in Figure 8-4.

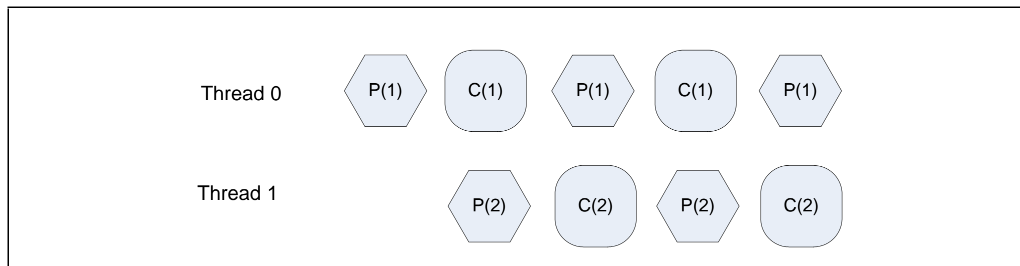


Figure 8-4. Interlaced Variation of the Producer Consumer Model

Example 8-3 shows the basic structure of a thread function that can be used in this interlaced producer-consumer model.

Example 8-3. Thread Function for an Interlaced Producer Consumer Model

```
// master thread starts first iteration, other thread must wait
// one iteration
void producer_consumer_thread(int master)
{
int mode = 1 - master; // track which thread and its designated
                                    // buffer index
unsigned int iter_num = workamount >> 1;
unsigned int i=0;

iter_num += master & workamount & 1;

    if (master)  // master thread starts the first iteration
    {
          produce(buffs[mode],count);
          Signal(sigp[1-mode1],1);  // notify producer task in follower
                                               // thread that it can proceed
     consume(buffs[mode],count);
          Signal(sigc[1-mode],1);
     i = 1;
    }


    for (; i < iter_num; i++)
    {
     WaitForSignal(sigp[mode]);
          produce(buffs[mode],count); // notify the producer task in
                                                    // other thread
          Signal(sigp[1-mode],1);

    WaitForSignal(sigc[mode]);
     consume(buffs[mode],count);
          Signal(sigc[1-mode],1);
    }
}
```

8.2.4 Tools for Creating Multithreaded Applications

Programming directly to a multithreading application programming interface (API) is not the only method for creating multithreaded applications. New tools (such as the Intel compiler) have become available with capabilities that make the challenge of creating multithreaded application easier.

Features available in the latest Intel compilers are:

- Generating multithreaded code using OpenMP* directives[4].
- Generating multithreaded code automatically from unmodified high-level code[5].

---

4. Intel Compiler 5.0 and later supports OpenMP directives. Visit http://developer.intel.com/software/products for details.

8.2.4.1 Programming with OpenMP Directives

OpenMP provides a standardized, non-proprietary, portable set of Fortran and C++ compiler directives supporting shared memory parallelism in applications. OpenMP supports directive-based processing. This uses special preprocessors or modified compilers to interpret parallelism expressed in Fortran comments or C/C++ pragmas. Benefits of directive-based processing include:

- The original source can be compiled unmodified.
- It is possible to make incremental code changes. This preserves algorithms in the original code and enables rapid debugging.
- Incremental code changes help programmers maintain serial consistency. When the code is run on one processor, it gives the same result as the unmodified source code.
- Offering directives to fine tune thread scheduling imbalance.
- Intel's implementation of OpenMP runtime can add minimal threading overhead relative to hand-coded multithreading.

8.2.4.2 Automatic Parallelization of Code

While OpenMP directives allow programmers to quickly transform serial applications into parallel applications, programmers must identify specific portions of the application code that contain parallelism and add compiler directives. Intel Compiler 6.0 supports a new (-QPARALLEL) option, which can identify loop structures that contain parallelism. During program compilation, the compiler automatically attempts to decompose the parallelism into threads for parallel processing. No other intervention or programmer is needed.

8.2.4.3 Supporting Development Tools

See Appendix A, "Application Performance Tools" for information on the various tools that Intel provides for software development.

8.3 OPTIMIZATION GUIDELINES

This section summarizes optimization guidelines for tuning multithreaded applications. Five areas are listed (in order of importance):

- Thread synchronization.
- Bus utilization.
- Memory optimization.
- Front end optimization.
- Execution resource optimization.

Practices associated with each area are listed in this section. Guidelines for each area are discussed in greater depth in sections that follow.

Most of the coding recommendations improve performance scaling with processor cores; and scaling due to HT Technology. Techniques that apply to only one environment are noted.

8.3.1 Key Practices of Thread Synchronization

Key practices for minimizing the cost of thread synchronization are summarized below:

- Insert the PAUSE instruction in fast spin loops and keep the number of loop repetitions to a minimum to improve overall system performance.

---

5. Intel Compiler 6.0 supports auto-parallelization.

- Replace a spin-lock that may be acquired by multiple threads with pipelined locks such that no more than two threads have write accesses to one lock. If only one thread needs to write to a variable shared by two threads, there is no need to acquire a lock.
- Use a thread-blocking API in a long idle loop to free up the processor.
- Prevent “false-sharing” of per-thread-data between two threads.
- Place each synchronization variable alone, separated by 128 bytes or in a separate cache line.

See Section 8.4, “Thread Synchronization,” for details.

8.3.2 Key Practices of System Bus Optimization

Managing bus traffic can significantly impact the overall performance of multithreaded software and MP systems. Key practices of system bus optimization for achieving high data throughput and quick response are:

- Improve data and code locality to conserve bus command bandwidth.
- Avoid excessive use of software prefetch instructions and allow the automatic hardware prefetcher to work. Excessive use of software prefetches can significantly and unnecessarily increase bus utilization if used inappropriately.
- Consider using overlapping multiple back-to-back memory reads to improve effective cache miss latencies.
- Use full write transactions to achieve higher data throughput.

See Section 8.5, “System Bus Optimization,” for details.

8.3.3 Key Practices of Memory Optimization

Key practices for optimizing memory operations are summarized below:

- Use cache blocking to improve locality of data access. Target one quarter to one half of cache size when targeting processors supporting HT Technology.
- Minimize the sharing of data between threads that execute on different physical processors sharing a common bus.
- Minimize data access patterns that are offset by multiples of 64-KBytes in each thread.
- Adjust the private stack of each thread in an application so the spacing between these stacks is not offset by multiples of 64 KBytes or 1 MByte (prevents unnecessary cache line evictions) when targeting processors supporting HT Technology.
- Add a per-instance stack offset when two instances of the same application are executing in lock steps to avoid memory accesses that are offset by multiples of 64 KByte or 1 MByte when targeting processors supporting HT Technology.

See Section 8.6, “Memory Optimization,” for details.

8.3.4 Key Practices of Execution Resource Optimization

Each physical processor has dedicated execution resources. Logical processors in physical processors supporting HT Technology share specific on-chip execution resources. Key practices for execution resource optimization include:

- Optimize each thread to achieve optimal frequency scaling first.
- Optimize multithreaded applications to achieve optimal scaling with respect to the number of physical processors.
- Use on-chip execution resources cooperatively if two threads are sharing the execution resources in the same physical processor package.

- For each processor supporting HT Technology, consider adding functionally uncorrelated threads to increase the hardware resource utilization of each physical processor package.

See Section 8.8, “Affinities and Managing Shared Platform Resources,” for details.

8.3.5 Generality and Performance Impact

The next five sections cover the optimization techniques in detail. Recommendations discussed in each section are ranked by importance in terms of estimated local impact and generality.

Rankings are subjective and approximate. They can vary depending on coding style, application and threading domain. The purpose of including high, medium and low impact ranking with each recommendation is to provide a relative indicator as to the degree of performance gain that can be expected when a recommendation is implemented.

It is not possible to predict the likelihood of a code instance across many applications, so an impact ranking cannot be directly correlated to application-level performance gain. The ranking on generality is also subjective and approximate.

Coding recommendations that do not impact all three scaling factors are typically categorized as medium or lower.

8.4 THREAD SYNCHRONIZATION

Applications with multiple threads use synchronization techniques in order to ensure correct operation. However, thread synchronization that are improperly implemented can significantly reduce performance.

The best practice to reduce the overhead of thread synchronization is to start by reducing the application’s requirements for synchronization. Intel Thread Profiler can be used to profile the execution timeline of each thread and detect situations where performance is impacted by frequent occurrences of synchronization overhead.

Several coding techniques and operating system (OS) calls are frequently used for thread synchronization. These include spin-wait loops, spin-locks, critical sections, to name a few. Choosing the optimal OS call for the circumstance and implementing synchronization code with parallelism in mind are critical in minimizing the cost of handling thread synchronization.

SSE3 provides two instructions (MONITOR/MWAIT) to help multithreaded software improve synchronization between multiple agents. In the first implementation of MONITOR and MWAIT, these instructions are available to operating system so that operating system can optimize thread synchronization in different areas. For example, an operating system can use MONITOR and MWAIT in its system idle loop (known as C0 loop) to reduce power consumption. An operating system can also use MONITOR and MWAIT to implement its C1 loop to improve the responsiveness of the C1 loop. See Chapter 8 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

8.4.1 Choice of Synchronization Primitives

Thread synchronization often involves modifying some shared data while protecting the operation using synchronization primitives. There are many primitives to choose from. Guidelines that are useful when selecting synchronization primitives are:

- Favor compiler intrinsics or an OS provided interlocked API for atomic updates of simple data operation, such as increment and compare/exchange. This will be more efficient than other more complicated synchronization primitives with higher overhead.

  For more information on using different synchronization primitives, see the white paper *Developing Multi-threaded Applications: A Platform Consistent Approach*. See http://www3.intel.com/cd/ids/developer/asmo-na/eng/53797.htm.
- When choosing between different primitives to implement a synchronization construct, using Intel Thread Checker and Thread Profiler can be very useful in dealing with multithreading functional

correctness issue and performance impact under multi-threaded execution. Additional information on the capabilities of Intel Thread Checker and Thread Profiler are described in Appendix A.

Table 8-1 is useful for comparing the properties of three categories of synchronization objects available to multi-threaded applications.

Table 8-1. Properties of Synchronization Objects

Characteristics	Operating System Synchronization Objects	Light Weight User Synchronization	Synchronization Object based on MONITOR/MWAIT
Cycles to acquire and release (if there is a contention)	Thousands or Tens of thousands cycles	Hundreds of cycles	Hundreds of cycles
Power consumption	Saves power by halting the core or logical processor if idle	Some power saving if using PAUSE	Saves more power than PAUSE
Scheduling and context switching	Returns to the OS scheduler if contention exists (can be tuned with earlier spin loop count)	Does not return to OS scheduler voluntarily	Does not return to OS scheduler voluntarily
Ring level	Ring 0	Ring 3	Ring 0
Miscellaneous	Some objects provide intra-process synchronization and some are for inter-process communication	Must lock accesses to synchronization variable if several threads may write to it simultaneously. Otherwise can write without locks.	Same as light weight. Can be used only on systems supporting MONITOR/MWAIT
Recommended use conditions	▪ Number of active threads is greater than number of cores ▪ Waiting thousands of cycles for a signal ▪ Synchronization among processes	▪ Number of active threads is less than or equal to number of cores ▪ Infrequent contention ▪ Need inter process synchronization	▪ Same as light weight objects ▪ MONITOR/MWAIT available

8.4.2 Synchronization for Short Periods

The frequency and duration that a thread needs to synchronize with other threads depends application characteristics. When a synchronization loop needs very fast response, applications may use a spin-wait loop.

A spin-wait loop is typically used when one thread needs to wait a short amount of time for another thread to reach a point of synchronization. A spin-wait loop consists of a loop that compares a synchronization variable with some pre-defined value. See Example 8-4(a).

On a modern microprocessor with a superscalar speculative execution engine, a loop like this results in the issue of multiple simultaneous read requests from the spinning thread. These requests usually execute out-of-order with each read request being allocated a buffer resource. On detection of a write by a worker thread to a load that is in progress, the processor must guarantee no violations of memory order occur. The necessity of maintaining the order of outstanding memory operations inevitably costs the processor a severe penalty that impacts all threads.

This penalty occurs on the Pentium M processor, the Intel Core Solo and Intel Core Duo processors. However, the penalty on these processors is small compared with penalties suffered on the Pentium 4 and Intel Xeon processors. There the performance penalty for exiting the loop is about 25 times more severe.

On a processor supporting HT Technology, spin-wait loops can consume a significant portion of the execution bandwidth of the processor. One logical processor executing a spin-wait loop can severely impact the performance of the other logical processor.

Example 8-4. Spin-wait Loop and PAUSE Instructions

(a) An un-optimized spin-wait loop experiences performance penalty when exiting the loop. It consumes execution resources without contributing computational work.

```
do {
    // This loop can run faster than the speed of memory access,
    // other worker threads cannot finish modifying sync_var until
    // outstanding loads from the spinning loops are resolved.
} while( sync_var != constant_value);
```

(b) Inserting the PAUSE instruction in a fast spin-wait loop prevents performance-penalty to the spinning thread and the worker thread

```
do {
_asm   pause
    // Ensure this loop is de-pipelined, i.e. preventing more than one
    // load request to sync_var to be outstanding,
    // avoiding performance penalty when the worker thread updates
    // sync_var and the spinning thread exiting the loop.

}
while( sync_var != constant_value);
```

(c) A spin-wait loop using a "test, test-and-set" technique to determine the availability of the synchronization variable. This technique is recommended when writing spin-wait loops to run on Intel 64 and IA-32 architecture processors.

```
Spin_Lock:
    CMP lockvar, 0 ;            // Check if lock is free.
    JE  Get_lock
        PAUSE;                  // Short delay.
        JMP Spin_Lock;
Get_Lock:
        MOV EAX, 1;
        XCHG EAX, lockvar;      // Try to get lock.
        CMP EAX, 0;             // Test if successful.
        JNE Spin_Lock;
Critical_Section:
        <critical section code>
        MOV lockvar, 0;          // Release lock.
```

User/Source Coding Rule 18. (M impact, H generality) *Insert the PAUSE instruction in fast spin loops and keep the number of loop repetitions to a minimum to improve overall system performance.*

On processors that use the Intel NetBurst microarchitecture core, the penalty of exiting from a spin-wait loop can be avoided by inserting a PAUSE instruction in the loop. In spite of the name, the PAUSE instruction improves performance by introducing a slight delay in the loop and effectively causing the memory read requests to be issued at a rate that allows immediate detection of any store to the synchronization variable. This prevents the occurrence of a long delay due to memory order violation.

One example of inserting the PAUSE instruction in a simplified spin-wait loop is shown in Example 8-4(b). The PAUSE instruction is compatible with all Intel 64 and IA-32 processors. On IA-32 processors prior to Intel NetBurst microarchitecture, the PAUSE instruction is essentially a NOP instruction. Additional examples of optimizing spin-wait loops using the PAUSE instruction are available in Application note AP-949, "Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor." See http://www3.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/knowledgebase/19083.htm.

Inserting the PAUSE instruction has the added benefit of significantly reducing the power consumed during the spin-wait because fewer system resources are used.

8.4.3 Optimization with Spin-Locks

Spin-locks are typically used when several threads needs to modify a synchronization variable and the synchronization variable must be protected by a lock to prevent un-intentional overwrites. When the lock is released, however, several threads may compete to acquire it at once. Such thread contention significantly reduces performance scaling with respect to frequency, number of discrete processors, and HT Technology.

To reduce the performance penalty, one approach is to reduce the likelihood of many threads competing to acquire the same lock. Apply a software pipelining technique to handle data that must be shared between multiple threads.

Instead of allowing multiple threads to compete for a given lock, no more than two threads should have write access to a given lock. If an application must use spin-locks, include the PAUSE instruction in the wait loop. Example 8-4(c) shows an example of the “test, test-and-set” technique for determining the availability of the lock in a spin-wait loop.

User/Source Coding Rule 19. (M impact, L generality) *Replace a spin lock that may be acquired by multiple threads with pipelined locks such that no more than two threads have write accesses to one lock. If only one thread needs to write to a variable shared by two threads, there is no need to use a lock.*

8.4.4 Synchronization for Longer Periods

When using a spin-wait loop not expected to be released quickly, an application should follow these guidelines:

- Keep the duration of the spin-wait loop to a minimum number of repetitions.
- Applications should use an OS service to block the waiting thread; this can release the processor so that other runnable threads can make use of the processor or available execution resources.

On processors supporting HT Technology, operating systems should use the HLT instruction if one logical processor is active and the other is not. HLT will allow an idle logical processor to transition to a halted state; this allows the active logical processor to use all the hardware resources in the physical package. An operating system that does not use this technique must still execute instructions on the idle logical processor that repeatedly check for work. This “idle loop” consumes execution resources that could otherwise be used to make progress on the other active logical processor.

If an application thread must remain idle for a long time, the application should use a thread blocking API or other method to release the idle processor. The techniques discussed here apply to traditional MP system, but they have an even higher impact on processors that support HT Technology.

Typically, an operating system provides timing services, for example Sleep(*dwMilliseconds*)[6]; such variables can be used to prevent frequent checking of a synchronization variable.

Another technique to synchronize between worker threads and a control loop is to use a thread-blocking API provided by the OS. Using a thread-blocking API allows the control thread to use less processor cycles for spinning and waiting. This gives the OS more time quanta to schedule the worker threads on available processors. Furthermore, using a thread-blocking API also benefits from the system idle loop optimization that OS implements using the HLT instruction.

User/Source Coding Rule 20. (H impact, M generality) *Use a thread-blocking API in a long idle loop to free up the processor.*

Using a spin-wait loop in a traditional MP system may be less of an issue when the number of runnable threads is less than the number of processors in the system. If the number of threads in an application is expected to be greater than the number of processors (either one processor or multiple processors), use a thread-blocking API to free up processor resources. A multithreaded application adopting one control thread to synchronize multiple worker threads may consider limiting worker threads to the number of processors in a system and use thread-blocking APIs in the control thread.

---

6. The Sleep() API is not thread-blocking, because it does not guarantee the processor will be released. Example 8-5(a) shows an example of using Sleep(0), which does not always realize the processor to another thread.

8.4.4.1 Avoid Coding Pitfalls in Thread Synchronization

Synchronization between multiple threads must be designed and implemented with care to achieve good performance scaling with respect to the number of discrete processors and the number of logical processor per physical processor. No single technique is a universal solution for every synchronization situation.

The pseudo-code example in Example 8-5(a) illustrates a polling loop implementation of a control thread. If there is only one runnable worker thread, an attempt to call a timing service API, such as Sleep(0), may be ineffective in minimizing the cost of thread synchronization. Because the control thread still behaves like a fast spinning loop, the only runnable worker thread must share execution resources with the spin-wait loop if both are running on the same physical processor that supports HT Technology. If there are more than one runnable worker threads, then calling a thread blocking API, such as Sleep(0), could still release the processor running the spin-wait loop, allowing the processor to be used by another worker thread instead of the spinning loop.

A control thread waiting for the completion of worker threads can usually implement thread synchronization using a thread-blocking API or a timing service, if the worker threads require significant time to complete. Example 8-5(b) shows an example that reduces the overhead of the control thread in its thread synchronization.

Example 8-5. Coding Pitfall using Spin Wait Loop

```
(a) A spin-wait loop attempts to release the processor incorrectly. It experiences a performance penalty if the only
worker thread and the control thread runs on the same physical processor package.
// Only one worker thread is running,
//  the control loop waits for the worker thread to complete.

ResumeWorkThread(thread_handle);
While (!task_not_done ) {
   Sleep(0)   // Returns immediately back to spin loop.
 ...
}
(b) A polling loop frees up the processor correctly.

// Let a worker thread run and wait for completion.
ResumeWorkThread(thread_handle);
While (!task_not_done ) {
  Sleep(FIVE_MILISEC)

//  This processor is released for some duration, the processor
//  can be used by other threads.
...
}
```

In general, OS function calls should be used with care when synchronizing threads. When using OS-supported thread synchronization objects (critical section, mutex, or semaphore), preference should be given to the OS service that has the least synchronization overhead, such as a critical section.

8.4.5 Prevent Sharing of Modified Data and False-Sharing

Depending on the cache topology relative to processor/core topology and the specific underlying micro-architecture, sharing of modified data can incur some degree of performance penalty when a software thread running on one core tries to read or write data that is currently present in modified state in the local cache of another core. This will cause eviction of the modified cache line back into memory and reading it into the first-level cache of the other core. The latency of such cache line transfer is much higher than using data in the immediate first level cache or second level cache.

False sharing applies to data used by one thread that happens to reside on the same cache line as different data used by another thread. These situations can also incur a performance delay depending on the topology of the logical processors/cores in the platform.

False sharing can experience a performance penalty when the threads are running on logical processors reside on different physical processors or processor cores. For processors that support HT Technology, false-sharing incurs a performance penalty when two threads run on different cores, different physical processors, or on two logical processors in the physical processor package. In the first two cases, the performance penalty is due to cache evictions to maintain cache coherency. In the latter case, performance penalty is due to memory order machine clear conditions.

A generic approach for multi-threaded software to prevent incurring false-sharing penalty is to allocate separate critical data or locks with alignment granularity according to a "false-sharing threshold" size. The following steps will allow software to determine the "false-sharing threshold" across Intel processors:

1. If the processor supports CLFLUSH instruction, i.e. CPUID.01H:EDX.CLFLUSH[bit 19] =1:

   Use the CLFLUSH line size, i.e. the integer value of CPUID.01H:EBX[15:8], as the "false-sharing threshold".

2. If CLFLUSH line size is not available, use CPUID leaf 4 as described below:

   Determine the "false-sharing threshold" by evaluating the largest system coherency line size among valid cache types that are reported via the sub-leaves of CPUID leaf 4. For each sub-leaf n, its associated system coherency line size is (CPUID.(EAX=4, ECX=n):EBX[11:0] + 1).

3. If neither CLFLUSH line size is available, nor CPUID leaf 4 is available, then software may choose the "false-sharing threshold" from one of the following:

   a. Query the descriptor tables of CPUID leaf 2 and choose from available descriptor entries.

   b. A Family/Model-specific mechanism available in the platform or a Family/Model-specific known value.

   c. Default to a safe value 64 bytes.

User/Source Coding Rule 21. (H impact, M generality) *Beware of false sharing within a cache line or within a sector. Allocate critical data or locks separately using alignment granularity not smaller than the "false-sharing threshold".*

When a common block of parameters is passed from a parent thread to several worker threads, it is desirable for each work thread to create a private copy (each copy aligned to multiples of the *"false-sharing threshold")* of frequently accessed data in the parameter block.

8.4.6 Placement of Shared Synchronization Variable

On processors based on Intel NetBurst microarchitecture, bus reads typically fetch 128 bytes into a cache, the optimal spacing to minimize eviction of cached data is 128 bytes. To prevent false-sharing, synchronization variables and system objects (such as a critical section) should be allocated to reside alone in a 128-byte region and aligned to a 128-byte boundary.

Example 8-6 shows a way to minimize the bus traffic required to maintain cache coherency in MP systems. This technique is also applicable to MP systems using processors with or without HT Technology.

Example 8-6. Placement of Synchronization and Regular Variables

```
int regVar;
int padding[32];
int SynVar[32*NUM_SYNC_VARS];
int AnotherVar;
```

On Pentium M, Intel Core Solo, Intel Core Duo processors, and processors based on Intel Core microarchitecture; a synchronization variable should be placed alone and in separate cache line to avoid false-sharing. Software must not allow a synchronization variable to span across page boundary.

User/Source Coding Rule 22. (M impact, ML generality) *Place each synchronization variable alone, separated by 128 bytes or in a separate cache line.*

User/Source Coding Rule 23. (H impact, L generality) *Do not place any spin lock variable to span a cache line boundary.*

At the code level, false sharing is a special concern in the following cases:

- Global data variables and static data variables that are placed in the same cache line and are written by different threads.
- Objects allocated dynamically by different threads may share cache lines. Make sure that the variables used locally by one thread are allocated in a manner to prevent sharing the cache line with other threads.

Another technique to enforce alignment of synchronization variables and to avoid a cacheline being shared is to use compiler directives when declaring data structures. See Example 8-7.

Example 8-7. Declaring Synchronization Variables without Sharing a Cache Line

```
__declspec(align(64)) unsigned __int64 sum;
struct sync_struct {…};
__declspec(align(64)) struct sync_struct sync_var;
```

Other techniques that prevent false-sharing include:

- Organize variables of different types in data structures (because the layout that compilers give to data variables might be different than their placement in the source code).
- When each thread needs to use its own copy of a set of variables, declare the variables with:
  — Directive threadprivate, when using OpenMP.
  — Modifier __declspec (thread), when using Microsoft compiler.
- In managed environments that provide automatic object allocation, the object allocators and garbage collectors are responsible for layout of the objects in memory so that false sharing through two objects does not happen.
- Provide classes such that only one thread writes to each object field and close object fields, in order to avoid false sharing.

One should not equate the recommendations discussed in this section as favoring a sparsely populated data layout. The data-layout recommendations should be adopted when necessary and avoid unnecessary bloat in the size of the work set.

8.4.7 Pause Latency in Skylake Microarchitecture

The PAUSE instruction is typically used with software threads executing on two logical processors located in the same processor core, waiting for a lock to be released. Such short wait loops tend to last between tens and a few hundreds of cycles, so performance-wise it is more beneficial to wait while occupying the CPU than yielding to the OS. When the wait loop is expected to last for thousands of cycles or more, it is preferable to yield to the operating system by calling one of the OS synchronization API functions, such as WaitForSingleObject on Windows* OS.

The PAUSE instruction is intended to:

- Temporarily provide the sibling logical processor (ready to make forward progress exiting the spin loop) with competitively shared hardware resources. The competitively-shared microarchitectural resources that the sibling logical processor can utilize in the Skylake microarchitecture are:
  — More front end slots in the Decode ICache, LSD and IDQ.

— More execution slots in the RS.

- Save power consumed by the processor core compared to executing equivalent spin loop instruction sequence in the following configurations:
  — One logical processor is inactive (e.g. entering a C-state).
  — Both logical processors in the same core execute the PAUSE instruction.
  — HT is disabled (e.g. using BIOS options).

The latency of PAUSE instruction in prior generation microarchitecture is about 10 cycles, whereas on Skylake microarchitecture it has been extended to as many as 140 cycles.

The increased latency (allowing more effective utilization of competitively-shared microarchitectural resources to the logical processor ready to make forward progress) has a small positive performance impact of 1-2% on highly threaded applications. It is expected to have negligible impact on less threaded applications if forward progress is not blocked on executing a fixed number of looped PAUSE instructions. There's also a small power benefit in 2-core and 4-core systems.

As the PAUSE latency has been increased significantly, workloads that are sensitive to PAUSE latency will suffer some performance loss.

8.5 SYSTEM BUS OPTIMIZATION

The system bus services requests from bus agents (e.g. logical processors) to fetch data or code from the memory sub-system. The performance impact due data traffic fetched from memory depends on the characteristics of the workload, and the degree of software optimization on memory access, locality enhancements implemented in the software code. A number of techniques to characterize memory traffic of a workload is discussed in Appendix A. Optimization guidelines on locality enhancement is also discussed in Section 3.6.11, "Locality Enhancement," and Section 7.6.11, "Hardware Prefetching and Cache Blocking Techniques."

The techniques described in Chapter 3 and Chapter 7 benefit application performance in a platform where the bus system is servicing a single-threaded environment. In a multi-threaded environment, the bus system typically services many more logical processors, each of which can issue bus requests independently. Thus, techniques on locality enhancements, conserving bus bandwidth, reducing large-stride-cache-miss-delay can have strong impact on processor scaling performance.

8.5.1 Conserve Bus Bandwidth

In a multithreading environment, bus bandwidth may be shared by memory traffic originated from multiple bus agents (These agents can be several logical processors and/or several processor cores). Preserving the bus bandwidth can improve processor scaling performance. Also, effective bus bandwidth typically will decrease if there are significant large-stride cache-misses. Reducing the amount of large-stride cache misses (or reducing DTLB misses) will alleviate the problem of bandwidth reduction due to large-stride cache misses.

One way for conserving available bus command bandwidth is to improve the locality of code and data. Improving the locality of data reduces the number of cache line evictions and requests to fetch data. This technique also reduces the number of instruction fetches from system memory.

User/Source Coding Rule 24. (M impact, H generality) *Improve data and code locality to conserve bus command bandwidth.*

Using a compiler that supports profiler-guided optimization can improve code locality by keeping frequently used code paths in the cache. This reduces instruction fetches. Loop blocking can also improve the data locality. Other locality enhancement techniques can also be applied in a multithreading environment to conserve bus bandwidth (see Section 7.6, "Memory Optimization Using Prefetch").

Because the system bus is shared between many bus agents (logical processors or processor cores), software tuning should recognize symptoms of the bus approaching saturation. One useful technique is to examine the queue depth of bus read traffic. When the bus queue depth is high, locality enhancement

to improve cache utilization will benefit performance more than other techniques, such as inserting more software prefetches or masking memory latency with overlapping bus reads. An approximate working guideline for software to operate below bus saturation is to check if bus read queue depth is significantly below 5.

Some MP and workstation platforms may have a chipset that provides two system buses, with each bus servicing one or more physical processors. The guidelines for conserving bus bandwidth described above also applies to each bus domain.

8.5.2 Understand the Bus and Cache Interactions

Be careful when parallelizing code sections with data sets that results in the total working set exceeding the second-level cache and /or consumed bandwidth exceeding the capacity of the bus. On an Intel Core Duo processor, if only one thread is using the second-level cache and / or bus, then it is expected to get the maximum benefit of the cache and bus systems because the other core does not interfere with the progress of the first thread. However, if two threads use the second-level cache concurrently, there may be performance degradation if one of the following conditions is true:

- Their combined working set is greater than the second-level cache size.
- Their combined bus usage is greater than the capacity of the bus.
- They both have extensive access to the same set in the second-level cache, and at least one of the threads writes to this cache line.

To avoid these pitfalls, multithreading software should try to investigate parallelism schemes in which only one of the threads access the second-level cache at a time, or where the second-level cache and the bus usage does not exceed their limits.

8.5.3 Avoid Excessive Software Prefetches

Pentium 4 and Intel Xeon Processors have an automatic hardware prefetcher. It can bring data and instructions into the unified second-level cache based on prior reference patterns. In most situations, the hardware prefetcher is likely to reduce system memory latency without explicit intervention from software prefetches. It is also preferable to adjust data access patterns in the code to take advantage of the characteristics of the automatic hardware prefetcher to improve locality or mask memory latency. Processors based on Intel Core microarchitecture also provides several advanced hardware prefetching mechanisms. Data access patterns that can take advantage of earlier generations of hardware prefetch mechanism generally can take advantage of more recent hardware prefetch implementations.

Using software prefetch instructions excessively or indiscriminately will inevitably cause performance penalties. This is because excessively or indiscriminately using software prefetch instructions wastes the command and data bandwidth of the system bus.

Using software prefetches delays the hardware prefetcher from starting to fetch data needed by the processor core. It also consumes critical execution resources and can result in stalled execution. In some cases, it may be fruitful to evaluate the reduction or removal of software prefetches to migrate towards more effective use of hardware prefetch mechanisms. The guidelines for using software prefetch instructions are described in Chapter 3. The techniques for using automatic hardware prefetcher is discussed in Chapter 7.

User/Source Coding Rule 25. (M impact, L generality) *Avoid excessive use of software prefetch instructions and allow automatic hardware prefetcher to work. Excessive use of software prefetches can significantly and unnecessarily increase bus utilization if used inappropriately.*

8.5.4 Improve Effective Latency of Cache Misses

System memory access latency due to cache misses is affected by bus traffic. This is because bus read requests must be arbitrated along with other requests for bus transactions. Reducing the number of outstanding bus transactions helps improve effective memory access latency.

One technique to improve effective latency of memory read transactions is to use multiple overlapping bus reads to reduce the latency of sparse reads. In situations where there is little locality of data or when memory reads need to be arbitrated with other bus transactions, the effective latency of scattered memory reads can be improved by issuing multiple memory reads back-to-back to overlap multiple outstanding memory read transactions. The average latency of back-to-back bus reads is likely to be lower than the average latency of scattered reads interspersed with other bus transactions. This is because only the first memory read needs to wait for the full delay of a cache miss.

User/Source Coding Rule 26. (M impact, M generality) *Consider using overlapping multiple back-to-back memory reads to improve effective cache miss latencies.*

Another technique to reduce effective memory latency is possible if one can adjust the data access pattern such that the access strides causing successive cache misses in the last-level cache is predominantly less than the trigger threshold distance of the automatic hardware prefetcher. See Section 7.6.3, "Example of Effective Latency Reduction with Hardware Prefetch."

User/Source Coding Rule 27. (M impact, M generality) *Consider adjusting the sequencing of memory references such that the distribution of distances of successive cache misses of the last level cache peaks towards 64 bytes.*

8.5.5 Use Full Write Transactions to Achieve Higher Data Rate

Write transactions across the bus can result in write to physical memory either using the full line size of 64 bytes or less than the full line size. The latter is referred to as a partial write. Typically, writes to write-back (WB) memory addresses are full-size and writes to write-combine (WC) or uncacheable (UC) type memory addresses result in partial writes. Both cached WB store operations and WC store operations utilize a set of six WC buffers (64 bytes wide) to manage the traffic of write transactions. When competing traffic closes a WC buffer before all writes to the buffer are finished, this results in a series of 8-byte partial bus transactions rather than a single 64-byte write transaction.

User/Source Coding Rule 28. (M impact, M generality) *Use full write transactions to achieve higher data throughput.*

Frequently, multiple partial writes to WC memory can be combined into full-sized writes using a software write-combining technique to separate WC store operations from competing with WB store traffic. To implement software write-combining, uncacheable writes to memory with the WC attribute are written to a small, temporary buffer (WB type) that fits in the first level data cache. When the temporary buffer is full, the application copies the content of the temporary buffer to the final WC destination.

When partial-writes are transacted on the bus, the effective data rate to system memory is reduced to only 1/8 of the system bus bandwidth.

8.6 MEMORY OPTIMIZATION

Efficient operation of caches is a critical aspect of memory optimization. Efficient operation of caches needs to address the following:

- Cache blocking.
- Shared memory optimization.
- Eliminating 64-KByte aliased data accesses.
- Preventing excessive evictions in first-level cache.

8.6.1 Cache Blocking Technique

Loop blocking is useful for reducing cache misses and improving memory access performance. The selection of a suitable block size is critical when applying the loop blocking technique. Loop blocking is applicable to single-threaded applications as well as to multithreaded applications running on processors with or without HT Technology. The technique transforms the memory access pattern into blocks that efficiently fit in the target cache size.

When targeting Intel processors supporting HT Technology, the loop blocking technique for a unified cache can select a block size that is no more than one half of the target cache size, if there are two logical processors sharing that cache. The upper limit of the block size for loop blocking should be determined by dividing the target cache size by the number of logical processors available in a physical processor package. Typically, some cache lines are needed to access data that are not part of the source or destination buffers used in cache blocking, so the block size can be chosen between one quarter to one half of the target cache (see Chapter 3, "General Optimization Guidelines").

Software can use the deterministic cache parameter leaf of CPUID to discover which subset of logical processors are sharing a given cache (see Chapter 7, "Optimizing Cache Usage"). Therefore, guideline above can be extended to allow all the logical processors serviced by a given cache to use the cache simultaneously, by placing an upper limit of the block size as the total size of the cache divided by the number of logical processors serviced by that cache. This technique can also be applied to single-threaded applications that will be used as part of a multitasking workload.

User/Source Coding Rule 29. (H impact, H generality) *Use cache blocking to improve locality of data access. Target one quarter to one half of the cache size when targeting Intel processors supporting HT Technology or target a block size that allow all the logical processors serviced by a cache to share that cache simultaneously.*

8.6.2 Shared-Memory Optimization

Maintaining cache coherency between discrete processors frequently involves moving data across a bus that operates at a clock rate substantially slower that the processor frequency.

8.6.2.1 Minimize Sharing of Data between Physical Processors

When two threads are executing on two physical processors and sharing data, reading from or writing to shared data usually involves several bus transactions (including snooping, request for ownership changes, and sometimes fetching data across the bus). A thread accessing a large amount of shared memory is likely to have poor processor-scaling performance.

User/Source Coding Rule 30. (H impact, M generality) *Minimize the sharing of data between threads that execute on different bus agents sharing a common bus. The situation of a platform consisting of multiple bus domains should also minimize data sharing across bus domains.*

One technique to minimize sharing of data is to copy data to local stack variables if it is to be accessed repeatedly over an extended period. If necessary, results from multiple threads can be combined later by writing them back to a shared memory location. This approach can also minimize time spent to synchronize access to shared data.

8.6.2.2 Batched Producer-Consumer Model

The key benefit of a threaded producer-consumer design, shown in Figure 8-5, is to minimize bus traffic while sharing data between the producer and the consumer using a shared second-level cache. On an Intel Core Duo processor and when the work buffers are small enough to fit within the first-level cache, re-ordering of producer and consumer tasks are necessary to achieve optimal performance. This is because fetching data from L2 to L1 is much faster than having a cache line in one core invalidated and fetched from the bus.

Figure 8-5 illustrates a batched producer-consumer model that can be used to overcome the drawback of using small work buffers in a standard producer-consumer model. In a batched producer-consumer model, each scheduling quanta batches two or more producer tasks, each producer working on a designated buffer. The number of tasks to batch is determined by the criteria that the total working set be greater than the first-level cache but smaller than the second-level cache.

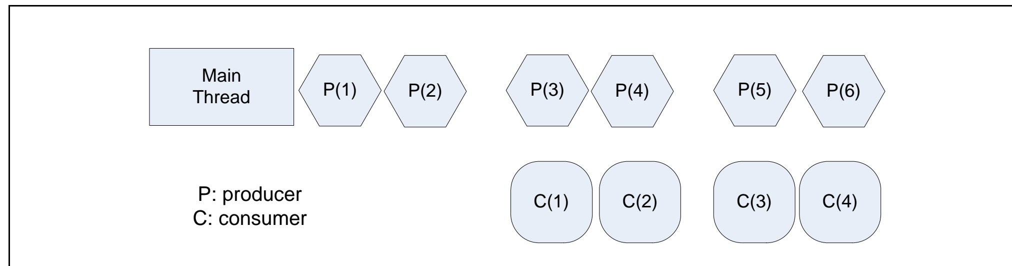


Figure 8-5. Batched Approach of Producer Consumer Model

Example 8-8 shows the batched implementation of the producer and consumer thread functions.

Example 8-8. Batched Implementation of the Producer Consumer Threads

```
void producer_thread()
{    int iter_num = workamount - batchsize;
     int mode1;
for (mode1=0; mode1 < batchsize; mode1++)
{         produce(buffs[mode1],count);  }

     while (iter_num--)
     {      Signal(&signal1,1);
            produce(buffs[mode1],count); // placeholder function
            WaitForSignal(&end1);
            mode1++;
            if (mode1 > batchsize)
                  mode1 = 0;
     }
}

void consumer_thread()
{    int mode2 = 0;
     int iter_num = workamount - batchsize;
     while (iter_num--)
     {      WaitForSignal(&signal1);
            consume(buffs[mode2],count); // placeholder function
            Signal(&end1,1);
            mode2++;
            if (mode2 > batchsize)
                  mode2 = 0;

     }
     for (i=0;i<batchsize;i++)
     {      consume(buffs[mode2],count);
            mode2++;
            if (mode2 > batchsize)
                  mode2 = 0;
     }
}
```

8.6.3 Eliminate 64-KByte Aliased Data Accesses

The 64-KByte aliasing condition is discussed in Chapter 3. Memory accesses that satisfy the 64-KByte aliasing condition can cause excessive evictions of the first-level data cache. Eliminating 64-KByte aliased data accesses originating from each thread helps improve frequency scaling in general. Furthermore, it enables the first-level data cache to perform efficiently when HT Technology is fully utilized by software applications.

User/Source Coding Rule 31. (H impact, H generality) *Minimize data access patterns that are offset by multiples of 64 KBytes in each thread.*

The presence of 64-KByte aliased data access can be detected using Pentium 4 processor performance monitoring events. Appendix B includes an updated list of Pentium 4 processor performance metrics. These metrics are based on events accessed using the Intel VTune Performance Analyzer.

Performance penalties associated with 64-KByte aliasing are applicable mainly to current processor implementations of HT Technology or Intel NetBurst microarchitecture. The next section discusses memory optimization techniques that are applicable to multithreaded applications running on processors supporting HT Technology.

8.7 FRONT END OPTIMIZATION

For dual-core processors where the second-level unified cache is shared by two processor cores (Intel Core Duo processor and processors based on Intel Core microarchitecture), multi-threaded software should consider the increase in code working set due to two threads fetching code from the unified cache as part of front end and cache optimization. For quad-core processors based on Intel Core microarchitecture, the considerations that applies to Intel Core 2 Duo processors also apply to quad-core processors.

8.7.1 Avoid Excessive Loop Unrolling

Unrolling loops can reduce the number of branches and improve the branch predictability of application code. Loop unrolling is discussed in detail in Chapter 3. Loop unrolling must be used judiciously. Be sure to consider the benefit of improved branch predictability and the cost of under-utilization of the loop stream detector (LSD).

User/Source Coding Rule 32. (M impact, L generality) *Avoid excessive loop unrolling to ensure the LSD is operating efficiently.*

8.8 AFFINITIES AND MANAGING SHARED PLATFORM RESOURCES

Modern OSes provide either API and/or data constructs (e.g. affinity masks) that allow applications to manage certain shared resources , e.g. logical processors, Non-Uniform Memory Access (NUMA) memory sub-systems.

Before multithreaded software considers using affinity APIs, it should consider the recommendations in Table 8-2.

Table 8-2. Design-Time Resource Management Choices

Runtime Environment	Thread Scheduling/Processor Affinity Consideration	Memory Affinity Consideration
A single-threaded application	Support OS scheduler objectives on system response and throughput by letting OS scheduler manage scheduling. OS provides facilities for end user to optimize runtime specific environment.	Not relevant; let OS do its job.
A multi-threaded application requiring: **i) less than all processor resource in the system,** **ii) share system resource with other concurrent applications,** **iii) other concurrent applications may have higher priority.**	Rely on OS default scheduler policy. Hard-coded affinity-binding will likely harm system response and throughput; and/or in some cases hurting application performance.	Rely on OS default scheduler policy. Use API that could provide transparent NUMA benefit without managing NUMA explicitly.
A multi-threaded application requiring **i) foreground and higher priority,** **ii) uses less than all processor resource in the system,** **iii) share system resource with other concurrent applications,** **iv) but other concurrent applications have lower priority.**	If application-customized thread binding policy is considered, a cooperative approach with OS scheduler should be taken instead of hard-coded thread affinity binding policy. For example, the use of SetThreadIdealProcessor() can provide a floating base to anchor a next-free-core binding policy for locality-optimized application binding policy, and cooperate with default OS policy.	Use API that could provide transparent NUMA benefit without managing NUMA explicitly. Use performance event to diagnose non-local memory access issue if default OS policy cause performance issue.
A multi-threaded application runs in foreground, requiring all processor resource in the system and not sharing system resource with concurrent applications; MPI-based multi-threading.	Application-customized thread binding policy can be more efficient than default OS policy. Use performance event to help optimize locality and cache transfer opportunities. A multi-threaded application that employs its own explicit thread affinity-binding policy should deploy with some form of opt-in choice granted by the end-user or administrator. For example, permission to deploy explicit thread affinity-binding policy can be activated after permission is granted after installation.	Application-customized memory affinity binding policy can be more efficient than default OS policy. Use performance event to diagnose non-local memory access issues related to either OS or custom policy

8.8.1 Topology Enumeration of Shared Resources

Whether multithreaded software ride on OS scheduling policy or need to use affinity APIs for customized resource management, understanding the topology of the shared platform resource is essential. The processor topology of logical processors (SMT), processor cores, and physical processors in the platform can enumerated using information provided by CPUID. This is discussed in Chapter 8, “Multiple-Processor Management” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. A white paper and reference code is also available from Intel.

8.8.2 Non-Uniform Memory Access

Platforms using two or more Intel Xeon processors based on Intel microarchitecture code name Nehalem support non-uniform memory access (NUMA) topology because each physical processor provides its own local memory controller. NUMA offers system memory bandwidth that can scale with the number of physical processors. System memory latency will exhibit asymmetric behavior depending on the memory transaction occurring locally in the same socket or remotely from another socket. Additionally, OS-specific construct and/or implementation behavior may present additional complexity at the API level that the multi-threaded software may need to pay attention to memory allocation/initialization in a NUMA environment.

Generally, latency sensitive workload would favor memory traffic to stay local over remote. If multiple threads shares a buffer, the programmer will need to pay attention to OS-specific behavior of memory allocation/initialization on a NUMA system.

Bandwidth sensitive workloads will find it convenient to employ a data composition threading model and aggregates application threads executing in each socket to favor local traffic on a per-socket basis to achieve overall bandwidth scalable with the number of physical processors.

The OS construct that provides the programming interface to manage local/remote NUMA traffic is referred to as memory affinity. Because OS manages the mapping between physical address (populated by system RAM) to linear address (accessed by application software); and paging allows dynamic reassignment of a physical page to map to different linear address dynamically, proper use of memory affinity will require a great deal of OS-specific knowledge.

To simplify application programming, OS may implement certain APIs and physical/linear address mapping to take advantage of NUMA characteristics transparently in certain situations. One common technique is for OS to delay commit of physical memory page assignment until the first memory reference on that physical page is accessed in the linear address space by an application thread. This means that the allocation of a memory buffer in the linear address space by an application thread does not necessarily determine which socket will service local memory traffic when the memory allocation API returns to the program. However, the memory allocation API that supports this level of NUMA transparency varies across different OSes. For example, the portable C-language API “malloc” provides some degree of transparency on Linux*, whereas the API “VirtualAlloc” behave similarly on Windows*. Different OSes may also provide memory allocation APIs that require explicit NUMA information, such that the mapping between linear address to local/remote memory traffic are fixed at allocation.

Example 8-9 shows an example that multi-threaded application could undertake the least amount of effort dealing with OS-specific APIs and to take advantage of NUMA hardware capability. This parallel

approach to memory buffer initialization is conducive to having each worker thread keep memory traffic local on NUMA systems.

Example 8-9. Parallel Memory Initialization Technique Using OpenMP and NUMA

```
#ifdef _LINUX // Linux implements malloc to commit physical page at first touch/access
    buf1 = (char *) malloc(DIM*(sizeof (double))+1024);
    buf2 = (char *) malloc(DIM*(sizeof (double))+1024);
    buf3 = (char *) malloc(DIM*(sizeof (double))+1024);
#endif
#ifdef windows
// Windows implements malloc to commit physical page at allocation, so use VirtualAlloc
    buf1 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
    buf2 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
    buf3 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
#endif
            (continue)
    a = (double *) buf1;
    b = (double *) buf2;
    c = (double *) buf3;
#pragma omp parallel
{ // use OpenMP threads to execute each iteration of the loop
// number of OpenMP threads can be specified by default or via environment variable
#pragma omp for private(num)
// each loop iteration is dispatched to execute in different OpenMP threads using private iterator
    for(num=0;num<len;num++)
    {// each thread perform first-touches to its own subset of memory address, physical pages
//      mapped to the local memory controller of the respective threads
            a[num]=10.;
            b[num]=10.;
            c[num]=10.;
    }
}
```

Note that the example shown in Example 8-9 implies that the memory buffers will be freed after the worker threads created by OpenMP have ended. This situation avoids a potential issue of repeated use of malloc/free across different application threads. Because if the local memory that was initialized by one thread and subsequently got freed up by another thread, the OS may have difficulty in tracking/re-allocating memory pools in linear address space relative to NUMA topology. In Linux, another API, "numa_local_alloc" may be used.

8.9 OPTIMIZATION OF OTHER SHARED RESOURCES

Resource optimization in multi-threaded application depends on the cache topology and execution resources associated within the hierarchy of processor topology. Processor topology and an algorithm for software to identify the processor topology are discussed in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

In platforms with shared buses, the bus system is shared by multiple agents at the SMT level and at the processor core level of the processor topology. Thus multi-threaded application design should start with an approach to manage the bus bandwidth available to multiple processor agents sharing the same bus link in an equitable manner. This can be done by improving the data locality of an individual application thread or allowing two threads to take advantage of a shared second-level cache (where such shared cache topology is available).

In general, optimizing the building blocks of a multi-threaded application can start from an individual thread. The guidelines discussed in Chapter 3 through Chapter 9 largely apply to multi-threaded optimization.

Tuning Suggestion 2. *Optimize single threaded code to maximize execution throughput first.*

Tuning Suggestion 3. *Employ efficient threading model, leverage available tools (such as Intel Threading Building Block, Intel Thread Checker, Intel Thread Profiler) to achieve optimal processor scaling with respect to the number of physical processors or processor cores.*

8.9.1 Expanded Opportunity for HT Optimization

The Hyper-Threading Technology (HT) implementation in Intel microarchitecture code name Nehalem differs from previous generations of HT implementations. It offers broader opportunity for multi-threaded software to take advantage of HT and achieve higher system throughput over a broader range of application problems. This section provide a few heuristic recommendations and illustrates some of those situations that HT in Intel microarchitecture code name Nehalem provides more optimization opportunities.

Chapter 2, “Intel® 64 and IA-32 Architectures” covered some of the microarchitectural capability enhancement in Hyper-Threading Technology. Many of these enhancements centers around the basic needs of multi-threaded software in terms of sharing common hardware resources that may be used by more than one thread context.

Different software algorithms and workload characteristics may produce different performance characteristics due to their demands on critical microarchitectural resources that may be shared amongst several logical processors. A brief comparison of the various microarchitectural subsystem that can play a significant role in software tuning for HT is summarized in Table 8-3.

Table 8-3. Microarchitectural Resources Comparisons of HT Implementations

Microarchitectural Subsystem	**Intel Microarchitecture Code Name Nehalem** **06_1AH**	**Intel NetBurst Microarchitecture** **0F_02H, 0F_03H, 0F_04H, 0F_06H**
Issue ports, execution units	Three issue ports (0, 1, 5) distributed to handle ALU, SIMD, FP computations	Unbalanced ports, fast ALU SIMD and FP sharing the same port (port 1).
Buffering	More entries in ROB, RS, fill buffers, etc. with moderate pipeline depths	Less balance between buffer entries and pipeline depths
Branch Prediction and Misaligned memory access	More robust speculative execution with immediate reclamation after misprediction; efficient handling of cache splits	More microarchitectural hazards resulting in pipeline cleared for both threads.
Cache hierarchy	Larger and more efficient	More microarchitectural hazards to work around
Memory and bandwidth	NUMA, three channels per socket to DDR3, up to 32GB/s per socket	SMP, FSB, or dual FSB, up to 12.8 GB/s per FSB

For compute bound workloads, the HT opportunity in Intel NetBurst microarchitecture tend to favor thread contexts that executes with relatively high CPI (average cycles to retire consecutive instructions). At a hardware level, this is in part due to the issue port imbalance in the microarchitecture, as port 1 is shared by fast ALU, slow ALU (more heavy-duty integer operations), SIMD, and FP computations. At a software level, some of the cause for high CPI and may appear as benign catalyst for providing HT benefit may include: long latency instructions (port 1), some L2 hits, occasional branch mispredictions, etc. But the length of the pipeline in Intel NetBurst microarchitecture often impose additional internal hardware constraints that limits software's ability to take advantage of HT.

The microarchitectural enhancements listed in Table 8-3 is expected to provide broader software optimization opportunities for compute-bound workloads. Whereas contention in the same execution unit by two compute-bound threads might be a concern to choose a functional-decomposition threading model over data-composition threading. Intel microarchitecture code name Nehalem will likely be more accommodating to support the programmer to choose the optimal threading decomposition models.

Memory intensive workloads can exhibit a wide range of performance characteristics, ranging from completely parallel memory traffic (saturating system memory bandwidth, as in the well-known example of Stream), memory traffic dominated by memory latency, or various mixtures of compute operations and memory traffic of either kind.

The HT implementation in Intel NetBurst microarchitecture may provide benefit to some of the latter two types of workload characteristics. The HT capability in the Intel microarchitecture code name Nehalem can broaden the operating envelop of the two latter types workload characteristics to deliver higher system throughput, due to its support for non-uniform memory access (NUMA), more efficient link protocol, and system memory bandwidth that scales with the number of physical processors.

Some cache levels of the cache hierarchy may be shared by multiple logical processors. Using the cache hierarchy is an important means for software to improve the efficiency of memory traffic and avoid saturating the system memory bandwidth. Multi-threaded applications employing cache-blocking technique may wish to partition a target cache level to take advantage of Hyper-Threading Technology. Alternately two logical processors sharing the same L1 and L2, or logical processors sharing the L3 may wish to manage the shared resources according to their relative topological relationship. A white paper on processor topology enumeration and cache topology enumeration with companion reference code has been published (see reference at the end of Chapter 1).

CHAPTER 9
64-BIT MODE CODING GUIDELINES

9.1 INTRODUCTION

This chapter describes coding guidelines for application software written to run in 64-bit mode. Some coding recommendations applicable to 64-bit mode are covered in Chapter 3. The guidelines in this chapter should be considered as an addendum to the coding guidelines described in Chapter 3 through Chapter 8.

Software that runs in either compatibility mode or legacy non-64-bit modes should follow the guidelines described in Chapter 3 through Chapter 8.

9.2 CODING RULES AFFECTING 64-BIT MODE

9.2.1 Use Legacy 32-Bit Instructions When Data Size Is 32 Bits

64-bit mode makes 16 general purpose 64-bit registers available to applications. If application data size is 32 bits, there is no need to use 64-bit registers or 64-bit arithmetic.

The default operand size for most instructions is 32 bits. The behavior of those instructions is to make the upper 32 bits all zeros. For example, when zeroing out a register, the following two instruction streams do the same thing, but the 32-bit version saves one instruction byte:

32-bit version:

```
xor eax, eax; Performs xor on lower 32bits and zeroes the upper 32 bits.
```

64-bit version:

```
xor rax, rax; Performs xor on all 64 bits.
```

This optimization holds true for the lower 8 general purpose registers: EAX, ECX, EBX, EDX, ESP, EBP, ESI, EDI. To access the data in registers R9-R15, the REX prefix is required. Using the 32-bit form there does not reduce code size.

Assembly/Compiler Coding Rule 64. (H impact, M generality) *Use the 32-bit versions of instructions in 64-bit mode to reduce code size unless the 64-bit version is necessary to access 64-bit data or additional registers.*

9.2.2 Use Extra Registers to Reduce Register Pressure

64-bit mode makes 8 additional 64-bit general purpose registers and 8 additional XMM registers available to applications. To access the additional registers, a single byte REX prefix is necessary. Using 8 additional registers can prevent the compiler from needing to spill values onto the stack.

Note that the potential increase in code size, due to the REX prefix, can increase cache misses. This can work against the benefit of using extra registers to access the data. When eight registers are sufficient for an algorithm, don’t use the registers that require an REX prefix. This keeps the code size smaller.

Assembly/Compiler Coding Rule 65. (M impact, MH generality) *When they are needed to reduce register pressure, use the 8 extra general purpose registers for integer code and 8 extra XMM registers for floating-point or SIMD code.*

9.2.3 Effective Use of 64-Bit by 64-Bit Multiplies

Integer multiplies of 64-bit by 64-bit operands can produce a result that's 128-bit wide. The upper 64-bits of a 128-bit result may take a few cycles longer to be ready than the lower 64 bits. In a dependent chain of addition of integers wider than 128-bits, accessing the high 64-bits result of the multiply should be delayed relative to the low 64-bit multiply result for optimal software pipelining.

If the compiler can determine at compile time that the result of a multiply will not exceed 64 bits, then the compiler should generate the multiply instruction that produces a 64-bit result. If the compiler or assembly programmer can not determine that the result will be less than 64 bits, then a multiply that produces a 128-bit result is necessary.

Assembly/Compiler Coding Rule 66. (ML impact, M generality) *Prefer 64-bit by 64-bit integer multiplies that produce 64-bit results over multiplies that produce 128-bit results.*

Assembly/Compiler Coding Rule 67. (ML impact, M generality) *Stagger accessing the high 64-bit result of a 128-bit multiply after accessing the low 64-bit results.*

In Intel microarchitecture code name Sandy Bridge, the low 64-bit result of a 128-bit multiply is ready to use in 3 cycles and the high 64-bit result is ready one cycle after the low 64-bit result. This can speed up the calculation of integer multiply and division of large integers.

9.2.4 Replace 128-bit Integer Division with 128-bit Multiplies

Modern compiler can transform expressions of integer division in high-level language code with a constant divisor into assembly sequences that uses IMUL/MUL to replace IDIV/DIV instructions. Typically, compilers will replace divisor value that is within the range of 32-bits and if the divisor value is known at compile time. If the divisor value is not known at compile time or the divisor is greater than those represented by 32-bits, DIV or IDIV will be generated.

The latency of DIV instruction with a 128-bit dividend is quite long. For dividend value greater than 64-bits, the latency can range from 70-90 cycles.

The basic technique that compiler employs to transform integer division into 128-bit integer multiply is based on the congruence principle of modular arithmetic. It can be easily extended to handle larger divisor values to take advantage of fast 128-bit IMUL/MUL operation.

The integer equation:

Dividend = Q * divisor + R, or

Q = floor(Dividend/Divisor), R = Dividend - Q * Divisor

Transform to the real number domain:

floor(Dividend/Divisor) = Dividend/Divisor - R/Divisor, is equivalent to

Q * C2 = Dividend * (C2 /divisor) + R*C2/Divisor

One can choose C2 = 2^N to control the rounding of the last term, then

Q = ((Dividend * (C2 /divisor)) >> N) + ((R*C2/Divisor) >> N).

If "divisor" is known at compile time, (C2/Divisor) can be pre-computed into a congruent constant Cx = Ceil(C2/divisor), then the quotient can computed by an integer multiple, followed by a shift:

Q = (Dividend * Cx) >> N;

R = Dividend - ((Dividend * Cx) >> N) * divisor;

The 128-bit IDIV/DIV instructions restrict the range of divisor, quotient, and remainder to be within 64-bits to avoid causing numerical exceptions. This presents a challenge for situations with either of the

three having a value near the upper bounds of 64-bit and for dividend values nearing the upper bound of 128 bits.

This challenge can be overcome with choosing a larger shift count N, and extending the (Dividend * Cx) operation from 128-bit range to the next computing-efficient range. For example, if (Dividend * Cx) is greater and 128 bits and N is greater than 63 bits, one can take advantage of computing bits 191:64 of the 192-bit results using 128-bit MUL without implementing a full 192-bit multiply.

A convenient way to choose the congruent constant Cx is as follows:

- If the range of dividend is within 64 bits: N_{min} ~ BSR(Divisor) + 63.
- In situations of disparate dynamic range of quotient/remainder relative to the range of divisor, raise N accordingly so that quotient/remainder can be computed efficiently.

Consider the computation of quotient/remainder computation for the divisor 10^16 on unsigned dividends near the range of 64-bits. Example 9-1 illustrates using "MUL r64" instruction to handle 64-bit dividend with 64-bit divisors.

Example 9-1. Compute 64-bit Quotient and Remainder with 64-bit Divisor

```
_Cx10to16:        ; Congruent constant for 10^16 with shift count 'N' = 117
    DD      0c44de15ch       ; floor ( (2^117 / 10^16) + 1)
    DD      0e69594beh       ; Optimize length of Cx to reduce # of 128-bit multiplies
_tento16:                    ; 10^16
    DD      6fc10000h
    DD      002386f2h

    mov     r9, qword ptr [rcx]      ; load 64-bit dividend value
    mov     rax, r9
    mov     rsi, _Cx10to16   ; Congruent Constant for 10^16 with shift count 117
    mul     [rsi]            ; 128-bit multiply
    mov     r10, qword ptr 8[rsi]    ; load divisor 10^16
    shr     rdx, 53;         ;
    mov     r8, rdx


    mov     rax, r8
    mul     r10     ; 128-bit multiply
    sub     r9, rax;         ;
    jae     remain
    sub     r8, 1   ; this may be off by one due to round up
    mov     rax, r8
    mul     r10              ; 128-bit multiply
    sub     r9, rax;         ;
remain:
    mov     rdx, r8          ; quotient
    mov     rax, r9          ; remainder
```

Example 9-2 shows a similar technique to handle 128-bit dividend with 64-bit divisors.

Example 9-2. Quotient and Remainder of 128-bit Dividend with 64-bit Divisor

```
    mov     rax, qword ptr [rcx]    ; load bits 63:0 of 128-bit dividend from memory
    mov     rsi, _Cx10to16      ; Congruent Constant for 10^16 with shift count 117
    mov     r9, qword ptr [rsi]     ; load Congruent Constant
    mul     r9                  ; 128-bit multiply
    xor     r11, r11                ; clear accumulator
    mov     rax, qword ptr 8[rcx]   ; load bits 127:64 of 128-bit dividend
    shr     rdx, 53;            ;
    mov     r10, rdx            ; initialize bits 127:64 of 192 b it result
    mul     r9                  ; Accumulate to bits 191:128
    add     rax, r10;           ;
    adc     rdx, r11;           ;
    shr     rax, 53;            ;
    shl     rdx, 11;            ;
    or      rdx, rax;           ;
    mov     r8, qword ptr 8[rsi]    ; load Divisor 10^16
    mov     r9, rdx;            ; approximate quotient, may be off by 1
    mov     rax, r8
    mul     r9                  ; will quotient * divisor > dividend?
    sub     rdx, qword ptr 8[rcx]   ;
    sbb     rax, qword ptr [rcx]    ;


    jb      remain
    sub     r9, 1           ; this may be off by one due to round up
    mov     rax, r8     ; retrieve Divisor 10^16
    mul     r9          ; final quotient * divisor
    sub     rax, qword ptr [rcx]    ;
    sbb     rdx, qword ptr 8[rcx]   ;
remain:
    mov     rdx, r9         ; quotient
    neg     rax             ; remainder
```

The techniques illustrated in Example 9-1 and Example 9-2 can increase the speed of remainder/quotient calculation of 128-bit dividends to at or below the cost of a 32-bit integer division.

Extending the technique above to deal with divisor greater than 64-bits is relatively straightforward. One optimization worth considering is to choose shift count N > 128 bits. This can reduce the number of 128-bit MUL needed to compute the relevant upper bits of (Dividend * Cx).

9.2.5 Sign Extension to Full 64-Bits

When in 64-bit mode, processors based on Intel NetBurst microarchitecture can sign-extend to 64 bits in a single micro-op. In 64-bit mode, when the destination is 32 bits, the upper 32 bits must be zeroed.

Zeroing the upper 32 bits requires an extra micro-op and is less optimal than sign extending to 64 bits. While sign extending to 64 bits makes the instruction one byte longer, it reduces the number of micro-ops that the trace cache has to store, improving performance.

For example, to sign-extend a byte into ESI, use:

```
movsx rsi, BYTE PTR[rax]
```

instead of:

```
movsx esi, BYTE PTR[rax]
```

If the next instruction uses the 32-bit form of esi register, the result will be the same. This optimization can also be used to break an unintended dependency. For example, if a program writes a 16-bit value to a register and then writes the register with an 8-bit value, if bits 15:8 of the destination are not needed, use the sign-extended version of writes when available.

For example:

```
mov r8w, r9w; Requires a merge to preserve
       ; bits 63:15.
mov r8b, r10b; Requires a merge to preserve bits 63:8
```

Can be replaced with:

```
movsx r8, r9w ; If bits 63:8 do not need to be
       ; preserved.
movsx r8, r10b  ; If bits 63:8 do not need to
       ; be preserved.
```

In the above example, the moves to R8W and R8B both require a merge to preserve the rest of the bits in the register. There is an implicit real dependency on R8 between the 'MOV R8W, R9W' and 'MOV R8B, R10B'. Using MOVSX breaks the real dependency and leaves only the output dependency, which the processor can eliminate through renaming.

For processors based on Intel Core microarchitecture, zeroing the upper 32 bits is faster than sign-extend to 64 bits. For processors based on Intel microarchitecture code name Nehalem, zeroing or sign-extend the upper bits is single micro-op.

9.3 ALTERNATE CODING RULES FOR 64-BIT MODE

9.3.1 Use 64-Bit Registers Instead of Two 32-Bit Registers for 64-Bit Arithmetic Result

Legacy 32-bit mode offers the ability to support extended precision integer arithmetic (such as 64-bit arithmetic). However, 64-bit mode offers native support for 64-bit arithmetic. When 64-bit integers are desired, use the 64-bit forms of arithmetic instructions.

In 32-bit legacy mode, getting a 64-bit result from a 32-bit by 32-bit integer multiply requires three registers; the result is stobbred in 32-bit chunks in the EDX:EAX pair. When the instruction is available in 64-bit mode, using the 32-bit version of the instruction is not the optimal implementation if a 64-bit result is desired. Use the extended registers.

For example, the following code sequence loads the 32-bit values sign-extended into the 64-bit registers and performs a multiply:

```
movsx rax, DWORD PTR[x]
movsx rcx, DWORD PTR[y]
imul  rax, rcx
```

The 64-bit version above is more efficient than using the following 32-bit version:

```
mov eax, DWORD PTR[x]
mov ecx, DWORD PTR[y]
imul ecx
```

In the 32-bit case above, EAX is required to be a source. The result ends up in the EDX:EAX pair instead of in a single 64-bit register.

Assembly/Compiler Coding Rule 68. (ML impact, M generality) *Use the 64-bit versions of multiply for 32-bit integer multiplies that require a 64 bit result.*

To add two 64-bit numbers in 32-bit legacy mode, the add instruction followed by the addc instruction is used. For example, to add two 64-bit variables (X and Y), the following four instructions could be used:

```
mov eax, DWORD PTR[X]
mov edx, DWORD PTR[X+4]
add eax, DWORD PTR[Y]
adc edx, DWORD PTR[Y+4]
```

The result will end up in the two-register EDX:EAX.

In 64-bit mode, the above sequence can be reduced to the following:

```
mov rax, QWORD PTR[X]
add rax, QWORD PTR[Y]
```

The result is stored in rax. One register is required instead of two.

Assembly/Compiler Coding Rule 69. (ML impact, M generality) *Use the 64-bit versions of add for 64-bit adds.*

9.3.2 CVTSI2SS and CVTSI2SD

In processors based on Intel Core microarchitecture and later, CVTSI2SS and CVTSI2SD are improved significantly over those in Intel NetBurst microarchitecture, in terms of latency and throughput. The improvements applies equally to 64-bit and 32-bit versions.

9.3.3 Using Software Prefetch

Intel recommends that software developers follow the recommendations in Chapter 3 and Chapter 7 when considering the choice of organizing data access patterns to take advantage of the hardware prefetcher (versus using software prefetch).

Assembly/Compiler Coding Rule 70. (L impact, L generality) *If software prefetch instructions are necessary, use the prefetch instructions provided by SSE.*

CHAPTER 10 SSE4.2 AND SIMD PROGRAMMING FOR TEXT-PROCESSING/LEXING/PARSING

String/text processing spans a discipline that often employs techniques different from traditional SIMD integer vector processing. Much of the traditional string/text algorithms are character based, where characters may be represented by encodings (or code points) of fixed or variable byte sizes. Textual data represents a vast amount of raw data and often carrying contextual information. The contextual information embedded in raw textual data often requires algorithmic processing dealing with a wide range of attributes, such as character values, character positions, character encoding formats, subsetting of character sets, strings of explicit or implicit lengths, tokens, delimiters; contextual objects may be represented by sequential characters within a pre-defined character subsets (e.g. decimal-valued strings); textual streams may contain embedded state transitions separating objects of different contexts (e.g. tag-delimited fields).

Traditional Integer SIMD vector instructions may, in some simpler situations, be successful to speed up simple string processing functions. SSE4.2 includes four new instructions that offer advances to computational algorithms targeting string/text processing, lexing and parsing of either unstructured or structured textual data.

10.1 SSE4.2 STRING AND TEXT INSTRUCTIONS

SSE4.2 provides four instructions, PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM that can accelerate string and text processing by combining the efficiency of SIMD programming techniques and the lexical primitives that are embedded in these 4 instructions. Simple examples of these instructions include string length determination, direct string comparison, string case handling, delimiter/token processing, locating word boundaries, locating sub-string matches in large text blocks. Sophisticated application of SSE4.2 can accelerate XML parsing and Schema validation.

Processor's support for SSE4.2 is indicated by the feature flag value returned in ECX [bit 20] after executing CPUID instruction with EAX input value of 1 (i.e. SSE4.2 is supported if CPUID.01H:ECX.SSE4_2 [bit 20] = 1). Therefore, software must verify CPUID.01H:ECX.SSE4_2 [bit 20] is set before using these 4 instructions. (Verifying CPUID.01H:ECX.SSE4_2 = 1 is also required before using PCMPGTQ or CRC32. Verifying CPUID.01H:ECX.POPCNT[Bit 23] = 1 is required before using the POPCNT instruction.)

These string/text processing instructions work by performing up to 256 comparison operations on text fragments. Each text fragment can be 16 bytes. They can handle fragments of different formats: either byte or word elements. Each of these four instructions can be configured to perform four types of parallel comparison operation on two text fragments.

The aggregated intermediate result of a parallel comparison of two text fragments become a bit patterns: 16 bits for processing byte elements or 8 bits for word elements. These instruction provide additional flexibility, using bit fields in the immediate operand of the instruction syntax, to configure an unary transformation (polarity) on the first intermediate result.

Lastly, the instruction's immediate operand offers a output selection control to further configure the flexibility of the final result produced by the instruction. The rich configurability of these instruction is summarized in Figure 10-1.

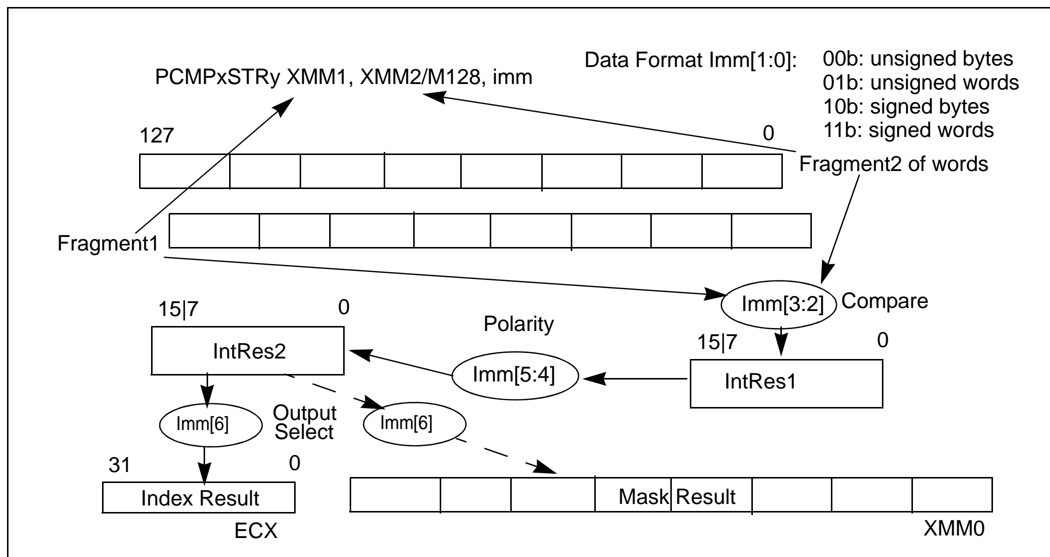


Figure 10-1. SSE4.2 String/Text Instruction Immediate Operand Control

The PCMPxSTRI instructions produce final result as an integer index in ECX, the PCMPxSTRM instructions produce final result as a bit mask in the XMM0 register. The PCMPISTRy instructions support processing string/text fragments using implicit length control via null termination for handling string/text of unknown size. the PCMPESTRy instructions support explicit length control via EDX:EAX register pair to specify the length text fragments in the source operands.

The first intermediate result, IntRes1, is an aggregated result of bit patterns from parallel comparison operations done on pairs of data elements from each text fragment, according to the imm[3:2] bit field encoding, see Table 10-1.

Table 10-1. SSE4.2 String/Text Instructions Compare Operation on N-elements

Imm[3:2]	Name	IntRes1[i] is TRUE if	Potential Usage
00B	Equal Any	Element i in fragment2 matches any element j in fragment1	Tokenization, XML parser
01B	Ranges	Element i in fragment2 is within any range pairs specified in fragment1	Subsetting, Case handling, XML parser, Schema validation
10B	Equal Each	Element i in fragment2 matches element i in fragment1	Strcmp()
11B	Equal Ordered	Element i and subsequent, consecutive valid elements in fragment2 match fully or partially with fragment1 starting from element 0	Substring Searches, KMP, Strstr()

Input data element format selection using imm[1:0] can support signed or unsigned byte/word elements.

The bit field imm[5:4] allows applying a unary transformation on IntRes1, see Table 10-2.

Table 10-2. SSE4.2 String/Text Instructions Unary Transformation on IntRes1

Imm[5:4]	Name	IntRes2[i] =	Potential Usage
00B	No Change	IntRes1[i]	
01B	Invert	-IntRes1[i]	
10B	No Change	IntRes1[i]	
11B	Mask Negative	IntRes1[i] if element i of fragment2 is invalid, otherwise -IntRes1[i]	

The output selection field, imm[6] is described in Table 10-3.

Table 10-3. SSE4.2 String/Text Instructions Output Selection Imm[6]

Imm[6]	Instruction	Final Result	Potential Usage
0B	PCMPxSTRI	ECX = offset of least significant bit set in IntRes2 if IntRes2 != 0, otherwise ECX = number of data element per 16 bytes	
0B	PCMPxSTRM	XMM0 = ZeroExtend(IntRes2);	
1B	PCMPxSTRI	ECX = offset of most significant bit set in IntRes2 if IntRes2 != 0, otherwise ECX = number of data element per 16 bytes	
1B	PCMPxSTRM	Data element i of XMM0 = SignExtend(IntRes2[i]);	

The comparison operation on each data element pair is defined in Table 10-4. Table 10-4 defines the type of comparison operation between valid data elements (last row of Table 10-4) and boundary conditions when the fragment in a source operand may contain invalid data elements (rows 1 through 3 of Table 10-4). Arithmetic comparison are performed only if both data elements are valid element in fragment1 and fragment2, as shown in row 4 of Table 10-4.

Table 10-4. SSE4.2 String/Text Instructions Element-Pair Comparison Definition

Fragment1 Element	Fragment2 Element	Imm[3:2]= 00B, Equal Any	Imm[3:2]= 01B, Ranges	Imm[3:2]= 10B, Equal Each	Imm[3:2]= 11B, Equal Ordered
Invalid	Invalid	Force False	Force False	Force True	Force True
Invalid	Valid	Force False	Force False	Force False	Force True
Valid	Invalid	Force False	Force False	Force False	Force False
Valid	Valid	Compare	Compare	Compare	Compare

The string and text processing instruction provides several aid to handle end-of-string situations, see Table 10-5. Additionally, the PCMPxSTRy instructions are designed to not require 16-byte alignment to simplify text processing requirements.

Table 10-5. SSE4.2 String/Text Instructions Eflags Behavior

EFLAGs	Description	Potential Usage
CF	Reset if IntRes2 = 0; Otherwise set	When CF=0, ECX= #of data element to scan next
ZF	Reset if entire 16-byte fragment2 is valid	likely end-of-string
SF	Reset if entire 16-byte fragment1 is valid	
OF	IntRes2[0];	

10.1.1 CRC32

CRC32 instruction computes the 32-bit cyclic redundancy checksum signature for byte/word/dword or qword stream of data. It can also be used as a hash function. For example, a dictionary uses hash indices to de-reference strings. CRC32 instruction can be easily adapted for use in this situation.

Example 10-1 shows a straight forward hash function that can be used to evaluate the hash index of a string to populate a hash table. Typically, the hash index is derived from the hash value by taking the remainder of the hash value modulo the size of a hash table.

Example 10-1. A Hash Function Examples

```
unsigned int hash_str(unsigned char* pStr)
{   unsigned int hVal = (unsigned int)(*pStr++);
   while (*pStr)
    {   hVal = (hashVal * CONST_A) + (hVal >> 24) + (unsigned int)(*pStr++);
    }
    return hVal;
}
```

CRC32 instruction can be use to derive an alternate hash function. Example 10-2 takes advantage the 32-bit granular CRC32 instruction to update signature value of the input data stream. For string of small to moderate sizes, using the hardware accelerated CRC32 can be twice as fast as Example 10-1.

Example 10-2. Hash Function Using CRC32

```
static unsigned cn_7e = 0x7efefeff, Cn_81 = 0x81010100;

unsigned int hash_str_32_crc32x(unsigned char* pStr)
{   unsigned  *pDW = (unsigned  *) &pStr[1];
    unsigned  short *pWd = (unsigned short *) &pStr[1];
    unsigned int tmp, hVal = (unsigned int)(*pStr);
    if( !pStr[1]) ;
    else {
        tmp = ((pDW[0] +cn_7e ) ^(pDW[0]^ -1)) & Cn_81;
        while ( !tmp ) // loop until there is byte in *pDW had 0x00
        {
            hVal = _mm_crc32_u32 (hVal, *pDW ++);
            tmp = ((pDW[0] +cn_7e ) ^(pDW[0]^ -1)) & Cn_81;
        };
        if(!pDW[0]);
        else  if(pDW[0] < 0x100) { // finish last byte that's non-zero
            hVal = _mm_crc32_u8 (hVal, pDW[0]);
        }
```

Example 10-2. Hash Function Using CRC32 (Contd.)

```
        else  if(pDW[0] < 0x10000) { // finish last two byte that's non-zero
            hVal = _mm_crc32_u16 (hVal, pDW[0]);
        }
        else  { // finish last three byte that's non-zero
            hVal = _mm_crc32_u32 (hVal, pDW[0]);
        }
    }
    return hVal;
}
```

10.2 USING SSE4.2 STRING AND TEXT INSTRUCTIONS

String libraries provided by high-level languages or as part of system library are used in a wide range of situations across applications and privileged system software. These situations can be accelerated using a replacement string library that implements PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM.

Although system-provided string library provides standardized string handling functionality and interfaces, most situations dealing with structured document processing requires considerable more sophistication, optimization, and services not available from system-provided string libraries. For example, structured document processing software often architect different class objects to provide building block functionality to service specific needs of the application. Often application may choose to disperse equivalent string library services into separate classes (string, lexer, parser) or integrate memory management capability into string handling/lexing/parsing objects.

PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM instructions are general-purpose primitives that software can use to build replacement string libraries or build class hierarchy to provide lexing/parsing services for structured document processing. XML parsing and schema validation are examples of the latter situations.

Unstructured, raw text/string data consist of characters, and have no natural alignment preferences. Therefore, PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM instructions are architected to not require the 16-Byte alignment restrictions of other 128-bit SIMD integer vector processing instructions.

With respect to memory alignment, PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM support unaligned memory loads like other unaligned 128-bit memory access instructions, e.g. MOVDQU.

Unaligned memory accesses may encounter special situations that require additional coding techniques, depending on the code running in ring 3 application space or in privileged space. Specifically, an unaligned 16-byte load may cross page boundary. Section 10.2.1 discusses a technique that application code can use. Section 10.2.2 discusses the situation string library functions needs to deal with. Section 10.3 gives detailed examples of using PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM instructions to implement equivalent functionality of several string library functions in situations that application code has control over memory buffer allocation.

10.2.1 Unaligned Memory Access and Buffer Size Management

In application code, the size requirements for memory buffer allocation should consider unaligned SIMD memory semantics and application usage.

For certain types of application usage, it may be desirable to make distinctions between valid buffer range limit versus valid application data size (e.g. a video frame). The former must be greater or equal to the latter.

To support algorithms requiring unaligned 128-bit SIMD memory accesses, memory buffer allocation by a caller function should consider adding some pad space so that a callee function can safely use the address pointer safely with unaligned 128-bit SIMD memory operations.

The minimal padding size should be the width of the SIMD register that might be used in conjunction with unaligned SIMD memory access.

10.2.2 Unaligned Memory Access and String Library

String library functions may be used by application code or privileged code. String library functions must be careful not to violate memory access rights. Therefore, a replacement string library that employ SIMD unaligned access must employ special techniques to ensure no memory access violation occur.

Section 10.3.6 provides an example of a replacement string library function implemented with SSE4.2 and demonstrates a technique to use 128-bit unaligned memory access without unintentionally crossing page boundary.

10.3 SSE4.2 APPLICATION CODING GUIDELINE AND EXAMPLES

Software implementing SSE4.2 instruction must use CPUID feature flag mechanism to verify processor's support for SSE4.2. Details can be found in CHAPTER 12 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* and in CPUID of CHAPTER 3 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

In the following sections, we use several examples in string/text processing of progressive complexity to illustrates the basic techniques of adapting the SIMD approach to implement string/text processing using PCMPxSTRy instructions in SSE4.2. For simplicity, we will consider string/text in byte data format in situations that caller functions have allocated sufficient buffer size to support unaligned 128-bit SIMD loads from memory without encountering side-effects of cross page boundaries.

10.3.1 Null Character Identification (Strlen equivalent)

The most widely used string function is probably strlen(). One can view the lexing requirement of strlen() is to identify the null character in a text block of unknown size (end of string condition). Brute-force, byte-granular implementation fetches data inefficiently by loading one byte at a time.

Optimized implementation using general-purpose instructions can take advantage of dword operations in 32-bit environment (and qword operations in 64-bit environment) to reduce the number of iterations.

A 32-bit assembly implementation of strlen() is shown Example 10-3. The peak execution throughput of handling EOS condition is determined by eight ALU instructions in the main loop.

Example 10-3. Strlen() Using General-Purpose Instructions

```
int strlen_asm(const  char* s1)
{int len = 0;
   _asm{
   mov  ecx, s1
   test  ecx, 3 ; test addr aligned to dword
   je     short _main_loop1  ; dword aligned loads would be faster
_malign_str1:
   mov  al, byte ptr [ecx] ; read one byte at a time
   add   ecx, 1
   test  al, al ; if we find a null, go calculate the length
   je     short _byte3a
```

(continue)

Example 10-3. Strlen() Using General-Purpose Instructions (Contd.)

```
   test  ecx, 3; test if addr is now aligned to dword
   jne   short _malign_str1; if not, repeat
   align16
_main_loop1:; read each 4-byte block and check for a NULL char in the dword
   mov   eax, [ecx]; read 4 byte to reduce loop count
   mov   edx, 7efefeffh
   add   edx, eax
   xor   eax, -1
   xor   eax, edx
   add   ecx, 4; increment address pointer by 4
   test  eax, 81010100h ; if no null code in 4-byte stream, do the next 4 bytes
   je    short _main_loop1
   ;     there is a null char in the dword we just read,
   ;     since we already advanced pointer ecx by 4, and the dword is lost
   mov   eax, [ecx -4]; re-read the dword that contain at least a null char
   test  al, al     ; if byte0 is null
   je    short _byte0a; the least significant byte is null
   test  ah, ah   ; if byte1 is null
   je    short _byte1a
   test  eax, 00ff0000h; if byte2 is null
   je    short _byte2a
   test  eax, 00ff000000h; if byte3 is null
   je    short _byte3a
   jmp   short _main_loop1
_byte3a:
   ; we already found the null, but pointer already advanced by 1
   lea   eax, [ecx-1]; load effective address corresponding to null code
   mov   ecx, s1
   sub   eax, ecx; difference between null code and start address
   jmp   short _resulta
_byte2a:

   lea   eax, [ecx-2]
   mov   ecx, s1
   sub   eax, ecx
   jmp   short _resulta
_byte1a:
   lea   eax, [ecx-3]
   mov   ecx, s1
   sub   eax, ecx
   jmp   short _resulta
_byte0a:
   lea   eax, [ecx-4]
   mov   ecx, s1
   sub   eax, ecx
_resulta:
   mov    len, eax; store result
   }
   return len;
}
```

The equivalent functionality of EOS identification can be implemented using PCMPISTRI. Example 10-4 shows a simplistic SSE4.2 implementation to scan a text block by loading 16-byte text fragments and locate the null termination character. Example 10-5 shows the optimized SSE4.2 implementation that demonstrates the importance of using memory disambiguation to improve instruction-level parallelism.

Example 10-4. Sub-optimal PCMPISTRI Implementation of EOS handling

```
static char ssch2[16]= {0x1, 0xff, 0x00, }; // range values for non-null characters

int strlen_un_optimized(const  char* s1)
{int len = 0;
  _asm{
  mov  eax, s1
  movdquxmm2, ssch2 ; load character pair as range (0x01 to 0xff)
  xor    ecx, ecx ; initial offset to 0
              (continue)
_loopc:
  add    eax, ecx   ; update addr pointer to start of text fragment
  pcmpistri  xmm2, [eax], 14h; unsigned bytes, ranges, invert, lsb index returned to ecx
      ; if there is a null char in the 16Byte fragment at [eax], zf will be set.
      ; if all 16 bytes of the fragment are non-null characters, ECX will return 16,
  jnz    short _loopc; xmm1 has no null code, ecx has 16, continue search
  ;        we have a null code in xmm1, ecx has the offset of the null code i
  add    eax, ecx       ; add ecx to the address of the last fragment2/xmm1
  mov   edx, s1; retrieve effective address of the input string
  sub    eax, edx;the string length
  mov   len, eax; store result
  }
  return len;
}
```

The code sequence shown in Example 10-4 has a loop consisting of three instructions. From a performance tuning perspective, the loop iteration has loop-carry dependency because address update is done using the result (ECX value) of a previous loop iteration. This loop-carry dependency deprives the out-of-order engine's capability to have multiple iterations of the instruction sequence making forward progress. The latency of memory loads, the latency of these instructions, any bypass delay could not be amortized by OOO execution in the presence of loop-carry dependency.

A simple optimization technique to eliminate loop-carry dependency is shown in Example 10-5.

Using memory disambiguation technique to eliminate loop-carry dependency, the cumulative latency exposure of the 3-instruction sequence of Example 10-5 is amortized over multiple iterations, the net cost of executing each iteration (handling 16 bytes) is less then 3 cycles. In contrast, handling 4 bytes of string data using 8 ALU instructions in Example 10-3 will also take a little less than 3 cycles per iteration. Whereas each iteration of the code sequence in Example 10-4 will take more than 10 cycles because of loop-carry dependency.

Example 10-5. Strlen() Using PCMPISTRI without Loop-Carry Dependency

```
int strlen_sse4_2(const  char* s1)
{int len = 0;
  _asm{
  mov  eax, s1
  movdquxmm2, ssch2 ; load character pair as range (0x01 to 0xff)
  xor    ecx, ecx ; initial offset to 0
  sub    eax, 16 ; address arithmetic to eliminate extra instruction and a branch
```

Example 10-5. Strlen() Using PCMPISTRI without Loop-Carry Dependency (Contd.)

```
_loopc:
   add   eax, 16 ; adjust address pointer and disambiguate load address for each iteration
   pcmpistri  xmm2, [eax], 14h; unsigned bytes, ranges, invert, lsb index returned to ecx
       ; if there is a null char in [eax] fragment, zf will be set.
       ; if all 16 bytes of the fragment are non-null characters, ECX will return 16,
   jnz short _loopc ; ECX will be 16 if there is no null byte in [eax], so we disambiguate
_endofstring:
   add   eax, ecx       ; add ecx to the address of the last fragment
   mov   edx, s1; retrieve effective address of the input string
   sub   eax, edx;the string length
   mov   len, eax; store result
   }
   return len;
}
```

SSE4.2 Coding Rule 5. (H impact, H generality) *Loop-carry dependency that depends on the ECX result of PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM for address adjustment must be minimized. Isolate code paths that expect ECX result will be 16 (bytes) or 8 (words), replace these values of ECX with constants in address adjustment expressions to take advantage of memory disambiguation hardware.*

10.3.2 White-Space-Like Character Identification

Character-granular-based text processing algorithms have developed techniques to handle specific tasks to remedy the efficiency issue of character-granular approaches. One such technique is using look-up tables for character subset classification. For example, some application may need to separate alpha-numeric characters from white-space-like characters. More than one character may be treated as white-space characters.

Example 10-6 illustrates a simple situation of identifying white-space-like characters for the purpose of marking the beginning and end of consecutive non-white-space characters.

Example 10-6. WordCnt() Using C and Byte-Scanning Technique

```
// Counting words involves locating the boundary of contiguous non-whitespace characters.
// Different software may choose its own mapping of white space character set.
// This example employs a simple definition for tutorial purpose:
// Non-whitespace character set will consider: A-Z, a-z, 0-9, and the apostrophe mark '
// The example uses a simple technique to map characters into bit patterns of square waves
// we can simply count the number of falling edges

static char alphnrange[16]= {0x27, 0x27, 0x30, 0x39, 0x41, 0x5a, 0x61, 0x7a, 0x0};
static char alp_map8[32] = {0x0, 0x0, 0x0, 0x0, 0x80,0x0,0xff, 0x3,0xfe, 0xff, 0xff, 0x7, 0xfe, 0xff, 0xff, 0x7}; // 32
byte lookup table, 1s map to bit patterns of alpha numerics in alphnrange
int wordcnt_c(const  char* s1)
{int  i, j, cnt = 0;
char cc, cc2;
char  flg[3]; // capture the a wavelet to locate a falling edge
   cc2 = cc = s1[0];
   // use the compacted bit pattern to consolidate multiple comparisons into one look up
   if( alp_map8[cc>>3] & ( 1<< ( cc & 7) ) )
   {   flg[1] = 1; } // non-white-space char that is part of a word,
              (continue)
```

Example 10-6. WordCnt() Using C and Byte-Scanning Technique (Contd.)

```
    // we're including apostrophe in this example since counting the
    // following 's' as a separate word would be kind of silly
    else
    {   flg[1] = 0; }  // 0: whitespace, punctuations not be considered as part of a word

    i = 1; // now we're ready to scan through the rest of the block
    // we'll try to pick out each falling edge of the bit pattern to increment word count.
    // this works with consecutive white spaces, dealing with punctuation marks, and
    // treating hyphens as connecting two separate words.
    while (cc2 )
    {   cc2 = s1[i];
        if( alp_map8[cc2>>3] & ( 1<< ( cc2 & 7) ) )
        {   flg[2] = 1;} // non-white-space
        else
        {   flg[2] = 0;} // white-space-like

        if( !flg[2] && flg[1] )
        {   cnt ++; }// found the falling edge
        flg[1] = flg[2];
        i++;
    }
    return cnt;
}
```

In Example 10-6, a 32-byte look-up table is constructed to represent the ascii code values 0x0-0xff, and partitioned with each bit of 1 corresponding to the specified subset of characters. While this bit-lookup technique simplifies the comparison operations, data fetching remains byte-granular.

Example 10-7 shows an equivalent implementation of counting words using PCMPISTRM. The loop iteration is performed at 16-byte granularity instead of byte granularity. Additionally, character set subsetting is easily expressed using range value pairs and parallel comparisons between the range values and each byte in the text fragment are performed by executing PCMPISTRI once.

Example 10-7. WordCnt() Using PCMPISTRM

```
// an SSE4.2 example of counting words using the definition of non-whitespace character
// set of {A-Z, a-z, 0-9, '}. Each text fragment (up to 16 bytes) are mapped to a
// 16-bit pattern, which may contain one or more falling edges. Scanning bit-by-bit
// would be inefficient and goes counter to leveraging SIMD programming techniques.
// Since each falling edge must have a preceding rising edge, we take a finite
// difference approach to derive a pattern where each rising/falling edge maps to 2-bit pulse,
// count the number of bits in the 2-bit pulses using popcnt and divide by two.
int wdcnt_sse4_2(const  char* s1)
{int len = 0;
  _asm{
  mov   eax, s1
  movdquxmm3, alphnrange ; load range value pairs to detect non-white-space codes
  xor    ecx, ecx
  xor    esi, esi
  xor    edx, edx
              (continue)
```

Example 10-7. WordCnt() Using PCMPISTRM (Contd.)

```
    movdquxmm1, [eax]
    pcmpistrm  xmm3, xmm1, 04h ; white-space-like char becomes 0 in xmm0[15:0]
    movdqa xmm4, xmm0
    movdqa     xmm1, xmm0
    psrld xmm4, 15    ; save MSB to use in next iteration
    movdqa xmm5, xmm1
    psllw xmm5, 1; lsb is effectively mapped to a white space
    pxor  xmm5, xmm0; the first edge is due to the artifact above
    pextrd edi, xmm5, 0
    jz       _lastfragment; if xmm1 had a null, zf would be set
    popcnt edi, edi; the first fragment will include a rising edge
    add      esi, edi
    mov   ecx, 16
_loopc:
    add    eax, ecx ; advance address pointer
    movdquxmm1, [eax]
    pcmpistrm  xmm3, xmm1, 04h ; white-space-like char becomes 0 in xmm0[15:0]
    movdqa xmm5, xmm4  ; retrieve the MSB of the mask from last iteration
    movdqa xmm4, xmm0
    psrld xmm4, 15    ; save mSB of this iteration for use in next iteration
    movdqa xmm1, xmm0

    psllw xmm1, 1
    por    xmm5, xmm1  ; combine MSB of last iter and the rest from current iter
    pxor   xmm5, xmm0; differentiate binary wave form into pattern of edges
    pextrdedi, xmm5, 0 ; the edge patterns has (1 bit from last, 15 bits from this round)
    jz       _lastfragment; if xmm1 had a null, zf would be set
    mov   ecx, 16; xmm1, had no null char, advance 16 bytes
    popcntedi, edi; count both rising and trailing edges
    add    esi, edi; keep a running count of both edges
    jmp    short _loopc
_lastfragment:
    popcntedi, edi; count both rising and trailing edges
    add    esi, edi; keep a running count of both edges
    shr    esi, 1 ; word count corresponds to the trailing edges
    mov   len, esi
    }
    return len;
}
```

10.3.3 Substring Searches

Strstr() is a common function in the standard string library. Typically, A library may implement strstr(sTarg, sRef) with a brute-force, byte-granular technique of iterative comparisons between the reference string with a round of string comparison with a subset of the target string. Brute-force, byte-granular techniques provide reasonable efficiency when the first character of the target substring and the reference string are different, allowing subsequent string comparisons of target substrings to proceed forward to the next byte in the target string.

When a string comparison encounters partial matches of several characters (i.e. the sub-string search found a partial match starting from the beginning of the reference string) and determined the partial match led to a false-match. The brute-force search process need to go backward and restart string comparisons from a location that had participated in previous string comparison operations. This is referred to as re-trace inefficiency of the brute-force substring search algorithm. See Figure 10-2.

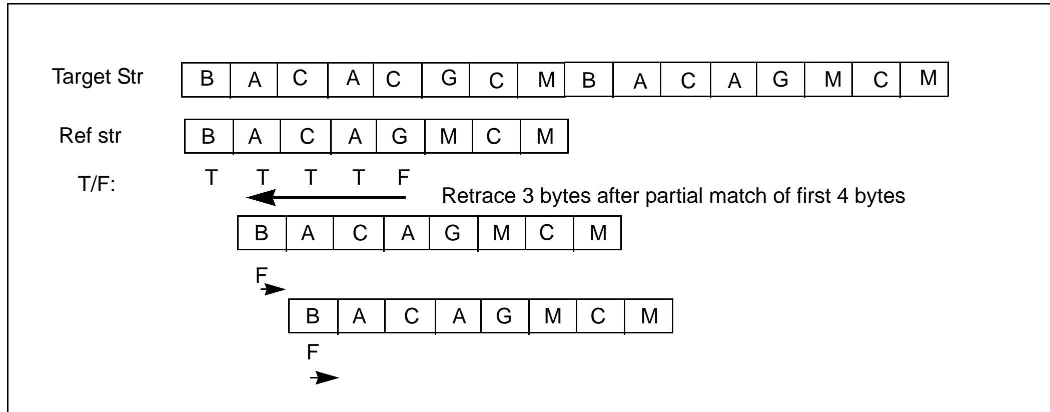


Figure 10-2. Retrace Inefficiency of Byte-Granular, Brute-Force Search

The Knuth, Morris, Pratt algorithm[1] (KMP) provides an elegant enhancement to overcome the re-trace inefficiency of brute-force substring searches. By deriving an overlap table that is used to manage retrace distance when a partial match leads to a false match, KMP algorithm is very useful for applications that search relevant articles containing keywords from a large corpus of documents.

Example 10-8 illustrates a C-code example of using KMP substring searches.

Example 10-8. KMP Substring Search in C

```
// s1 is the target string of length cnt1
// s2 is the reference string of length cnt2
// j is the offset in target string s1 to start each round of string comparison
// i is the offset in reference string s2 to perform byte granular comparison
                (continue)
```

```
int str_kmp_c(const  char* s1, int cnt1, const  char* s2, int cnt2 )
{ int i, j;
    i = 0; j = 0;
    while ( i+j < cnt1) {
        if( s2[i] == s1[i+j]) {
            i++;
            if( i == cnt2) break; // found full match
        }
        else {
            j = j+i - ovrlap_tbl[i]; // update the offset in s1 to start next round of string compare
            if( i > 0) {
                i = ovrlap_tbl[i]; // update the offset of s2 for next string compare should start at
            }
        }
    };
 return j;
}
```

1. Donald E. Knuth, James H. Morris, and Vaughan R. Pratt; SIAM J. Comput. Volume 6, Issue 2, pp. 323-350 (1977)

Example 10-8. KMP Substring Search in C (Contd.)

```
void kmp_precalc(const char * s2, int cnt2)
{int i = 2;
char nch = 0;
  ovrlap_tbl[0] = -1; ovrlap_tbl[1] = 0;
     // pre-calculate KMP table
  while( i < cnt2) {
     if( s2[i-1] == s2[nch]) {
        ovrlap_tbl[i] =  nch +1;
        i++; nch++;
     }
     else if ( nch > 0) nch = ovrlap_tbl[nch];
     else {
        ovrlap_tbl[i] = 0;
        i++;
     }
  };
  ovrlap_tbl[cnt2] = 0;
}
```

Example 10-8 also includes the calculation of the KMP overlap table. Typical usage of KMP algorithm involves multiple invocation of the same reference string, so the overhead of precalculating the overlap table is easily amortized. When a false match is determined at offset i of the reference string, the overlap table will predict where the next round of string comparison should start (updating the offset j), and the offset in the reference string that byte-granular character comparison should resume/restart.

While KMP algorithm provides efficiency improvement over brute-force byte-granular substring search, its best performance is still limited by the number of byte-granular operations. To demonstrate the versatility and built-in lexical capability of PCMPISTRI, we show an SSE4.2 implementation of substring search using brute-force 16-byte granular approach in Example 10-9, and combining KMP overlap table with substring search using PCMPISTRI in Example 10-10.

Example 10-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic

```
int strsubs_sse4_2i(const  char* s1, int cnt1, const  char* s2, int cnt2 )
{ int  kpm_i=0, idx;
int  ln1= 16, ln2=16, rcnt1 = cnt1, rcnt2= cnt2;
__m128i *p1 = (__m128i *) s1;
__m128i *p2 = (__m128i *) s2;
__m128ifrag1, frag2;
int  cmp, cmp2,  cmp_s;
__m128i *pt = NULL;
   if( cnt2 > cnt1 || !cnt1) return -1;
   frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
   frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment


            (continue)
```

Example 10-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic (Contd.)

```
while(rcnt1 > 0)
{    cmp_s = _mm_cmpestrs(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
     cmp = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
     if( !cmp) { // we have a partial match that needs further analysis
         if( cmp_s) { // if we're done with s2
             if( pt)
             {idx = (int) ((char *) pt - (char *) s1) ; }
             else
             {idx = (int) ((char *) p1 - (char *) s1) ;  }
             return idx;
         }
         // we do a round of string compare to verify full match till end of s2
         if( pt == NULL) pt = p1;
         cmp2 = 16;
         rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
         while(  cmp2 == 16  && rcnt2) {  // each 16B frag matches,
             rcnt1 = cnt1  - 16 -(int) ((char *)p1-(char *)s1);
             rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
             if( rcnt1 <=0 || rcnt2 <= 0 ) break;
             p1 = (__m128i *)(((char *)p1) + 16);
             p2 = (__m128i *)(((char *)p2) + 16);
             frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
             frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
             cmp2 = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x18); // lsb, eq each
         };
         if( !rcnt2 || rcnt2 == cmp2) {
             idx = (int) ((char *) pt - (char *) s1) ;
             return idx;
         }
         else if ( rcnt1 <= 0) { // also cmp2 < 16, non match
             if( cmp2 == 16 && ((rcnt1 + 16) >= (rcnt2+16) ) )
             {idx = (int) ((char *) pt - (char *) s1) ;
             return idx;
             }
             else return -1;
         }
             (continue)
```

Example 10-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic (Contd.)

```
            else { // in brute force, we advance fragment offset in target string s1 by 1
                p1 = (__m128i *)(((char *)pt) + 1);  // we're  not taking advantage of kmp
                rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
                pt = NULL;
                p2 = (__m128i *)((char *)s2) ;
                rcnt2 = cnt2  -(int) ((char *)p2-(char *)s2);
                frag1 = _mm_loadu_si128(p1);// load next fragment from s1
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
            }
        }
        else{
            if( cmp == 16) p1 = (__m128i *)(((char *)p1) + 16);
            else p1 = (__m128i *)(((char *)p1) + cmp);
            rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
            if( pt && cmp ) pt = NULL;
            frag1 = _mm_loadu_si128(p1);// load next fragment from s1
        }
    }
    return idx;
}
```

In Example 10-9, address adjustment using a constant to minimize loop-carry dependency is practised in two places:

- In the inner while loop of string comparison to determine full match or false match (the result cmp2 is not used for address adjustment to avoid dependency).
- In the last code block when the outer loop executed PCMPISTRI to compare 16 sets of ordered compare between a target fragment with the first 16-byte fragment of the reference string, and all 16 ordered compare operations produced false result (producing cmp with a value of 16).

Example 10-10 shows an equivalent intrinsic implementation of substring search using SSE4.2 and KMP overlap table. When the inner loop of string comparison determines a false match, the KMP overlap table is consulted to determine the address offset for the target string fragment and the reference string fragment to minimize retrace.

It should be noted that a significant portions of retrace with retrace distance less than 15 bytes are avoided even in the brute-force SSE4.2 implementation of Example 10-9. This is due to the order-compare primitive of PCMPISTRI. "Ordered compare" performs 16 sets of string fragment compare, and many false match with less than 15 bytes of partial matches can be filtered out in the same iteration that executed PCMPISTRI.

Retrace distance of greater than 15 bytes does not get filtered out by the Example 10-9. By consulting with the KMP overlap table, Example 10-10 can eliminate retraces of greater than 15 bytes.

Example 10-10. Substring Search Using PCMPISTRI and KMP Overlap Table

```
int strkmp_sse4_2(const  char* s1, int cnt1, const  char* s2, int cnt2 )
{ int  kpm_i=0, idx;
int  ln1= 16, ln2=16, rcnt1 = cnt1, rcnt2= cnt2;
__m128i *p1 = (__m128i *) s1;
__m128i *p2 = (__m128i *) s2;
__m128ifrag1, frag2;
                (continue)
```

Example 10-10. Substring Search Using PCMPISTRI and KMP Overlap Table (Contd.)

```
int  cmp, cmp2,  cmp_s;
__m128i *pt = NULL;
    if( cnt2 > cnt1 || !cnt1) return -1;
    frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
    frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment

    while(rcnt1 > 0)
    {    cmp_s = _mm_cmpestrs(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
        cmp = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
        if( !cmp) { // we have a partial match that needs further analysis
            if( cmp_s) { // if we've reached the end with s2
                if( pt)
                {idx = (int) ((char *) pt - (char *) s1) ; }
                else
                {idx = (int) ((char *) p1 - (char *) s1) ;  }
                return idx;
            }
        // we do a round of string compare to verify full match till end of s2
            if( pt == NULL) pt = p1;
            cmp2 = 16;
            rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
            while(  cmp2 == 16 && rcnt2) {  // each 16B frag matches
                rcnt1 = cnt1  - 16 -(int) ((char *)p1-(char *)s1);
                rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
                if( rcnt1 <=0 || rcnt2 <= 0 ) break;
                p1 = (__m128i *)(((char *)p1) + 16);
                p2 = (__m128i *)(((char *)p2) + 16);
                frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
                cmp2 = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x18); // lsb, eq each
            };
            if( !rcnt2 || rcnt2 == cmp2) {
                idx = (int) ((char *) pt - (char *) s1) ;
                return idx;
            }
            else if ( rcnt1 <= 0 ) { // also cmp2 < 16, non match
                return -1;
            }
                (continue)
```

Example 10-10. Substring Search Using PCMPISTRI and KMP Overlap Table (Contd.)

```
            else { // a partial match led to false match, consult KMP overlap table for addr adjustment
                kpm_i = (int) ((char *)p1 - (char *)pt)+ cmp2 ;
                p1 = (__m128i *)(((char *)pt) + (kpm_i - ovrlap_tbl[kpm_i])); // use kmp to skip retrace
                rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
                pt = NULL;
                p2 = (__m128i *)(((char *)s2) + (ovrlap_tbl[kpm_i]));
                rcnt2 = cnt2  -(int) ((char *)p2-(char *)s2);
                frag1 = _mm_loadu_si128(p1);// load next fragment from s1
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
            }
        }
        else{
                if( kpm_i && ovrlap_tbl[kpm_i]) {
                p2 = (__m128i *)(((char *)s2) );
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
                //p1 = (__m128i *)(((char *)p1) );

                //rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
                if( pt && cmp ) pt = NULL;
                rcnt2 = cnt2  ;
                //frag1 = _mm_loadu_si128(p1);// load next fragment from s1
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
                kpm_i = 0;
                }
                else { // equ order comp resulted in sub-frag match or non-match
                if( cmp == 16) p1 = (__m128i *)(((char *)p1) + 16);
                else p1 = (__m128i *)(((char *)p1) + cmp);
                rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
                if( pt && cmp ) pt = NULL;
                frag1 = _mm_loadu_si128(p1);// load next fragment from s1
                }
        }
    }
    return idx;
}
```

The relative speed up of byte-granular KMP, brute-force SSE4.2, and SSE4.2 with KMP overlap table over byte-granular brute-force substring search is illustrated in the graph that plots relative speedup over percentage of retrace for a reference string of 55 bytes long. A retrace of 40% in the graph meant, after a partial match of the first 22 characters, a false match is determined.

So when brute-force, byte-granular code has to retrace, the other three implementation may be able to avoid the need to retrace because:

- Example 10-8 can use KMP overlap table to predict the start offset of next round of string compare operation after a partial-match/false-match, but forward movement after a first-character-false-match is still byte-granular.

- Example 10-9 can avoid retrace of shorter than 15 bytes but will be subject to retrace of 21 bytes after a partial-match/false-match at byte 22 of the reference string. Forward movement after each order-compare-false-match is 16 byte granular.
- Example 10-10 avoids retrace of 21 bytes after a partial-match/false-match, but KMP overlap table lookup incurs some overhead. Forward movement after each order-compare-false-match is 16 byte granular.

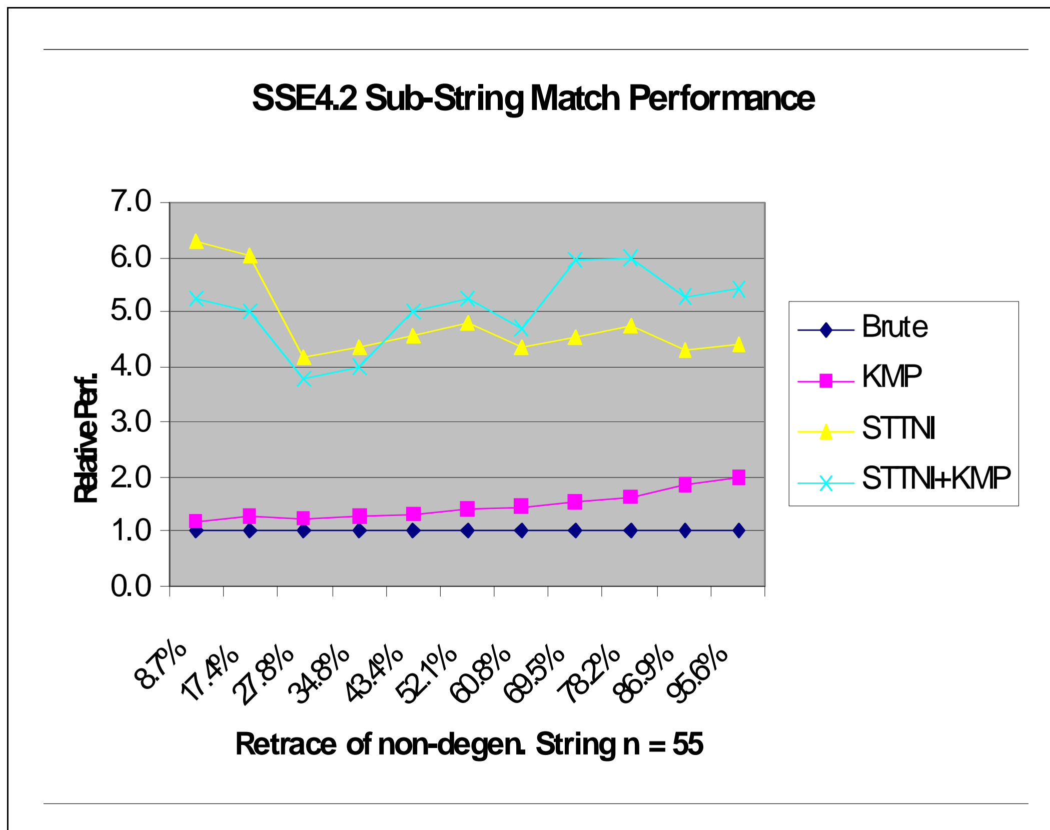


Figure 10-3. SSE4.2 Speedup of SubString Searches

10.3.4 String Token Extraction and Case Handling

Token extraction is a common task in text/string handling. It is one of the foundation of implementing lexer/parser objects of higher sophistication. Indexing services also build on tokenization primitives to sort text data from streams.

Tokenization requires the flexibility to use an array of delimiter characters.

A library implementation of Strtok_s() may employ a table-lookup technique to consolidate sequential comparisons of the delimiter characters into one comparison (similar to Example 10-6). An SSE4.2 implementation of the equivalent functionality of strtok_s() using intrinsic is shown in Example 10-11.

Example 10-11. I Equivalent Strtok_s() Using PCMPISTRI Intrinsic

```
char ws_map8[32]; // packed bit lookup table for delimiter characters

char * strtok_sse4_2i(char* s1, char *sdlm, char ** pCtxt)
{
__m128i *p1 = (__m128i *) s1;
__m128ifrag1, stmpz, stmp1;
int cmp_z, jj =0;
int start, endtok, s_idx, ldx;
   if (sdlm == NULL || pCtxt == NULL) return NULL;
   if( p1 == NULL && *pCtxt == NULL) return NULL;
   if( s1 == NULL)  {
       if( *pCtxt[0] == 0 ) { return NULL; }
       p1 = (__m128i *) *pCtxt;
       s1 = *pCtxt;
   }
   else p1 = (__m128i *) s1;
   memset(&ws_map8[0], 0, 32);
   while (sdlm[jj] ) {
       ws_map8[ (sdlm[jj] >> 3) ] |= (1 << (sdlm[jj] & 7) ); jj ++
   }
   frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
   stmpz  = _mm_loadu_si128((__m128i *)sdelimiter);
   // if the first char is not a delimiter , proceed to check non-delimiter,
   // otherwise need to skip leading delimiter chars
   if( ws_map8[s1[0]>>3] & (1 <<  (s1[0]&7)) ) {
    start = s_idx = _mm_cmpistri(stmpz, frag1, 0x10);// unsigned bytes/equal any, invert, lsb
   }
   else start = s_idx = 0;
```

```
   // check if we're dealing with short input string less than 16 bytes
   cmp_z = _mm_cmpistrz(stmpz, frag1, 0x10);
   if( cmp_z) {  // last fragment
       if( !start) {
           endtok = ldx = _mm_cmpistri(stmpz, frag1, 0x00);
           if( endtok == 16) { // didn't find delimiter at the end, since it's null-terminated
               // find where is the null byte
               *pCtxt = s1+ 1+ _mm_cmpistri(frag1, frag1, 0x40);
               return s1;
           }
           else { // found a delimiter that ends this word
           s1[start+endtok] = 0;
           *pCtxt = s1+start+endtok+1;
           }
       }
             (continue)
```

Example 10-11. I Equivalent Strtok_s() Using PCMPISTRI Intrinsic (Contd.)

```
        else {
            if(!s1[start] ) {
                *pCtxt = s1 + start +1;
                return NULL;
            }
            p1 = (__m128i *)(((char *)p1) + start);
            frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
            endtok = ldx = _mm_cmpistri(stmpz, frag1, 0x00);// unsigned bytes/equal any, lsb
            if( endtok == 16) { // looking for delimiter, found none
                *pCtxt = (char *)p1 + 1+ _mm_cmpistri(frag1, frag1, 0x40);
                return s1+start;
            }
            else { // found delimiter before null byte
                s1[start+endtok] = 0;
                *pCtxt = s1+start+endtok+1;
            }
        }
    }
```

```
    else
    {   while ( !cmp_z && s_idx == 16) {
            p1 = (__m128i *)(((char *)p1) + 16);
            frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
            s_idx = _mm_cmpistri(stmpz, frag1, 0x10);// unsigned bytes/equal any, invert, lsb
            cmp_z = _mm_cmpistrz(stmpz, frag1, 0x10);
        }
        if(s_idx != 16) start = ((char *) p1  -s1) + s_idx;
        else { // corner case if we ran to the end looking for delimiter and never found a non-dilimiter
            *pCtxt = (char *)p1  +1+ _mm_cmpistri(frag1, frag1, 0x40);
            return NULL;
        }
        if( !s1[start] ) { // in case a null byte follows delimiter chars
            *pCtxt = s1 + start+1;
            return NULL;
        }
        // now proceed to find how many non-delimiters are there
        p1 = (__m128i *)(((char *)p1) + s_idx);
        frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
        endtok = ldx = _mm_cmpistri(stmpz, frag1, 0x00);// unsigned bytes/equal any, lsb
        cmp_z = 0;
        while ( !cmp_z && ldx == 16) {
            p1 = (__m128i *)(((char *)p1) + 16);
            frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
            ldx = _mm_cmpistri(stmpz, frag1, 0x00);// unsigned bytes/equal any, lsb
            cmp_z = _mm_cmpistrz(stmpz, frag1, 0x00);
            if(cmp_z) { endtok += ldx; }
        }
            (continue)
```

Example 10-11. I Equivalent Strtok_s() Using PCMPISTRI Intrinsic (Contd.)

```
      if( cmp_z ) { // reached the end of s1
         if(  ldx < 16) // end of word found by finding a delimiter
            endtok += ldx;
         else {  // end of word found by finding the null
            if( s1[start+endtok]) // ensure this frag don't start with null byte
            endtok += 1+ _mm_cmpistri(frag1, frag1, 0x40);
         }
      }
      *pCtxt = s1+start+endtok+1;
      s1[start+endtok] = 0;
   }
   return (char *) (s1+ start);
}
```

An SSE4.2 implementation of the equivalent functionality of strupr() using intrinsic is shown in Example 10-12.

Example 10-12. I Equivalent Strupr() Using PCMPISTRM Intrinsic

```
static char uldelta[16]= {0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
0x20, 0x20};
static char ranglc[6]= {0x61, 0x7a, 0x00, 0x00, 0x00, 0x00};
char * strup_sse4_2i(  char* s1)
{int len = 0, res = 0;
__m128i *p1 = (__m128i *) s1;
__m128ifrag1, ranglo, rmsk, stmpz, stmp1;
int cmp_c, cmp_z, cmp_s;
   if( !s1[0]) return (char *) s1;
   frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
   ranglo = _mm_loadu_si128((__m128i *)ranglc);// load up to 16 bytes of fragment
   stmpz  = _mm_loadu_si128((__m128i *)uldelta);
   cmp_z = _mm_cmpistrz(ranglo, frag1, 0x44);// range compare, produce byte masks
   while (!cmp_z)
   {
      rmsk = _mm_cmpistrm(ranglo, frag1, 0x44); // producing byte mask
      stmp1 = _mm_blendv_epi8(stmpz, frag1, rmsk);  // bytes of lc preserved, other bytes replaced by const
      stmp1 =_mm_sub_epi8(stmp1, stmpz);  // bytes of lc becomes uc, other bytes are now zero
      stmp1 = _mm_blendv_epi8(frag1, stmp1, rmsk); //bytes of lc replaced by uc, other bytes unchanged
      _mm_storeu_si128(p1, stmp1);//
      p1 = (__m128i *)(((char *)p1) + 16);
      frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
      cmp_z = _mm_cmpistrz(ranglo, frag1, 0x44);
   }
            (continue)
```

Example 10-12. I Equivalent Strupr() Using PCMPISTRM Intrinsic (Contd.)

```
    if( *(char *)p1 == 0) return (char *) s1;
    rmsk = _mm_cmpistrm(ranglo, frag1, 0x44);// byte mask, valid lc bytes are 1, all other 0
    stmp1 = _mm_blendv_epi8(stmpz, frag1, rmsk);  // bytes of lc continue, other bytes replaced by const
    stmp1 =_mm_sub_epi8(stmp1, stmpz);  // bytes of lc becomes uc, other bytes are now zero
    stmp1 = _mm_blendv_epi8(frag1, stmp1, rmsk); //bytes of lc replaced by uc, other bytes unchanged
    rmsk = _mm_cmpistrm(frag1, frag1, 0x44);// byte mask, valid  bytes are 1, invalid bytes are zero
    _mm_maskmoveu_si128(stmp1, rmsk, (char *) p1);//
    return (char *) s1;
}
```

10.3.5 Unicode Processing and PCMPxSTRy

Unicode representation of string/text data is required for software localization. UTF-16 is a common encoding scheme for localized content. In UTF-16 representation, each character is represented by a code point. There are two classes of code points: 16-bit code points and 32-bit code points which consists of a pair of 16-bit code points in specified value range, the latter is also referred to as a surrogate pair.

A common technique in unicode processing uses a table-loop up method, which has the benefit of reduced branching. As a tutorial example we compare the analogous problem of determining properly-encoded UTF-16 string length using general purpose code with table-lookup vs. SSE4.2.

Example 10-13 lists the C code sequence to determine the number of properly-encoded UTF-16 code points (either 16-bit or 32-bit code points) in a unicode text block. The code also verifies if there are any improperly-encoded surrogate pairs in the text block.

Example 10-13. UTF16 VerStrlen() Using C and Table Lookup Technique

```
// This example demonstrates validation of surrogate pairs (32-bit code point) and
// tally the number of16-bit and 32-bit code points in the text block
// Parameters: s1 is pointer to input utf-16 text block.
// pLen: store count of utf-16 code points
// return the number of 16-bit code point encoded in the surrogate range but do not form
// a properly encoded surrogate pair. if 0: s1 is a properly encoded utf-16 block,
// If return value >0 then s1 contains invalid encoding of surrogates

int u16vstrlen_c(const  short* s1, unsigned * pLen)
{int  i, j, cnt = 0, cnt_invl = 0, spcnt= 0;
unsigned short cc, cc2;
char  flg[3];

 cc2 = cc = s1[0];
 // map each word in s1into bit patterns of 0, 1or 2 using a table lookup
// the first half of a surrogate pair must be encoded between D800-DBFF and mapped as 2
// the 2nd half of a surrogate pair must be encoded between DC00-DFFF and mapped as 1
// regular 16-bit encodings are mapped to 0, except null code mapped to 3
flg[1] = utf16map[cc];
 flg[0] = flg[1];
 if(!flg[1]) cnt ++;
 i = 1;
                (continue)
```

Example 10-13. UTF16 VerStrlen() Using C and Table Lookup Technique (Contd.)

```
 while (cc2 ) // examine each non-null word encoding
 { cc2 = s1[i];
   flg[2] = utf16map[cc2];
   if( (flg[1] && flg[2] && (flg[1]-flg[2] == 1) )  )
   {   spcnt ++; }// found a surrogate pair
   else if(flg[1] == 2 && flg[2] != 1)
   {  cnt_invl += 1; } // orphaned 1st half
   else if( !flg[1]  && flg[2] == 1)
   {  cnt_invl += 1; } // orphaned 2nd half
   else
   {   if(!flg[2]) cnt ++;// regular non-null code16-bit code point
       else ;
   }
   flg[0] = flg[1];// save the pair sequence for next iteration
   flg[1] = flg[2];
   i++;
 }
 *pLen = cnt + spcnt;
 return cnt_invl;
}
```

The VerStrlen() function for UTF-16 encoded text block can be implemented using SSE4.2.

Example 10-14 shows the listing of SSE4.2 assembly implementation and Example 10-15 shows the listing of SSE4.2 intrinsic listings of VerStrlen().

Example 10-14. Assembly Listings of UTF16 VerStrlen() Using PCMPISTRI

```
// complementary range values for detecting either halves of 32-bit UTF-16 code point
static short ssch0[16]= {0x1, 0xd7ff, 0xe000, 0xffff, 0, 0};
// complementary range values for detecting the 1st half of 32-bit UTF-16 code point
static short ssch1[16]= {0x1, 0xd7ff, 0xdc00, 0xffff, 0, 0};
// complementary range values for detecting the 2nd half of 32-bit UTF-16 code point
static short ssch2[16]= {0x1, 0xdbff, 0xe000, 0xffff, 0, 0};


int utf16slen_sse4_2a(const  short* s1, unsigned * pLen)
{int len = 0, res = 0;
   _asm{
   mov   eax, s1
   movdquxmm2, ssch0 ; load range value to identify either halves
   movdquxmm3, ssch1 ; load range value to identify 1st half (0xd800 to 0xdbff)
   movdquxmm4, ssch2 ; load range value to identify 2nd half (0xdc00 to 0xdfff)
   xor    ecx, ecx
   xor    edx, edx; store # of 32-bit code points (surrogate pairs)
   xor    ebx, ebx; store # of non-null 16-bit code points
   xor    edi, edi ; store # of invalid word encodings

                (continue)
```

Example 10-14. Assembly Listings of UTF16 VerStrlen() Using PCMPISTRI (Contd.)

```
_loopc:
   shl    ecx, 1; pcmpistri with word processing return ecx in word granularity, multiply by 2 to get byte offset
   add    eax, ecx
   movdquxmm1, [eax] ; load a string fragment of up to 8 words
   pcmpistri  xmm2, xmm1, 15h; unsigned words, ranges, invert, lsb index returned to ecx
       ; if there is a utf-16 null wchar in xmm1, zf will be set.
       ; if all 8 words in the comparison matched range,
       ;  none of bits in the intermediate result will be set after polarity inversions,
       ;  and ECX will return with a value of 8
   jz     short _lstfrag; if null code, handle last fragment
   ; if ecx < 8, ecx point to a word of either 1st or 2nd half of a 32-bit code point
   cmp    ecx, 8
   jne    _chksp
   add    ebx, ecx   ; accumulate # of 16-bit non-null code points
   mov    ecx, 8 ; ecx must be 8 at this point, we want to avoid loop carry dependency
   jmp    _loopc

_chksp:; this fragment has word encodings in the surrogate value range
   add    ebx, ecx ; account for the 16-bit code points
   shl    ecx, 1; pcmpistri with word processing return ecx in word granularity, multiply by 2 to get byte offset
   add    eax, ecx
   movdquxmm1, [eax] ; ensure the fragment start with word encoding in either half
   pcmpistri  xmm3, xmm1, 15h; unsigned words, ranges, invert, lsb index returned to ecx
   jz     short _lstfrag2; if null code, handle the last fragment
   cmp    ecx, 0 ; properly encoded 32-bit code point must start with 1st half
   jg     _invalidsp; some invalid s-p code point exists in the fragment
   pcmpistri  xmm4, xmm1, 15h; unsigned words, ranges, invert, lsb index returned to ecx
   cmp    ecx, 1 ; the 2nd half must follow the first half
   jne    _invalidsp
   add    edx, 1; accumulate # of valid surrogate pairs
   add    ecx, 1 ; we want to advance two words
   jmp    _loopc
_invalidsp:; the first word of this fragment is either the 2nd half or an un-paired 1st half
   add    edi, 1  ; we have an invalid code point (not a surrogate pair)
   mov    ecx, 1 ; advance one word and continue scan for 32-bit code points
   jmp    _loopc
_lstfrag:
   add    ebx, ecx ; account for the non-null 16-bit code points
_morept:
   shl    ecx, 1; pcmpistri with word processing return ecx in word granularity, multiply by 2 to get byte offset
   add    eax, ecx
   mov    si, [eax] ; need to check for null code
   cmp    si, 0
   je     _final
   movdquxmm1, [eax] ; load remaining word elements which start with either 1st/2nd half
   pcmpistri  xmm3, xmm1, 15h; unsigned words, ranges, invert, lsb index returned to ecx
_lstfrag2:
   cmp    ecx, 0 ; a valid 32-bit code point must start from 1st half
   jne    _invalidsp2
   pcmpistri  xmm4, xmm1, 15h; unsigned words, ranges, invert, lsb index returned to ecx
   cmp    ecx, 1
   jne    _invalidsp2
               (continue)
```

Example 10-14. Assembly Listings of UTF16 VerStrlen() Using PCMPISTRI (Contd.)

```
   add   edx, 1
   mov   ecx, 2
   jmp   _morept
_invalidsp2:
   add   edi, 1
   mov   ecx, 1
   jmp   _morept
_final:
   add   edx, ebx; add # of 16-bit and 32-bit code points
   mov   ecx, pLen; retrieve address of pointer provided by caller
   mov   [ecx], edx; store result of string length to memory
   mov   res, edi
   }
   return res;
}
```

Example 10-15. Intrinsic Listings of UTF16 VerStrlen() Using PCMPISTRI

```
int utf16slen_i(const  short* s1, unsigned * pLen)
{int len = 0, res = 0;
__m128i *pF = (__m128i *) s1;
__m128iu32 =_mm_loadu_si128((__m128i *)ssch0);
__m128i  u32a =_mm_loadu_si128((__m128i *)ssch1);
__m128i  u32b =_mm_loadu_si128((__m128i *)ssch2);
__m128ifrag1;
int offset1 = 0, cmp, cmp_1, cmp_2;
intcnt_16 = 0, cnt_sp=0, cnt_invl= 0;
short *ps;
while (1) {
      pF = (__m128i *)(((short *)pF) + offset1);
      frag1 = _mm_loadu_si128(pF);// load up to 8 words
      // does frag1 contain either halves of a 32-bit UTF-16 code point?
      cmp = _mm_cmpistri(u32, frag1, 0x15);// unsigned bytes, equal order, lsb index returned to ecx

      if (_mm_cmpistrz(u32, frag1, 0x15))// there is a null code in frag1
      {   cnt_16 += cmp;
          ps = (((short *)pF) + cmp);
          while (ps[0])
          {   frag1 = _mm_loadu_si128( (__m128i *)ps);
              cmp_1 = _mm_cmpistri(u32a, frag1, 0x15);
              if(!cmp_1)
              {   cmp_2 = _mm_cmpistri(u32b, frag1, 0x15);
                  if( cmp_2 ==1) { cnt_sp++; offset1 = 2;}
                  else {cnt_invl++;  offset1= 1;}
              }
               (continue)
```

Example 10-15. Intrinsic Listings of UTF16 VerStrlen() Using PCMPISTRI (Contd.)

```
                else
                {   cmp_2 = _mm_cmpistri(u32b, frag1, 0x15);
                    if(!cmp_2) {cnt_invl ++; offset1 = 1;}
                    else {cnt_16 ++; offset1 = 1; }
                }
                ps = (((short *)ps) + offset1);
            }
            break;
        }
```

```
        if(cmp != 8) // we have at least some half of 32-bit utf-16 code points
        {   cnt_16 += cmp; // regular 16-bit UTF16 code points
            pF = (__m128i *)(((short *)pF) + cmp);
            frag1 = _mm_loadu_si128(pF);
            cmp_1 = _mm_cmpistri(u32a, frag1, 0x15);
            if(!cmp_1)
            {   cmp_2 = _mm_cmpistri(u32b, frag1, 0x15);
                if( cmp_2 ==1) { cnt_sp++; offset1 = 2;}
                else {cnt_invl++;  offset1= 1;}
            }
            else
            {   cnt_invl ++;
                offset1 = 1;
            }
        }
        else {
            offset1 = 8; // increment address by 16 bytes to handle next fragment
            cnt_16+= 8;
        }
};
        *pLen = cnt_16 + cnt_sp;
        return cnt_invl;
}
```

10.3.6 Replacement String Library Function Using SSE4.2

Unaligned 128-bit SIMD memory access can fetch data cross page boundary, since system software manages memory access rights with page granularity.

Implementing a replacement string library function using SIMD instructions must not cause memory access violation. This requirement can be met by adding a small amounts of code to check the memory address of each string fragment. If a memory address is found to be within 16 bytes of crossing over to the next page boundary, string processing algorithm can fall back to byte-granular technique.

Example 10-16 shows an SSE4.2 implementation of strcmp() that can replace byte-granular implementation supplied by standard tools.

Example 10-16. Replacement String Library Strcmp Using SSE4.2

```
// return 0 if strings are equal, 1 if greater, -1 if less
int strcmp_sse4_2(const char *src1, const char *src2)
{
    int val;
    __asm{
          mov         esi, src1 ;
          mov         edi, src2
          mov         edx, -16 ; common index relative to base of either string pointer
          xor         eax, eax
topofloop:
          add         edx, 16 ; prevent loop carry dependency
next:
          lea         ecx, [esi+edx] ; address of fragment that we want to load
          and         ecx, 0x0fff ; check least significant12 bits of addr for page boundary
          cmp         ecx, 0x0ff0
          jg          too_close_pgb ; branch to byte-granular if within 16 bytes of boundary
          lea         ecx, [edi+edx] ; do the same check for each fragment of 2nd string
          and         ecx, 0x0fff
          cmp         ecx, 0x0ff0
          jg          too_close_pgb
          movdqu      xmm2, BYTE PTR[esi+edx]
          movdqu      xmm1, BYTE PTR[edi+edx]
          pcmpistri   xmm2, xmm1, 0x18 ; equal each
          ja          topofloop
          jnc         ret_tag
          add         edx, ecx ; ecx points to the byte offset that differ
not_equal:
          movzx       eax, BYTE PTR[esi+edx]
          movzx       edx, BYTE PTR[edi+edx]
          cmp         eax, edx
          cmova       eax, ONE
          cmovb       eax, NEG_ONE
          jmp         ret_tag
too_close_pgb:
          add         edx, 1 ; do byte granular compare
          movzx       ecx, BYTE PTR[esi+edx-1]
          movzx       ebx, BYTE PTR[edi+edx-1]
          cmp         ecx, ebx
          jne         inequality
          add         ebx, ecx
          jnz         next
          jmp         ret_tag
inequality:
          cmovb       eax, NEG_ONE
          cmova       eax, ONE
              (continue)
```

Example 10-16. Replacement String Library Strcmp Using SSE4.2 (Contd.)

```
ret_tag:
        mov         [val], eax
    }
    return(val);
}
```

In Example 10-16, 8 instructions were added following the label "next" to perform 4KByte boundary checking of address that will be used to load two string fragments into registers. If either address is found to be within 16 bytes of crossing over to the next page, the code branches to byte-granular comparison path following the label "too_close_pgb".

The return values of Example 10-16 uses the convention of returning 0, +1, -1 using CMOV. It is straight forward to modify a few instructions to implement the convention of returning 0, positive integer, negative integer.

10.4 SSE4.2 ENABLED NUMERICAL AND LEXICAL COMPUTATION

SSE4.2 can enable SIMD programming techniques to explore byte-granular computational problems that were considered unlikely candidates for using SIMD instructions. We consider a common library function atol() in its full 64-bit flavor of converting a sequence of alpha numerical characters within the range representable by the data type __int64.

There are several attributes of this string-to-integer problem that poses as difficult challenges for using prior SIMD instruction sets (before the introduction of SSE4.2) to accelerate the numerical computation aspect of string-to-integer conversions:

- Character subset validation: Each character in the input stream must be validated with respect to the character subset definitions and conform to data representation rules of white space, signs, numerical digits. SSE4.2 provides the perfect tools for character subset validation.
- State-dependent nature of character validation: While SIMD computation instructions can expedite the arithmetic operations of "multiply by 10 and add", the arithmetic computation requires the input byte stream to consist of numerical digits only. For example, the validation of numerical digits, white-space, and the presence/absence of sign, must be validated in mid-stream. The flexibility of the SSE4.2 primitive can handle these state-dependent validation well.
- Additionally, exit condition to wrap up arithmetic computation can happen in mid-stream due to invalid characters, or due to finite representable range of the data type (~10^19 for int64, no more than 10 non-zero-leading digits for int32) may lead one to believe this type data stream consisting of short bursts are not suited for exploring SIMD ISA and be content with byte-granular solutions.

Because of the character subset validation and state-dependent nature, byte-granular solutions of the standard library function tends to have a high start-up cost (for example, converting a single numerical digit to integer may take 50 or 60 cycles), and low throughput (each additional numeric digit in the input character stream may take 6-8 cycles per byte).

A high level pseudo-operation flow of implementing a library replacement of atol() is described in Example 10-17.

Example 10-17. High-level flow of Character Subset Validation for String Conversion

1. Check Early_Out Exit Conditions (e.g. first byte is not valid).
2. Check if 1st byte is white space and skip any additional leading white space.
3. Check for the presence of a sign byte.
4. Check the validity of the remaining byte stream if they are numeric digits.
5. If the byte stream starts with '0', skip all leading digits that are '0'.
6. Determine how many valid non-zero-leading numeric digits.
7. Convert up to 16 non-zero-leading digits to int64 value.
8. load up to the next 16 bytes safely and check for consecutive numeric digits
9. Normalize int64 value converted from the first 16 digits, according to # of remaining digits,
10. Check for out-of-bound results of normalized intermediate int64 value,
11. Convert remaining digits to int64 value and add to normalized intermediate result,
12. Check for out-of-bound final results.

Example 10-18 shows the code listing of an equivalent functionality of atol() capable of producing int64 output range. Auxiliary function and data constants are listed in Example 10-19.

Example 10-18. Intrinsic Listings of atol() Replacement Using PCMPISTRI

```
__int64 sse4i_atol(const  char* s1)
{char  *p = ( char *) s1;
  int NegSgn = 0;
  __m128i mask0;
  __m128i  value0, value1;
  __m128i  w1, w1_l8, w1_u8, w2, w3 = _mm_setzero_si128();
  __int64 xxi;
  int index, cflag, sflag, zflag, oob=0;
// check the first character is valid via lookup
if ( (BtMLValDecInt[ *p >> 3] & (1 << ((*p) & 7)) ) == 0) return 0;
// if the first character is white space, skip remaining white spaces
if (BtMLws[*p >>3] & (1 <<((*p) & 7)) )
{  p ++;
   value0 = _mm_loadu_si128 ((__m128i *) listws);
skip_more_ws:
     mask0 = __m128i_strloadu_page_boundary (p);
     /* look for the 1st non-white space character */
     index = _mm_cmpistri (value0, mask0,  0x10);
     cflag = _mm_cmpistrc (value0, mask0,   0x10);
     sflag = _mm_cmpistrs (value0, mask0,   0x10);
     if( !sflag && !cflag)
     {  p = ( char *) ((size_t) p + 16);
        goto skip_more_ws;
     }
     else        p = (  char *) ((size_t) p + index);
}
                                                     (continue)
```

Example 10-18. Intrinsic Listings of atol() Replacement Using PCMPISTRI (Contd.)

```
if( *p == '-')
{  p++;
   NegSgn = 1;
}
else if( *p == '+') p++;

/* load up to 16 byte safely and check how many valid numeric digits we can do SIMD */
value0 = _mm_loadu_si128 ((__m128i *) rangenumint);
mask0 = __m128i_strloadu_page_boundary (p);
index = _mm_cmpistri (value0, mask0,  0x14);
zflag = _mm_cmpistrz (value0, mask0,  0x14);
```

```
/* index points to the first digit that is not a valid numeric digit */
if( !index) return 0;
else if (index == 16)
{  if( *p == '0') /* if all 16 bytes are numeric digits */
   {  /* skip leading zero */
      value1 = _mm_loadu_si128 ((__m128i *) rangenumintzr);
      index = _mm_cmpistri (value1, mask0,  0x14);
      zflag = _mm_cmpistrz (value1, mask0,  0x14);
      while(index == 16 && !zflag )
      {  p = ( char *) ((size_t) p + 16);
         mask0 = __m128i_strloadu_page_boundary (p);
         index = _mm_cmpistri (value1, mask0,  0x14);
         zflag = _mm_cmpistrz (value1, mask0,  0x14);
      }
      /* now the 1st digit is non-zero, load up to 16 bytes and update index  */
      if( index < 16)
         p = ( char *) ((size_t) p + index);
      /* load up to 16 bytes of non-zero leading numeric digits */
      mask0 = __m128i_strloadu_page_boundary (p);
      /* update index to point to non-numeric character or indicate we may have more than 16 bytes */
      index = _mm_cmpistri (value0, mask0,  0x14);
   }
}
if( index == 0)  return 0;
else if( index == 1)   return (NegSgn? (long long) -(p[0]-48): (long long) (p[0]-48));
// Input digits in xmm are ordered in reverse order. the LS digit of output is next to eos
// least sig numeric digit aligned to byte 15 , and subtract 0x30 from each ascii code
mask0  = ShfLAlnLSByte( mask0, 16 -index);
w1_u8 = _mm_slli_si128 ( mask0, 1);
w1 = _mm_add_epi8( mask0, _mm_slli_epi16 (w1_u8, 3)); /* mul by 8  and add */
w1 = _mm_add_epi8( w1, _mm_slli_epi16 (w1_u8, 1)); /* 7 LS bits per byte, in bytes 0, 2, 4, 6, 8, 10, 12, 14*/
w1 = _mm_srli_epi16( w1, 8);  /* clear out upper bits of each wd*/
w2 = _mm_madd_epi16(w1, _mm_loadu_si128(  (__m128i *) &MulplyPairBaseP2[0]) ); /* multiply base^2, add adjacent
word,*/
w1_u8 = _mm_packus_epi32 ( w2, w2);  /* pack 4 low word of each dword into 63:0 */
w1 = _mm_madd_epi16(w1_u8, _mm_loadu_si128(  (__m128i *) &MulplyPairBaseP4[0]) ); /* multiply base^4, add
adjacent word,*/
w1 = _mm_cvtepu32_epi64( w1);  /*   converted dw was in 63:0, expand to qw */
w1_l8 = _mm_mul_epu32(w1, _mm_setr_epi32(  100000000, 0, 0, 0) );
w2 = _mm_add_epi64(w1_l8, _mm_srli_si128 (w1, 8) );

            (continue)
```

Example 10-18. Intrinsic Listings of atol() Replacement Using PCMPISTRI (Contd.)

```
if( index < 16)
{  xxi = _mm_extract_epi64(w2, 0);
   return (NegSgn? (long long) -xxi: (long long) xxi);
}
/* 64-bit integer allow up to 20 non-zero-leading digits. */
/* accumulate each 16-digit fragment*/
w3 = _mm_add_epi64(w3, w2);
/* handle next batch of up to 16 digits, 64-bit integer only allow 4 more digits */
p = ( char *) ((size_t) p + 16);
if( *p == 0)
{  xxi = _mm_extract_epi64(w2, 0);
   return (NegSgn? (long long) -xxi: (long long) xxi);
}
mask0 = __m128i_strloadu_page_boundary (p);
/* index points to first non-numeric digit */
index = _mm_cmpistri (value0, mask0,  0x14);
zflag = _mm_cmpistrz (value0, mask0,  0x14);
if( index  == 0) /* the first char is not valid numeric digit */
{  xxi = _mm_extract_epi64(w2, 0);
   return (NegSgn? (long long) -xxi: (long long) xxi);
}
if ( index > 3)  return (NegSgn? (long long) RINT64VALNEG: (long long) RINT64VALPOS);
/* multiply low qword by base^index */
w1 =  _mm_mul_epu32( _mm_shuffle_epi32( w2,  0x50), _mm_setr_epi32( MulplyByBaseExpN
    [index - 1] , 0, MulplyByBaseExpN[index-1], 0));
w3 = _mm_add_epi64(w1, _mm_slli_epi64 (_mm_srli_si128(w1, 8), 32 ) );
mask0  = ShfLAlnLSByte( mask0, 16 -index);
// convert upper 8 bytes of xmm: only least sig. 4 digits of output will be added to prev 16 digits
w1_u8 = _mm_cvtepi8_epi16(_mm_srli_si128 ( mask0, 8));
/* merge 2 digit  at a time with multiplier into each dword*/
w1_u8 = _mm_madd_epi16(w1_u8, _mm_loadu_si128(  (__m128i *) &MulplyQuadBaseExp3To0 [ 0]));
/* bits 63:0 has two dword integer, bits 63:32 is the LS dword of output; bits 127:64 is not needed*/
w1_u8 = _mm_cvtepu32_epi64( _mm_hadd_epi32(w1_u8, w1_u8) );
w3 = _mm_add_epi64(w3, _mm_srli_si128( w1_u8, 8) );
xxi = _mm_extract_epi64(w3, 0);
if( xxi >> 63 )
        return (NegSgn? (long long) RINT64VALNEG: (long long) RINT64VALPOS);
else   return (NegSgn? (long long) -xxi: (long long) xxi);

}
```

The general performance characteristics of an SSE4.2 enhanced atol() replacement have a start-up cost that is somewhat lower than byte-granular implementations generated from C code.

Example 10-19. Auxiliary Routines and Data Constants Used in sse4i_atol() listing

```
// bit lookup table of valid ascii code for decimal string conversion, white space, sign, numeric digits
static  char BtMLValDecInt[32] = {0x0, 0x3e, 0x0, 0x0, 0x1, 0x28, 0xff, 0x03,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
                (continue)
```

Example 10-19. Auxiliary Routines and Data Constants Used in sse4i_atol() listing (Contd.)

```
// bit lookup table, white space only
static char BtMLws[32] = {0x0, 0x3e, 0x0, 0x0, 0x1, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
// list of white space for sttni use
static  char listws[16] =
        {0x20, 0x9, 0xa, 0xb, 0xc, 0xd, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
// list of numeric digits for sttni use
static  char rangenumint[16] =
        {0x30, 0x39, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
static  char rangenumintzr[16] =
        {0x30, 0x30, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};

// we use pmaddwd to merge two adjacent short integer pair, this is the second step of merging each pair of 2-digit
integers
static short MulplyPairBaseP2[8] =
{ 100, 1, 100, 1, 100, 1, 100, 1};

// Multiplier-pair for two adjacent short integer pair, this is the third step of merging each pair of 4-digit integers
static short MulplyPairBaseP4[8] =
{ 10000, 1, 10000, 1, 10000, 1, 10000, 1 };

// multiplier for pmulld for normalization of > 16 digits
static int MulplyByBaseExpN[8] =
{  10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000};

static short MulplyQuadBaseExp3To0[8] =
{ 1000, 100, 10, 1, 1000, 100, 10, 1};
```

```
__m128i __m128i_shift_right (__m128i value, int offset)
{ switch (offset)
   {
   case 1: value = _mm_srli_si128 (value, 1); break;
   case 2: value = _mm_srli_si128 (value, 2); break;
   case 3: value = _mm_srli_si128 (value, 3); break;
   case 4: value = _mm_srli_si128 (value, 4); break;
   case 5: value = _mm_srli_si128 (value, 5); break;
   case 6: value = _mm_srli_si128 (value, 6); break;
   case 7: value = _mm_srli_si128 (value, 7); break;
   case 8: value = _mm_srli_si128 (value, 8); break;
   case 9: value = _mm_srli_si128 (value, 9); break;
   case 10: value = _mm_srli_si128 (value, 10); break;
   case 11: value = _mm_srli_si128 (value, 11); break;
   case 12: value = _mm_srli_si128 (value, 12); break;
   case 13: value = _mm_srli_si128 (value, 13); break;
   case 14: value = _mm_srli_si128 (value, 14); break;
   case 15: value = _mm_srli_si128 (value, 15); break;
   }
  return value;
}
               (continue)
```

Example 10-19. Auxiliary Routines and Data Constants Used in sse4i_atol() listing (Contd.)

```
/* Load string at S near page boundary safely.  */

__m128i __m128i_strloadu_page_boundary (const  char *s)
{
 int offset = ((size_t) s & (16 - 1));
 if (offset)
 {
    __m128i v = _mm_load_si128 ((__m128i *) (s - offset));
    __m128i zero = _mm_setzero_si128 ();
    int bmsk = _mm_movemask_epi8 (_mm_cmpeq_epi8 (v, zero));
    if ( (bmsk >> offset) != 0 )  return __m128i_shift_right (v, offset);
 }
 return _mm_loadu_si128 ((__m128i *) s);
}

__m128i ShfLAlnLSByte( __m128i value, int offset)
{
  /*now remove constant bias, so each byte element are unsigned byte int */
  value = _mm_sub_epi8(value,  _mm_setr_epi32(0x30303030, 0x30303030, 0x30303030, 0x30303030));
  switch (offset)
  {
  case 1:
    value = _mm_slli_si128 (value, 1);   break;
  case 2:
    value = _mm_slli_si128 (value, 2);   break;
  case 3:
    value = _mm_slli_si128 (value, 3);   break;
  case 4:
    value = _mm_slli_si128 (value, 4);   break;
  case 5:
    value = _mm_slli_si128 (value, 5);   break;
  case 6:
    value = _mm_slli_si128 (value, 6);   break;
  case 7:
    value = _mm_slli_si128 (value, 7);   break;
  case 8:
    value = _mm_slli_si128 (value, 8);   break;
  case 9:
    value = _mm_slli_si128 (value, 9);   break;
  case 10:
    value = _mm_slli_si128 (value, 10);   break;
  case 11:
    value = _mm_slli_si128 (value, 11);   break;
  case 12:
    value = _mm_slli_si128 (value, 12);   break;
  case 13:
    value = _mm_slli_si128 (value, 13);   break;
  case 14:
    value = _mm_slli_si128 (value, 14);   break;
  case 15:
    value = _mm_slli_si128 (value, 15);   break;
  }
  return value;
}
```

With an input byte stream no more than 16 non-zero-leading digits, it has a constant performance. An input string consisting of more than 16 bytes of non-zero-leading digits can be processed in about 100 cycles or less, compared byte-granular solution needing around 200 cycles. Even for shorter input strings of 9 non-zero-leading digits, SSE4.2 enhanced replacement can also achieve ~2X performance of byte-granular solutions.

10.5 NUMERICAL DATA CONVERSION TO ASCII FORMAT

Conversion of binary integer data to ASCII format gets used in many situations from simple C library functions to computations with finances. Some C libraries provides exported conversion functions like itoa, ltoa; other libraries implement internal equivalents to support data formatting needs of standard output functions. Among the most common binary integer to ascii conversion is conversion based on radix 10. Example 10-20 shows the basic technique implemented in many libraries for base 10 conversion to ascii of a 64-bit integer. For simplicity, the example produces lower-case output format.

Example 10-20. Conversion of 64-bit Integer to ASCII

```
// Convert 64-bit signed binary integer to lower-case ASCII format

static char lc_digits[]= "0123456789abcdefghijklmnopqrstuvwxyz";

int lltoa_cref( __int64 x,  char* out)
{const char *digits = &lc_digits[0];
char lbuf[32]  // base 10 conversion of 64-bit signed integer need only 21 digits
char * p_bkwd = &lbuf[2];
 __int64  y ;
unsigned int  base = 10, len = 0, r, cnt;
   if( x < 0)
   {    y = -x;
       while (y > 0)
       {    r = (int) (y % base); // one digit at a time from least significant digit
           y = y /base;
           * --p_bkwd = digits[r];
           len ++;
       }
       *out++ = '-';
       cnt = len +1;
       while( len--) *out++ = p_bkwd++; // copy each converted digits
   } else
   {
       y = x;
       while (y > 0)
       {    r = (int) (y % base); // one digit at a time from least significant digit
           y = y /base;
           * --p_bkwd = digits[r];
           len ++;
       }
       cnt = len;;
       while( len--) *out++ = p_bkwd++; // copy each converted digits
   }
               (continue)
```

Example 10-20. Conversion of 64-bit Integer to ASCII (Contd.)

```
   out[cnt] = 0;
   return (int) cnt;
}
```

Example 10-20 employs iterative sequence that process one digit at a time using the hardware native integer divide instruction. The reliance on integer divide can be replaced by fixed-point multiply technique discussed in Chapter 9. This is shown in Example 10-21.

Example 10-21. Conversion of 64-bit Integer to ASCII without Integer Division

```
// Convert 64-bit signed binary integer to lower-case ASCII format and
// replace integer division with fixed-point multiply
;__int64 umul_64x64(__int64* p128, __int64 u, __int64 v)
umul_64x64 PROC
        mov       rax, rdx  ; 2nd parameter
        mul       r8  ; u * v
        mov       qword ptr [rcx], rax
        mov       qword ptr [rcx+8], rdx
        ret 0
umul_64x64 ENDP
#define   cg_10_pms3 0xccccccccccccccdull
static char lc_digits[]= "0123456789";

int lltoa_cref( __int64 x,  char* out)
{const char *digits = &lc_digits[0];
char lbuf[32]  // base 10 conversion of 64-bit signed integer need only 21 digits
char * p_bkwd = &lbuf[2];
 __int64  y, z128[2];
unsigned __int64   q;
unsigned int  base = 10, len = 0, r, cnt;

   if( x < 0)
   {   y = -x;
       while (y > 0)
       {  umul_64x64( &z128[0], y, cg_10_pms3);
           q = z128[1] >> 3;
           q = (y < q * (unsigned __int64) base)? q-1: q;
           r = (int) (y - q * (unsigned __int64) base); // one digit at a time from least significant digit
           y =q;
           * --p_bkwd = digits[r];
           len ++;
       }
       *out++ = '-';
       cnt = len +1;
       while( len--) *out++ = p_bkwd++; // copy each converted digits
   } else
              (continue)
```

Example 10-21. Conversion of 64-bit Integer to ASCII without Integer Division (Contd.)

```
    {
        y = x;
        while (y > 0)
        {   umul_64x64( &z128[0], y, cg_10_pms3);
            q = z128[1] >> 3;
            q = (y < q * (unsigned __int64) base)? q-1: q;
            r = (int) (y - q * (unsigned __int64) base); // one digit at a time from least significant digit
            y =q;
            * --p_bkwd = digits[r];
            len ++;
        }
        cnt = len;;
        while( len--) *out++ = p_bkwd++; // copy each converted digits
    }
    out[cnt] = 0;
    return cnt;
}
```

Example 10-21 provides significant speed improvement by eliminating the reliance on integer divisions. However, the numeric format conversion problem is still constrained by the dependent chain that process one digit at a time.

SIMD technique can apply to this class of integer numeric conversion problem by noting that an unsigned 64-bit integer can expand a dynamic range of up to 20 digits. Such a wide dynamic range can be expressed as polynomial expressions of the form:

a_0 + a_1 *10^4 + a_2 *10^8 + a_3 *10^12 + a_4 *10^16 where

the dynamic range of a_i is between [0, 9999].

Reduction of an unsigned 64-bit integer into up-to 5 reduced-range coefficients can be computed using fixed-point multiply in stages. Once the dynamic range of coefficients are reduced to no more than 4 digits, one can apply SIMD techniques to compute ascii conversion in parallel.

The SIMD technique to convert an unsigned 16-bit integer via radix 10 with input dynamic range [0, 9999] is shown in Figure 10-4. This technique can also be generalized to apply to other non-power-of-2 radix that is less than 16.

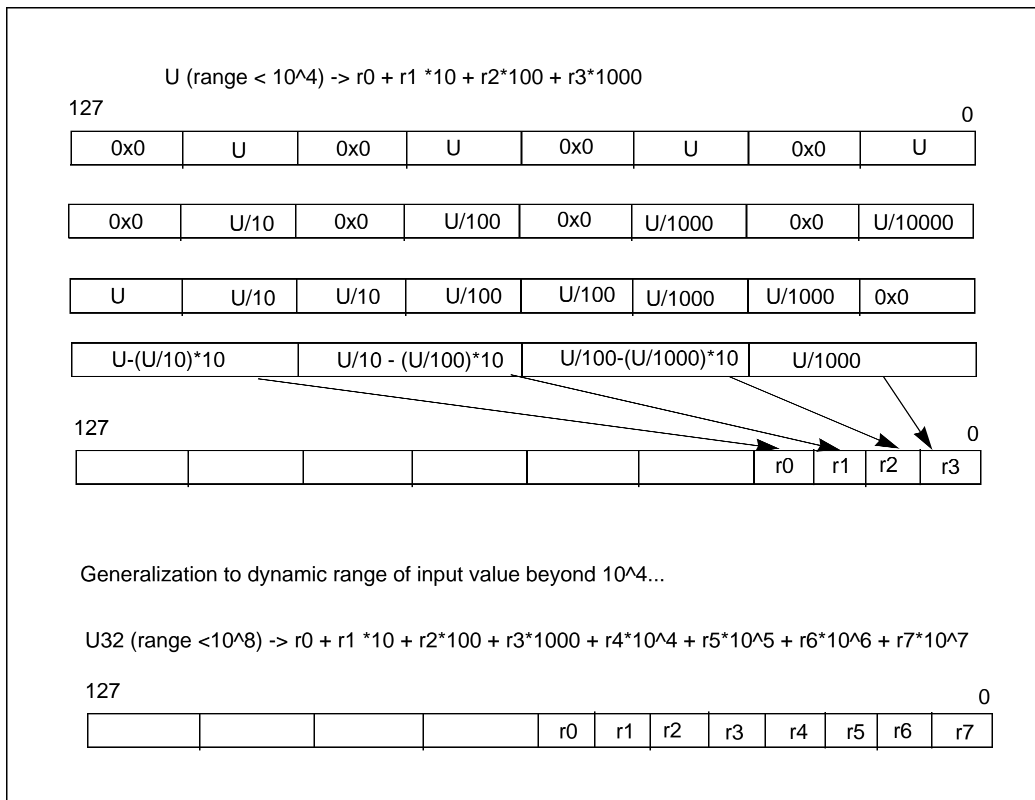


Figure 10-4. Compute Four Remainders of Unsigned Short Integer in Parallel

To handle greater input dynamic ranges, the input is reduced into multiple unsigned short integers and converted sequentially. The most significant U16 conversion is computed first, followed by the conversion of the next four significant digits.

Example 10-22 shows the fixed-point multiply combined with parallel remainder computation using SSE4 instructions for 64-bit integer conversion up to 19 digits.

Example 10-22. Conversion of 64-bit Integer to ASCII Using SSE4

```
#include <smmintrin.h>
#include <stdio.h>
#define    QWCG10to8        0xabcc77118461cefdull
#define    QWCONST10to8     100000000ull

/* macro to convert input parameter of short integer "hi4" into output variable "x3" which is __m128i;
   the input value "hi4" is assume to be less than 10^4;
   the output is 4  single-digit integer between 0-9, located in the low byte of each dword,
   most significant digit in lowest DW.
   implicit overwrites: locally allocated __m128i variable "x0", "x2"
*/
              (continue)
```

Example 10-22. Conversion of 64-bit Integer to ASCII Using SSE4 (Contd.)

```
#define    __ParMod10to4SSSE3( x3, hi4 ) \
{                                        \
       x0 = _mm_shuffle_epi32( _mm_cvtsi32_si128(  (hi4)), 0); \
       x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d));\
       x2 = _mm_srli_epi32( _mm_madd_epi16( x2, _mm_loadu_si128( (__m128i *) quo4digComp_mulplr_d)), 10); \
       (x3) = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) (hi4), 1);  \
       (x3) = _mm_or_si128(x2, (x3));\
       (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;\
}

/* macro to convert input parameter of the 3rd dword element of "t5" ( __m128i type)
  into output variable "x3" which is __m128i;
  the third dword element "t5" is assume to be less than 10^4, the 4th dword must be 0;
  the output is 4  single-digit integer between 0-9, located in the low byte of each dword,
  MS digit in LS DW.
  implicit overwrites: locally allocated __m128i variable "x0", "x2"
*/

#define    __ParMod10to4SSSE3v( x3, t5 ) \
{                                        \
       x0 = _mm_shuffle_epi32( t5, 0xaa);  \
       x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d));\
       x2 = _mm_srli_epi32( _mm_madd_epi16( x2, _mm_loadu_si128( (__m128i *) quo4digComp_mulplr_d)), 10); \
       (x3) = _mm_or_si128(_mm_slli_si128(x2, 6),   _mm_srli_si128(t5, 6));  \
       (x3) = _mm_or_si128(x2, (x3));\
       (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;\
}
```

```
static __attribute__ ((aligned(16)) ) short  quo4digComp_mulplr_d[8] =
{  1024, 0, 64, 0, 8, 0, 0, 0};
static __attribute__ ((aligned(16)) )  short  quoTenThsn_mulplr_d[8] =
{  0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0};
static __attribute__ ((aligned(16)) )  short  mten_mulplr_d[8] =
{ -10, 1, -10,  1, -10,  1, -10, 1};
static __attribute__ ((aligned(16)) )  unsigned short bcstpklodw[8] =
{0x080c, 0x0004, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080};
static __attribute__ ((aligned(16)) )  unsigned short bcstpkdw1[8] =
{0x8080, 0x8080, 0x080c, 0x0004, 0x8080, 0x8080, 0x8080, 0x8080};
static __attribute__ ((aligned(16)) )  unsigned short bcstpkdw2[8] =
{0x8080, 0x8080, 0x8080, 0x8080, 0x080c, 0x0004, 0x8080, 0x8080};
static __attribute__ ((aligned(16)) )  unsigned short bcstpkdw3[8] =
{0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x080c, 0x0004};
static __attribute__ ((aligned(16)) )  int  asc0bias[4] =
{0x30, 0x30, 0x30, 0x30};
static __attribute__ ((aligned(16)) )  int  asc0reversebias[4] =
{0xd0d0d0d0, 0xd0d0d0d0, 0xd0d0d0d0, 0xd0d0d0d0};
static __attribute__ ((aligned(16)) )  int   pr_cg_10to4[4] =
{  0x68db8db, 0 , 0x68db8db, 0};
static __attribute__ ((aligned(16)) )  int   pr_1_m10to4[4] =
{  -10000, 0 , 1, 0};

                (continue)
```

Example 10-22. Conversion of 64-bit Integer to ASCII Using SSE4 (Contd.)

```
/*input value "xx" is less than 2^63-1 */
/* In environment that does not support binary integer arithmetic on __int128_t,
   this helper can be done as asm routine
*/
__inline __int64_t u64mod10to8( __int64_t * pLo, __int64_t xx)
{__int128_t t, b = (__int128_t)QWCG10to8;
__int64_t q;
  t = b * (__int128_t)xx;
  q = t>>(64 +26); // shift count associated with QWCG10to8
  *pLo = xx - QWCONST10to8 * q;
  return q;
}


/* convert integer between 2^63-1 and 0 to ASCII string */
int  sse4i_q2a_u63 ( __int64_t xx, char *ps)
{int  j, tmp, idx=0, cnt;
__int64_t lo8, hi8, abv16, temp;
__m128i x0, m0, x1, x2, x3, x4, x5, x6, m1;
long long  w, u;
  if ( xx < 10000 )
  {   j = ubs_Lt10k_2s_i2 ( (unsigned ) xx, ps);
      ps[j] = 0;        return j;
  }
  if (xx < 100000000 )   // dynamic range of xx is less than 32-bits
  {   m0 = _mm_cvtsi32_si128( xx);
      x1 = _mm_shuffle_epi32(m0, 0x44);  // broadcast to dw0 and dw2
      x3 = _mm_mul_epu32(x1, _mm_loadu_si128( (__m128i *) pr_cg_10to4 ));
      x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
      m0 = _mm_add_epi32( _mm_srli_si128( x1, 8), x3); // quotient in dw2, remainder in dw0
      __ParMod10to4SSSE3v( x3, m0); // pack single digit from each dword to dw0
      x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
      __ParMod10to4SSSE3v( x3, _mm_slli_si128(m0, 8));   // move the remainder to dw2 first
      x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw1) );
      x4 = _mm_or_si128(x4, x5); // pack  digits in bytes 0-7 with leading 0
      cnt = 8;
  }
  else
  {   hi8 = u64mod10to8(&lo8, xx);
      if ( hi8 < 10000) // decompose lo8 dword into quotient and remainder mod 10^4
      {   m0 = _mm_cvtsi32_si128( lo8);
          x2 = _mm_shuffle_epi32(m0, 0x44);
          x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
          x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
          m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);  // quotient in dw0
          __ParMod10to4SSSE3( x3, hi8); // handle digist 11:8 first
          x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
          __ParMod10to4SSSE3v( x3, m0);  // handle digits 7:4
          x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw1) );
          x4 = _mm_or_si128(x4, x5);
          __ParMod10to4SSSE3v( x3, _mm_slli_si128(m0, 8));
          x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw2) );
          x4 = _mm_or_si128(x4, x5); // pack single digist in bytes 0-11 with leading 0
          cnt = 12;
      }
              (continue)
```

Example 10-22. Conversion of 64-bit Integer to ASCII Using SSE4 (Contd.)

```
        else
        {    cnt = 0;
            if ( hi8 >=  100000000)  // handle input greater than 10^16
            {    abv16 = u64mod10to8(&temp, (__int64_t)hi8);
                hi8 = temp;
                __ParMod10to4SSSE3( x3, abv16);
                x6 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
                cnt = 4;
            } // start with handling digits 15:12
            m0 = _mm_cvtsi32_si128( hi8);
            x2 = _mm_shuffle_epi32(m0, 0x44);
            x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
            x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
            m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
            m1 = _mm_cvtsi32_si128( lo8);
            x2 = _mm_shuffle_epi32(m1, 0x44);
            x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
            x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
            m1 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
            __ParMod10to4SSSE3v( x3, m0);
            x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
            __ParMod10to4SSSE3v( x3, _mm_slli_si128(m0, 8));
            x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw1) );
            x4 = _mm_or_si128(x4, x5);
            __ParMod10to4SSSE3v( x3, m1);
            x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw2) );
            x4 = _mm_or_si128(x4, x5);
            __ParMod10to4SSSE3v( x3, _mm_slli_si128(m1, 8));
            x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw3) );
            x4 = _mm_or_si128(x4, x5);
            cnt += 16;
        }
    }
    m0 = _mm_loadu_si128( (__m128i *) asc0reversebias);
    if( cnt > 16)
    {    tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(x6,_mm_setzero_si128()) );
        x6 = _mm_sub_epi8(x6, m0);
    } else {
        tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(x4,_mm_setzero_si128()) );
    }
```

```
#ifndef __USE_GCC__
    __asm__ ("bsfl %1, %%ecx; movl %%ecx, %0;" :"=r"(idx) :"r"(tmp) : "%ecx");
#else
    _BitScanForward(&idx, tmp);
#endif
    x4 = _mm_sub_epi8(x4, m0);
    cnt  -= idx;
    w = _mm_cvtsi128_si64(x4);

    switch(cnt)
    {    case5:*ps++ = (char) (w >>24);  *(unsigned *) ps = (w >>32);
            break;
        case6:*(short *)ps = (short) (w >>16);  *(unsigned *) (&ps[2]) = (w >>32);
            break;
                (continue)
```

Example 10-22. Conversion of 64-bit Integer to ASCII Using SSE4 (Contd.)

```
        case7:*ps = (char) (w >>8);   *(short *) (&ps[1]) = (short) (w >>16);
            *(unsigned *) (&ps[3]) = (w >>32);
            break;
        case 8: *(long long *)ps = w;
        break;
        case9:*ps++ = (char) (w >>24);
            *(long long *) (&ps[0]) =  _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
            break;
        case10:*(short *)ps = (short) (w >>16);
            *(long long *) (&ps[2])  = _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
            break;
        case11:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
                *(long long *) (&ps[3])  = _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
            break;
        case 12: *(unsigned *)ps = w;
            *(long long *) (&ps[4])  = _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
            break;
        case13:*ps++ = (char) (w >>24);  *(unsigned *) ps = (w >>32);
            *(long long *) (&ps[4]) =  _mm_cvtsi128_si64( _mm_srli_si128(x4, 8));
            break;
        case14:*(short *)ps = (short) (w >>16);  *(unsigned *) (&ps[2]) = (w >>32);
            *(long long *) (&ps[6]) =  _mm_cvtsi128_si64( _mm_srli_si128(x4, 8));
        break;
        case15: *ps = (char) (w >>8);
            *(short *) (&ps[1]) = (short) (w >>16);     *(unsigned *) (&ps[3]) = (w >>32);
            *(long long *) (&ps[7])  = _mm_cvtsi128_si64( _mm_srli_si128(x4, 8));
            break;
        case 16: _mm_storeu_si128( (__m128i *) ps, x4);
            break;

        case17:u = _mm_cvtsi128_si64(x6);  *ps++ = (char) (u >>24);
            _mm_storeu_si128( (__m128i *) &ps[0], x4);
            break;
        case18:u = _mm_cvtsi128_si64(x6);  *(short *)ps = (short) (u >>16);
            _mm_storeu_si128( (__m128i *) &ps[2], x4);
            break;
        case19:u = _mm_cvtsi128_si64(x6);  *ps = (char) (u >>8);
            *(short *) (&ps[1]) = (short) (u >>16);
            _mm_storeu_si128( (__m128i *) &ps[3], x4);
            break;
        case20:u = _mm_cvtsi128_si64(x6);  *(unsigned *)ps = (short) (u);
            _mm_storeu_si128( (__m128i *) &ps[4], x4);
            break;
    }
    return cnt;
}
```

(continue)

Example 10-22. Conversion of 64-bit Integer to ASCII Using SSE4 (Contd.)

```
/* convert input value into 4 single digits via parallel fixed-point arithmetic with each dword
   element, and pack each digit into low dword element and write to buffer without leading
   white space; input value must be < 10000 and > 9
*/
__inline int ubs_Lt10k_2s_i2(int x_Lt10k, char *ps)
{int tmp;
__m128i x0, m0, x2, x3, x4, compv;
// Use a set of scaling constant to compensate for lack for per-element shift count
   compv = _mm_loadu_si128( (__m128i *) quo4digComp_mulplr_d);
// broadcast input value to each dword element
   x0 = _mm_shuffle_epi32( _mm_cvtsi32_si128( x_Lt10k), 0);
   // low to high dword in x0 : u16, u16, u16, u16
   m0 = _mm_loadu_si128( (__m128i *) quoTenThsn_mulplr_d); // load 4 congruent consts
   x2 = _mm_mulhi_epu16(x0, m0); // parallel fixed-point multiply for base 10,100, 1000, 10000
   x2 = _mm_srli_epi32( _mm_madd_epi16( x2, compv), 10);
// dword content in x2: u16/10, u16/100, u16/1000, u16/10000
   x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1);
   //word content in x3: 0, u16, 0, u16/10, 0, u16/100, 0, u16/1000

   x4 = _mm_or_si128(x2, x3);
   // perform parallel remainder operation with each word pair to derive 4 unbiased single-digit result
   x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;
   x2 = _mm_add_epi32( x4, _mm_loadu_si128( (__m128i *) asc0bias) ) ;
   // pack each ascii-biased digits from respective dword to the low dword element
   x3 = _mm_shuffle_epi8(x2, _mm_loadu_si128( (__m128i *) bcstpklodw) );
   // store ascii result to buffer without leading white space
   if (x_Lt10k > 999 )
   {   *(int *) ps = _mm_cvtsi128_si32( x3);
       return 4;
   }
   else if (x_Lt10k > 99 )
   {   tmp = _mm_cvtsi128_si32( x3);
       *ps = (char ) (tmp >>8);
       *((short *) (++ps)) = (short ) (tmp >>16);
       return 3;
   }
   else if ( x_Lt10k > 9) // take advantage of reduced dynamic range > 9 to reduce branching
   {   *((short *) ps) = (short ) _mm_extract_epi16( x3, 1);
       return 2;
   }
   *ps = '0' + x_Lt10k;
   return 1;
}
```

(continue)

Example 10-22. Conversion of 64-bit Integer to ASCII Using SSE4 (Contd.)

```
char lower_digits[] = "0123456789";

int ltoa_sse4 (const long long s1, char * buf)
{long long temp ;
int j = 1, len = 0;
const char *digits = &lower_digits[0];
  if( s1 < 0) {
      temp = -s1;
      len ++;
       beg[0] = '-';
      if( temp < 10) beg[1] = digits[ (int) temp];
      else len += sse4i_q2a_u63( temp, &buf[ 1]); // parallel conversion in 4-digit granular operation
  }
  else {
  if( s1 < 10) beg[ 0 ] = digits[(int)s1];
  else len += sse4i_q2a_u63( s1, &buf[ 1] );
  }
  buf[len] = 0;
  return len;
}
```

When an ltoa()-like utility implementation executes native IDIV instruction to convert one digit at a time, it can produce output at a speed of about 45-50 cycles per digit. Using fixed-point multiply to replace IDIV (like Example 10-21) can reduce 10-15 cycles per digit. Using 128-bit SIMD technique to perform parallel fixed-point arithmetic, the output speed can further improve to 4-5 cycles per digit with recent Intel microarchitectures like Sandy Bridge and Nehalem.

The range-reduction technique demonstrated in Example 10-22 reduces up-to 19 levels of dependency chain down to 5 hierarchy and allows parallel SIMD technique to perform 4-wide numeric conversion. This technique can also be done with only SSSE3, and with similar speed improvement.

Support for conversion to wide character strings can be easily adapted using the code snippet shown in Example 10-23.

Example 10-23. Conversion of 64-bit Integer to Wide Character String Using SSE4

```
static __attribute__ ((aligned(16)) ) int  asc0bias[4] =
{0x30, 0x30, 0x30, 0x30};

// exponent_x must be < 10000 and > 9
__inline  int ubs_Lt10k_2wcs_i2(int x_Lt10k, wchar_t *ps)
{
__m128i x0, m0, x2, x3, x4,  compv;
  compv = _mm_loadu_si128( (__m128i *) quo4digComp_mulplr_d);
  x0 = _mm_shuffle_epi32( _mm_cvtsi32_si128(  x_Lt10k), 0);  // low to high dw: u16, u16, u16, u16
  m0 = _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d);
  // u16, 0, u16, 0, u16, 0, u16, 0
  x2 = _mm_mulhi_epu16(x0, m0);
  x2 = _mm_srli_epi32( _mm_madd_epi16( x2, compv), 10); // u16/10, u16/100, u16/1000, u16/10000

  x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1);  // 0, u16, 0, u16/10, 0, u16/100, 0, u16/1000
  x4 = _mm_or_si128(x2, x3);
  x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;
            (continue)
```

Example 10-23. Conversion of 64-bit Integer to Wide Character String Using SSE4 (Contd.)

```
    x2 = _mm_add_epi32( x4, _mm_loadu_si128( (__m128i *) asc0bias) ) ;
    x2 = _mm_shuffle_epi32(x2, 0x1b);  // switch sequence
    if (x_Lt10k > 999 ) {
        _mm_storeu_si128( (__m128i *) ps, x2);
        return 4;
    }
    else if (x_Lt10k > 99 ) {
        *ps++ =  (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x2, 4));
        *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x2, 8));
        return 3;
    }
    else if ( x_Lt10k > 9){ // take advantage of reduced dynamic range > 9 to reduce branching
        *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x2, 8));
        return 2;
    }
    *ps = L'0' + x_Lt10k;
    return 1;
}
```

```
long long  sse4i_q2wcs_u63 ( __int64_t xx, wchar_t *ps)
{int  j, tmp, idx=0, cnt;
__int64_t lo8, hi8, abv16, temp;
__m128i x0, m0, x1, x2, x3, x4, x5, x6, x7, m1;

    if ( xx < 10000 ) {
        j = ubs_Lt10k_2wcs_i2 ( (unsigned ) xx, ps); ps[j] = 0; return j;
    }
    if (xx < 100000000 ) {  // dynamic range of xx is less than 32-bits
       m0 = _mm_cvtsi32_si128( xx);
       x1 = _mm_shuffle_epi32(m0, 0x44);  // broadcast to dw0 and dw2
       x3 = _mm_mul_epu32(x1, _mm_loadu_si128( (__m128i *) pr_cg_10to4 ));
       x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
       m0 = _mm_add_epi32( _mm_srli_si128( x1, 8), x3); // quotient in dw2, remainder in dw0
       __ParMod10to4SSSE3v( x3, m0);
       //x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
       x3 = _mm_shuffle_epi32(x3, 0x1b);
       __ParMod10to4SSSE3v( x4, _mm_slli_si128(m0, 8));   // move the remainder to dw2 first
       x4 = _mm_shuffle_epi32(x4, 0x1b);
       cnt = 8;
    } else {
       hi8 = u64mod10to8(&lo8, xx);
       if( hi8 < 10000) {
        m0 = _mm_cvtsi32_si128( lo8);
        x2 = _mm_shuffle_epi32(m0, 0x44);
        x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
        x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));

        m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
        __ParMod10to4SSSE3( x3, hi8);
        x3 = _mm_shuffle_epi32(x3, 0x1b);
        __ParMod10to4SSSE3v( x4, m0);
        x4 = _mm_shuffle_epi32(x4, 0x1b);
               (continue)
```

Example 10-23. Conversion of 64-bit Integer to Wide Character String Using SSE4 (Contd.)

```
      __ParMod10to4SSSE3v( x5, _mm_slli_si128(m0, 8));
      x5 = _mm_shuffle_epi32(x5, 0x1b);
      cnt = 12;
     } else {
      cnt = 0;
      if ( hi8 > 100000000) {
        abv16 = u64mod10to8(&temp, (__int64_t)hi8);
        hi8 = temp;
        __ParMod10to4SSSE3( x7, abv16);
        x7 = _mm_shuffle_epi32(x7, 0x1b);
        cnt = 4;
      }
      m0 = _mm_cvtsi32_si128( hi8);
      x2 = _mm_shuffle_epi32(m0, 0x44);
      x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
      x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
      m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
      m1 = _mm_cvtsi32_si128( lo8);
      x2 = _mm_shuffle_epi32(m1, 0x44);
      x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
      x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
      m1 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
      __ParMod10to4SSSE3v( x3, m0);
      x3 = _mm_shuffle_epi32(x3, 0x1b);
      __ParMod10to4SSSE3v( x4, _mm_slli_si128(m0, 8));
      x4 = _mm_shuffle_epi32(x4, 0x1b);
      __ParMod10to4SSSE3v( x5, m1);
      x5 = _mm_shuffle_epi32(x5, 0x1b);
      __ParMod10to4SSSE3v( x6, _mm_slli_si128(m1, 8));
      x6 = _mm_shuffle_epi32(x6, 0x1b);
      cnt += 16;
    }
  }

  m0 = _mm_loadu_si128( (__m128i *) asc0bias);
  if( cnt > 16) {
      tmp = _mm_movemask_epi8( _mm_cmpgt_epi32(x7,_mm_setzero_si128()) ) ;
      //x7 = _mm_add_epi32(x7, m0);
  } else {
      tmp = _mm_movemask_epi8( _mm_cmpgt_epi32(x3,_mm_setzero_si128()) );
  }
#ifndef __USE_GCC__
  __asm__ ("bsfl %1, %%ecx; movl %%ecx, %0;" :"=r"(idx) :"r"(tmp) : "%ecx");
#else
  _BitScanForward(&idx, tmp);
#endif
  x3 = _mm_add_epi32(x3, m0);
  cnt  -= (idx >>2);
  x4 = _mm_add_epi32(x4, m0);
  switch(cnt) {
  case5:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 12));
        _mm_storeu_si128( (__m128i *) ps, x4);
  break;
  case6:*(long long *)ps =  _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
        _mm_storeu_si128( (__m128i *) &ps[2], x4);
  break;
                    (continue)
```

Example 10-23. Conversion of 64-bit Integer to Wide Character String Using SSE4 (Contd.)

```
case7:*ps++ =  (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 4));
  *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
  _mm_storeu_si128( (__m128i *) &ps[2], x4);
break;
case 8: _mm_storeu_si128( (__m128i *) &ps[0], x3);
    _mm_storeu_si128( (__m128i *) &ps[4], x4);
break;
case9:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 12));
    x5 = _mm_add_epi32(x5, m0);
    _mm_storeu_si128( (__m128i *) ps, x4);
    _mm_storeu_si128( (__m128i *) &ps[4], x5);
break;
case10:*(long long *)ps =  _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
    x5 = _mm_add_epi32(x5, m0);
    _mm_storeu_si128( (__m128i *) &ps[2], x4);
    _mm_storeu_si128( (__m128i *) &ps[6], x5);
break;
```

```
case11:*ps++ =  (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 4));
    *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
    x5 = _mm_add_epi32(x5, m0);
    _mm_storeu_si128( (__m128i *) &ps[2], x4);
    _mm_storeu_si128( (__m128i *) &ps[6], x5);
break;
case 12: _mm_storeu_si128( (__m128i *) &ps[0], x3);
    x5 = _mm_add_epi32(x5, m0);
    _mm_storeu_si128( (__m128i *) &ps[4], x4);
    _mm_storeu_si128( (__m128i *) &ps[8], x5);
break;
case13:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 12));
    x5 = _mm_add_epi32(x5, m0);
    _mm_storeu_si128( (__m128i *) ps, x4);
    x6 = _mm_add_epi32(x6, m0);
    _mm_storeu_si128( (__m128i *) &ps[4], x5);
    _mm_storeu_si128( (__m128i *) &ps[8], x6);
break;
case14:*(long long *)ps =  _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
    x5 = _mm_add_epi32(x5, m0);
    _mm_storeu_si128( (__m128i *) &ps[2], x4);
    x6 = _mm_add_epi32(x6, m0);
    _mm_storeu_si128( (__m128i *) &ps[6], x5);
    _mm_storeu_si128( (__m128i *) &ps[10], x6);
break;
case15:*ps++ =  (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 4));
    *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
    x5 = _mm_add_epi32(x5, m0);
    _mm_storeu_si128( (__m128i *) &ps[2], x4);
    x6 = _mm_add_epi32(x6, m0);
    _mm_storeu_si128( (__m128i *) &ps[6], x5);
    _mm_storeu_si128( (__m128i *) &ps[10], x6);
break;
        (continue)
```

Example 10-23. Conversion of 64-bit Integer to Wide Character String Using SSE4 (Contd.)

```
    case 16: _mm_storeu_si128( (__m128i *) &ps[0], x3);
        x5 = _mm_add_epi32(x5, m0);
        _mm_storeu_si128( (__m128i *) &ps[4], x4);
        x6 = _mm_add_epi32(x6, m0);
        _mm_storeu_si128( (__m128i *) &ps[8], x5);
        _mm_storeu_si128( (__m128i *) &ps[12], x6);
    break;

    case17:x7 = _mm_add_epi32(x7, m0);
        *ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x7, 12));
        x5 = _mm_add_epi32(x5, m0);
        _mm_storeu_si128( (__m128i *) ps, x3);
        x6 = _mm_add_epi32(x6, m0);
        _mm_storeu_si128( (__m128i *) &ps[4], x4);
        _mm_storeu_si128( (__m128i *) &ps[8], x5);
        _mm_storeu_si128( (__m128i *) &ps[12], x6);
    break;
    case18:x7 = _mm_add_epi32(x7, m0);
        *(long long *)ps =  _mm_cvtsi128_si64( _mm_srli_si128( x7, 8));
        x5 = _mm_add_epi32(x5, m0);
        _mm_storeu_si128( (__m128i *) &ps[2], x3);
        x6 = _mm_add_epi32(x6, m0);
        _mm_storeu_si128( (__m128i *) &ps[6], x4);
        _mm_storeu_si128( (__m128i *) &ps[10], x5);
        _mm_storeu_si128( (__m128i *) &ps[14], x6);
    break;
    case19:x7 = _mm_add_epi32(x7, m0);
        *ps++ = (wchar_t) _mm_cvtsi128_si64( _mm_srli_si128( x7, 4));
        *(long long *)ps =  _mm_cvtsi128_si64( _mm_srli_si128( x7, 8));
        x5 = _mm_add_epi32(x5, m0);
        _mm_storeu_si128( (__m128i *) &ps[2], x3);
        x6 = _mm_add_epi32(x6, m0);
        _mm_storeu_si128( (__m128i *) &ps[6], x4);
        _mm_storeu_si128( (__m128i *) &ps[10], x5);
        _mm_storeu_si128( (__m128i *) &ps[14], x6);
    break;
    case20:x7 = _mm_add_epi32(x7, m0);
        _mm_storeu_si128( (__m128i *) &ps[0], x7);
        x5 = _mm_add_epi32(x5, m0);
        _mm_storeu_si128( (__m128i *) &ps[4], x3);
        x6 = _mm_add_epi32(x6, m0);
        _mm_storeu_si128( (__m128i *) &ps[8], x4);
        _mm_storeu_si128( (__m128i *) &ps[12], x5);
        _mm_storeu_si128( (__m128i *) &ps[16], x6);
    break;
    }
    return cnt;
}
```

10.5.1 Large Integer Numeric Computation

10.5.1.1 MULX Instruction and Large Integer Numeric Computation

The MULX instruction is similar to the MUL instruction but does not read or write arithmetic flags and is enhanced with more flexibility in register allocations for the destination operands. These enhancements allow better out-of-order operation of the hardware and for software to intermix add-carry instruction without corrupting the carry chain.

For computations calculating large integers (e.g. 2048-bit RSA key), MULX can improve performance significantly over techniques based on MUL/ADC chain sequences (see http://download.intel.com/embedded/processor/whitepaper/327831.pdf). AVX2 can be used to build efficient techniques, see Section 11.16.2.

Example 10-24 gives an example of how MULX is used to improve the carry chain computation of integer numeric greater than 64-bit wide.

Example 10-24. MULX and Carry Chain in Large Integer Numeric

mov rax, [rsi+8*1]
mul rbp ; rdx:rax = rax * rbp
mov r8, rdx
add r9, rax
adc r8, 0
add r9, rbx
adc r8, 0
	mulx rbx, r8, [rsi+8*1] ; rbx:r8 = rdx * [rsi+8*1]
	add r8, r9
	adc rbx, 0
	add r8, rbp
	adc rbx, 0

Using MULX to implement 128-bit integer output can be a useful building block for implementing library functions ranging from atof/strtod or intermediate mantisa computation or mantissa/exponent normalization in 128-bit binary decimal floating-point operations. Example 10-25 gives examples of building-block macros, used in 128-bit binary-decimal floating-point operations, which can take advantage MULX to calculate intermediate results of multiple-precision integers of widths between 128 to 256 bits. Details of binary-integer-decimal (BID) floating-point format and library implementation of BID operation can be found at http://software.intel.com/en-us/articles/intel-decimal-floating-point-math-library.

Example 10-25. Building-block Macro Used in Binary Decimal Floating-point Operations

```
// Portable C macro of 64x64-bit product using 32-bit word granular operations
// Output: BID_UINT128 P128
#define __mul_64x64_to_128MACH(P128, CX64, CY64)  \
{                                                 \
  BID_UINT64 CXH,CXL,CYH,CYL,PL,PH,PM,PM2;     \
  CXH = (CX64) >> 32;                           \
  CXL = (BID_UINT32)(CX64);                     \
  CYH = (CY64) >> 32;                           \
  CYL = (BID_UINT32)(CY64);                     \
  PM = CXH*CYL;                                 \
  PH = CXH*CYH;                                 \
  PL = CXL*CYL;                                 \
  PM2 = CXL*CYH;                                \
  PH += (PM>>32);                               \
  PM = (BID_UINT64)((BID_UINT32)PM)+PM2+(PL>>32);          \
  (P128).w[1] = PH + (PM>>32);                  \
  (P128).w[0] = (PM<<32)+(BID_UINT32)PL;              \
}
```

```
// 64x64-bit product using intrinsic producing 128-bit output in 64-bit mode
// Output: BID_UINT128 P128
#define __mul_64x64_to_128MACH_x64(P128, CX64, CY64)  \
{                                                 \
(P128).w[0] = mulx_u64(CX64, CY64, &( (P128).w[1]) );              \
}
```

CHAPTER 11
OPTIMIZATIONS FOR INTEL® AVX, FMA AND AVX2

Intel® Advanced Vector Extension (Intel® AVX), is a major enhancement to Intel Architecture. It extends the functionality of previous generations of 128-bit SSE vector instructions and increased the vector register width to support 256-bit operations. The Intel AVX ISA enhancement is focused on float-point instructions. Some 256-bit integer vectors are supported via floating-point to integer and integer to floating-point conversions.

Intel microarchitecture code name Sandy Bridge implements the Intel AVX instructions, in most cases, on 256-bit hardware. Thus, each core has 256-bit floating-point Add and Multiply units. The Divide and Square-root units are not enhanced to 256-bits. Thus, Intel AVX instructions use the 128-bit hardware in two steps to complete these 256-bit operations.

Prior generations of Intel® Streaming SIMD Extensions (Intel® SSE) instructions generally are two-operand syntax, where one of the operands serves both as source and as destination. Intel AVX instructions are encoded with a VEX prefix, which includes a bit field to encode vector lengths and support three-operand syntax. A typical instruction has two sources and one destination. Four operand instructions such as VBLENDVPS and VBLENDVPD exist as well. The added operand enables non destructive source (NDS) and it eliminates the need for register duplication using MOVAPS operations.

With the exception of MMX instructions, almost all legacy 128-bit SSE instructions have AVX equivalents that support three operand syntax. 256-bit AVX instructions employ three-operand syntax and some with 4-operand syntax.

The 256-bit vector register, YMM, extends the 128-bit XMM register to 256 bits. Thus the lower 128-bits of YMM is aliased to the legacy XMM registers.

While 256-bit AVX instructions writes 256 bits of results to YMM, 128-bit AVX instructions writes 128-bits of results into the XMM register and zeros the upper bits above bit 128 of the corresponding YMM. 16 vector registers are available in 64-bit mode. Only the lower 8 vector registers are available in non-64-bit modes.

Software can continue to use any mixture of legacy SSE code, 128-bit AVX code and 256-bit AVX code. Section covers guidelines to deliver optimal performance across mixed-vector-length code modules without experiencing transition delays between legacy SSE and AVX code. There are no transition delays of mixing 128-bit AVX code and 256-bit AVX code.

The optimal memory alignment of an Intel AVX 256-bit vector, stored in memory, is 32 bytes. Some data-movement 256-bit Intel AVX instructions enforce 32-byte alignment and will signal #GP fault if memory operand is not properly aligned. The majority of 256-bit Intel AVX instructions do not require address alignment. These instructions generally combine load and compute operations, so any non-aligned memory address can be used in these instructions.

For best performance, software should pay attention to align the load and store addresses to 32 bytes whenever possible.

The major differences between using AVX instructions and legacy SSE instructions are summarized in Table 11-1 below:

Table 11-1. Features between 256-bit AVX, 128-bit AVX and Legacy SSE Extensions

Features	256-bit AVX	128-bit AVX	Legacy SSE-AESNI
Functionality Scope	Floating-point operation, Data Movement.	Matches legacy SIMD ISA (except MMX).	128-bit FP and integer SIMD ISA.
Register Operand	YMM.	XMM.	XMM.
Operand Syntax	Up to 4; non-destructive source.	Up to 4; non-destructive source.	2 operand syntax; destructive source.
Memory alignment	Load-Op semantics do not require alignment.	Load-Op semantics do not require alignment.	Always enforce 16B alignment.
Aligned Move Instructions	32 byte alignment.	16 byte alignment.	16 byte alignment.
Non-destructive source operand	Yes.	Yes.	No.
Register State Handling	Updates bits 255:0.	Updates 127:0; Zeroes bits above 128.	Updates 127:0; Bits above 128 unmodified.
Intrinsic Support	▪ New 256-bit data types. ▪ _mm256 prefix for promoted functionality. ▪ New intrinsics for new functionalities.	▪ Existing data types. ▪ Inherit same prototype for exiting functionalities. ▪ Use "_mm" prefix for new VEX-128 functionalities.	Baseline datatypes and prototype definitions.
128-bit Lanes	Applies to most 256-bit operations.	One 128-bit lane.	One 128-bit lane.
Mixed Code Handling	Use VZEROUPPER to avoid transition penalty.	No transition penalty.	Transition penalty after executing 256-bit AVX code.

11.1 INTEL® AVX INTRINSICS CODING

256-bit AVX instructions have new intrinsics. Specifically, 256-bit AVX instruction that are promoted to 256-bit vector length from existing SSE functionality are generally prototyped with a "_mm256" prefix instead of the "_mm" prefix and using new data types defined for 256-bit operation. New functionality in 256-bit AVX instructions have brand new prototype.

The 128-bit AVX instruction that were promoted from legacy SIMD ISA uses the same prototype as before. Newer functionality common in 256-bit and 128-bit AVX instructions are prototyped with "_mm256" and "_mm" prefixes respectively.

Thus porting from legacy SIMD code written in intrinsic can be ported to 256-bit AVX code with a modest effort.

The following guidelines show how to convert a simple intrinsic from Intel SSE code sequence to Intel AVX:

- Align statically and dynamically allocated buffers to 32-bytes.
- May need to double supplemental buffer size.
- Change __mm_ intrinsic name prefix with __mm256_.
- Change variable data types names from __m128 to __m256.
- Divide by 2 iteration count (or double stride length).

This example below on Cartesian coordinate transformation demonstrates the Intel AVX Instruction format, 32 byte YMM registers, dynamic and static memory allocation with data alignment of 32bytes, and the C data type representing 8 floating-point elements in a YMM register.

Example 11-1. Cartesian Coordinate Transformation with Intrinsics

```
//Use SSE intrinsic
#include "wmmintrin.h"

int main()
{ int len = 3200;
  //Dynamic memory allocation with 16byte
  //alignment
  float* pInVector = (float*) _mm_malloc(len*sizeof(float),
16);
  float* pOutVector = (float*) _mm_malloc(len*sizeof(float),
16);
//init data
for(int i=0; i<len; i++) pInVector[i] = 1;

float cos_teta = 0.8660254037;
  float sin_teta = 0.5;
//Static memory allocation of 4 floats with 16byte
alignment
__declspec(align(16)) float cos_sin_teta_vec[4] = {cos_teta,
sin_teta, cos_teta, sin_teta};

__declspec(align(16)) float sin_cos_teta_vec[4] = {sin_teta,
cos_teta, sin_teta, cos_teta};

//__m128 data type represents an xmm
  //register with 4 float elements
  __m128 Xmm_cos_sin = _mm_load_ps(cos_sin_teta_vec);

  //SSE 128bit packed single load
  __m128 Xmm_sin_cos = _mm_load_ps(sin_cos_teta_vec);

  __m128 Xmm0, Xmm1, Xmm2, Xmm3
//processing 8 elements in an unrolled twice loop

for(int i=0; i<len; i+=8)
 {
   Xmm0 = _mm_load_ps(pInVector+i);
   Xmm1 = _mm_moveldup_ps(Xmm0);
   Xmm2 = _mm_movehdup_ps(Xmm0);
   Xmm1 = _mm_mul_ps(Xmm1,Xmm_cos_sin);
   Xmm2 = _mm_mul_ps(Xmm2,Xmm_sin_cos);
   Xmm3 = _mm_addsub_ps(Xmm1, Xmm2);
   _mm_store_ps(pOutVector + i, Xmm3);
```

```
// Use Intel AVX intrinsic
#include "immintrin.h"

int main()
{ int len = 3200;
  //Dynamic memory allocation with 16byte
  //alignment
  float* pInVector = (float*) _mm_malloc(len*sizeof(float),
32);
  float* pOutVector = (float*) _mm_malloc(len*sizeof(float),
32);
//init data
for(int i=0; i<len; i++) pInVector[i] = 1;

float cos_teta = 0.8660254037;
  float sin_teta = 0.5;
//Static memory allocation of 8 floats with 32byte
alignment
__declspec(align(32)) float cos_sin_teta_vec[8] = {cos_teta,
sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta,
sin_teta};

__declspec(align(32)) float sin_cos_teta_vec[8] = {sin_teta,
cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta,
cos_teta };

  //__m256 data type holds 8 float elements
  __m256 Ymm_cos_sin =
_mm256_load_ps(cos_sin_teta_vec);

  //AVX 256bit packed single load
  __m256 Ymm_sin_cos =
_mm256_load_ps(sin_cos_teta_vec);

  __m256 Ymm0, Ymm1, Ymm2, Ymm3;

  //processing 8 elements in an unrolled twice loop

for(int i=0; i<len; i+=16)
 {
   Ymm0 = _mm256_load_ps(pInVector+i);
   Ymm1 = _mm256_moveldup_ps(Ymm0);
   Ymm2 = _mm256_movehdup_ps(Ymm0);
   Ymm1 = _mm256_mul_ps(Ymm1,Ymm_cos_sin);
   Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
   Ymm3 = _mm256_addsub_ps(Ymm1, Ymm2);
   _mm256_store_ps(pOutVector + i, Ymm3);
```

Example 11-1. Cartesian Coordinate Transformation with Intrinsics (Contd.)

`Xmm0 = _mm_load_ps(pInVector+i+4);`
`Xmm1 = _mm_moveldup_ps(Xmm0);`
`Xmm2 = _mm_movehdup_ps(Xmm0);`
`Xmm1 = _mm_mul_ps(Xmm1,Xmm_cos_sin);`
`Xmm2 = _mm_mul_ps(Xmm2,Xmm_sin_cos);`
`Xmm3 = _mm_addsub_ps(Xmm1, Xmm2);`
`_mm_store_ps(pOutVector+i+4, Xmm3);`
`}`
`_mm_free(pInVector);`
`_mm_free(pOutVector);`
`return 0;`
`}`
	`Ymm0 = _mm256_load_ps(pInVector+i+8);`
	`Ymm1 = _mm256_moveldup_ps(Ymm0);`
	`Ymm2 = _mm256_movehdup_ps(Ymm0);`
	`Ymm1 = _mm256_mul_ps(Ymm1,Ymm_cos_sin);`
	`Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);`
	`Ymm3 = _mm256_addsub_ps(Ymm1, Ymm2);`
	`_mm256_store_ps(pOutVector+i+8, Ymm3);`
	`}`
	`_mm_free(pInVector);`
	`_mm_free(pOutVector);`
	`return 0;`
	`}`

11.1.1 Intel® AVX Assembly Coding

Similar to the intrinsic porting guidelines, assembly porting guidelines is listed below:

- Align statically and dynamically allocated buffers to 32-bytes.
- Double the supplemental buffer sizes if needed.
- Add a “v” prefix to instruction names.
- Change register names from xmm to ymm.
- Add destination registers to computational Intel AVX instructions.
- Divide the iteration count by two (or double stride length).

Example 11-2. Cartesian Coordinate Transformation with Assembly

`//Use SSE Assembly`
`int main()`
`{`
`int len = 3200;`
`//Dynamic memory allocation with 16byte`
`//alignment`
`float* pInVector = (float*) _mm_malloc(len*sizeof(float), 16);`
`float* pOutVector = (float*) _mm_malloc(len*sizeof(float), 16);`
 	
`//init data`
`for(int i=0; i<len; i++)`
`pInVector[i] = 1;`
 	
`//Static memory allocation of 4 floats`
`//with 16byte alignment`
`float cos_teta = 0.8660254037;`
`float sin_teta = 0.5;`
`__declspec(align(16)) float cos_sin_teta_vec[4] = {cos_teta, sin_teta, cos_teta, sin_teta};`
	`// Use Intel AVX assembly`
	`int main()`
	`{`
	`int len = 3200;`
	`//Dynamic memory allocation with 32byte`
	`//alignment`
	`float* pInVector = (float*) _mm_malloc(len*sizeof(float), 32);`
	`float* pOutVector = (float*) _mm_malloc(len*sizeof(float), 32);`
	
	`//init data`
	`for(int i=0; i<len; i++)`
	`pInVector[i] = 1;`
	
	`//Static memory allocation of 8 floats`
	`//with 32byte alignment`
	`float cos_teta = 0.8660254037;`
	`float sin_teta = 0.5;`
	`__declspec(align(32)) float cos_sin_teta_vec[8] = {cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta};`

Example 11-2. Cartesian Coordinate Transformation with Assembly (Contd.)

```
 __declspec(align(16)) float sin_cos_teta_vec[4] = {sin_teta,
cos_teta, sin_teta, cos_teta};

//processing 8 elements in an unrolled-twice loop
__asm
{
  mov eax, pInVector
  mov ebx, pOutVector
// Load into an xmm register of 16 bytes
  movups xmm3,
        xmmword ptr[cos_sin_teta_vec]
  movups xmm4,
        xmmword ptr[sin_cos_teta_vec]

  mov edx, len
  xor ecx, ecx
loop1:
  movsldup  xmm0, [eax+ecx]
  movshdup  xmm1, [eax+ecx]
//example: mulps has 2 operands
  mulps xmm0, xmm3
  mulps xmm1, xmm4
  addsubps xmm0, xmm1
// 16 byte store from an xmm register
  movaps [ebx+ecx], xmm0

  movsldup  xmm0, [eax+ecx+16]
  movshdup  xmm1, [eax+ecx+16]
  mulps xmm0, xmm3
  mulps xmm1, xmm4
  addsubps xmm0, xmm1
// offset of 16 bytes from previous store
  movaps [ebx+ecx+16], xmm0

// Processed 32bytes in this loop
//(The code is unrolled twice)
  add ecx, 32
  cmp ecx, edx
  jl loop1
}
  _mm_free(pInVector);
  _mm_free(pOutVector);
  return 0;
}
```

```
 __declspec(align(32)) float sin_cos_teta_vec[8] = {sin_teta,
cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta,
cos_teta};

//processing 16 elements in an unrolled-twice loop
__asm
 {
   mov eax, pInVector
   mov ebx, pOutVector
// Load into an ymm register of 32 bytes
   vmovups ymm3,
          ymmword ptr[cos_sin_teta_vec]
   vmovups ymm4,
          ymmword ptr[sin_cos_teta_vec]

   mov edx, len
   xor ecx, ecx
loop1:
   vmovsldup  ymm0, [eax+ecx]
   vmovshdup  ymm1, [eax+ecx]
//example: vmulps has 3 operands
   vmulps ymm0, ymm0, ymm3
   vmulps ymm1, ymm1, ymm4
   vaddsubps ymm0, ymm0, ymm1
// 32 byte store from an ymm register
   vmovaps [ebx+ecx], ymm0

   vmovsldup  ymm0, [eax+ecx+32]
   vmovshdup  ymm1, [eax+ecx+32]
   vmulps ymm0, ymm0, ymm3
   vmulps ymm1, ymm1, ymm4
   vaddsubps ymm0, ymm0, ymm1
// offset of 32 bytes from previous store
   vmovaps [ebx+ecx+32], ymm0

// Processed 64bytes in this loop
//(The code is unrolled twice)
   add ecx, 64
   cmp ecx, edx
   jl loop1
 }
 _mm_free(pInVector);
 _mm_free(pOutVector);
 return 0;
}
```

11.2 NON-DESTRUCTIVE SOURCE (NDS)

Most Intel AVX instructions have three operands. A typical instruction has two sources and one destination, with both source operands unmodified by the instruction. This section describes how using the NDS

feature to save register copies, reduce amount of instructions, reduce amount of micro-ops and improve performance. In this example, the Intel AVX code is more than 2x faster than the Intel SSE code.

The following example uses a vectorized calculation of the polynomial A3+A2+A. The polynomial calculation pseudo code is:

```
While (i<len)
{
  B[i] := A[i]3 + A[i]2 + A[i]
  i++
}
```

The left side of Example 11-3 shows the vectorized implementation using SSE assembly. In this code, A is copied by an additional load from memory to a register and A2 is copied using a register to register assignment. The code uses 10 micro-ops to process four elements.

The middle example uses 128-bit AVX and takes advantage of NDS. The additional load and register copies are eliminated. This code uses 8 micro-ops to process 4 elements and is about 30% faster than the baseline above.

The right-most example uses 256-bit Intel AVX instructions. It uses 8 micro-ops to process 8 elements. Combining the NDS feature with doubling of vector width, this speeds up over the baseline by more than 2x.

Example 11-3. Direct Polynomial Calculation

SSE code	128-bit AVX code	256-bit AVX code

float* pA = InputBuffer;
float* pB = OutputBuffer;
int len = miBufferWidth-4;
 	
__asm
{
mov rax, pA
mov rbx, pB
movsxd r8, len
 	
loop1:
//Load A
movupsxmm0, [rax+r8*4]
//Copy A
movupsxmm1, [rax+r8*4]
//A^2
mulpsxmm1, xmm1
//Copy A^2
movupsxmm2, xmm1
//A^3
mulps xmm2, xmm0
//A + A^2
addps xmm0, xmm1
//A + A^2 + A^3
addps xmm0, xmm2
//Store result
movups[rbx+r8*4], xmm0
	float* pA = InputBuffer1;
	float* pB = OutputBuffer1;
	int len = miBufferWidth-4;
	
	__asm
	{
	mov rax, pA
	mov rbx, pB
	movsxd r8, len
	
	loop1:
	//Load A
	vmovups xmm0, [rax+r8*4]
	
	//A^2
	vmulps xmm1, xmm0, xmm0
	
	//A^3
	vmulps xmm2, xmm1, xmm0
	//A+A^2
	vaddps xmm0, xmm0, xmm1
	//A+A^2+A^3
	vaddps xmm0, xmm0, xmm2
	//Store result
	vmovups[rbx+r8*4], xmm0
		float* pA = InputBuffer1;
		float* pB = OutputBuffer1;
		int len = miBufferWidth-8;
		
		__asm
		{
		mov rax, pA
		mov rbx, pB
		movsxd r8, len
		
		loop1:
		//Load A
		vmovups ymm0, [rax+r8*4]
		
		//A^2
		vmulpsymm1, ymm0, ymm0
		
		//A^3
		vmulps ymm2, ymm1, ymm0
		//A+A^2
		vaddps ymm0, ymm0, ymm1
		//A+A^2+A^3
		vaddps ymm0, ymm0, ymm2
		//Store result
		vmovups [rbx+r8*4], ymm0

Example 11-3. Direct Polynomial Calculation (Contd.)

SSE code	128-bit AVX code	256-bit AVX code

sub r8, 4
jge loop1
}
	sub r8, 4
	jge loop1
	}
		sub r8, 8
		jge loop1
		}

11.3 MIXING AVX CODE WITH SSE CODE

The Intel AVX architecture allows programmers to port a large code base gradually, resulting is mixed AVX code and SSE code. If your code includes both Intel AVX and Intel SSE, consider the following:

- Recompilation of Intel SSE code with the Intel compiler and the option "/QxAVX" in Windows or "-xAVX" in Linux. This transforms all SSE instructions to 128-bit Intel AVX instructions automatically. This refers to inline assembly and intrinsic code. "GCC -c -mAVX" will generate AVX code, including assembly files. GCC assembler also supports "-msse2AVX" switch to generate AVX code from SSE.
- Intel AVX and Intel SSE code can co-exist and execute in the same run. This can happen if your application includes third party libraries with Intel SSE code, a new DLL using AVX code is deployed that calls other modules running SSE code, or you cannot recompile all your application at once. In these cases, the AVX code must use the VZEROUPPER instruction to avoid AVX/SSE transition penalty.

AVX instruction always modifies the upper bits of YMM registers and SSE instructions do not modify the upper bits. From a hardware perspective, the upper bits of the YMM register collection can be considered to be in one of three states:

- Clean: All upper bits of YMM are zero. This is the state when processor starts from RESET.
- Modified and Unsaved (In Table 11-2, this is abbreviated as M/U): The execution of one AVX instruction (either 256-bit or 128-bit) modifies the upper bits of the destination YMM. This is also referred to as dirty upper YMM state. In this state, bits 255:128 and bits 127:0 of a given YMM are related to the most recent 256-bit or 128-bit AVX instruction that operated on that register.
- Preserved/Non_INIT Upper State (In Table 11-2, this is abbreviated as P/N): In this state, the upper YMM state is not zero. The upper 128 bits of a YMM and the lower 128 bits may be unrelated to the last AVX instruction executed in the processor as a result of XRSTOR from a saved image with dirty upper YMM state.

If software inter-mixes AVX and SSE instructions without using VZEROUPPER properly, it can experience an AVX/SSE transition penalty. The situations of executing SSE, AVX, or managing the YMM state using XSAVE/XRSTOR/VZEROUPPER/VZEROALL is illustrated in Figure 11-1. The penalty associated with transitions into or out of the processor state "Modified and Unsaved" is implementation specific, depending on the microarchitecture.

Figure 11-1 depicts the situations that a transition penalty will occur for recent generations of microarchitectures that support AVX, up to and including the Broadwell microarchitecture. The transition penalty of A and B occurs with each instruction execution that would cause the transition. It is largely the cost of copying the entire YMM state to internal storage.

To minimize the occurrence of YMM state transitions related to the "Preserved/Non_INIT Upper State", software that uses XSAVE/XRSTOR family of instructions to save/restore the YMM state should write a "Clean" upper YMM state to the XSAVE region in memory. Restoring a dirty YMM image from memory into the YMM registers can experience a penalty. This is illustrated in Figure 11-1.

The Skylake microarchitecture implements a different state machine than prior generations to manage the YMM state transition associated with mixing SSE and AVX instructions. It no longer saves the entire upper YMM state when executing an SSE instruction when in "Modified and Unsaved" state, but saves the upper bits of individual register. As a result, mixing SSE and AVX instructions will experience a penalty associated with partial register dependency of the destination registers being used and additional blend

operation on the upper bits of the destination registers. Figure 11-2 depicts the transition penalty applicable to the Skylake microarchitecture.

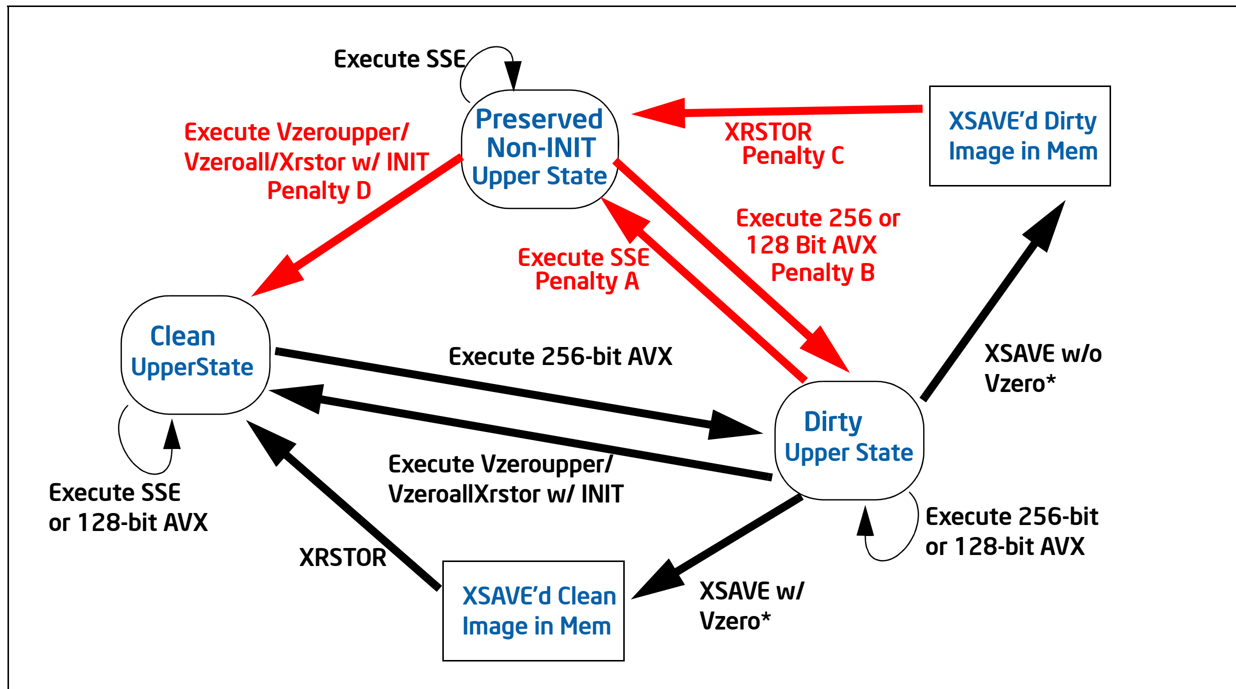


Figure 11-1. AVX-SSE Transitions in the Broadwell, and Prior Generation Microarchitectures

Table 11-2 lists the effect of mixing Intel AVX and Intel SSE code, with the bottom row indicating the types of penalty that might arise depending on the initial YMM state (the row marked 'Begin') and the ending state. Table 11-2 also includes the effect of transition penalty (Type C and D) associated with restoring a dirty YMM state image stored in memory.

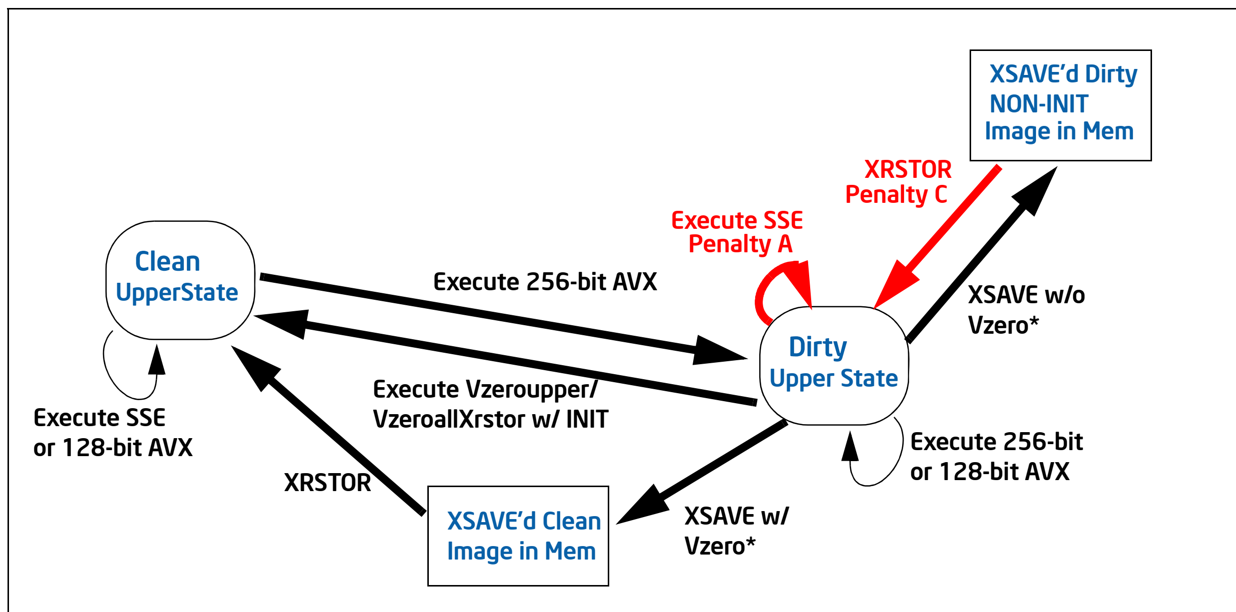


Figure 11-2. AVX-SSE Transitions in the Skylake Microarchitecture

Table 11-2. State Transitions of Mixing AVX and SSE Code

	Execute SSE			Execute AVX-128			Execute AVX-256			VZeroupper	XRSTOR	
Begin	Clean	M/U	P/N	Clean	M/U	P/S	Clean	M/U	P/N	P/N	Dirty Image	Clean Image
End	Clean	P/N	P/N	Clean	M/U	M/U	M/U	M/U	M/U	Clean	P/N	Clean
Penalty	No	A	No	No	No	B	No	No	B	D	C	No

The magnitude of each type of transition penalty can vary across different microarchitectures. In Skylake microarchitecture, some of the transition penalty is reduced. The transition diagram and associated penalty is depicted in Figure 11-2. Table 11-3 gives approximate order of magnitude of the different transition penalty types across recent microarchitectures.

Table 11-3. Approximate Magnitude of AVX-SSE Transition Penalties in Different Microarchitectures

Type	Haswell Microarchitecture	Broadwell Microarchitecture	Skylake Microarchitecture
A	~XSAVE	~XSAVE	Partial Register Dependency + Blend
B	~XSAVE	~XSAVE	NA
C	~Fraction of XSAVE	~Fraction of XSAVE	~XSAVE
D	~XSAVE	~XSAVE	NA

To enable fast transitions between 256-bit Intel AVX and Intel SSE code blocks, use the VZEROUPPER instruction before and after an AVX code block that would need to switch to execute SSE code. The VZEROUPPER instruction resets the upper 128 bits of all Intel AVX registers. This instruction has zero latency. In addition, the processor changes back to a Clean state, after which execution of SSE instructions or Intel AVX instructions has no transition penalty with prior microarchitectures. In Skylake microarchitecture, the SSE block can executed from a Clean state without the penalty of upper-bits dependency and blend operation.

128-bit Intel AVX instructions zero the upper 128-bits of the destination registers. Therefore, 128-bit and 256-bit Intel AVX instructions can be mixed with no penalty.

Assembly/Compiler Coding Rule 71. (H impact, H generality) *Whenever a 256-bit AVX code block and 128-bit SSE code block might execute in sequence, use the VZEROUPPER instruction to facilitate a transition to a "Clean" state for the next block to execute from.*

11.3.1 Mixing Intel® AVX and Intel SSE in Function Calls

Intel AVX to Intel SSE transitions can occur unexpectedly when calling functions or returning from functions. For example, if a function that uses 256-bit Intel AVX, calls another function, the callee might be using SSE code. Similarly, after a 256-bit Intel AVX function returns, the caller might be executing Intel SSE code.

Assembly/Compiler Coding Rule 72. (H impact, H generality) *Add VZEROUPPER instruction after 256-bit AVX instructions are executed and before any function call that might execute SSE code. Add VZEROUPPER at the end of any function that uses 256-bit AVX instructions.*

Example 11-4. Function Calls and AVX/SSE transitions

```
__attribute__((noinline)) void SSE_function()
{
    __asm addps xmm1, xmm2
    __asm xorps xmm3, xmm4
}

__attribute__((noinline)) void AVX_function_no_zeroupper()
{
    __asm vaddps ymm1, ymm2, ymm3
    __asm vxorps ymm4, ymm5, ymm6
}
__attribute__((noinline)) void AVX_function_with_zeroupper()
{
    __asm vaddps ymm1, ymm2, ymm3
    __asm vxorps ymm4, ymm5, ymm6
    //add vzeroupper when returning from an AVX function
    __asm vzeroupper
}
```

```
// Code encounter transition penalty

__asm vaddps ymm1, ymm2, ymm3
..

//penalty
SSE_function();
AVX_function_no_zeroupper();
//penalty
__asm addps xmm1, xmm2
```

```
// Code mitigated transition penalty

__asm vaddps ymm1, ymm2, ymm3
//add vzeroupper before
//calling SSE function from AVX code
__asm vzeroupper   //no penalty
SSE_function();
AVX_function_with_zeroupper();
//no penalty
__asm addps xmm1, xmm2
```

Table 11-2 summarizes a heuristic of the performance impact of using or not using VZEROUPPER to bridge transitions of inter-function calls that changes between AVX code implementation and SSE code.

Table 11-4. Effect of VZEROUPPER with Inter-Function Calls Between AVX and SSE Code

Inter-Function Call	Prior Microarchitectures	Skylake Microarchitecture
With VZEROUPPER	1X (baseline)	~1
No VZEROUPPER	< 0.1X	Fraction of baseline

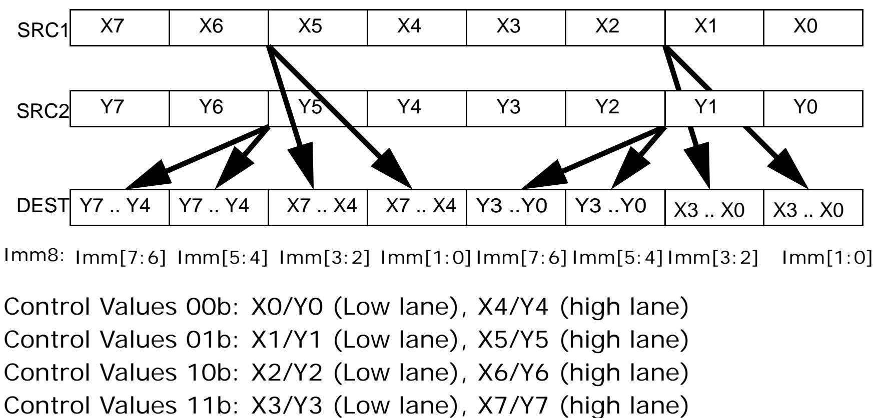
11.4 128-BIT LANE OPERATION AND AVX

256-bit operations in Intel AVX are generally performed in two halves of 128-bit lanes. Most of the 256-bit Intel AVX instructions are defined as in-lane: the destination elements in each lane are calculated using source elements only from the same lane. There are only a few cross-lane instructions, which are described below.

The majority of SSE computational instructions perform computation along vertical slots with each data elements. The 128-bit lanes does not affect porting 128-bit code into 256-bit AVX code. VADDPS is one example of this.

Many 128-bit SSE instruction moves data elements horizontally, e.g. SHUFPS uses an imm8 byte to control the horizontal movement of data elements.

Intel AVX promotes these horizontal 128-bit SIMD instruction in-lane into 256-bit operation by using the same control field within the low 128-bit land and the high 128-bit lane. For example, the 256-bit VSHUFPS instruction uses a control byte containing 4 control values to select the source location of each destination element in a 128-bit lane. This is shown below.



11.4.1 Programming With the Lane Concept

Using the lane concept, algorithms implemented with SSE instruction set can be easily converted to use 256-bit Intel AVX. An SSE algorithm that executes iterations 0 to n can be converted such that the calculation of iteration i is done in the low lane and the calculation of iteration i+k is done in the high lane. For consecutive iterations k equals one.

Some vectorized algorithms implemented with SSE instructions cannot use a simple conversion described above. For example, shuffles that move elements within 16 bytes cannot be naturally converted to shuffles with 32 byte since 32 byte shuffles can't cross lanes.

You can use the following instructions as building blocks for working with lanes:

- VINSERTF128 - insert packed floating-point values.
- VEXTRACTF128 - extract packed floating-point values.
- VPERM2F128 - permute floating-point values.
- VBROADCAST - load with broadcast.

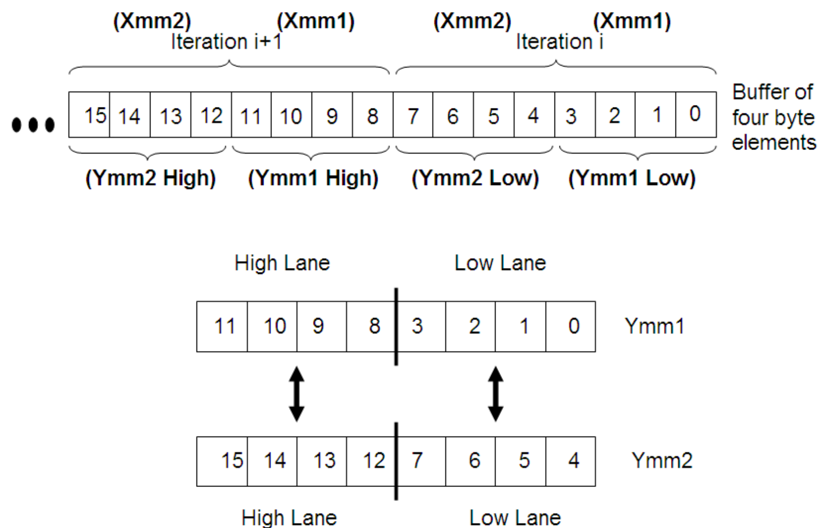
The sections below describe two techniques: the strided loads and the cross register overlap. These methods implement the in lane data arrangement described above and are useful in many algorithms that initially seem to require cross lane calculations.

11.4.2 Strided Load Technique

The strided load technique is a programming method that uses Intel AVX instructions and is useful for algorithms that involve unsupported cross-lane shuffles.

The method describes how to arrange data to avoid cross-lane shuffles. The main idea is to use 128-bit loads in a way that mimics the corresponding Intel SSE algorithm, and enables the 256 Intel AVX instruc-

tions to execute iterations i of the loop in the low lanes and the iteration and i+k in the high lanes. In the following example, k equals one.

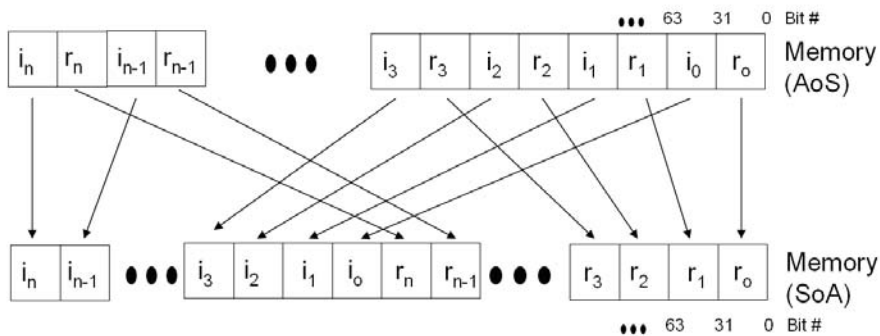


The values in the low lanes of Ymm1 and Ymm2 in the figure above correspond to iteration i in the SSE implementation. Similarly, the values in the high lanes of Ymm1 and Ymm2 correspond to iteration i+1.

The following example demonstrates the strided load method in a conversion of an Array of Structures (AoS) to a Structure of Arrays (SoA). In this example, the input buffer contains complex numbers in an AoS format. Each complex number is made of a real and an imaginary float values. The output buffer is arranged as SoA. All the real components of the complex numbers are located at the first half of the output buffer and all the imaginary components are located at the second half of the buffer. The following pseudo code and figure illustrate the conversion:

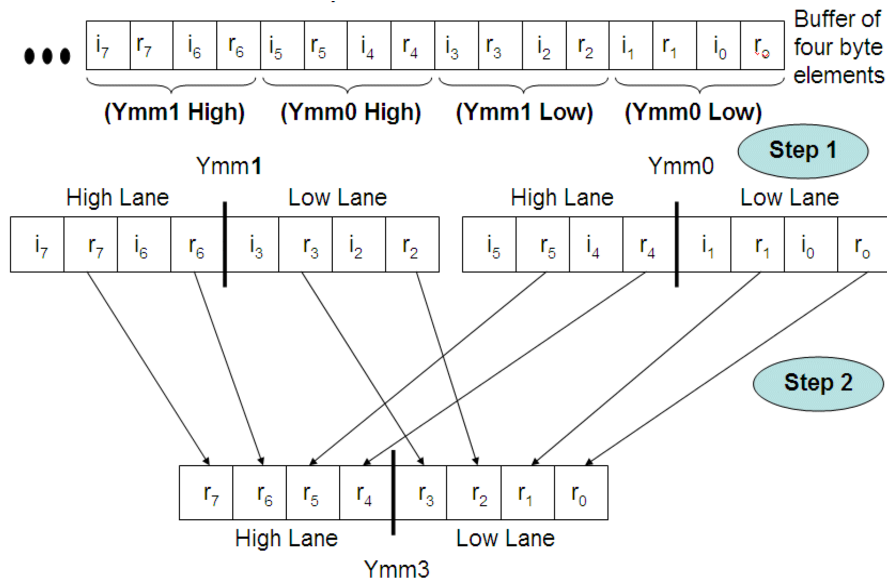
Example 11-5. AoS to SoA Conversion of Complex Numbers in C Code

```
Ifor (i=0 I < N)
{
  Real[i] =Complex[i].Real
  Imaginary[i] =Complex[i].Imaginary
}
```



A simple extension of the Intel SSE algorithm from 16-byte to 32-byte operations would require cross-lane data transition, as shown in the following figure. However, this is not possible with Intel AVX architecture and a different technique is required.

The challenge of cross-lane shuffle can be overcome with Intel AVX for AoS to SoA conversion. Using VINSERTF128 to load 16 bytes to the appropriate lane in the YMM registers obviates the need for cross-lane shuffle. Once the data is organized properly in the YMM registers for step 1, 32-byte VSHUFPS can be used to move the data in lanes, as shown in step 2.



The following code compares the Intel SSE implementation of AoS to SoA with the 256-bit Intel AVX implementation and demonstrates the performance gained.

Example 11-6. Aos to SoA Conversion of Complex Numbers Using AVX

SSE Code	AVX Code
xor rbx, rbx	xor rbx, rbx
xor rdx, rdx	xor rdx, rdx
mov rcx, len	mov rcx, len
mov rdi, inPtr	mov rdi, inPtr
mov rsi, outPtr1	mov rsi, outPtr1
mov rax, outPtr2	mov rax, outPtr2
loop1:	loop1:
movups xmm0, [rdi+rbx]	vmovups xmm0, [rdi+rbx]
//i1 r1 i0 r0	//i1 r1 i0 r0
movups xmm1, [rdi+rbx+16]	vmovups xmm1, [rdi+rbx+16]
// i3 r3 i2 r2	// i3 r3 i2 r2
movups xmm2, xmm0	vinsertf128 ymm0, ymm0, [rdi+rbx+32] , 1
	//i5 r5 i4 r4; i1 r1 i0 r0
shufps xmm0, xmm1, 0xdd	vinsertf128 ymm1, ymm1, [rdi+rbx+48] , 1
//i3 i2 i1 i0	//i7 r7 i6 r6; i3 r3 i2 r2
shufps xmm2, xmm1, 0x88	vshufps ymm2, ymm0, ymm1, 0xdd
//r3 r2 r1 r0	//i7 i6 i5 i4; i3 i2 i1 i0
	vshufps ymm3, ymm0, ymm1, 0x88
	//r7 r6 r5 r4; r3 r2 r1 r0

Example 11-6. Aos to SoA Conversion of Complex Numbers Using AVX (Contd.)

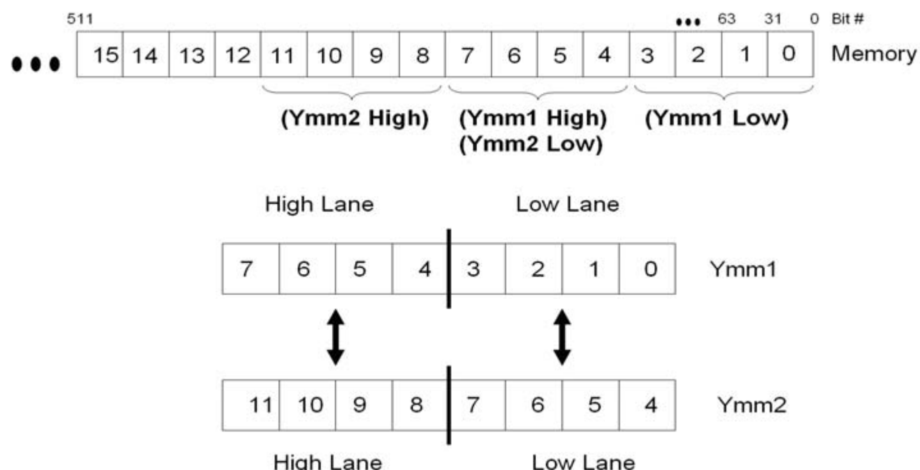
SSE Code	AVX Code

movups [rax+rdx], xmm0
movups [rsi+rdx], xmm2
add rdx, 16
add rbx, 32
cmp rcx, rbx
jnz loop1
	vmovups [rax+rdx], ymm2
	vmovups [rsi+rdx], ymm3
	add rdx, 32
	add rbx, 64
	cmp rcx, rbx
	jnz loop1

11.4.3 The Register Overlap Technique

The register overlap technique is useful for algorithms that use shuffling. Similar to the strided load technique, the register overlap technique arranges data to avoid cross-lane shuffles.

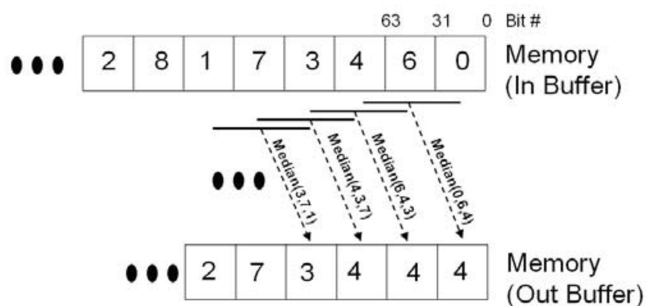
This technique is useful for algorithm that process continues data, which is partially shared by sequential iterations. The following figure illustrates the desired data layout. This is enabled by using overlapping 256-bit loads, or by using the VPERM2F128 instruction.



The Median3 code sample below demonstrates the register overlap technique. The median3 technique calculates the median of every three consecutive elements in a vector.

Y[i] = Median(X[i], X[i+1], X[i+2])

Where Y is the output vector and X is the input vector. The following figure illustrates the calculation done by the median algorithm.



Following are three implementations of the Median3 algorithm. Alternative 1 is the Intel SSE implementation. Alternatives 2 and 3 implement the register overlap technique in two ways. Alternative 2 loads the data from the input buffer into the YMM registers using overlapping 256-bit load operations. Alternative 3 loads the data from the input buffer into the YMM registers using a 256-bit load operation and VPERM2F128. Alternatives 2 and 3 gain performance by using wider vectors.

.

Example 11-7. Register Overlap Method for Median of 3 Numbers

1: SSE Code	2: 256-bit AVX w/ Overlapping Loads	3: 256-bit AVX with VPERM2F128

xor ebx, ebx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr
movaps xmm0, [rdi]
 	
loop_start:
movaps xmm4, [rdi+16]
movaps xmm2, [rdi]
movaps xmm1, [rdi]
movaps xmm3, [rdi]
 	
add rdi, 16
add rbx, 4
shufps xmm2, xmm4, 0x4e
shufps xmm1, xmm2, 0x99
minps xmm3, xmm1
maxps xmm0, xmm1
minps xmm0, xmm2
maxps xmm0 ,xmm3
movaps [rsi], xmm0
movaps xmm0, xmm4
add rsi, 16
cmp rbx, rcx
jl loop_start
	xor ebx, ebx
	mov rcx, len
	mov rdi, inPtr
	mov rsi, outPtr
	vmovaps ymm0, [rdi]
	
	loop_start:
	vshufps ymm2, ymm0,
	[rdi+16], 0x4E
	vshufps ymm1, ymm0,
	ymm2, 0x99
	
	add rbx, 8
	add rdi, 32
	
	vminps ymm4, ymm0 ,ymm1
	vmaxps ymm0, ymm0, ymm1
	vminps ymm3, ymm0, ymm2
	vmaxps ymm5, ymm3, ymm4
	vmovaps [rsi], ymm5
	add rsi, 32
	vmovaps ymm0, [rdi]
	cmp rbx, rcx
	jl loop_start
		xor ebx, ebx
		mov rcx, len
		mov rdi, inPtr
		mov rsi, outPtr
		vmovaps ymm0, [rdi]
		
		loop_start:
		add rdi, 32
		vmovaps ymm6, [rdi]
		vperm2f128 ymm1, ymm0, ymm6, 0x21
		vshufps ymm3, ymm0, ymm1, 0x4E
		
		vshufps ymm2,ymm0,ymm3,0x99
		add rbx, 8
		vminps ymm5, ymm0, ymm2
		vmaxps ymm0, ymm0, ymm2
		vminps ymm4, ymm0, ymm3
		vmaxps ymm7, ymm4, ymm5
		vmovaps ymm0, ymm6
		vmovaps [rsi], ymm7
		add rsi, 32
		cmp rbx, rcx
		jl loop_start

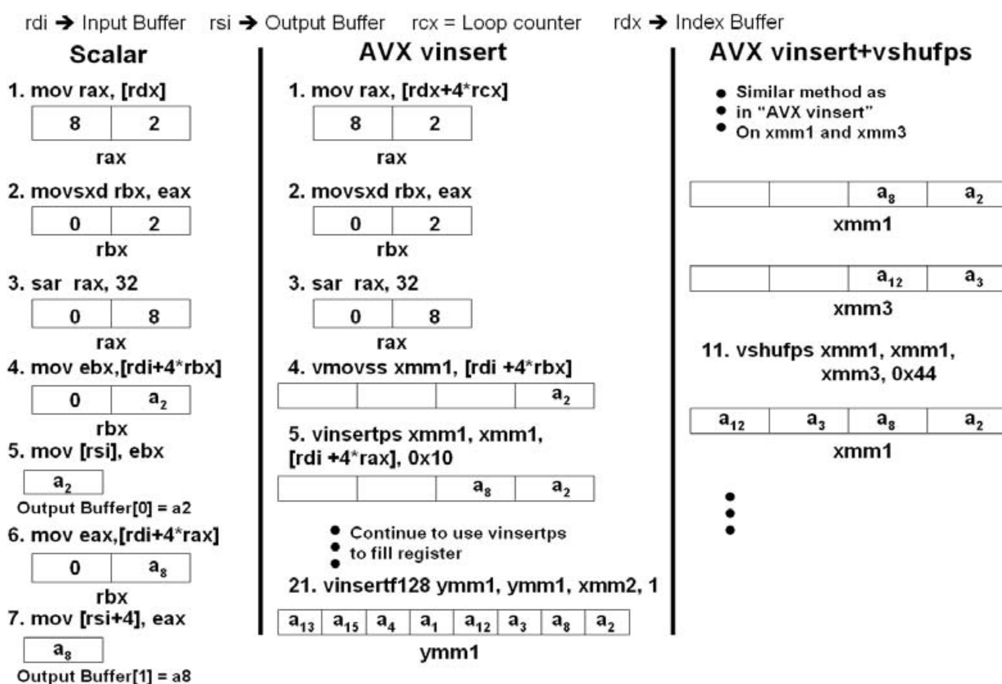
11.5 DATA GATHER AND SCATTER

This section describes techniques for implementing data gather and scatter operations using Intel AVX instructions.

11.5.1 Data Gather

The gather operation reads elements from an input buffer based on indexes specified in an index buffer. The gathered elements are written in an output buffer. The following figure illustrates an example for a gather operation.

Following are 3 implementations for the gather operation from an array of 4 byte elements. Alternative 1 is a scalar implementation using general purpose registers. Alternative 2 and 3 use Intel AVX instructions. The figure below shows code snippets from Example 11-8 assuming that it runs the first iteration on data from the previous figure.



Performance of the Intel AVX examples is similar to the performance of a corresponding Intel SSE implementation. The table below shows the three gather implementations.

Example 11-8. Data Gather - AVX versus Scalar Code

1: Scalar Code	2: AVX w/ VINSRT	3: VINSRT+VSHUFPS

movr di, InBuf
mov rsi, OutBuf
mov rdx, Index
xor rcx, rcx
	movr di, InBuf
	mov rsi, OutBuf
	mov rdx, Index
	xor rcx, rcx
		movr di, InBuf
		mov rsi, OutBuf
		mov rdx, Index
		xor rcx, rcx

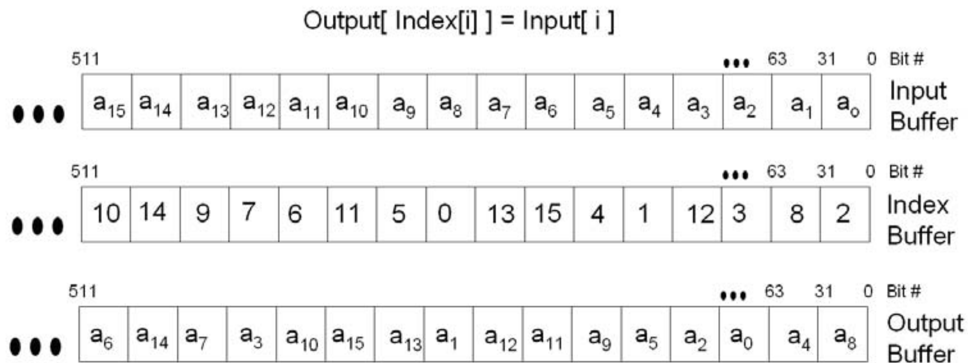
Example 11-8. Data Gather - AVX versus Scalar Code (Contd.)

1: Scalar Code	2: AVX w/ VINSRT	3: VINSRT+VSHUFPS

loop1:
mov rax, [rdx]
movsxd rbx, eax
sar rax, 32
mov ebx,[rdi+4*rbx]
mov [rsi], ebx
mov eax,[rdi+4*rax]
mov [rsi+4], eax
 	
mov rax, [rdx+8]
movsxd rbx, eax
sar rax, 32
mov ebx, [rdi+4*rbx]
mov [rsi+8], ebx
mov eax,[rdi+4*rax]
mov [rsi+12], eax
 	
mov rax, [rdx+16]
movsxd rbx, eax
sar rax, 32
mov ebx, [rdi+4*rbx]
mov [rsi+16], ebx
mov eax, [rdi+4*rax]
mov [rsi+20], eax
 	
mov rax, [rdx+24]
movsxd rbx, eax
sar rax, 32
mov ebx, [rdi+4*rbx]
mov [rsi+24], ebx
mov eax, [rdi+4*rax]
mov [rsi+28], eax
 	
addrsi, 32
addrdx, 32
addrcx, 8
cmprcx, len
jl loop1
	loop1:
	mov rax, [rdx+4*rcx]
	movsxd rbx, eax
	sar rax, 32
	vmovss xmm1, [rdi +4*rbx]
	vinsertps xmm1, xmm1,
	[rdi +4*rax], 0x10
	mov rax, [rdx + 8+4*rcx]
	movsxd rbx, eax
	sar rax, 32
	vinsertps xmm1, xmm1,
	[rdi +4*rbx], 0x20
	
	vinsertps xmm1, xmm1,
	[rdi +4*rax], 0x30
	
	mov rax, [rdx + 16+4*rcx]
	movsxd rbx, eax
	sar rax, 32
	vmovss xmm2, [rdi +4*rbx]
	vinsertps xmm2, xmm2,
	[rdi +4*rax], 0x10
	
	mov rax,[rdx + 24+4*rcx]
	movsxd rbx, eax
	sar rax, 32
	vinsertps xmm2, xmm2,
	[rdi +4*rbx], 0x20
	
	vinsertps xmm2, xmm2,
	[rdi +4*rax], 0x30
	
	vinsertf128 ymm1, ymm1,
	xmm2, 1
	
	vmovaps [rsi+4*rcx], ymm1
	add rcx, 8
	cmp rcx, len
	jl loop1
		loop1:
		mov rax, [rdx+4*rcx]
		movsxd rbx, eax
		sar rax, 32
		vmovss xmm1, [rdi +4*rbx]
		vinsertps xmm1, xmm1,
		[rdi +4*rax], 0x10
		mov rax, [rdx + 8+4*rcx]
		movsxd rbx, eax
		sar rax, 32
		vmovss xmm3, [rdi +4*rbx]
		vinsertps xmm3, xmm3,
		[rdi +4*rax], 0x10
		
		vshufps xmm1, xmm1,
		xmm3, 0x44
		
		mov rax, [rdx + 16+4*rcx]
		movsxd rbx, eax
		sar rax, 32
		vmovss xmm2, [rdi +4*rbx]
		vinsertps xmm2, xmm2,
		[rdi +4*rax], 0x10
		
		Mov rax,[rdx + 24+4*rcx]
		movsxd rbx, eax
		sar rax, 32
		vmovss xmm4, [rdi +4*rbx]
		vinsertps xmm4, xmm4,
		[rdi +4*rax], 0x10
		
		vshufpsxmm2, xmm2,
		xmm4, 0x44
		
		vinsertf128 ymm1, ymm1,
		xmm2, 1
		
		vmovaps [rsi+4*rcx], ymm1
		add rcx, 8
		cmp rcx, len
		jl loop1

11.5.2 Data Scatter

The scatter operation reads elements from an input buffer sequentially. It then writes them to an output buffer based on indexes specified in an index buffer. The following figure illustrates an example for a scatter operation.



The following table includes a scalar implementation and an Intel AVX implementation of a scatter sequence. The Intel AVX examples consist mainly of 128-bit Intel AVX instructions. Performance of the Intel AVX examples is similar to the performance of corresponding Intel SSE implementation.

Example 11-9. Scatter Operation Using AVX

Scalar Code	AVX Code
mov rdi, InBuf	mov rdi, InBuf
mov rsi, OutBuf	mov rsi, OutBuf
mov rdx, Index	mov rdx, Index
xorr cx, rcx	xorr cx, rcx
loop1:	loop1:
movsxd rax, [rdx]	vmovaps ymm0, [rdi + 4*rcx]
mov ebx, [rdi]	movsxd rax, [rdx + 4*rcx]
mov [rsi + 4*rax], ebx	movsxd rbx, [rdx + 4*rcx + 4]
movsxd rax, [rdx + 4]	vmovss [rsi + 4*rax], xmm0
mov ebx, [rdi + 4]	movsxd rax, [rdx + 4*rcx + 8]
mov [rsi + 4*rax], ebx	vpalignr xmm1, xmm0, xmm0, 4
movsxd rax, [rdx + 8]	
	vmovss [rsi + 4*rbx], xmm1
mov ebx, [rdi + 8]	movsxd rbx, [rdx + 4*rcx + 12]
mov [rsi + 4*rax], ebx	vpalignr xmm2, xmm0, xmm0, 8
movsxd rax, [rdx + 12]	vmovss [rsi + 4*rax], xmm2
mov ebx, [rdi + 12]	movsxd rax, [rdx + 4*rcx + 16]
mov [rsi + 4*rax], ebx	vpalignr xmm3, xmm0, xmm0, 12
movsxd rax, [rdx + 16]	vmovss [rsi + 4*rbx], xmm3
mov ebx, [rdi + 16]	movsxd rbx, [rdx + 4*rcx + 20]
mov [rsi + 4*rax], ebx	vextractf128 xmm0, ymm0, 1
movsxd rax, [rdx + 20]	vmovss [rsi + 4*rax], xmm0
mov ebx, [rdi + 20]	movsxd rax, [rdx + 4*rcx + 24]
mov [rsi + 4*rax], ebx	vpalignr xmm1, xmm0, xmm0, 4
movsxd rax, [rdx + 24]	vmovss [rsi + 4*rbx], xmm1
mov ebx, [rdi + 24]	movsxd rbx, [rdx + 4*rcx + 28]
mov [rsi + 4*rax], ebx	vpalignr xmm2, xmm0, xmm0, 8
movsxd rax, [rdx + 28]	vmovss [rsi + 4*rax], xmm2
mov ebx, [rdi + 28]	vpalignr xmm3, xmm0, xmm0, 12
mov [rsi + 4*rax], ebx	vmovss [rsi + 4*rbx], xmm3

Example 11-9. Scatter Operation Using AVX (Contd.)

Scalar Code	AVX Code

add rdi, 32
add rdx, 32
add rcx, 8
cmp rcx, len
jl loop1
	add rcx, 8
	cmp rcx, len
	jl loop1

11.6 DATA ALIGNMENT FOR INTEL® AVX

This section explains the benefit of aligning data that is used by Intel AVX instructions and proposes some methods to improve performance when such alignment is not possible. Most examples in this section are variations of the SAXPY kernel. SAXPY is the Scalar Alpha * X + Y algorithm.

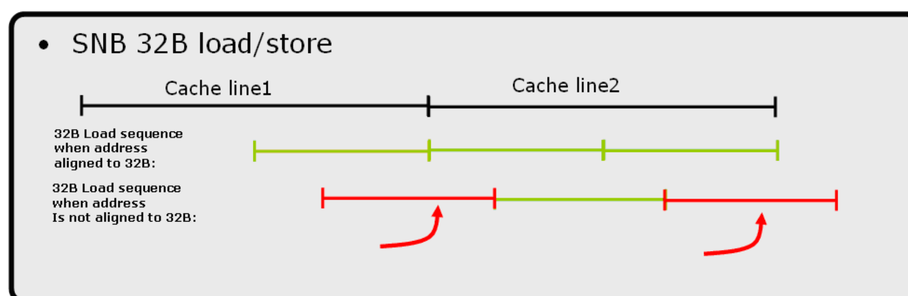
The C code below is a C implementation of SAXPY.

```
for (int i = 0; i < n; i++)
{ c[i] = alpha * a[i] + b[i]; }
```

11.6.1 Align Data to 32 Bytes

Aligning data to vector length is recommended. When using 16-byte SIMD instructions, loaded data should be aligned to 16 bytes. Similarly, for best results when using Intel AVX instructions with 32-byte registers align the data to 32-bytes.

When using Intel AVX with unaligned 32-byte vectors, every second load is a cache-line split, since the cache-line is 64 bytes. This doubles the cache line split rate compared to Intel SSE code that uses 16-byte vectors. Even though split line access penalties were reduced significantly since Intel microarchitecture code name Nehalem, a high cache-line split rate in memory-intensive code may cause performance degradation.



Example 11-10. SAXPY using Intel AVX

```
mov     rax, src1
mov     rbx, src2
mov     rcx, dst
mov     rdx, len
xor     rdi, rdi
vbroadcastss    ymm0, alpha
```

Example 11-10. SAXPY using Intel AVX (Contd.)

```
start_loop:
    vmovups         ymm1, [ rax + rdi ]
    vmulps          ymm1, ymm1, ymm0
    vmovups         ymm2, [ rbx + rdi ]
    vaddps          ymm1, ymm1, ymm2
    vmovups         [ rcx + rdi ], ymm1
    vmovups     ymm1, [ rax + rdi  + 32 ]
    vmulps      ymm1, ymm1, ymm0
    vmovups     ymm2, [ rbx + rdi + 32]
    vaddps      ymm1, ymm1, ymm2
    vmovups     [ rcx + rdi + 32 ], ymm1

    add         rdi, 64
    cmp         rdi, rdx
    jl          start_loop
```

SAXPY is a memory intensive kernel that emphasizes the importance of data alignment. Optimal performance requires both data source address to be 32-byte aligned and destination address also 32-byte aligned. If only one of the three address is not aligned to 32-byte boundary, the performance may be halved. If all three addresses are mis-aligned relative to 32 byte, the performance degrades further. In some cases, unaligned accesses may result in lower performance for Intel AVX code compared to Intel SSE code. Other Intel AVX kernels typically have more computation which can reduce the effect of the data alignment penalty.

Assembly/Compiler Coding Rule 73. (H impact, M generality) *Align data to 32-byte boundary when possible. Prefer store alignment over load alignment.*

You can use dynamic data alignment using the _mm_malloc intrinsic instruction with the Intel® Compiler, or _aligned_malloc of the Microsoft* Compiler. For example:

//dynamically allocating 32byte aligned buffer with 2048 float elements.

InputBuffer = (float*) _mm_malloc (2048*sizeof(float), 32);

You can use static data alignment using __declspec(align(32)). For example:

//Statically allocating 32byte aligned buffer with 2048 float elements.

__declspec(align(32)) float InputBuffer[2048];

11.6.2 Consider 16-Byte Memory Access when Memory is Unaligned

For best results use Intel AVX 32-byte loads and align data to 32-bytes. However, there are cases where you cannot align the data, or data alignment is unknown. This can happen when you are writing a library function and the input data alignment is unknown. In these cases, using 16-Byte memory accesses may be the best alternative. The following method uses 16-byte loads while still benefiting from the 32-byte YMM registers.

Consider replacing unaligned 32-byte memory accesses using a combination of VMOVUPS, VINSERTF128, and VEXTRACTF128 instructions.

Example 11-11. Using 16-Byte Memory Operations for Unaligned 32-Byte Memory Operation

```
Convert 32-byte loads as follows:
    vmovups      ymm0, mem   ->   vmovups xmm0, mem
                                  vinsertf128   ymm0, ymm0, mem+16, 1
Convert 32-byte stores as follows:
    vmovups mem, ymm0        ->   vmovups mem, xmm0
                                  vextractf128 mem+16, ymm0, 1

The following intrinsics are available to handle unaligned 32-byte memory operating using 16-byte memory accesses:
    _mm256_loadu2_m128 ( float const * addr_hi, float const * addr_lo);
    _mm256_loadu2_m128d ( double const * addr_hi, double const * addr_lo);
    _mm256_loadu2_m128 i( __m128i const * addr_hi, __m128i const * addr_lo);
    _mm256_storeu2_m128 ( float * addr_hi, float * addr_lo, __m256 a);
    _mm256_storeu2_m128d ( double * addr_hi, double * addr_lo, __m256d a);
    _mm256_storeu2_m128 i( __m128i * addr_hi, __m128i * addr_lo, __m256i a);
```

Example 11-12 shows two implementations for SAXPY with unaligned addresses. Alternative 1 uses 32 byte loads and alternative 2 uses 16 byte loads. These code samples are executed with two source buffers, src1, src2, at 4 byte offset from 32-Byte alignment, and a destination buffer, DST, that is 32-Byte aligned. Using two 16-byte memory operations in lieu of 32-byte memory access performs faster.

Example 11-12. SAXPY Implementations for Unaligned Data Addresses

AVX with 32-byte memory operation	AVX using two 16-byte memory operations

mov rax, src1
mov rbx, src2
mov rcx, dst
mov rdx, len
xor rdi, rdi
vbroadcastss ymm0, alpha
start_loop:
vmovups ymm1, [rax + rdi]
vmulps ymm1, ymm1, ymm0
vmovups ymm2, [rbx + rdi]
vaddps ymm1, ymm1, ymm2
vmovups [rcx + rdi], ymm1
 	
vmovups ymm1, [rax+rdi+32]
vmulps ymm1, ymm1, ymm0
 	
vmovups ymm2, [rbx+rdi+32]
vaddps ymm1, ymm1, ymm2
vmovups [rcx+rdi+32], ymm1
 	
add rdi, 64
cmp rdi, rdx
jl start_loop
	mov rax, src1
	mov rbx, src2
	mov rcx, dst
	mov rdx, len
	xor rdi, rdi
	vbroadcastss ymm0, alpha
	start_loop:
	vmovups xmm2, [rax+rdi]
	vinsertf128 ymm2, ymm2, [rax+rdi+16], 1
	vmulps ymm1, ymm0, ymm2
	vmovups xmm2, [rbx + rdi]
	vinsertf128 ymm2, ymm2, [rbx+rdi+16], 1
	vaddps ymm1, ymm1, ymm2
	vaddps ymm1, ymm1, ymm2
	vmovaps [rcx+rdi], ymm1
	vmovups xmm2, [rax+rdi+32]
	vinsertf128 ymm2, ymm2, [rax+rdi+48], 1
	vmulps ymm1, ymm0, ymm2
	vmovups xmm2, [rbx+rdi+32]
	vinsertf128 ymm2, ymm2, [rbx+rdi+48], 1
	vaddps ymm1, ymm1, ymm2
	vmovups [rcx+rdi+32], ymm1
	add rdi, 64
	cmp rdi, rdx
	jl start_loop

Assembly/Compiler Coding Rule 74. (M impact, H generality) *Align data to 32-byte boundary when possible. Prefer store alignment over load alignment.*

11.6.3 Prefer Aligned Stores Over Aligned Loads

There are cases where it is possible to align only a subset of the processed data buffers. In these cases, aligning data buffers used for store operations usually yields better performance than aligning data buffers used for load operations.

Unaligned stores are likely to cause greater performance degradation than unaligned loads, since there is a very high penalty on stores to a split cache-line that crosses pages. This penalty is estimated at 150 cycles. Loads that cross a page boundary are executed at retirement. In Example 11-12, unaligned store address can affect SAXPY performance for 3 unaligned addresses to about one quarter of the aligned case.

11.7 L1D CACHE LINE REPLACEMENTS

When a load misses the L1D Cache, a cache line with the requested data is brought from a higher memory hierarchy level. In memory intensive code where the L1 DCache is always active, replacing a cache line in the L1 DCache may delay other loads. In Intel microarchitecture code name Sandy Bridge and Ivy Bridge, the penalty for 32-Byte loads may be higher than the penalty for 16-Byte loads. Therefore, memory intensive Intel AVX code with 32-Byte loads and with data set larger than the L1 DCache may be slower than similar code with 16-Byte loads.

When Example 11-12 is run with a data set that resides in the L2 Cache, the 16-byte memory access implementation is slightly faster than the 32-byte memory operation.

Be aware that the relative merit of 16-Byte memory accesses versus 32-byte memory access is implementation specific across generations of microarchitectures.

With Intel microarchitecture code name Haswell, the L1 DCache can support two 32-byte fetch each cycle, this cache line replacement concern does not apply.

11.8 4K ALIASING

4-KByte memory aliasing occurs when the code stores to one memory location and shortly after that it loads from a different memory location with a 4-KByte offset between them. For example, a load to linear address 0x400020 follows a store to linear address 0x401020.

The load and store have the same value for bits 5 - 11 of their addresses and the accessed byte offsets should have partial or complete overlap.

4K aliasing may have a five-cycle penalty on the load latency. This penalty may be significant when 4K aliasing happens repeatedly and the loads are on the critical path. If the load spans two cache lines it might be delayed until the conflicting store is committed to the cache. Therefore 4K aliasing that happens on repeated unaligned Intel AVX loads incurs a higher performance penalty.

To detect 4K aliasing, use the LD_BLOCKS_PARTIAL.ADDRESS_ALIAS event that counts the number of times Intel AVX loads were blocked due to 4K aliasing.

To resolve 4K aliasing, try the following methods in the following order:

- Align data to 32 Bytes.
- Change offsets between input and output buffers if possible.
- Use 16-Byte memory accesses on memory which is not 32-Byte aligned.

11.9 CONDITIONAL SIMD PACKED LOADS AND STORES

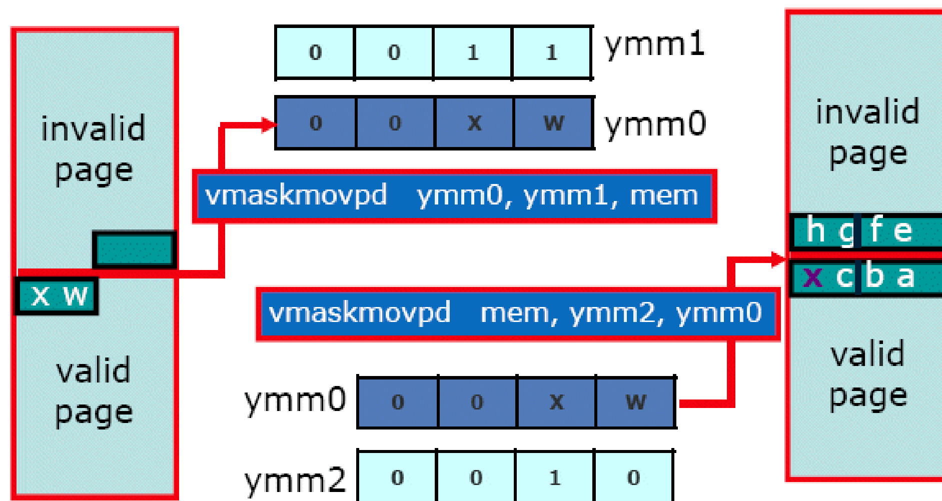
The VMASKMOV instruction conditionally moves packed data elements to/from memory, depending on the mask bits associated with each data element. The mask bit for each data element is the most significant bit of the corresponding element in the mask register.

When performing a mask load, the returned value is 0 for elements which have a corresponding mask value of 0. The mask store instruction writes to memory only the elements with a corresponding mask value of 1, while preserving memory values for elements with a corresponding mask value of 0. Faults can occur only for memory accesses that are required by the mask. Faults do not occur due to referencing any memory location if the corresponding mask bit value for that memory location is zero. For example, no faults are detected if the mask bits are all zero.

The following figure shows an example for a mask load and a mask store which does not cause a fault. In this example, the mask register for the load operation is ymm1 and the mask register for the store operation is ymm2.

When using masked load or store consider the following:

- The address of a VMASKMOV store is considered as resolved only after the mask is known. Loads that follow a masked store can be blocked until the mask value is known (unless relieved by the memory disambiguator).
- If the mask is not all 1 or all 0, loads that depend on the masked store have to wait until the store data is written to the cache. If the mask is all 1 the data can be forwarded from the masked store to the dependent loads. If the mask is all 0 the loads do not depend on the masked store.



- Masked loads including an illegal address range do not result in an exception if the range is under a zero mask value. However, the processor may take a multi-hundred-cycle “assist” to determine that no part of the illegal range have a one mask value. This assist may occur even when the mask is “zero” and it seems obvious to the programmer that the load should not be executed.

When using VMASKMOV, consider the following:

- Use VMASKMOV only in cases where VMOVUPS cannot be used.
- Use VMASKMOV on 32Byte aligned addresses if possible.
- If possible use valid address range for masked loads, even if the illegal part is masked with zeros.
- Determine the mask as early as possible.
- Avoid store-forwarding issues by performing loads prior to a VMASKMOV store if possible.

- Be aware of mask values that would cause the VMASKMOV instruction to require assist (if an assist is required, the latency of VMASKMOV to load data will increase dramatically):
  - Load data using VMASKMOV with a mask value selecting 0 elements from an illegal address will require an assist.
  - Load data using VMASKMOV with a mask value selecting 0 elements from a legal address expressed in some addressing form (e.g. [base+index], disp[base+index])will require an assist.

With processors based on the Skylake microarchitecture, the performance characteristics of VMASKMOV instructions have the following notable items:

- Loads that follow a masked store is not longer blocked until the mask value is known.
- Store data using VMASKMOV with a mask value permitting 0 elements to be written to an illegal address will require an assist.

11.9.1 Conditional Loops

VMASKMOV enables vectorization of loops that contain conditional code. There are two main benefits in using VMASKMOV over the scalar implementation in these cases:

- VMASKMOV code is vectorized.
- Branch mispredictions are eliminated.

Below is a conditional loop C code:

Example 11-13. Loop with Conditional Expression

```
for(int i = 0; i < miBufferWidth; i++)
{       if(A[i]>0)
        {           B[i] = (E[i]*C[i]);
        }
        else
        {           B[i] = (E[i]*D[i]);
        }
}
```

Example 11-14. Handling Loop Conditional with VMASKMOV

Scalar	AVX using VMASKMOV

float* pA = A;
float* pB = B;
float* pC = C;
float* pD = D;
float* pE = E;
uint64 len = (uint64) (miBuffer-Width)*sizeof(float);
__asm
{
    mov rax, pA
    mov rbx, pB
    mov rcx, pC
    mov rdx, pD
    mov rsi, pE
    mov r8, len
	float* pA = A;
	float* pB = B;
	float* pC = C;
	float* pD = D;
	float* pE = E;
	uint64 len = (uint64) (miBufferWidth)*sizeof(float);
	__asm
	{
	    mov rax, pA
	    mov rbx, pB
	    mov rcx, pC
	    mov rdx, pD
	    mov rsi, pE
	    mov r8, len

Example 11-14. Handling Loop Conditional with VMASKMOV (Contd.)

Scalar	AVX using VMASKMOV

//xmm8 all zeros
    vxorps xmm8, xmm8, xmm8
 	
    xor r9,r9
loop1:
    vmovss xmm1, [rax+r9]
    vcomiss xmm1, xmm8
    jbe a_le
a_gt:
    vmovss xmm4, [rcx+r9]
    jmp mul
a_le:
    vmovss xmm4, [rdx+r9]
mul:
    vmulss xmm4, xmm4, [rsi+r9]
    vmovss [rbx+r9], xmm4
    add r9, 4
    cmp r9, r8
  jl loop1
}
	//ymm8 all zeros
	  vxorps ymm8, ymm8, ymm8
	    //ymm9 all ones
	  vcmpps ymm9, ymm8, ymm8, 0
	  xor r9,r9
	loop1:
	  vmovups ymm1, [rax+r9]
	  vcmpps ymm2, ymm8, ymm1, 1
	  vmaskmovps ymm4, ymm2, [rcx+r9]
	  vxorps ymm2, ymm2, ymm9
	  vmaskmovps ymm5, ymm2, [rdx+r9]
	  vorps ymm4, ymm4, ymm5
	  vmulps ymm4,ymm4, [rsi+r9]
	  vmovups [rbx+r9], ymm4
	  add r9, 32
	  cmp r9, r8
	  jl loop1
	}

The performance of the left side of Example 11-14 is sensitive to branch mis-predictions and can be an order of magnitude slower than the VMASKMOV example which has no data-dependent branches.

11.10 MIXING INTEGER AND FLOATING-POINT CODE

Integer SIMD functionalities in Intel AVX instructions are limited to 128-bit. There are some algorithm that uses mixed integer SIMD and floating-point SIMD instructions. Therefore, porting such legacy 128-bit code into 256-bit AVX code requires special attention.

For example, PALINGR (Packed Align Right) is an integer SIMD instruction that is useful arranging data elements for integer and floating-point code. But VPALINGR instruction does not have a corresponding 256-bit instruction in AVX.

There are two approaches to consider when porting legacy code consisting of mostly floating-point with some integer operations into 256-bit AVX code:

- Locate a 256-bit AVX alternative to replace the critical128-bit Integer SIMD instructions if such an AVX instructions exist. This is more likely to be true with integer SIMD instruction that re-arranges data elements.
- Mix 128-bit AVX and 256-bit AVX instructions.

The performance gain from these two approaches may vary. Where possible, use method (1), since this method utilizes the full 256-bit vector width.

In case the code is mostly integer, convert the code from 128-bit SSE to 128 bit AVX instructions and gain from the Non destructive Source (NDS) feature.

Example 11-15. Three-Tap Filter in C Code

```
for(int i = 0; i < len -2; i++)
{
    pOut[i] = A[i]*coeff[0]+A[i+1]*coeff[1]+A[i+2]*coeff[2];{B[i] = (E[i]*D[i]);
}
```

Example 11-16. Three-Tap Filter with 128-bit Mixed Integer and FP SIMD

```
    xor ebx, ebx
    mov rcx, len
    mov rdi, inPtr
    mov rsi, outPtr
    mov r15, coeffs
    movss xmm2, [r15]    //load coeff 0
    shufps xmm2, xmm2, 0 //broadcast coeff 0
    movss xmm1, [r15+4]  //load coeff 1
    shufps xmm1, xmm1, 0 //broadcast coeff 1
    movss xmm0, [r15+8]  //coeff 2
    shufps xmm0, xmm0, 0 //broadcast coeff 2
    movaps xmm5, [rdi]   //xmm5={A[n+3],A[n+2],A[n+1],A[n]}


loop_start:
    movaps xmm6, [rdi+16] //xmm6={A[n+7],A[n+6],A[n+5],A[n+4]}
    movaps xmm7, xmm6
    movaps xmm8, xmm6
    add rdi, 16          //inPtr+=32
    add rbx, 4           //loop counter
    palignr xmm7, xmm5, 4 //xmm7={A[n+4],A[n+3],A[n+2],A[n+1]}
    palignr xmm8, xmm5, 8 //xmm8={A[n+5],A[n+4],A[n+3],A[n+2]}
    mulps   xmm5, xmm2   //xmm5={C0*A[n+3],C0*A[n+2],C0*A[n+1], C0*A[n]}



    mulps   xmm7, xmm1   //xmm7={C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
    mulps   xmm8, xmm0   //xmm8={C2*A[n+5],C2*A[n+4] C2*A[n+3],C2*A[n+2]}
    addps   xmm7 ,xmm5
    addps   xmm7, xmm8
    movaps [rsi], xmm7
    movaps  xmm5, xmm6
    add rsi, 16        //outPtr+=16
    cmp rbx, rcx
    jl  loop_start
```

Example 11-17. 256-bit AVX Three-Tap Filter Code with VSHUFPS

```
    xor ebx, ebx
    mov rcx, len
    mov rdi, inPtr
    mov rsi, outPtr
    mov r15, coeffs
    vbroadcastss ymm2, [r15]   //load and broadcast coeff 0
    vbroadcastss ymm1, [r15+4] //load and broadcast coeff 1
    vbroadcastss ymm0, [r15+8] //load and broadcast coeff 2
```

Example 11-17. 256-bit AVX Three-Tap Filter Code with VSHUFPS (Contd.)

```
loop_start:
    vmovaps ymm5, [rdi]          // Ymm5={A[n+7],A[n+6],A[n+5],A[n+4];
                                 // A[n+3],A[n+2],A[n+1] , A[n]}
    vshufps ymm6,ymm5,[rdi+16],0x4e // ymm6={A[n+9],A[n+8],A[n+7],A[n+6];
                                 // A[n+5],A[n+4],A[n+3],A[n+2]}
    vshufps ymm7,ymm5,ymm6,0x99     // ymm7={A[n+8],A[n+7],A[n+6],A[n+5];
                                 // A[n+4],A[n+3],A[n+2],A[n+1]}
    vmulps  ymm3,ymm5,ymm2// ymm3={C0*A[n+7],C0*A[n+6],C0*A[n+5],C0*A[n+4];
                             // C0*A[n+3],C0*A[n+2],C0*A[n+1],C0*A[n]}
    vmulps  ymm9,ymm7,ymm1    // ymm9={C1*A[n+8],C1*A[n+7],C1*A[n+6],C1*A[n+5];
                             // C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
    vmulps  ymm4,ymm6,ymm0    // ymm4={C2*A[n+9],C2*A[n+8],C2*A[n+7],C2*A[n+6];
                             // C2*A[n+5],C2*A[n+4],C2*A[n+3],C2*A[n+2]}
    vaddps    ymm8 ,ymm3,ymm4
    vaddps    ymm10, ymm8, ymm9
    vmovaps   [rsi], ymm10
    add       rdi, 32 //inPtr+=32
    add       rbx, 8  //loop counter
    add       rsi, 32 //outPtr+=32
    cmp       rbx, rcx
    jl        loop_start
```

Example 11-18. Three-Tap Filter Code with Mixed 256-bit AVX and 128-bit AVX Code

```
    xor ebx, ebx
    mov rcx, len
    mov rdi, inPtr
    mov rsi, outPtr
    mov r15, coeffs
    vbroadcastss ymm2, [r15]   //load and broadcast coeff 0
    vbroadcastss ymm1, [r15+4] //load and broadcast coeff 1
    vbroadcastss ymm0, [r15+8] //load and broadcast coeff 2
    vmovaps xmm3, [rdi]    //xmm3={A[n+3],A[n+2],A[n+1],A[n]}
    vmovaps xmm4, [rdi+16] //xmm4={A[n+7],A[n+6],A[n+5],A[n+4]}
    vmovaps xmm5, [rdi+32] //xmm5={A[n+11], A[n+10],A[n+9],A[n+8]}
loop_start:
    vinsertf128 ymm3, ymm3, xmm4, 1 // ymm3={A[n+7],A[n+6],A[n+5],A[n+4];
                                 // A[n+3], A[n+2],A[n+1],A[n]}
    vpalignr xmm6, xmm4, xmm3, 4    // xmm6={A[n+4],A[n+3],A[n+2],A[n+1]}
    vpalignr xmm7, xmm5, xmm4, 4    // xmm7={A[n+8],A[n+7],A[n+6],A[n+5]}
     vinsertf128 ymm6,ymm6,xmm7,1      // ymm6={A[n+8],A[n+7],A[n+6],A[n+5];
                                 // A[n+4],A[n+3],A[n+2],A[n+1]}
    vpalignr xmm8,xmm4,xmm3,8       // xmm8={A[n+5],A[n+4],A[n+3],A[n+2]}
    vpalignr xmm9, xmm5, xmm4, 8    // xmm9={A[n+9],A[n+8],A[n+7],A[n+6]}
    vinsertf128 ymm8, ymm8, xmm9,1  // ymm8={A[n+9],A[n+8],A[n+7],A[n+6];
                                 // A[n+5],A[n+4],A[n+3],A[n+2]}
    vmulps   ymm3,ymm3,ymm2 // Ymm3={C0*A[n+7],C0*A[n+6],C0*A[n+5], C0*A[n+4];
                              // C0*A[n+3],C0*A[n+2],C0*A[n+1],C0*A[n]}
    vmulps   ymm6,ymm6,ymm1    // Ymm9={C1*A[n+8],C1*A[n+7],C1*A[n+6],C1*A[n+5];
                              // C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
```

Example 11-18. Three-Tap Filter Code with Mixed 256-bit AVX and 128-bit AVX Code (Contd.)

```
vmulps   ymm8,ymm8,ymm0 // Ymm4={C2*A[n+9],C2*A[n+8],C2*A[n+7],C2*A[n+6];
                         // C2*A[n+5],C2*A[n+4],C2*A[n+3],C2*A[n+2]}
vaddps   ymm3 ,ymm3,ymm6
vaddps   ymm3, ymm3, ymm8
vmovaps  [rsi], ymm3
vmovaps xmm3, xmm5
add      rdi, 32 //inPtr+=32
add      rbx, 8  //loop counter
add      rsi, 32 //outPtr+=32
cmp      rbx, rcx
jl       loop_start
```

Example 11-17 uses 256-bit VSHUFPS to replace the PALIGNR in 128-bit mixed SSE code. This speeds up almost 70% over the 128-bit mixed SSE code of Example 11-16 and slightly ahead of Example 11-18.

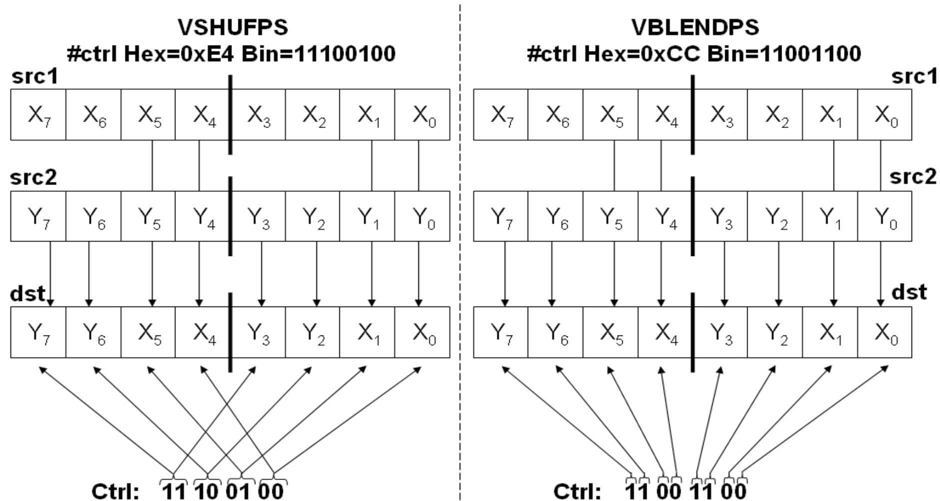
For code that includes integer instructions and is written with 256-bit Intel AVX instructions, replace the integer instruction with floating-point instructions that have similar functionality and performance. If there is no similar floating-point instruction, consider using a 128-bit Intel AVX instruction to perform the required integer operation.

11.11 HANDLING PORT 5 PRESSURE

Port 5 in Intel microarchitecture code name Sandy Bridge includes shuffle units and it frequently becomes a performance bottleneck. Sometimes it is possible to replace shuffle instructions that dispatch only on port 5, with different instructions and improve performance by reducing port 5 pressure. For more information, see Table 2-15.

11.11.1 Replace Shuffles with Blends

There are a few cases where shuffles such as VSHUFPS or VPERM2F128 can be replaced by blend instructions. Intel AVX shuffles are executed only on port 5, while blends are also executed on port 0. Therefore, replacing shuffles with blends could reduce port 5 pressure. The following figure shows how a VSHUFPS is implemented using VBLENDPS.



The following example shows two implementations of an 8x8 Matrix transpose. In both cases, the bottleneck is Port 5 pressure. Alternative 1 uses 12 vshufps instructions that are executed only on port 5. Alternative 2 replaces eight of the vshufps instructions with the vblendps instruction which can be executed on Port 0.

Example 11-19. 8x8 Matrix Transpose - Replace Shuffles with Blends

256-bit AVX using VSHUFPS	AVX replacing VSHUFPS with VBLENDPS
movrcx, inpBuf	movrcx, inpBuf
movrdx, outBuf	movrdx, outBuf
movr10, NumOfLoops	movr10, NumOfLoops
movrbx, rdx	movrbx, rdx
loop1:	loop1:
vmovaps ymm9, [rcx]	vmovaps ymm9, [rcx]
vmovaps ymm10, [rcx+32]	vmovaps ymm10, [rcx+32]
vmovaps ymm11, [rcx+64]	vmovaps ymm11, [rcx+64]
vmovaps ymm12, [rcx+96]	vmovaps ymm12, [rcx+96]
vmovaps ymm13, [rcx+128]	vmovaps ymm13, [rcx+128]
vmovaps ymm14, [rcx+160]	vmovaps ymm14, [rcx+160]
vmovaps ymm15, [rcx+192]	vmovaps ymm15, [rcx+192]
vmovaps ymm2, [rcx+224]	vmovaps ymm2, [rcx+224]
vunpcklps ymm6, ymm9, ymm10	vunpcklps ymm6, ymm9, ymm10
vunpcklps ymm1, ymm11, ymm12	vunpcklps ymm1, ymm11, ymm12
vunpckhps ymm8, ymm9, ymm10	vunpckhps ymm8, ymm9, ymm10
vunpcklps ymm0, ymm13, ymm14	vunpcklps ymm0, ymm13, ymm14
vunpcklps ymm9, ymm15, ymm2	vunpcklps ymm9, ymm15, ymm2
vshufps ymm3, ymm6, ymm1, 0x4E	vshufps ymm3, ymm6, ymm1, 0x4E
vshufps ymm10, ymm6, ymm3, 0xE4	vblendps ymm10, ymm6, ymm3, 0xCC
vshufps ymm6, ymm0, ymm9, 0x4E	vshufps ymm6, ymm0, ymm9, 0x4E
vunpckhps ymm7, ymm11, ymm12	vunpckhps ymm7, ymm11, ymm12
vshufps ymm11, ymm0, ymm6, 0xE4	vblendps ymm11, ymm0, ymm6, 0xCC
vshufps ymm12, ymm3, ymm1, 0xE4	vblendps ymm12, ymm3, ymm1, 0xCC
vperm2f128 ymm3, ymm10, ymm11, 0x20	vperm2f128 ymm3, ymm10, ymm11, 0x20
vmovaps [rdx], ymm3	vmovaps [rdx], ymm3
vunpckhps ymm5, ymm13, ymm14	vunpckhps ymm5, ymm13, ymm14
vshufps ymm13, ymm6, ymm9, 0xE4	vblendps ymm13, ymm6, ymm9, 0xCC
vunpckhps ymm4, ymm15, ymm2	vunpckhps ymm4, ymm15, ymm2
vperm2f128 ymm2, ymm12, ymm13, 0x20	vperm2f128 ymm2, ymm12, ymm13, 0x20
vmovaps 32[rdx], ymm2	vmovaps 32[rdx], ymm2
vshufps ymm14, ymm8, ymm7, 0x4	vshufps ymm14, ymm8, ymm7, 0x4E
vshufps ymm15, ymm14, ymm7, 0xE4	vblendps ymm15, ymm14, ymm7, 0xCC
vshufps ymm7, ymm5, ymm4, 0x4E	vshufps ymm7, ymm5, ymm4, 0x4E
vshufps ymm8, ymm8, ymm14, 0xE4	vblendps ymm8, ymm8, ymm14, 0xCC
vshufps ymm5, ymm5, ymm7, 0xE4	vblendps ymm5, ymm5, ymm7, 0xCC
vperm2f128 ymm6, ymm8, ymm5, 0x20	vperm2f128 ymm6, ymm8, ymm5, 0x20
vmovaps 64[rdx], ymm6	vmovaps 64[rdx], ymm6
vshufps ymm4, ymm7, ymm4, 0xE4	vblendps ymm4, ymm7, ymm4, 0xCC
vperm2f128 ymm7, ymm15, ymm4, 0x20	vperm2f128 ymm7, ymm15, ymm4, 0x20
vmovaps 96[rdx], ymm7	vmovaps 96[rdx], ymm7
vperm2f128 ymm1, ymm10, ymm11, 0x31	vperm2f128 ymm1, ymm10, ymm11, 0x31
vperm2f128 ymm0, ymm12, ymm13, 0x31	vperm2f128 ymm0, ymm12, ymm13, 0x31
vmovaps 128[rdx], ymm1	vmovaps 128[rdx], ymm1
vperm2f128 ymm5, ymm8, ymm5, 0x31	vperm2f128 ymm5, ymm8, ymm5, 0x31
vperm2f128 ymm4, ymm15, ymm4, 0x31	vperm2f128 ymm4, ymm15, ymm4, 0x31

Example 11-19. 8x8 Matrix Transpose - Replace Shuffles with Blends (Contd.)

256-bit AVX using VSHUFPS	AVX replacing VSHUFPS with VBLENDPS
vmovaps 160[rdx], ymm0 vmovaps 192[rdx], ymm5 vmovaps 224[rdx], ymm4 decr10 jnz loop1	vmovaps 160[rdx], ymm0 vmovaps 192[rdx], ymm5 vmovaps 224[rdx], ymm4 dec r10 jnz loop1

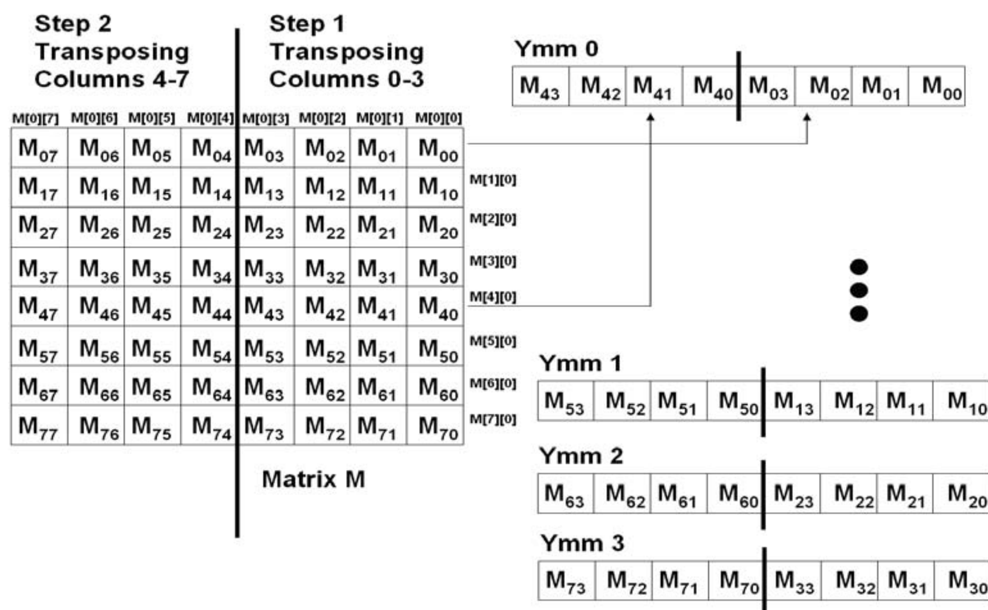
In Example 11-19, replacing VSHUFPS with VBLENDPS relieved port 5 pressure and can gain almost 40% speedup.

Assembly/Compiler Coding Rule 75. (M impact, M generality) *Use Blend instructions in lieu of shuffle instruction in AVX whenever possible.*

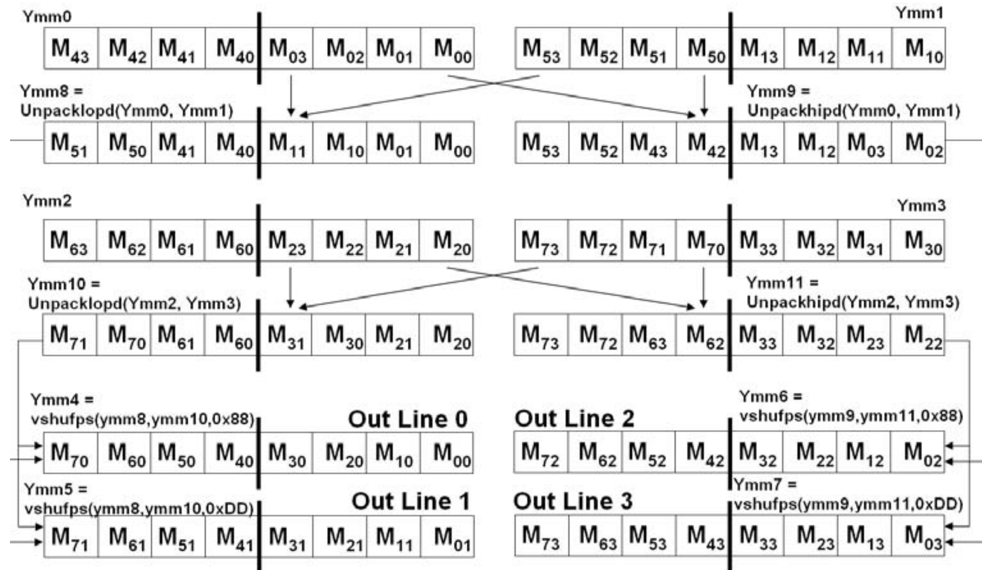
11.11.2 Design Algorithm With Fewer Shuffles

In some cases you can reduce port 5 pressure by changing the algorithm to use less shuffles. The figure below shows that the transpose moved all the elements in rows 0-4 to the low lanes, and all the elements in rows 4-7 to the high lanes. Therefore, using 256-bit loads in the beginning of the algorithm requires using VPERM2F128 in order to swap elements between the lanes. The processor executes the VPERM2F128 instruction only on port 5.

Example 11-19 used eight 256-bit loads and eight VPERM2F128 instructions. You can implement the same 8x8 Matrix Transpose using VINSERTF128 instead of the 256-bit loads and the eight VPERM2F128. Using VINSERTF128 from memory is executed in the load ports and on port 0 or 5. The original method required loads that are performed on the load ports and VPERM2F128 that is only performed on port 5. Therefore redesigning the algorithm to use VINSERTF128 reduces port 5 pressure and improves performance.



The following figure describes step 1 of the 8x8 matrix transpose with vinsertf128. Step 2 performs the same operations on different columns.



Example 11-20. 8x8 Matrix Transpose Using VINSRTPS

```
    mov     rcx, inpBuf
    mov     rdx, outBuf
    mov     r8, iLineSize
    mov     r10, NumOfLoops
loop1:
    vmovaps      xmm0, [rcx]
    vinsertf128  ymm0, ymm0, [rcx + 128], 1
    vmovaps      xmm1, [rcx + 32]
    vinsertf128  ymm1, ymm1, [rcx + 160], 1

    vunpcklpd    ymm8, ymm0, ymm1
    vunpckhpd    ymm9, ymm0, ymm1
    vmovaps      xmm2, [rcx+64]
    vinsertf128  ymm2, ymm2, [rcx + 192], 1
    vmovaps      xmm3, [rcx+96]
    vinsertf128  ymm3, ymm3, [rcx + 224], 1

    vunpcklpd    ymm10, ymm2, ymm3
    vunpckhpd    ymm11, ymm2, ymm3
    vshufps      ymm4, ymm8, ymm10, 0x88
    vmovaps      [rdx], ymm4
    vshufps      ymm5, ymm8, ymm10, 0xDD
    vmovaps      [rdx+32], ymm5
    vshufps      ymm6, ymm9, ymm11, 0x88
    vmovaps      [rdx+64], ymm6
    vshufps      ymm7, ymm9, ymm11, 0xDD
    vmovaps      [rdx+96], ymm7
```

Example 11-20. 8x8 Matrix Transpose Using VINSRTPS (Contd.)

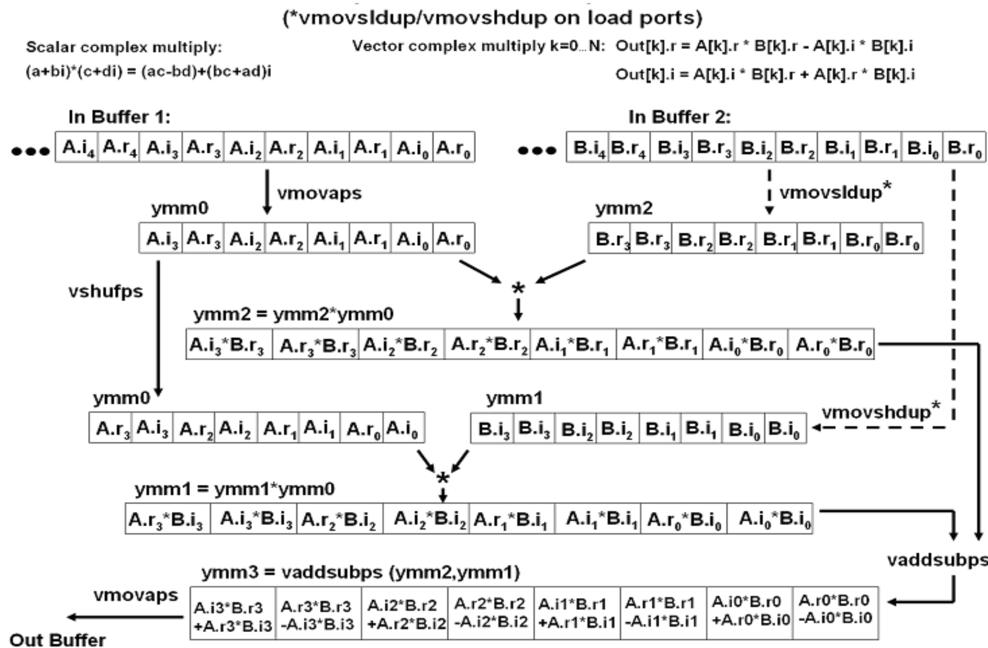
```
vmovaps      xmm0, [rcx+16]
vinsertf128  ymm0, ymm0, [rcx + 144], 1
vmovaps      xmm1, [rcx + 48]
vinsertf128 ymm1, ymm1, [rcx + 176], 1

vunpcklpd    ymm8, ymm0, ymm1
vunpckhpd    ymm9, ymm0, ymm1

vmovaps      xmm2, [rcx+80]
vinsertf128  ymm2, ymm2, [rcx + 208], 1
vmovaps      xmm3, [rcx+112]
vinsertf128  ymm3, ymm3, [rcx + 240], 1

vunpcklpd    ymm10, ymm2, ymm3
vunpckhpd    ymm11, ymm2, ymm3

vshufps      ymm4, ymm8, ymm10, 0x88
vmovaps      [rdx+128], ymm4
vshufps      ymm5, ymm8, ymm10, 0xDD
vmovaps      [rdx+160], ymm5
vshufps      ymm6, ymm9, ymm11, 0x88
vmovaps      [rdx+192], ymm6
vshufps      ymm7, ymm9, ymm11, 0xDD
vmovaps      [rdx+224], ymm7
dec          r10
jnz          loop1
```

In Example 11-20, this reduced port 5 pressure further than the combination of VSHUFPS with VBLENDPS in Example 11-19. It can gain 70% speedup relative to relying on VSHUFPS alone in Example 11-19.

11.11.3 Perform Basic Shuffles on Load Ports

Some shuffles can be executed in the load ports (ports 2, 3) if the source is from memory. The following example shows how moving some shuffles (vmovsldup/vmovshdup) from Port 5 to the load ports improves performance significantly.

The following figure describes an Intel AVX implementation of the complex multiply algorithm with vmovsldup/vmovshdup on the load ports.



Example 11-21 includes two versions of the complex multiply. Both versions are unrolled twice. Alternative 1 shuffles all the data in registers. Alternative 2 shuffles data while it is loaded from memory.

Example 11-21. Port 5 versus Load Port Shuffles

Shuffles data in registers	Shuffling loaded data
mov rax, inPtr1	mov rax, inPtr1
mov rbx, inPtr2	mov rbx, inPtr2
mov rdx, outPtr	mov rdx, outPtr
mov r8, len	mov r8, len
xor rcx, rcx	xor rcx, rcx
loop1:	loop1:
vmovaps ymm0, [rax +8*rcx]	vmovaps ymm0, [rax +8*rcx]
vmovaps ymm4, [rax +8*rcx +32] vmovaps	vmovaps ymm4, [rax +8*rcx +32]
ymm3, [rbx +8*rcx]	
vmovsldup ymm2, ymm3	vmovsldup ymm2, [rbx +8*rcx]
vmulps ymm2, ymm2, ymm0	vmulps ymm2, ymm2, ymm0
vshufps ymm0, ymm0, ymm0, 177	vshufps ymm0, ymm0, ymm0, 177
vmovshdup ymm1, ymm3	vmovshdup ymm1, [rbx +8*rcx]
vmulps ymm1, ymm1, ymm0	vmulps ymm1, ymm1, ymm0
vmovaps ymm7, [rbx +8*rcx +32]	vmovsldup ymm6, [rbx +8*rcx +32]
vmovsldup ymm6, ymm7	vmulps ymm6, ymm6, ymm4
vmulps ymm6, ymm6, ymm4	vaddsubps ymm3, ymm2, ymm1
vaddsubps ymm2, ymm2, ymm1	vmovshdup ymm5, [rbx +8*rcx +32]
vmovshdup ymm5, ymm7	

Example 11-21. Port 5 versus Load Port Shuffles (Contd.)

Shuffles data in registers	Shuffling loaded data
vmovaps [rdx+8*rcx], ymm2 vshufps ymm4, ymm4, ymm4, 177 vmulps ymm5, ymm5, ymm4 vaddsubps ymm6, ymm6, ymm5 vmovaps [rdx+8*rcx+32], ymm6 addrcx, 8 cmprcx, r8 jl loop1	vmovaps [rdx +8*rcx], ymm3 vshufps ymm4, ymm4, ymm4, 177 vmulps ymm5, ymm5, ymm4 vaddsubps ymm7, ymm6, ymm5 vmovaps [rdx +8*rcx +32], ymm7 addrcx, 8 cmprcx, r8 jl loop1

11.12 DIVIDE AND SQUARE ROOT OPERATIONS

In Intel microarchitectures prior to Skylake, the SSE divide and square root instructions DIVPS and SQRTPS have a latency of 14 cycles (or the neighborhood) and they are not pipelined. This means that the throughput of these instructions is one in every 14 cycles. The 256-bit Intel AVX instructions VDIVPS and VSQRTPS execute with 128-bit data path and have a latency of 28 cycles and they are not pipelined as well. Therefore, the performance of the Intel SSE divide and square root instructions is similar to the Intel AVX 256-bit instructions on Intel microarchitecture code name Sandy Bridge.

With the Skylake microarchitecture, 256-bit and 128-bit version of (V)DIVPS/(V)SQRTPS have the same latency because the 256-bit version can execute with a 256-bit data path. The latency is improved and is pipelined to execute with significantly improved throughput. See Appendix C, "IA-32 Instruction Latency and Throughput".

In microarchitectures that provide DIVPS/SQRTPS with high latency and low throughput, it is possible to speed up single-precision divide and square root calculations using the (V)RSQRTPS and (V)RCPPS instructions. For example, with 128-bit RCPPS/RSQRTPS at 5-cycle latency and 1-cycle throughput or with 256-bit implementation of these instructions at 7-cycle latency and 2-cycle throughput, a single Newton-Raphson iteration or Taylor approximation can achieve almost the same precision as the (V)DIVPS and (V)SQRTPS instructions. See Intel® 64 and IA-32 Architectures Software Developer's Manual for more information on these instructions.

In some cases, when the divide or square root operations are part of a larger algorithm that hides some of the latency of these operations, the approximation with Newton-Raphson can slow down execution, because more micro-ops, coming from the additional instructions, fill the pipe.

With the Skylake microarchitecture, choosing between approximate reciprocal instruction alternative versus DIVPS/SQRTPS for optimal performance of simple algebraic computations depend on a number of factors. Table 11-5 shows several algebraic formula the throughput comparison of implementations of different numeric accuracy tolerances. In each row, 24-bit accurate implementations are IEEE-compliant and using the respective instructions of 128-bit or 256-bit ISA. The columns of 22-bit and 11-bit accurate implementations are using approximate reciprocal instructions of the respective instruction set.

Table 11-5. Comparison of Numeric Alternatives of Selected Linear Algebra in Skylake Microarchitecture

Algorithm	Instruction Type	24-bit Accurate	22-bit Accurate	11-bit Accurate
Z = X/Y	SSE	1X	0.9X	1.3X
	256-bit AVX	1X	1.5X	2.6X
$Z = X^{0.5}$	SSE	1X	0.7X	2X
	256-bit AVX	1X	1.4X	3.4X
$Z = X^{-0.5}$	SSE	1X	1.7X	4.3X
	256-bit AVX	1X	3X	7.7X

Table 11-5. Comparison of Numeric Alternatives of Selected Linear Algebra in Skylake Microarchitecture

Algorithm	Instruction Type	24-bit Accurate	22-bit Accurate	11-bit Accurate
$Z = (X*Y + Y*Y)^{0.5}$	SSE	1X	0.75X	0.85X
	256-bit AVX	1X	1.1X	1.6X
Z = (X+2Y+3)/(Z-2Y-3)	SSE	1X	0.85X	1X
	256-bit AVX	1X	0.8X	1X

If targeting processors based on the Skylake microarchitecture, Table 11-5 can be summarized as:

- For 256- bit AVX code, Newton-Raphson approximation can be beneficial on Skylake microarchitecture when the algorithm contains only operations executed on the divide unit. However, when single precision divide or square root operations are part of a longer computation, the lower latency of the DIVPS or SQRTPS instructions can lead to better overall performance.
- For SSE or 128-bit AVX implementation, consider use of approximation for divide and square root instructions only for algorithms that do not require precision higher than 11-bit or algorithms that contain multiple operations executed on the divide unit.

Table 11-6 summarizes recommended calculation methods of divisions or square root when using single-precision instructions, based on the desired accuracy level across recent generations of Intel microarchitectures.

Table 11-6. Single-Precision Divide and Square Root Alternatives

Operation	Accuracy Tolerance	Recommendation
Divide	24 bits (IEEE)	DIVPS
	~ 22 bits	Skylake: Consult Table 11-5 Prior uarch: RCPPS + 1 Newton-Raphson Iteration + MULPS
	~ 11 bits	RCPPS + MULPS
Reciprocal square root	24 bits (IEEE)	SQRTPS + DIVPS
	~ 22 bits	RSQRTPS + 1 Newton-Raphson Iteration
	~ 11 bits	RSQRTPS
Square root	24 bits (IEEE)	SQRTPS
	~ 22 bits	Skylake: Consult Table 11-5 Prior uarch: RSQRTPS + 1 Newton-Raphson Iteration + MULPS
	~ 11 bits	RSQRTPS + RCPPS

11.12.1 Single-Precision Divide

To compute:

Z[i]=A[i]/B[i]

On a large vector of single-precision numbers, Z[i] can be calculated by a divide operation, or by multiplying 1/B[i] by A[i].

Denoting B[i] by N, it is possible to calculate 1/N using the (V)RCPPS instruction, achieving approximately 11-bit precision.

For better accuracy you can use the one Newton-Raphson iteration:

```
X_(0 )  ~= 1/N                ; Initial estimation, rcp(N)
X_(0 ) =  1/N*(1-E)
E=1-N*X_0                              ; E ~= 2^(-11)
X_1=X_0*(1+E)=1/N*(1-E^2 )      ; E^2 ~= 2^(-22)
```

X_1=X_0*(1+1-N*X_0)= 2 *X_0 - N*X_0^2

X_1 is an approximation of 1/N with approximately 22-bit precision.

Example 11-22. Divide Using DIVPS for 24-bit Accuracy

SSE code using DIVPS	Using VDIVPS

mov rax, pIn1
mov rbx, pIn2
mov rcx, pOut
mov rsi, iLen
xor rdx, rdx
 	
loop1:
movups xmm0, [rax+rdx*1]
movups xmm1, [rbx+rdx*1]
divps xmm0, xmm1
movups [rcx+rdx*1], xmm0
add rdx, 0x10
cmp rdx, rsi
jl loop1
	mov rax, pIn1
	mov rbx, pIn2
	mov rcx, pOut
	mov rsi, iLen
	xor rdx, rdx
	
	loop1:
	vmovups ymm0, [rax+rdx*1]
	vmovups ymm1, [rbx+rdx*1]
	vdivps ymm0, ymm0, ymm1
	vmovups [rcx+rdx*1], ymm0
	add rdx, 0x20
	cmp rdx, rsi
	jl loop1

Example 11-23. Divide Using RCPPS 11-bit Approximation

SSE code using RCPPS	Using VRCPPS

mov rax, pIn1
mov rbx, pIn2
mov rcx, pOut
mov rsi, iLen
xor rdx, rdx
 	
loop1:
movups xmm0,[rax+rdx*1]
movups xmm1,[rbx+rdx*1]
rcpps xmm1,xmm1
mulps xmm0,xmm1
movups [rcx+rdx*1],xmm0
add rdx, 16
cmp rdx, rsi
jl loop1
	mov rax, pIn1
	mov rbx, pIn2
	mov rcx, pOut
	mov rsi, iLen
	xor rdx, rdx
	
	loop1:
	vmovups ymm0, [rax+rdx]
	vmovups ymm1, [rbx+rdx]
	vrcpps ymm1, ymm1
	vmulps ymm0, ymm0, ymm1
	vmovups [rcx+rdx], ymm0
	add rdx, 32
	cmp rdx, rsi
	jl loop1

Example 11-24. Divide Using RCPPS and Newton-Raphson Iteration

RCPPS + MULPS ~ 22 bit accuracy	VRCPPS + VMULPS ~ 22 bit accuracy

mov rax, pIn1
mov rbx, pIn2
mov rcx, pOut
mov rsi, iLen
xor rdx, rdx
	mov rax, pIn1
	mov rbx, pIn2
	mov rcx, pOut
	mov rsi, iLen
	xor rdx, rdx

Example 11-24. Divide Using RCPPS and Newton-Raphson Iteration (Contd.)

RCPPS + MULPS ~ 22 bit accuracy	VRCPPS + VMULPS ~ 22 bit accuracy

loop1:
movups xmm0, [rax+rdx*1]
movups xmm1, [rbx+rdx*1]
rcpps xmm3, xmm1
movaps xmm2, xmm3
addps xmm3, xmm2
mulps xmm2, xmm2
mulps xmm2, xmm1
subps xmm3, xmm2
mulps xmm0, xmm3
movups xmmword ptr [rcx+rdx*1], xmm0
add rdx, 0x10
cmp rdx, rsi
jl loop1
	loop1:
	vmovups ymm0, [rax+rdx]
	vmovups ymm1, [rbx+rdx]
	vrcpps ymm3, ymm1
	
	vaddps ymm2, ymm3, ymm3
	vmulps ymm3, ymm3, ymm3
	vmulps ymm3, ymm3, ymm1
	vsubps ymm2, ymm2, ymm3
	vmulps ymm0, ymm0, ymm2
	vmovups [rcx+rdx], ymm0
	add rdx, 32
	cmp rdx, rsi
	jl loop1

Table 11-7. Comparison of Single-Precision Divide Alternatives

Accuracy	Method	SSE Performance	AVX Performance
24 bits	(V)DIVPS	Baseline	1X
~ 22 bits	(V)RCPPS + Newton-Raphson	2.7X	4.5X
~ 11 bits	(V)RCPPS	6X	8X

11.12.2 Single-Precision Reciprocal Square Root

To compute Z[i]=1/ (A[i]) ^0.5 on a large vector of single-precision numbers, denoting A[i] by N, it is possible to calculate 1/N using the (V)RSQRTPS instruction.

For better accuracy you can use one Newton-Raphson iteration:

X_0 ~=1/N ; Initial estimation RCP(N)

E=1-N*X_0^2

X_0= (1/N)^0.5 * ((1-E)^0.5) = (1/N)^0.5 * (1-E/2) ; E/2~= 2^(-11)

X_1=X_0*(1+E/2) ~= (1/N)^0.5 * (1-E^2/4) ; E^2/4?2^(-22)

X_1=X_0*(1+1/2-1/2*N*X_0^2)= 1/2*X_0*(3-N*X_0^2)

X1 is an approximation of (1/N)^0.5 with approximately 22-bit precision.

Example 11-25. Reciprocal Square Root Using DIVPS+SQRTPS for 24-bit Accuracy

Using SQRTPS, DIVPS	Using VSQRTPS, VDIVPS

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
movups xmm1, [rax+rdx]
sqrtps xmm0, xmm1
divps xmm0, xmm1
movups [rbx+rdx], xmm0
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	loop1:
	vmovups ymm1, [rax+rdx]
	vsqrtps ymm0,ymm1
	vdivps ymm0, ymm0, ymm1
	vmovups [rbx+rdx], ymm0
	add rdx, 32
	cmp rdx, rcx
	jl loop1

Example 11-26. Reciprocal Square Root Using RCPPS 11-bit Approximation

SSE code using RCPPS	Using VRCPPS

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
rsqrtps xmm0, [rax+rdx]
movups [rbx+rdx], xmm0
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	
	loop1:
	vrsqrtps ymm0, [rax+rdx]
	vmovups [rbx+rdx], ymm0
	add rdx, 32
	cmp rdx, rcx
	jl loop1

Example 11-27. Reciprocal Square Root Using RCPPS and Newton-Raphson Iteration

RCPPS + MULPS ~ 22 bit accuracy	VRCPPS + VMULPS ~ 22 bit accuracy

__declspec(align(16)) float minus_half[4] = {-0.5, -0.5, -0.5, -0.5};
__declspec(align(16)) float three[4] = {3.0, 3.0, 3.0, 3.0};
__asm
{
mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
movups xmm3, [three]
movups xmm4, [minus_half]
	__declspec(align(32)) float half[8] =
	{0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5};
	__declspec(align(32)) float three[8] =
	{3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0};
	__asm
	{
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	vmovups ymm3, [three]
	vmovups ymm4, [half]

Example 11-27. Reciprocal Square Root Using RCPPS and Newton-Raphson Iteration (Contd.)

RCPPS + MULPS ~ 22 bit accuracy	VRCPPS + VMULPS ~ 22 bit accuracy

loop1:
    movups xmm5, [rax+rdx]
    rsqrtps xmm0, xmm5
    movaps xmm2, xmm0
    mulps xmm0, xmm0
    mulps xmm0, xmm5
    subps xmm0, xmm3
    mulps xmm0, xmm2
    mulps xmm0, xmm4
    movups [rbx+rdx], xmm0
    add rdx, 16
    cmp rdx, rcx
    jl loop1
}
	loop1:
	    vmovups ymm5, [rax+rdx]
	    vrsqrtps ymm0, ymm5
	
	    vmulps ymm2, ymm0, ymm0
	    vmulps ymm2, ymm2, ymm5
	    vsubps ymm2, ymm3, ymm2
	    vmulps ymm0, ymm0, ymm2
	    vmulps ymm0, ymm0, ymm4
	
	    vmovups [rbx+rdx], ymm0
	    add rdx, 32
	    cmp rdx, rcx
	    jl loop1
	}

Table 11-8. Comparison of Single-Precision Reciprocal Square Root Operation

Accuracy	Method	SSE Performance	AVX Performance
24 bits	(V)SQRTPS + (V)DIVPS	Baseline	1X
~ 22 bits	(V)RCPPS + Newton-Raphson	5.2X	9.1X
~ 11 bits	(V)RCPPS	13.5X	17.5X

11.12.3 Single-Precision Square Root

To compute Z[i]= (A[i])^0.5 on a large vector of single-precision numbers, denoting A[i] by N, the approximation for N^0.5 is N multiplied by (1/N)^0.5 , where the approximation for (1/N)^0.5 is described in the previous section.

To get approximately 22-bit precision of N^0.5, use the following calculation:

N^0.5 = X_1*N = 1/2*N*X_0*(3-N*X_0^2)

Example 11-28. Square Root Using SQRTPS for 24-bit Accuracy

Using SQRTPS	Using VSQRTPS

mov rax, pIn
    mov rbx, pOut
    mov rcx, iLen
    xor rdx, rdx
loop1:
    movups xmm1, [rax+rdx]
    sqrtps xmm1, xmm1
    movups [rbx+rdx], xmm1
    add rdx, 16
    cmp rdx, rcx
    jl loop1
	mov rax, pIn
	    mov rbx, pOut
	    mov rcx, iLen
	    xor rdx, rdx
	loop1:
	    vmovups ymm1, [rax+rdx]
	    vsqrtps ymm1,ymm1
	    vmovups [rbx+rdx], ymm1
	    add rdx, 32
	    cmp rdx, rcx
	    jl loop1

Example 11-29. Square Root Using RCPPS 11-bit Approximation

SSE code using RCPPS	Using VRCPPS

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
movups xmm1, [rax+rdx]
xorps xmm8, xmm8
cmpneqps xmm8, xmm1
rsqrtps xmm1, xmm1
rcpps xmm1, xmm1
andps xmm1, xmm8
movups [rbx+rdx], xmm1
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	vxorps ymm8, ymm8, ymm8
	loop1:
	vmovups ymm1, [rax+rdx]
	vcmpneqps ymm9, ymm8, ymm1
	vrsqrtps ymm1, ymm1
	vrcpps ymm1, ymm1
	vandps ymm1, ymm1, ymm9
	vmovups [rbx+rdx], ymm1
	add rdx, 32
	cmp rdx, rcx
	jl loop1

Example 11-30. Square Root Using RCPPS and One Taylor Series Expansion

RCPPS + Taylor ~ 22 bit accuracy	VRCPPS + Taylor ~ 22 bit accuracy

__declspec(align(16)) float minus_half[4] = {-0.5, -0.5, -0.5, -0.5};
__declspec(align(16)) float three[4] = {3.0, 3.0, 3.0, 3.0};
 	
__asm
{
mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
movups xmm6, [three]
movups xmm7, [minus_half]
 	
loop1:
movups xmm3, [rax+rdx]
rsqrtps xmm1, xmm3
xorps xmm8, xmm8
cmpneqps xmm8, xmm3
andps xmm1, xmm8
movaps xmm4, xmm1
mulps xmm1, xmm3
movaps xmm5, xmm1
mulps xmm1, xmm4
	__declspec(align(32)) float three[8] = {3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0};
	
	__declspec(align(32)) float minus_half[8] = {-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5};
	
	__asm
	{
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	vmovups ymm6, [three]
	vmovups ymm7, [minus_half]
	vxorps ymm8, ymm8, ymm8
	
	loop1:
	vmovups ymm3, [rax+rdx]
	vrsqrtps ymm4, ymm3
	vcmpneqps ymm9, ymm8, ymm3
	vandps ymm4, ymm4, ymm9
	vmulps ymm1,ymm4, ymm3
	vmulps ymm2, ymm1, ymm4

Example 11-30. Square Root Using RCPPS and One Taylor Series Expansion (Contd.)

RCPPS + Taylor ~ 22 bit accuracy	VRCPPS + Taylor ~ 22 bit accuracy

subps xmm1, xmm6
mulps xmm1, xmm5
mulps xmm1, xmm7
movups [rbx+rdx], xmm1
add rdx, 16
cmp rdx, rcx
jl loop1
}
	vsubps ymm2, ymm2, ymm6
	vmulps ymm1, ymm1, ymm2
	vmulps ymm1, ymm1, ymm7
	vmovups [rbx+rdx], ymm1
	add rdx, 32
	cmp rdx, rcx
	jl loop1
	}

Table 11-9. Comparison of Single-Precision Square Root Operation

Accuracy	Method	SSE Performance	AVX Performance
24 bits	(V)SQRTPS	Baseline	1X
~ 22 bits	(V)RCPPS + Taylor-Expansion	2.3X	4.3X
~ 11 bits	(V)RCPPS	4.7X	5.9X

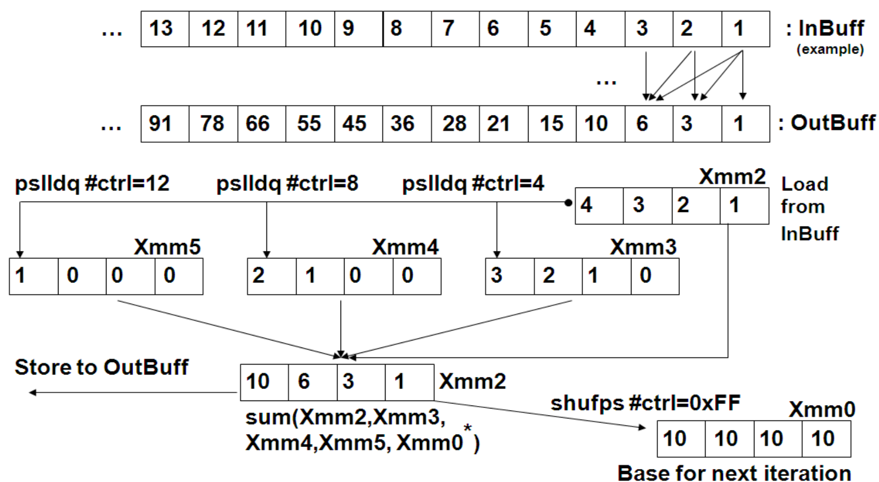
11.13 OPTIMIZATION OF ARRAY SUB SUM EXAMPLE

This section shows the transformation of SSE implementation of Array Sub Sum algorithm to Intel AVX implementation.

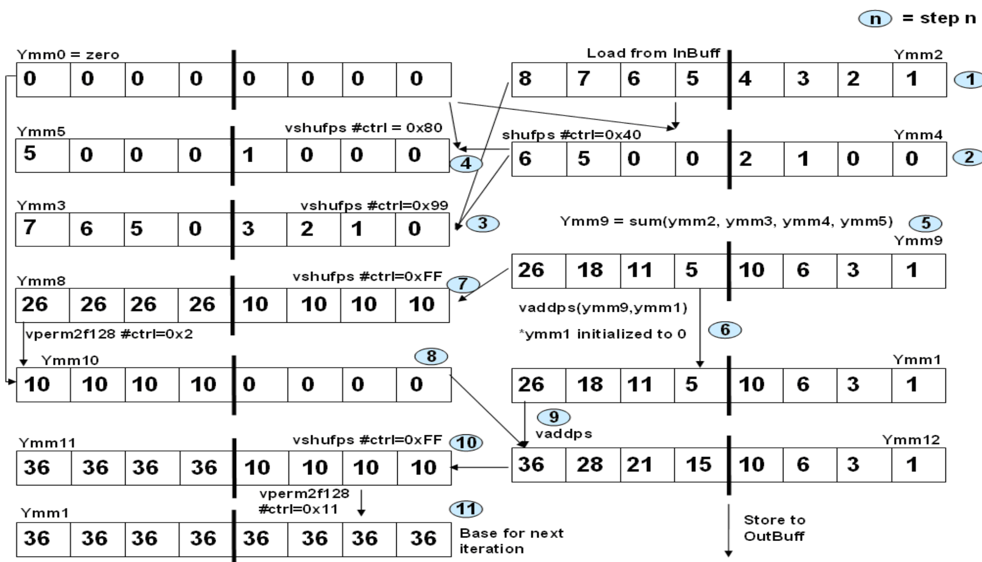
The Array Sub Sum algorithm is:

$Y_{[i]}$ = Sum of k from 0 to i ($X_{[k]}$) = $X_{[0]} + X_{[1]} + .. + X_{[i]}$

The following figure describes the SSE implementation.



The figure below describes the Intel AVX implementation of the Array Sub Sums algorithm. The PSLLDQ is an integer SIMD instruction which does not have a 256-bit equivalent. It is replaced by VSHUFPS.



Example 11-31. Array Sub Sums Algorithm

SSE code	AVX code
mov rax, InBuff	mov rax, InBuff
mov rbx, OutBuff	mov rbx, OutBuff
mov rdx, len	mov rdx, len
xor rcx, rcx	xor rcx, rcx
xorps xmm0, xmm0	vxorps ymm0, ymm0, ymm0
	vxorps ymm1, ymm1, ymm1
loop1:	loop1:
movaps xmm2, [rax+4*rcx]	vmovaps ymm2, [rax+4*rcx]
movaps xmm3, [rax+4*rcx]	vshufps ymm4, ymm0, ymm2, 0x40
movaps xmm4, [rax+4*rcx]	vshufps ymm3, ymm4, ymm2, 0x99
movaps xmm5, [rax+4*rcx]	vshufps ymm5, ymm0, ymm4, 0x80
pslldq xmm3, 4	vaddps ymm6, ymm2, ymm3
pslldq xmm4, 8	vaddps ymm7, ymm4, ymm5
pslldq xmm5, 12	vaddps ymm9, ymm6, ymm7
addps xmm2, xmm3	vaddps ymm1, ymm9, ymm1
addps xmm4, xmm5	vshufps ymm8, ymm9, ymm9, 0xff
addps xmm2, xmm4	vperm2f128 ymm10, ymm8, ymm0, 0x2
addps xmm2, xmm0	vaddps ymm12, ymm1, ymm10
movaps xmm0, xmm2	vshufps ymm11, ymm12, ymm12, 0xff
shufps xmm0, xmm2, 0xFF	vperm2f128 ymm1, ymm11, ymm11, 0x11
movaps [rbx+4*rcx], xmm2	vmovaps [rbx+4*rcx], ymm12
add rcx, 4	add rcx, 8
cmp rcx, rdx	cmp rcx, rdx
jl loop1	jl loop1

Example 11-31 shows SSE implementation of array sub summ and AVX implementation. The AVX code is about 40% faster.

11.14 HALF-PRECISION FLOATING-POINT CONVERSIONS

In applications that use floating-point and require only the dynamic range and precision offered by the 16-bit floating-point format, storing persistent floating-point data encoded in 16-bits has strong advantages in memory footprint and bandwidth conservation. These situations are encountered in some graphics and imaging workloads.

The encoding format of half-precision floating-point numbers can be found in Chapter 4, "Data Types" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Instructions to convert between packed, half-precision floating-point numbers and packed single-precision floating-point numbers is described in Chapter 14, "Programming with AVX, FMA and AVX2" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* and in the reference pages of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

To perform computations on half precision floating-point data, packed 16-bit FP data elements must be converted to single precision format first, and the single-precision results converted back to half precision format, if necessary. These conversions of 8 data elements using 256-bit instructions are very fast and handle the special cases of denormal numbers, infinity, zero and NaNs properly.

11.14.1 Packed Single-Precision to Half-Precision Conversion

To convert the data in single precision floating-point format to half precision format, without special hardware support like VCVTPS2PH, a programmer needs to do the following:

- Correct exponent bias to permitted range for each data element.
- Shift and round the significand of each data element.
- Copy the sign bit to bit 15 of each element.
- Take care of numbers outside the half precision range.
- Pack each data element to a register of half size.

Example 11-32 compares two implementations of floating-point conversion from single precision to half precision. The code on the left uses packed integer shift instructions that is limited to 128-bit SIMD instruction set. The code on right is unrolled twice and uses the VCVTPS2PH instruction.

Example 11-32. Single-Precision to Half-Precision Conversion

AVX-128 code	VCVTPS2PH code

__asm {
mov rax, pIn
mov rbx, pOut
mov rcx, bufferSize
add rcx, rax
vmovdqu xmm0,SignMask16
vmovdqu xmm1,ExpBiasFixAndRound
vmovdqu xmm4,SignMaskNot32
vmovdqu xmm5,MaxConvertibleFloat
vmovdqu xmm6,MinFloat
loop:
vmovdqu xmm2, [rax]
vmovdqu xmm3, [rax+16]
vpaddd xmm7, xmm2, xmm1
vpaddd xmm9, xmm3, xmm1
vpand xmm7, xmm7, xmm4
vpand xmm9, xmm9, xmm4
add rax, 32
	__asm {
	mov rax, pIn
	mov rbx, pOut
	mov rcx, bufferSize
	add rcx, rax
	loop:
	vmovups ymm0,[rax]
	vmovups ymm1,[rax+32]
	add rax, 64
	vcvtps2ph [rbx],ymm0, roundingCtrl
	vcvtps2ph [rbx+16],ymm1,roundingCtrl
	add rbx, 32
	cmp rax, rcx
	jl loop

Example 11-32. Single-Precision to Half-Precision Conversion (Contd.)

AVX-128 code	VCVTPS2PH code

<pre>vminps   xmm7, xmm7, xmm5
vminps   xmm9, xmm9, xmm5
vpcmpgtd   xmm8, xmm7, xmm6
vpcmpgtd   xmm10, xmm9, xmm6
vpand   xmm7, xmm8, xmm7
vpand   xmm9, xmm10, xmm9
vpackssdw   xmm2, xmm3, xmm2
vpsrad   xmm7, xmm7, 13
vpsrad   xmm8, xmm9, 13
vpand   xmm2, xmm2, xmm0
vpackssdw   xmm3, xmm7, xmm9
vpaddw   xmm3, xmm3, xmm2
vmovdqu   [rbx], xmm3
add   rbx, 16
cmp   rax, rcx
jl loop</pre>	

The code using VCVTPS2PH is approximately four times faster than the AVX-128 sequence. Although it is possible to load 8 data elements at once with 256-bit AVX, most of the per-element conversion operations require packed integer instructions which do not have 256-bit extensions yet. Using VCVTPS2PH is not only faster but also provides handling of special cases that do not encode to normal half-precision floating-point values.

11.14.2 Packed Half-Precision to Single-Precision Conversion

Example 11-33 compares two implementations using AVX-128 code and with VCVTPH2PS.

Conversion from half precision to single precision floating-point format is easier to implement, yet using VCVTPH2PS instruction performs about 2.5 times faster than the alternative AVX-128 code.

Example 11-33. Half-Precision to Single-Precision Conversion

AVX-128 code	VCVTPS2PH code

<pre>__asm {
mov   rax, pIn
mov   rbx, pOut
mov   rcx, bufferSize
add   rcx, rax
vmovdqu   xmm0,SignMask16
vmovdqu   xmm1,ExpBiasFix16
vmovdqu   xmm2,ExpMaskMarker
loop:
vmovdqu   xmm3, [rax]
add   rax, 16
vpandn   xmm4, xmm0, xmm3
vpand   xmm5, xmm3, xmm0
vpsrlw   xmm4, xmm4, 3
vpaddw   xmm6, xmm4, xmm1
vpcmpgtw   xmm7, xmm6, xmm2</pre>
	<pre> __asm {
	    mov rax, pIn
	    mov rbx, pOut
	    mov rcx, bufferSize
	    add rcx, rax
	loop:
	    vcvtph2ps ymm0,[rax]
	    vcvtph2ps ymm1,[rax+16]
	    add rax, 32
	    vmovups [rbx], ymm0
	    vmovups [rbx+32], ymm1
	    add rbx, 64
	    cmp rax, rcx
	    jl loop</pre>

Example 11-33. Half-Precision to Single-Precision Conversion (Contd.)

AVX-128 code	VCVTPS2PH code

vpand xmm6, xmm6, xmm7
vpand xmm8, xmm3, xmm7
vpor xmm6, xmm6, xmm5
vpsllw xmm8, xmm8, 13
vpunpcklwd xmm3, xmm8, xmm6
vpunpckhwd xmm4, xmm8, xmm6
vmovdqu [rbx], xmm3
vmovdqu [rbx+16], xmm4
add rbx, 32
cmp rax, rcx
jl loop	

11.14.3 Locality Consideration for using Half-Precision FP to Conserve Bandwidth

Example 11-32 and Example 11-33 demonstrate the performance advantage of using FP16C instructions when software needs to convert between half-precision and single-precision data. Half-precision FP format is more compact, consumes less bandwidth than single-precision FP format, but sacrifices dynamic range, precision, and incurs conversion overhead if arithmetic computation is required. Whether it is profitable for software to use half-precision data will be highly dependent on locality considerations of the workload.

This section uses an example based on the horizontal median filtering algorithm, "Median3". The Median3 algorithm calculates the median of every three consecutive elements in a vector:

Y[i] = Median3(X[i], X[i+1], X[i+2])

Where: Y is the output vector, and X is the input vector.

Example 11-34 shows two implementations of the Median3 algorithm; one uses single-precision format without conversion, the other uses half-precision format and requires conversion. Alternative 1 on the left works with single precision format using 256-bit load/store operations, each of which loads/stores eight 32-bit numbers. Alternative 2 uses 128-bit load/store operations to load/store eight 16-bit numbers in half precision format and VCVTPH2PS/VCVTPS2PH instructions to convert it to/from single precision floating-point format.

Example 11-34. Performance Comparison of Median3 using Half-Precision vs. Single-Precision

Single-Precision code w/o Conversion	Half-Precision code w/ Conversion

__asm {
xor rbx, rbx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr
vmovaps ymm0, [rdi]
loop:
add rdi, 32
vmovaps ymm6, [rdi]
vperm2f128 ymm1, ymm0, ymm6, 0x21
vshufps ymm3, ymm0, ymm1, 0x4E
vshufps ymm2, ymm0, ymm3, 0x99
vminps ymm5, ymm0 ,ymm2
vmaxps ymm0, ymm0, ymm2
	__asm {
	xor rbx, rbx
	mov rcx, len
	mov rdi, inPtr
	mov rsi, outPtr
	vcvtph2ps ymm0, [rdi]
	loop:
	add rdi,16
	vcvtph2ps ymm6, [rdi]
	vperm2f128 ymm1, ymm0, ymm6, 0x21
	vshufps ymm3, ymm0, ymm1, 0x4E
	vshufps ymm2, ymm0, ymm3, 0x99
	vminps ymm5, ymm0 ,ymm2
	vmaxps ymm0, ymm0, ymm2

Example 11-34. Performance Comparison of Median3 using Half-Precision vs. Single-Precision (Contd.)

Single-Precision code w/o Conversion	Half-Precision code w/ Conversion

vminps ymm4, ymm0, ymm3
vmaxps ymm7, ymm4, ymm5
vmovaps ymm0, ymm6
vmovaps [rsi], ymm7
add rsi, 32
add rbx, 8
cmp rbx, rcx
jl loop
	vminps ymm5, ymm0 ,ymm2
	vmaxps ymm0, ymm0, ymm2
	vminps ymm4, ymm0, ymm3
	vmaxps ymm7, ymm4, ymm5
	vmovaps ymm0, ymm6
	vcvtps2ph [rsi], ymm7, roundingCtrl
	add rsi, 16
	add rbx, 8
	cmp rbx, rcx
	jl loop

When the locality of the working set resides in memory, using half-precision format with processors based on Intel microarchitecture code name Ivy Bridge is about 30% faster than single-precision format, despite the conversion overhead. When the locality resides in L3, using half-precision format is still ~15% faster. When the locality resides in L1, using single-precision format is faster because the cache bandwidth of the L1 data cache is much higher than the rest of the cache/memory hierarchy and the overhead of the conversion becomes a performance consideration.

11.15 FUSED MULTIPLY-ADD (FMA) INSTRUCTIONS GUIDELINES

FMA instructions perform vectored operations of "a * b + c" on IEEE-754-2008 floating-point values, where the multiplication operations "a * b" are performed with infinite precision, the final results of the addition are rounded to produced the desired precision. Details of FMA rounding behavior and special case handling can be found in section 2.3 of Intel® Architecture Instruction Set Extensions Programming Reference.

FMA instruction can speed up and improve the accuracy of many FP calculations. Intel microarchitecture code name Haswell implements FMA instructions with execution units on port 0 and port 1 and 256-bit data paths. Dot product, matrix multiplication and polynomial evaluations are expected to benefit from the use of FMA, 256-bit data path and the independent executions on two ports. The peak throughput of FMA from each processor core are 16 single-precision and 8 double-precision results each cycle.

Algorithms designed to use FMA instruction should take into consideration that non-FMA sequence of MULPD/PS and ADDPD/PS likely will produce slightly different results compared to using FMA. For numerical computations involving a convergence criteria, the difference in the precision of intermediate results must be factored into the numeric formalism to avoid surprise in completion time due to rounding issues.

User/Source Coding Rule 33. *Factor in precision and rounding characteristics of FMA instructions when replacing multiply/add operations executing non-FMA instructions.* FMA improves performance when an algorithm is execution-port throughput limited, like DGEMM.

There may be situations where using FMA might not deliver better performance. Consider the vectored operation of "a * b + c * d" and data are ready at the same time:

In the three-instruction sequence of

```
VADDPS ( VMULPS (a,b) , VMULPS (c,b) );
```

VMULPS can be dispatched in the same cycle and execute in parallel, leaving the latency of VADDPS (3 cycle) exposed. With unrolling the exposure of VADDPS latency may be further amortized.

When using the two-instruction sequence of

```
VFMADD213PS ( c, d, VMULPS (a,b) );
```

The latency of FMA (5 cycle) is exposed for producing each vector result.

User/Source Coding Rule 34. *Factor in result-dependency, latency of FP add vs. FMA instructions when replacing FP add operations with FMA instructions.*

11.15.1 Optimizing Throughput with FMA and Floating-Point Add/MUL

In the Skylake microarchitecture, there are two pipes of executions supporting FMA, vector FP Multiply, and FP ADD instructions. All three categories of instructions have a latency of 4 cycles and can dispatch to either port 0 or port 1 to execute every cycle.

The arrangement of identical latency and number of pipes allows software to increase the performance of situations where floating-point calculations are limited by the floating-point add operations that follow FP multiplies. Consider a situation of vector operation $A_n = C_1 + C_2 * A_{n-1}$:

Example 11-35. FP Mul/FP Add Versus FMA

FP Mul/FP Add Sequence	FMA Sequence

mov eax, NumOfIterations
mov rbx, pA
mov rcx, pC1
mov rdx, pC2
vmovups ymm0, Ymmword ptr [rbx] // A
vmovups ymm1, Ymmword ptr [rcx] // C1
vmovups ymm2, Ymmword ptr [rdx] // C2
loop:
vmulps ymm4, ymm0 ,ymm2 // A * C2
vaddps ymm0, ymm1, ymm4
dec eax
jnz loop
 	
vmovups ymmword ptr[rbx], ymm0 // store An
	mov eax, NumOfIterations
	mov rbx, pA
	mov rcx, pC1
	mov rdx, pC2
	vmovups ymm0, Ymmword ptr [rbx] // A
	vmovups ymm1, Ymmword ptr [rcx] // C1
	vmovups ymm2, Ymmword ptr [rdx] // C2
	loop:
	vfmadd132ps ymm0, ymm1, ymm2 // C1 + A * C2
	dec eax
	jnz loop
	
	vmovups ymmword ptr[rbx], ymm0 // store An

Cost per iteration: ~ fp add latency + fp add latency	Cost per iteration: ~ fma latency

The overall throughput of the code sequence on the LHS is limited by the combined latency of the FP MUL and FP ADD instructions of specific microarchitecture. The overall throughput of the code sequence on the RHS is limited by the throughput of the FMA instruction of the corresponding microarchitecture.

A common situation where the latency of the FP ADD operation dominates performance is the following C code:

```
for ( int 1 = 0; i < arrLenght; i ++) result += arrToSum[i];
```

Example 11-35 shows two implementations with and without unrolling.

Example 11-36. Unrolling to Hide Dependent FP Add Latency

No Unroll	Unroll 8 times

mov eax, arrLength
mov rbx, arrToSum
vmovups ymm0, Ymmword ptr [rbx]
sub eax, 8
loop:
add rbx, 32
vaddps ymm0, ymm0, ymmword ptr [rbx]
sub eax, 8
jnz loop
	mov eax, arrLength
	mov rbx, arrToSum
	vmovups ymm0, ymmword ptr [rbx]
	vmovups ymm1, ymmword ptr 32[rbx]
	vmovups ymm2, ymmword ptr 64[rbx]
	vmovups ymm3, ymmword ptr 96[rbx]
	vmovups ymm4, ymmword ptr 128[rbx]
	vmovups ymm5, ymmword ptr 160[rbx]
	vmovups ymm6, ymmword ptr 192[rbx]
	vmovups ymm7, ymmword ptr 224[rbx]

Example 11-36. Unrolling to Hide Dependent FP Add Latency (Contd.)

No Unroll	Unroll 8 times

vextractf128 xmm1, ymm0, 1
vaddps xmm0, xmm0, xmm1
vpermilps xmm1, xmm0, 0xe
vaddps xmm0, xmm0, xmm1
vpermilps xmm1, xmm0, 0x1
vaddss xmm0, xmm0, xmm1
	sub eax, 64
	loop:
	add rbx, 256
	vaddps ymm0, ymm0, ymmword ptr [rbx]
	vaddps ymm1, ymm1, ymmword ptr 32[rbx]
	vaddps ymm2, ymm2, ymmword ptr 64[rbx]
	vaddps ymm3, ymm3, ymmword ptr 96[rbx]
	vaddps ymm4, ymm4, ymmword ptr 128[rbx]
	vaddps ymm5, ymm5, ymmword ptr 160[rbx]
	vaddps ymm6, ymm6, ymmword ptr 192[rbx]
	vaddps ymm7, ymm7, ymmword ptr 224[rbx]
	sub eax, 64
	jnz loop
	
	vaddps Ymm0, ymm0, ymm1
	vaddps Ymm2, ymm2, ymm3
	vaddps Ymm4, ymm4, ymm5
	vaddps Ymm6, ymm6, ymm7
	vaddps Ymm0, ymm0, ymm2
	vaddps Ymm4, ymm4, ymm6
	vaddps Ymm0, ymm0, ymm4

vmovss result, ymm0	vextractf128 xmm1, ymm0, 1 vaddps xmm0, xmm0, xmm1 vpermilps xmm1, xmm0, 0xe vaddps xmm0, xmm0, xmm1 vpermilps xmm1, xmm0, 0x1 vaddss xmm0, xmm0, xmm1 vmovss result, ymm0

Without unrolling (LHS of Example 11-35), the cost of summing every 8 array elements is about proportional to the latency of the FP ADD instruction, assuming the working set fit in L1. To use unrolling effectively, the number of unrolled operations should be at least "latency of the critical operation" * "number of pipes". The performance gain of optimized unrolling versus no unrolling, for a given microarchitecture, can approach "number of pipes" * "Latency of FP ADD".

User/Source Coding Rule 35. *Consider using unrolling technique for loops containing back-to-back dependent FMA, FP Add or Vector MUL operations, The unrolling factor can be chosen by considering the latency of the critical instruction of the dependency chain and the number of pipes available to execute that instruction.*

11.15.2 Optimizing Throughput with Vector Shifts

In the Skylake microarchitecture, many common vector shift instructions can dispatch into either port 0 or port 1, compared to only one port in prior generations, see Table 2-2 and Table 2-7.

A common situation where the latency of the FP ADD operation dominates performance is the following C code, where a, b, and c are integer arrays:

```
for ( int 1 = 0; i < len; i ++) c[i] += 4* a[i] + b[i]/2;
```

Example 11-35 shows two implementations with and without unrolling.

Example 11-37. FP Mul/FP Add Versus FMA

FP Mul/FP Add Sequence	FMA Sequence

mov eax, NumOfIterations
mov rbx, pA
mov rcx, pC1
mov rdx, pC2
vmovups ymm0, Ymmword ptr [rbx] // A
vmovups ymm1, Ymmword ptr [rcx] // C1
vmovups ymm2, Ymmword ptr [rdx] // C2
loop:
vmulps ymm4, ymm0 ,ymm2 // A * C2
vaddps ymm0, ymm1, ymm4
dec eax
jnz loop
 	
vmovups ymmword ptr[rbx], ymm0 // store An
	mov eax, NumOfIterations
	mov rbx, pA
	mov rcx, pC1
	mov rdx, pC2
	vmovups ymm0, Ymmword ptr [rbx] // A
	vmovups ymm1, Ymmword ptr [rcx] // C1
	vmovups ymm2, Ymmword ptr [rdx] // C2
	loop:
	vfmadd132ps ymm0, ymm1, ymm2 // C1 + A * C2
	dec eax
	jnz loop
	
	vmovups ymmword ptr[rbx], ymm0 // store An

Cost per iteration: ~ fp add latency + fp add latency	Cost per iteration: ~ fma latency

11.16 AVX2 OPTIMIZATION GUIDELINES

AVX2 instructions promotes the great majority of 128-bit SIMD integer instructions to operate on 256-bit YMM registers. AVX2 also adds a rich mix of broadcast/permute/variable-shift instructions to accelerate numerical computations. The 256-bit AVX2 instructions are supported by the Intel microarchitecture Haswell which implements 256-bit data path with low latency and high throughput.

Consider an intra-coding 4x4 block image transformation[1] shown in Figure 11-3.

A 128-bit SIMD implementation can perform this transformation by the following technique:

- Convert 8-bit pixels into 16-bit word elements and fetch two 4x4 image block as 4 row vectors.
- The matrix operation 1/128 * (B x R) can be evaluated with row vectors of the image block and column vectors of the right-hand-side coefficient matrix using a sequence of SIMD instructions of PMADDWD, PHADDD, packed shift and blend instructions.
- The two 4x4 word-granular, intermediate result can be re-arranged into column vectors.
- The left-hand-side coefficient matrix in row vectors and the column vectors of the intermediate block can be calculated (using PMADDWD, PHADDD, shift, blend) and written out.

1. C. Yeo, Y. H. Tan, Z. Li and S. Rahardja, “Mode-Dependent Fast Separable KLT for Block-based Intra Coding,” JCTVC-B024, Geneva, Switzerland, Jul 2010

$$\frac{1}{128}\begin{bmatrix} 29 & 55 & 74 & 84 \\ 74 & 74 & 0 & -74 \\ 84 & -29 & -74 & 55 \\ 55 & -84 & 74 & -29 \end{bmatrix} \times B \times \frac{1}{128}\begin{bmatrix} 64 & 64 & 64 & 64 \\ 84 & 35 & -35 & -84 \\ 64 & -64 & -64 & 64 \\ 35 & -84 & 84 & -35 \end{bmatrix}$$

L B R

Figure 11-3. 4x4 Image Block Transformation

The same technique can be implemented using AVX2 instructions in a straightforward manner. The AVX2 sequence is illustrated in Example 11-38 and Example 11-39.

Example 11-38. Macros for Separable KLT Intra-block Transformation Using AVX2

```
// b0: input row vector from 4 consecutive 4x4 image block of word pixels
// rmc0-3: columnar vector coefficient of the RHS matrix, repeated 4X for 256-bit
// min32km1: saturation constant vector to cap intermediate pixel to less than or equal to 32767
// w0: output row vector of garbled intermediate matrix, elements within each block are garbled
// e.g Low 128-bit of row 0 in descending order: y07, y05, y06, y04, y03, y01, y02, y00
            (continue)
```

```
#define __MyM_KIP_PxRMC_ROW_4x4Wx4(b0, w0, rmc0_256,
rmc1_256, rmc2_256, rmc3_256, min32km1)\

{__m256i  tt0, tt1, tt2, tt3;\
  tt0 = _mm256_madd_epi16(b0, (rmc0_256));\
  tt0 = _mm256_hadd_epi32(tt0, tt0) ;\
  tt1 = _mm256_madd_epi16(b0, rmc1_256);\
  tt1 = _mm256_blend_epi16(tt0, _mm256_hadd_epi32(tt1, tt1) , 0xf0);\
  tt1 = _mm256_min_epi32(_mm256_srai_epi32( tt1, 7), min32km1);\
  tt1 = _mm256_shuffle_epi32(tt1, 0xd8); \
  tt2 = _mm256_madd_epi16(b0, rmc2_256);\
  tt2 = _mm256_hadd_epi32(tt2, tt2) ;\
  tt3 = _mm256_madd_epi16(b0, rmc3_256);\
  tt3 = _mm256_blend_epi16(tt2, _mm256_hadd_epi32(tt3, tt3) , 0xf0);\
  tt3 = _mm256_min_epi32( _mm256_srai_epi32(tt3, 7), min32km1);\
  tt3 = _mm256_shuffle_epi32(tt3, 0xd8);\
  w0 = _mm256_blend_epi16(tt1, _mm256_slli_si256( tt3, 2),  0xaa);\
}
```

Example 11-38. Macros for Separable KLT Intra-block Transformation Using AVX2 (Contd.)

```
// t0-t3: 256-bit input vectors of un-garbled intermediate matrix 1/128 * (B x R)
// lmr_256: 256-bit vector of one row of LHS coefficient, repeated 4X
// min32km1: saturation constant vector to cap final pixel to less than or equal to 32767
// w0; Output row vector of final result in un-garbled order
#define __MyM_KIP_LMRxP_ROW_4x4Wx4(w0, t0, t1, t2, t3, lmr_256, min32km1)\
{__m256itb0, tb1, tb2, tb3;\
   tb0 = _mm256_madd_epi16( lmr_256, t0);\
   tb0 = _mm256_hadd_epi32(tb0, tb0) ;\
   tb1 = _mm256_madd_epi16( lmr_256, t1);\
   tb1 = _mm256_blend_epi16(tb0, _mm256_hadd_epi32(tb1, tb1), 0xf0 );\
   tb1 = _mm256_min_epi32( _mm256_srai_epi32( tb1, 7), min32km1);\
   tb1 = _mm256_shuffle_epi32(tb1, 0xd8);\
   tb2 = _mm256_madd_epi16( lmr_256, t2);\
   tb2 = _mm256_hadd_epi32(tb2, tb2) ;\
   tb3 = _mm256_madd_epi16( lmr_256, t3);\
   tb3 = _mm256_blend_epi16(tb2, _mm256_hadd_epi32(tb3, tb3) , 0xf0);\
   tb3 = _mm256_min_epi32( _mm256_srai_epi32( tb3, 7), min32km1);\
   tb3 = _mm256_shuffle_epi32(tb3, 0xd8); \
   tb3 = _mm256_slli_si256( tb3, 2);\
   tb3 = _mm256_blend_epi16(tb1, tb3,  0xaa);\
   w0 = _mm256_shuffle_epi8(tb3, _mm256_setr_epi32( 0x5040100, 0x7060302, 0xd0c0908, 0xf0e0b0a,
0x5040100, 0x7060302, 0xd0c0908, 0xf0e0b0a));\
}
```

In Example 11-39, matrix multiplication of 1/128 * (B xR) is evaluated first in a 4-wide manner by fetching from 4 consecutive 4x4 image block of word pixels. The first macro shown in Example 11-38 produces an output vector where each intermediate row result is in an garbled sequence between the two middle elements of each 4x4 block. In Example 11-39, undoing the garbled elements and transposing the intermediate row vector into column vectors are implemented using blend primitives instead of shuffle/unpack primitives.

In Intel microarchitecture code name Haswell, shuffle/pack/unpack primitives rely on the shuffle execution unit dispatched to port 5. In some situations of heavy SIMD sequences, port 5 pressure may become a determining factor in performance.

If 128-bit SIMD code faces port 5 pressure when running on Haswell, porting 128-bit code to use 256-bit AVX2 can improve performance and alleviate port 5 pressure.

Example 11-39. Separable KLT Intra-block Transformation Using AVX2

```
short __declspec(align(16))cst_rmc0[8] = {64, 84, 64, 35, 64, 84, 64, 35};
short __declspec(align(16))cst_rmc1[8] = {64, 35, -64, -84, 64, 35, -64, -84};
short __declspec(align(16))cst_rmc2[8] = {64, -35, -64, 84, 64, -35, -64, 84};
short __declspec(align(16))cst_rmc3[8] = {64, -84, 64, -35, 64, -84, 64, -35};
short __declspec(align(16))cst_lmr0[8] = {29, 55, 74, 84, 29, 55, 74, 84};
short __declspec(align(16))cst_lmr1[8] = {74, 74, 0, -74, 74, 74, 0, -74};
short __declspec(align(16))cst_lmr2[8] = {84, -29, -74, 44, 84, -29, -74, 55};
short __declspec(align(16)) cst_lmr3[8] = {55, -84, 74, -29, 55, -84, 74, -29};
```

```
void Klt_256_d(short * Input, short * Output, int iWidth, int iHeight)
{int iX, iY;
__m256i rmc0 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *) &cst_rmc0[0]));
__m256i rmc1 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_rmc1[0]));
__m256i rmc2 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_rmc2[0]));
__m256i rmc3 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_rmc3[0]));
__m256i lmr0 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr0[0]));
__m256i lmr1 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr1[0]));
__m256i lmr2 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr2[0]));
__m256i lmr3 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr3[0]));
__m256i min32km1 = _mm256_broadcastsi128_si256(   _mm_setr_epi32( 0x7fff7fff, 0x7fff7fff, 0x7fff7fff,
0x7fff7fff));
__m256i b0, b1, b2, b3, t0, t1, t2, t3;
__m256i w0, w1, w2, w3;
short* pImage = Input;
short* pOutImage = Output;
int hgt = iHeight, wid= iWidth;
                (continue)
```

```
// We implement 1/128 * (Mat_L x (1/128 * (Mat_B x Mat_R))) from the inner most parenthesis
for( iY = 0; iY < hgt; iY+=4) {
  for( iX = 0; iX < wid; iX+=16) {
      //load row 0 of 4 consecutive 4x4 matrix of word pixels
      b0 = _mm256_loadu_si256( (__m256i *) (pImage + iY*wid+ iX)) ;
      // multiply row 0 with columnar vectors of the RHS matrix coefficients
      __MyM_KIP_PxRMC_ROW_4x4Wx4(b0, w0, rmc0, rmc1, rmc2, rmc3, min32km1);
       // low 128-bit of garbled row 0, from hi->lo: y07, y05, y06, y04, y03, y01, y02, y00
      b1 = _mm256_loadu_si256( (__m256i *)  (pImage + (iY+1)*wid+ iX) );
      __MyM_KIP_PxRMC_ROW_4x4Wx4(b1, w1, rmc0, rmc1, rmc2, rmc3, min32km1);
       // hi->lo y17, y15, y16, y14, y13, y11, y12, y10
      b2 = _mm256_loadu_si256( (__m256i *)  (pImage + (iY+2)*wid+ iX) );
      __MyM_KIP_PxRMC_ROW_4x4Wx4(b2, w2, rmc0, rmc1, rmc2, rmc3, min32km1);
      b3 = _mm256_loadu_si256( (__m256i *) (pImage + (iY+3)*wid+ iX) );
      __MyM_KIP_PxRMC_ROW_4x4Wx4(b3, w3, rmc0, rmc1, rmc2, rmc3, min32km1);
```

Example 11-39. Separable KLT Intra-block Transformation Using AVX2 (Contd.)

```
        // unscramble garbled middle 2 elements of each 4x4 block, then
        // transpose into columnar vectors: t0 has 4 consecutive column 0 or 4 4x4 intermediate
        t0 = _mm256_blend_epi16( w0, _mm256_slli_epi64(w1, 16), 0x22);
        t0 = _mm256_blend_epi16( t0, _mm256_slli_epi64(w2, 32), 0x44);
        t0 = _mm256_blend_epi16( t0, _mm256_slli_epi64(w3, 48), 0x88);
        t1 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 32), _mm256_srli_epi64(w1, 16), 0x22);
        t1 = _mm256_blend_epi16( t1, w2, 0x44);
        t1 = _mm256_blend_epi16( t1, _mm256_slli_epi64(w3, 16), 0x88); // column 1
        t2 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 16), w1, 0x22);
        t2 = _mm256_blend_epi16( t2, _mm256_slli_epi64(w2, 16), 0x44);
        t2 = _mm256_blend_epi16( t2, _mm256_slli_epi64(w3, 32), 0x88); // column 2
        t3 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 48), _mm256_srli_epi64(w1, 32), 0x22);
        t3 = _mm256_blend_epi16( t3, _mm256_srli_epi64(w2, 16), 0x44);
        t3 = _mm256_blend_epi16( t3, w3, 0x88);// column 3

        // multiply row 0 of the LHS coefficient with 4 columnar vectors of intermediate blocks
        // final output row are arranged in normal order
        __MyM_KIP_LMRxP_ROW_4x4Wx4(w0, t0, t1, t2, t3, lmr0, min32km1);
        _mm256_store_si256( (__m256i *) (pOutImage+iY*wid+ iX), w0) ;

        __MyM_KIP_LMRxP_ROW_4x4Wx4(w1, t0, t1, t2, t3, lmr1, min32km1);
        _mm256_store_si256( (__m256i *) (pOutImage+(iY+1)*wid+ iX), w1) ;

        __MyM_KIP_LMRxP_ROW_4x4Wx4(w2, t0, t1, t2, t3, lmr2, min32km1);
        _mm256_store_si256( (__m256i *) (pOutImage+(iY+2)*wid+ iX), w2) ;

                (continue)

        __MyM_KIP_LMRxP_ROW_4x4Wx4(w3, t0, t1, t2, t3, lmr3, min32km1);
        _mm256_store_si256( (__m256i *) (pOutImage+(iY+3)*wid+ iX), w3) ;
    }
}
```

Although 128-bit SIMD implementation is not shown here, it can be easily derived.

When running 128-bit SIMD code of this KLT intra-coding transformation on Intel microarchitecture code name Sandy Bridge, the port 5 pressure are less because there are two shuffle units, and the effective throughput for each 4x4 image block transformation is around 50 cycles. Its speed-up relative to optimized scalar implementation is about 2.5X.

When the 128-bit SIMD code runs on Haswell, micro-ops issued to port 5 account for slightly less than 50% of all micro-ops, compared to about one third on prior microarchitecture, resulting in about 25% performance regression. On the other hand, AVX2 implementation can deliver effective throughput in less than 35 cycle per 4x4 block.

11.16.1 Multi-Buffering and AVX2

There are many compute-intensive algorithms (e.g. hashing, encryption, etc.) which operate on a stream of data buffers. Very often, the data stream may be partitioned and treated as multiple independent buffer streams to leverage SIMD instruction sets.

Detailed treatment of hashing several buffers in parallel can be found at http://www.scirp.org/journal/PaperInformation.aspx?paperID=23995 and at http://eprint.iacr.org/2012/476.pdf.

With AVX2 providing a full compliment of 256-bit SIMD instructions with rich functionality at multiple width granularities for logical and arithmetic operations. Algorithms that had leveraged XMM registers and prior generations of SSE instruction sets can extend those multi-buffering algorithms to use AVX2 on YMM and deliver even higher throughput. Optimized 256-bit AVX2 implementation may deliver up to 1.9X throughput when compared to 128-bit versions.

The image block transformation example discussed in Section 11.16 can be construed also as a multi-buffering implementation of 4x4 blocks. When the performance baseline is switched from a two-shuffle-port microarchitecture (Sandy Bridge) to single-shuffle-port microarchitecture, the 256-bit wide AVX2 provides a speed up of 1.9X relative to 128-bit SIMD implementation.

Greater details on multi-buffering can be found in the white paper at: https://www-ssl.intel.com/content/www/us/en/communications/communications-ia-multi-buffer-paper.html.

11.16.2 Modular Multiplication and AVX2

Modular multiplication of very large integers are often used to implement efficient modular exponentiation operations which are critical in public key cryptography, such as RSA 2048. Library implementation of modular multiplication is often done with MUL/ADC chain sequences. Typically, a MUL instruction can produce a 128-bit intermediate integer output, and add-carry chains must be used at 64-bit intermediate data granularity.

In AVX2, VPMULUDQ/VPADDQ/VPSRLQ/VPSLLQ/VPBROADCASTQ/VPERMQ allow vectorized approach to implement efficient modular multiplication/exponentiation for key lengths corresponding to RSA1024 and RSA2048. For details of modular exponentiation/multiplication and AVX2 implementation in OpenSSL, see http://rd.springer.com/chapter/10.1007%2F978-3-642-31662-3_9?LI=true.

The basic heuristic starts with reformulating the large integer input operands in 512/1024 bit exponentiation in redundant representations. For example, a 1024-bit integer can be represented using base 2^29 and 36 “**digits**”, where each “**digit**” is less than 2^29. A digit in such redundant representation can be placed in a dword slot of a vector register. Such redundant representation of large integer simplifies the requirement to perform carry-add chains across the hardware granularity of the intermediate results of unsigned integer multiplications.

Each VPMULUDQ in AVX2 using the **digits** from a redundant representation can produce 4 separate 64-bit intermediate result with sufficient headroom (e.g. 5 most significant bits are 0 excluding sign bit). Then, VPADDQ is sufficient to implement add-carry chain requirement without needing SIMD versions of equivalent of ADC-like instructions. More details are available in the reference cited in paragraph above, including the cost factor of conversion to redundant representation and effective speedup accounting for parallel output bandwidth of VPMULUDQ/VPADDQ chain.

11.16.3 Data Movement Considerations

Intel microarchitecture code name Haswell can support up to two 256-bit load and one 256-bit store micro-ops dispatched each cycle. Most existing binaries with heavy data-movement operation can benefit from this enhancement and the higher bandwidths of the L1 data cache and L2 without re-compilation, if the binary is already optimized for prior generation microarchitecture. For example, 256-bit SAXPY computation were limited by the number of load/store ports available in prior generation microarchitecture. It will benefit immediately on the Intel microarchitecture Haswell.

In some situation, there may be some intricate interactions between microarchitectural restrictions on the instruction set that is worth some discussion. We consider two commonly used library functions memcpy() and memset() and the optimal choice to implement them on the new microarchitecture.

With memcpy() on Intel microarchitecture code name Haswell, using REP MOVSB to implement memcpy operation for large copy length can take advantage the 256-bit store data path and deliver throughput of more than 20 bytes per cycle. For copy length that are smaller than a few hundred bytes, REP MOVSB approach is slower than using 128-bit SIMD technique described in Section 11.16.3.1.

11.16.3.1 SIMD Heuristics to implement Memcpy()

We start with a discussion of the general heuristic to attempt implementing memcpy() with 128-bit SIMD instructions, which revolves around three numeric factors (destination address alignment, source address alignment, bytes to copy) relative to the width of register width of the desired instruction set. The data movement work of memcpy can be separated into the following phases:

- An initial unaligned copy of 16 bytes, allows looping destination address pointer to become 16-byte aligned. Thus subsequent store operations can use as many 16-byte aligned stores.
- The remaining bytes-left-to-copy are decomposed into (a) multiples of unrolled 16-byte copy operations, plus (b) residual count that may include some copy operations of less than 16 bytes. For example, to unroll eight time to amortize loop iteration overhead, the residual count must handle individual cases from 1 to 8x16-1 = 127.
- Inside an 8X16 unrolled main loop, each 16 byte copy operation may need to deal with source pointer address is not aligned to 16-byte boundary and store 16 fresh data to 16B-aligned destination address. When the iterating source pointer is not 16B-aligned, the most efficient technique is a three instruction sequence of:
  - — Fetch an 16-byte chunk from an 16-byte-aligned adjusted pointer address and use a portion of this chunk with complementary portion from previous 16-byte-aligned fetch.
  - — Use PALIGNR to stitch a portion of the current chunk with the previous chunk.
  - — Stored stitched 16-byte fresh data to aligned destination address, and repeat this 3 instruction sequence.

  This 3-instruction technique allows the fetch:store instruction ratio for each 16-byte copy operation to remain at 1:1.

While the above technique (specifically, the main loop dealing with copying thousands of bytes of data) can achieve throughput of approximately 10 bytes per cycle on Intel microarchitecture Sandy Bridge and Ivy Bridge with 128-bit data path for store operations, an attempt to extend this technique to use wider data path will run into the following restrictions:

- To use 256-bit VPALIGNR with its 2X128-bit lane microarchitecture, stitching of two partial chunks of the current 256-bit 32-byte-aligned fetch requires another 256-bit fetch from an address 16-byte offset from the current 32-byte-aligned 256-bit fetch.
  - — The fetch:store ratio for each 32-byte copy operation becomes 2:1.
  - — The 32-byte-unaligned fetch (although aligned to 16-byte boundary) will experience a cache-line split penalty, once every 64-bytes of copy operation.

The net of this attempt to use 256-bit ISA to take advantage of the 256-bit store data-path microarchitecture was offset by the 4-instruction sequence and cacheline split penalty.

11.16.3.2 Memcpy() Implementation Using Enhanced REP MOVSB

It is interesting to compare the alternate approach of using enhanced REP MOVSB to implement memcpy(). In Intel microarchitecture code name Haswell and Ivy Bridge, REP MOVSB is an optimized, hardware provided, micro-op flow.

On Intel microarchitecture code name Ivy Bridge, a REP MOVSB implementation of memcpy can achieve throughput at slightly better than the 128-bit SIMD implementation when copying thousands of bytes. However, if the size of copy operation is less than a few hundred bytes, the REP MOVSB approach is less

efficient than the explicit residual copy technique described in phase 2 of Section 11.16.3.1. This is because handling 1-127 residual copy length (via jump table or switch/case, and is done before the main loop) plus one or two 8x16B iterations incurs less branching overhead than the hardware provided micro-op flows. For the grueling implementation details of 128-bit SIMD implementation of memcpy(), one can look up from the archived sources of open source library such as GLibC.

On Intel microarchitecture code name Haswell, using REP MOVSB to implement memcpy operation for large copy length can take advantage the 256-bit store data path and deliver throughput of more than 20 bytes per cycle. For copy length that are smaller than a few hundred bytes, REP MOVSB approach is still slower than treating the copy length as the residual phase of Section 11.16.3.1.

11.16.3.3 Memset() Implementation Considerations

The interface of Memset() has one address pointer as destination, which simplifies the complexity of managing address alignment scenarios to use 256-bit aligned store instruction. After an initial unaligned store, and adjusting the destination pointer to be 32-byte aligned, the residual phase follows the same consideration as described in Section 11.16.3.1, which may employ a large jump table to handle each residual value scenario with minimal branching, depending on the amount of unrolled 32B-aligned stores. The main loop is a simple YMM register to 32-byte-aligned store operation, which can deliver close to 30 bytes per cycle for lengths more than a thousand byte. The limiting factor here is due to each 256-bit VMOVDQA store consists of a store_address and a store_data micro-op flow. Only port 4 is available to dispatch the store_data micro-op each cycle.

Using REP STOSB to implement memset() has the code size advantage versus a SIMD implementation, like REP MOVSB for memcpy(). On Intel microarchitecture code name Haswell, a memset() routine implemented using REP STOSB will also benefit the from the 256-bit data path and increased L1 data cache bandwidth to deliver up to 32 bytes per cycle for large count values.

Comparing the performance of memset() implementations using REP STOSB vs. 256-bit AVX2 requires one to consider the pattern of invocation of memset(). The invocation pattern can lead to the necessity of using different performance measurement techniques. There may be side effects affecting the outcome of each measurement technique.

The most common measurement technique that is often used with a simple routine like memset() is to execute memset() inside a loop with a large iteration count, and wrap the invocation of RDTSC before and after the loop.

A slight variation of this measurement technique can apply to measuring memset() invocation patterns of multiple back-to-back calls to memset() with different count values with no other intervening instruction streams executed between calls to memset().

In both of the above memset() invocation scenarios, branch prediction can play a significant role in affecting the measured total cycles for executing the loop. Thus, measuring AVX2-implemented memset() under a large loop to minimize RDTSC overhead can produce a skewed result with the branch predictor being trained by the large loop iteration count.

In more realistic software stacks, the invocation patterns of memset() will likely have the characteristics that:

- There are intervening instruction streams being executed between invocations of memset(), the state of branch predictor prior to memset() invocation is not pre-trained for the branching sequence inside a memset() implementation.
- Memset() count values are likely to be uncorrected.

The proper measurement technique to compare memset() performance for more realistic memset() invocation scenarios will require a per-invocation technique that wraps two RDTSC around each invocation of memset().

With the per-invocation RDTSC measurement technique, the overhead of RDTSC and be pre-calibrated and post-validated outside of a measurement loop. The per-invocation technique may also consider cache warming effect by using a loop to wrap around the per-invocation measurements.

When the relevant skew factors of measurement techniques are taken into effect, the performance of memset() using REP STOSB, for count values smaller than a few hundred bytes, is generally faster than

the AVX2 version for the common memset() invocation scenarios. Only in the extreme scenarios of hundreds of unrolled memset() calls, all using count values less than a few hundred bytes and with no intervening instruction stream between each pair of memset() can the AVX2 version of memset() take advantage of the training effect of the branch predictor.

11.16.3.4 Hoisting Memcpy/Memset Ahead of Consuming Code

There may be situations where the data furnished by a call to memcpy/memset and subsequent instructions consuming the data can be re-arranged:

```
memcpy ( pBuf, pSrc, Cnt); // make a copy of some data with knowledge of Cnt
..... // subsequent instruction sequences are not consuming pBuf immediately
result = compute( pBuf); // memcpy result consumed here
```

When the count is known to be at least a thousand byte or more, using enhanced REP MOVSB/STOSB can provide another advantage to amortize the cost of the non-consuming code. The heuristic can be understood using a value of Cnt = 4096 and memset() as example:

- A 256-bit SIMD implementation of memset() will need to issue/execute retire 128 instances of 32-byte store operation with VMOVDQA, before the non-consuming instruction sequences can make their way to retirement.
- An instance of enhanced REP STOSB with ECX= 4096 is decoded as a long micro-op flow provided by hardware, but retires as one instruction. There are many store_data operation that must complete before the result of memset() can be consumed. Because the completion of store data operation is de-coupled from program-order retirement, a substantial part of the non-consuming code stream can process through the issue/execute and retirement, essentially cost-free if the non-consuming sequence does not compete for store buffer resources.

Software that use enhanced REP MOVSB/STOSB much check its availability by verifying CPUID.(EAX=07H, ECX=0):EBX.ERMSB (bit 9) reports 1.

11.16.3.5 256-bit Fetch versus Two 128-bit Fetches

On Intel microarchitecture code name Sandy Bridge and Ivy Bridge, using two 16-byte aligned loads are preferred due to the 128-bit data path limitation in the memory pipeline of the microarchitecture.

To take advantage of Intel microarchitecture code name Haswell's 256-bit data path microarchitecture, the use of 256-bit loads must consider the alignment implications. Instruction that fetched 256-bit data from memory should pay attention to be 32-byte aligned. If a 32-byte unaligned fetch would span across cache line boundary, it is still preferable to fetch data from two 16-byte aligned address instead.

11.16.3.6 Mixing MULX and AVX2 Instructions

Combining MULX and AVX2 instruction can further improve the performance of some common computation task, e.g. numeric conversion 64-bit integer to ascii format can benefit from the flexibility of MULX register allocation, wider YMM register, and variable packed shift primitive VPSRLVD for parallel moduli/remainder calculations.

Example 11-40 shows a macro sequence of AVX2 instruction to calculate one or two finite range unsigned short integer(s) into respective decimal digits, featuring VPSRLVD in conjunction with Montgomery reduction technique.

Example 11-40. Macros for Parallel Moduli/Remainder Calculation

```
static short quoTenThsn_mulplr_d[16] =
{ 0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0, 0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0};
static short mten_mulplr_d[16] = { -10, 1, -10, 1, -10, 1, -10, 1, -10, 1, -10, 1, -10, 1, -10, 1};
```

Example 11-40. Macros for Parallel Moduli/Remainder Calculation (Contd.)

```
// macro to convert input t5 (a __m256i type) containing quotient (dword 4) and remainder
// (dword 0) into single-digit integer (between 0-9) in output y3 ( a__m256i);
//both dword element "t5" is assume to be less than 10^4, the rest of dword must be 0;
//the output is 8 single-digit integer, located in the low byte of each dword, MS digit in dword 0
#define    __ParMod10to4AVX2dw4_0( y3, t5 ) \
{ __m256i x0, x2;              \
  x0 = _mm256_shuffle_epi32( t5, 0);  \
  x2 = _mm256_mulhi_epu16(x0, _mm256_loadu_si256( (__m256i *)  quoTenThsn_mulplr_d));\
  x2 = _mm256_srlv_epi32(  x2, _mm256_setr_epi32(0x0, 0x4, 0x7, 0xa, 0x0, 0x4, 0x7, 0xa) ); \
  (y3) = _mm256_or_si256(_mm256_slli_si256(x2, 6),   _mm256_slli_si256(t5, 2) );  \
  (y3) = _mm256_or_si256(x2, y3);\
  (y3) = _mm256_madd_epi16(y3, _mm256_loadu_si256( (__m256i *) mten_mulplr_d) ) ;\
}}
// parallel conversion of dword integer (< 10^4) to 4 single digit integer in __m128i
#define    __ParMod10to4AVX2dw( x3, dw32 ) \
{ __m128i x0, x2;              \
  x0 = _mm_broadcastd_epi32( _mm_cvtsi32_si128(  dw32));  \
  x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d));\
  x2 = _mm_srlv_epi32(  x2, _mm_setr_epi32(0x0, 0x4, 0x7, 0xa) ); \
  (x3) = _mm_or_si128(_mm_slli_si128(x2, 6),   _mm_slli_si128(_mm_cvtsi32_si128(  dw32), 2) );  \
  (x3) = _mm_or_si128(x2, (x3));\
  (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;\
}
```

Example 11-41 shows a helper utility and overall steps to reduce a 64-bit signed integer into 63-bit unsigned range. reduced-range integer quotient/remainder pairs using MULX.

Example 11-41. Signed 64-bit Integer Conversion Utility

```
#defineQWCG10to80xabcc77118461cefdull
static  short  quo4digComp_mulplr_d[8] = {  1024, 0, 64, 0, 8, 0, 0, 0};
```

```
static  int   pr_cg_10to4[8] = {  0x68db8db, 0 , 0, 0, 0x68db8db, 0, 0, 0};
static  int   pr_1_m10to4[8] = {  -10000, 0 , 0, 0 , 1, 0 , 0, 0};

char * i64toa_avx2i(  __int64 xx, char * p)
{int cnt;
  _mm256_zeroupper();
  if( xx < 0) cnt = avx2i_q2a_u63b(-xx, p);
  else   cnt = avx2i_q2a_u63b(xx, p);
  p[cnt] = 0;
  return p;
}
```

Example 11-41. Signed 64-bit Integer Conversion Utility (Contd.)

```
// Convert unsigned short (< 10^4) to ascii
__inline  int ubsAvx2_Lt10k_2s_i2(int x_Lt10k, char *ps)
{int tmp;
__m128i x0, m0, x2, x3, x4,  compv;
   if( x_Lt10k < 10) { *ps = '0' + x_Lt10k; return 1; }
   x0 = _mm_broadcastd_epi32( _mm_cvtsi32_si128(  x_Lt10k));
   // calculate quotients of divisors 10, 100, 1000, 10000
   m0 = _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d);
   x2 = _mm_mulhi_epu16(x0, m0);
   // u16/10, u16/100, u16/1000, u16/10000
   x2 = _mm_srlv_epi32(  x2, _mm_setr_epi32(0x0, 0x4, 0x7, 0xa) );
   // 0, u16, 0, u16/10, 0, u16/100, 0, u16/1000
   x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1);
   x4 = _mm_or_si128(x2, x3);
   // produce 4 single digits in low byte of each dword
   x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;// add bias for ascii encoding
   x2 = _mm_add_epi32( x4, _mm_set1_epi32( 0x30303030 ) );
   // pack 4 single digit into a dword, start with most significant digit
   x3 = _mm_shuffle_epi8(x2, _mm_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );
   if (x_Lt10k > 999 )    *(int *) ps = _mm_cvtsi128_si32(  x3);  return 4;
   else {
     tmp = _mm_cvtsi128_si32(  x3);
     if (x_Lt10k > 99 ) {
        *((short *) (ps)) = (short ) (tmp >>8);
        ps[2] = (char ) (tmp >>24);
        return 3;
     }

               (continue)
```

```
     else if ( x_Lt10k > 9){
        *((short *) ps) = (short ) tmp;
        return 2;
     }
   }
}
```

Example 11-42 shows the steps of numeric conversion of 63-bit dynamic range into ascii format according to a progressive range reduction technique using vectorized Montgomery reduction scheme.

Example 11-42. Unsigned 63-bit Integer Conversion Utility

```
unsigned   avx2i_q2a_u63b (unsigned __int64 xx, char *ps)
{ __m128i  v0;
  __m256i  m0, x1, x2, x3, x4, x5 ;
  unsigned __int64 xxi, xx2, lo64, hi64;
__int64  w;
  int j, cnt, abv16, tmp, idx, u;
    // conversion of less than 4 digits
   if ( xx < 10000 ) {
       j = ubsAvx2_Lt10k_2s_i2 ( (unsigned ) xx, ps); return j;
   } else if (xx < 100000000 ) {  // dynamic range of xx is less than 9 digits
    // conversion of 5-8 digits
     x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128(xx));  // broadcast to every dword
    // calculate quotient and remainder, each with reduced range (< 10^4)
     x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
     x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
   // quotient in dw4, remainder in dw0
     m0 = _mm256_add_epi32( _mm256_castsi128_si256( _mm_cvtsi32_si128(xx)), x3);
     __ParMod10to4AVX2dw4_0( x3, m0); // 8 digit in low byte of each dw
     x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
     x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080, 
0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );
   // pack 8 single-digit integer into first 8 bytes and set rest to zeros
     x4 = _mm256_permutevar8x32_epi32( x4, _mm256_setr_epi32(0x4, 0x0, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1)  );
     tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
     _BitScanForward((unsigned long *) &idx, tmp);
     cnt = 8 -idx; // actual number non-zero-leading digits to write to output
   } else {  // conversion of 9-12 digits
     lo64 = _mulx_u64(xx, (unsigned __int64) QWCG10to8, &hi64);
     hi64 >>= 26;
     xxi = _mulx_u64(hi64, (unsigned __int64)100000000, &xx2);
     lo64 = (unsigned __int64)xx - xxi;
```

(continue)

Example 11-42. Unsigned 63-bit Integer Conversion Utility (Contd.)

```
    if( hi64 < 10000) { // do digist 12-9 first
        __ParMod10to4AVX2dw(v0, hi64);
        v0 = _mm_add_epi32( v0, _mm_set1_epi32( 0x30303030 ) );
       // continue conversion of low 8 digits of a less-than 12-digit value
        x5 = _mm256_setzero_si256( );
        x5 = _mm256_castsi128_si256( _mm_cvtsi32_si128(lo64));
        x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128(lo64));  // broadcast to every dword
        x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
        x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
        m0 = _mm256_add_epi32( x5, x3); // quotient in dw4, remainder in dw0
        __ParMod10to4AVX2dw4_0( x3, m0);
        x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
        x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );
        x5 = _mm256_castsi128_si256( _mm_shuffle_epi8( v0, _mm_setr_epi32(0x80808080, 0x80808080,
0x0004080c, 0x80808080) ));
        x4 = _mm256_permutevar8x32_epi32( _mm256_or_si256(x4, x5), _mm256_setr_epi32(0x2, 0x4, 0x0, 0x1,
0x1, 0x1, 0x1, 0x1)  );
        tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
        _BitScanForward((unsigned long *) &idx, tmp);
        cnt = 12 -idx;
    } else { // handle greater than 12 digit input value
        cnt = 0;
        if ( hi64 >  100000000) { // case of input value has more than 16 digits
          xxi =  _mulx_u64(hi64, (unsigned __int64) QWCG10to8, &xx2) ;
          abv16 = xx2 >>26;
          hi64 -= _mulx_u64((unsigned __int64) abv16, (unsigned __int64) 100000000, &xx2);
          __ParMod10to4AVX2dw(v0,  abv16);
          v0 = _mm_add_epi32( v0, _mm_set1_epi32( 0x30303030 ) );
          v0 = _mm_shuffle_epi8(v0, _mm_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );
          tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(v0, _mm_set1_epi32( 0x30303030 )) );
          _BitScanForward((unsigned long *) &idx, tmp);
          cnt = 4 -idx;
       }

                   (continue)
```

Example 11-42. Unsigned 63-bit Integer Conversion Utility (Contd.)

```
    // conversion of lower 16 digits
     x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128(hi64));  // broadcast to every dword
     x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
     x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
     m0 = _mm256_add_epi32( _mm256_castsi128_si256( _mm_cvtsi32_si128(hi64)), x3);
     __ParMod10to4AVX2dw4_0( x3, m0);
     x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
     x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );
     x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128(lo64));  // broadcast to every dword
     x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
     x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
     m0 = _mm256_add_epi32( _mm256_castsi128_si256( _mm_cvtsi32_si128(hi64)), x3);
     __ParMod10to4AVX2dw4_0( x3, m0);
     x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
     x5 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x80808080, 0x80808080, 0x0004080c, 0x80808080,
0x80808080, 0x80808080, 0x0004080c, 0x80808080)  );
     x4 = _mm256_permutevar8x32_epi32( _mm256_or_si256(x4, x5), _mm256_setr_epi32(0x4, 0x0, 0x6, 0x2,
0x1,  0x1, 0x1, 0x1)  );
     cnt += 16;
     if (cnt <=  16) {
        tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
        _BitScanForward((unsigned long *) &idx, tmp);
        cnt -= idx;
     }
   }
 }
```

```
  w = _mm_cvtsi128_si64( _mm256_castsi256_si128(x4));
  switch(cnt) {
  case5:*ps++ = (char) (w >>24);  *(unsigned *) ps = (w >>32);
  break;
  case6:*(short *)ps = (short) (w >>16);  *(unsigned *) (&ps[2]) = (w >>32);
  break;
  case7:*ps = (char) (w >>8);   *(short *) (&ps[1]) = (short) (w >>16);
   *(unsigned *) (&ps[3]) = (w >>32);
  break;
  case 8: *(long long *)ps = w;
  break;
  case9:*ps++ = (char) (w >>24);  *(long long *) (&ps[0]) =  _mm_cvtsi128_si64(
_mm_srli_si128(_mm256_castsi256_si128(x4), 4));
  break;
```

(continue)

Example 11-42. Unsigned 63-bit Integer Conversion Utility (Contd.)

```
    case10:*(short *)ps = (short) (w >>16);
    *(long long *) (&ps[2])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
    break;
    case11:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
    *(long long *) (&ps[3])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
    break;
    case 12: *(unsigned *)ps = w; *(long long *) (&ps[4])  = _mm_cvtsi128_si64(
 _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
    break;
    case13:*ps++ = (char) (w >>24);  *(unsigned *) ps = (w >>32);
    *(long long *) (&ps[4]) =  _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
    break;
    case14:*(short *)ps = (short) (w >>16);  *(unsigned *) (&ps[2]) = (w >>32);
    *(long long *) (&ps[6]) =  _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
    break;
    case15:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
        *(unsigned *) (&ps[3]) = (w >>32);
    *(long long *) (&ps[7])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
    break;
    case 16: _mm_storeu_si128( (__m128i *) ps, _mm256_castsi256_si128(x4));
    break;
    case17:u = _mm_cvtsi128_si64(v0);  *ps++ = (char) (u >>24);
    _mm_storeu_si128( (__m128i *) &ps[0], _mm256_castsi256_si128(x4));
    break;
    case18:u = _mm_cvtsi128_si64(v0);  *(short *)ps = (short) (u >>16);
    _mm_storeu_si128( (__m128i *) &ps[2], _mm256_castsi256_si128(x4));
    break;
    case19:u = _mm_cvtsi128_si64(v0);  *ps = (char) (u >>8); *(short *) (&ps[1]) = (short) (u >>16);
    _mm_storeu_si128( (__m128i *) &ps[3], _mm256_castsi256_si128(x4));
    break;
    case20:u = _mm_cvtsi128_si64(v0);  *(unsigned *)ps = (short) (u);
    _mm_storeu_si128( (__m128i *) &ps[4], _mm256_castsi256_si128(x4));
    break;
    }

  return cnt;
 }
```

The AVX2 version of numeric conversion across the dynamic range of 3/9/17 output digits are approximately 23/57/54 cycles per input, compared to standard library implement ion's range of 85/260/560 cycles per input.

The techniques illustrated above can be extended to numeric conversion of other library, such as binary-integer-decimal (BID) encoded IEEE-754-2008 Decimal floating-point format. For BID-128 format, Example 11-42 can be adapted by adding another range-reduction stage using a pre-computed 256-bit constant to perform Montgomery reduction at modulus 10^16. The technique to construct the 256-bit

constant is covered in Chapter 10, "SSE4.2 and SIMD Programming For Text-Processing/LexING/Parsing"of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

11.16.4 Considerations for Gather Instructions

VGATHER family of instructions fetch multiple data elements specified by a vector index register containing relative offsets from a base address. Processors based on the Haswell microarchitecture is the first implementation of the VGATHER instruction and a single instruction results in multiple micro-ops being executed. In the Broadwell microarchitecture, the throughput of the VGATHER family of instructions have improved significantly; see Table C-5.

Depending on data organization and access patterns, it is possible to create equivalent code sequences without using VGATHER instruction that will execute faster and with fewer micro-ops than a single VGATHER instruction (e.g. see Section 11.5.1). Example 11-43 shows some of those situations where, use of VGATHER on Intel microarchitecture code name Haswell is unlikely to provide performance benefit.

Example 11-43. Access Patterns Favoring Non-VGATHER Techniques

Access Patterns	Recommended Instruction Selection
Sequential elements	Regular SIMD loads (MOVAPS/MOVUPS, MOVDQA/MOVDQU)
Fewer than 4 elements	Regular SIMD load + horizontal data-movement to re-arrange slots
Small Strides Transpositions	Load all nearby elements + shuffle/permute to collected strided elements: VMOVUPD YMM0, [sequential elements] VPERMQ YMM1, YMM0, 0x08 // the even elements VPERMQ YMM2, YMM0, 0x0d // the odd elements Regular SIMD loads + shuffle/permute/blend to transpose to columns

Redundant elements	Load once + shuffle/blend/logical to build data vectors in register. In this case, result[i] = x[index[i]] + x[index[i+1]], the technique below may be preferable to using multiple VGATHER:
	ymm0 <- VGATHER (x[index[k]]); // fetching 8 elements
	ymm1 <- VBLEND(VPERM (ymm0), VBROADCAST (x[indexx[k+8]]);
	ymm2 <- VPADD(ymm0, ymm1);

In other cases, using VGATHER instruction can reduce code size and execute faster with techniques including but not limited to amortizing the latency and throughput of VGATHER, or by hoisting the fetch operations well in advance of consumer code of the destination register of those fetches. Example 11-44 lists some patterns that can benefit from using VGATHER on Intel microarchitecture code name Haswell.

General tips for using VGATHER:

- Gathering more elements with a VGATHER instruction helps amortize the latency and throughput of VGATHER, and is more likely to provide performance benefit over an equivalent non-VGATHER flow. For example, the latency of 256-bit VGATHER is less than twice the equivalent 128-bit VGATHER and therefore more likely to show gains than two 128-bit equivalent ones. Also, using index size larger than data element size results in only half of the register slots utilized but not a proportional latency reduction. Therefore the dword index form of VGATHER is preferred over qword index if dwords or single-precision values are to be fetched.
- It is advantageous to hoist VGATHER well in advance of the consumer code.
- VGATHER merges the (unmasked) gathered elements with the previous value of the destination. Therefore, in cases where the previous value of the destination doesn't need to be merged (for instance, when no elements is masked off), it can be beneficial to break the dependency of the

VGATHER instruction on the previous writer of the destination register (by zeroing out the register with a VXOR instruction).

Example 11-44. Access Patterns Likely to Favor VGATHER Techniques

Access Patterns	Instruction Selection

4 or more elements with unknown masks	Code with conditional element gathers typically either will not vectorize without a VGATHER instruction or provide relatively poor performance due to data-dependent mis-predicted branches.
	C code with data-dependent branches:
	if (condition[i] > 0) { result[i] = x[index[i]] }
	AVX2 equivalent sequence:
	YMM0 <- VPCMPGT (condition, zeros) // compute vector mask
	YMM2 <- VGATHER (x[YMM1], YMM0) // addr=x[YMM1], mask=YMM0

Vectorized index calculation with 8 elements	Vectorized calculations to generate the index synergizes well with the VGATHER instruction functionality.
	C code snippet:
	x[index1[i] + index2[i]]
	AVX2 equivalent:
	YMM0 <- VPADD (index1, index2) // calc vector index
	YMM1 <- VGATHER (x[YMM0], mask) // addr=x[YMM0]

Performance of the VGATHER instruction compared to a multi-instruction gather equivalent flow can vary due to (1) differences in the base algorithm, (2) different data organization, and (3) the effectiveness of the equivalent flow. In performance critical applications it is advisable to evaluate both options before choosing one.

The throughput of GATHER instructions continue to improve from Broadwell to Skylake Microarchitecture. This is shown in Figure 11-4.

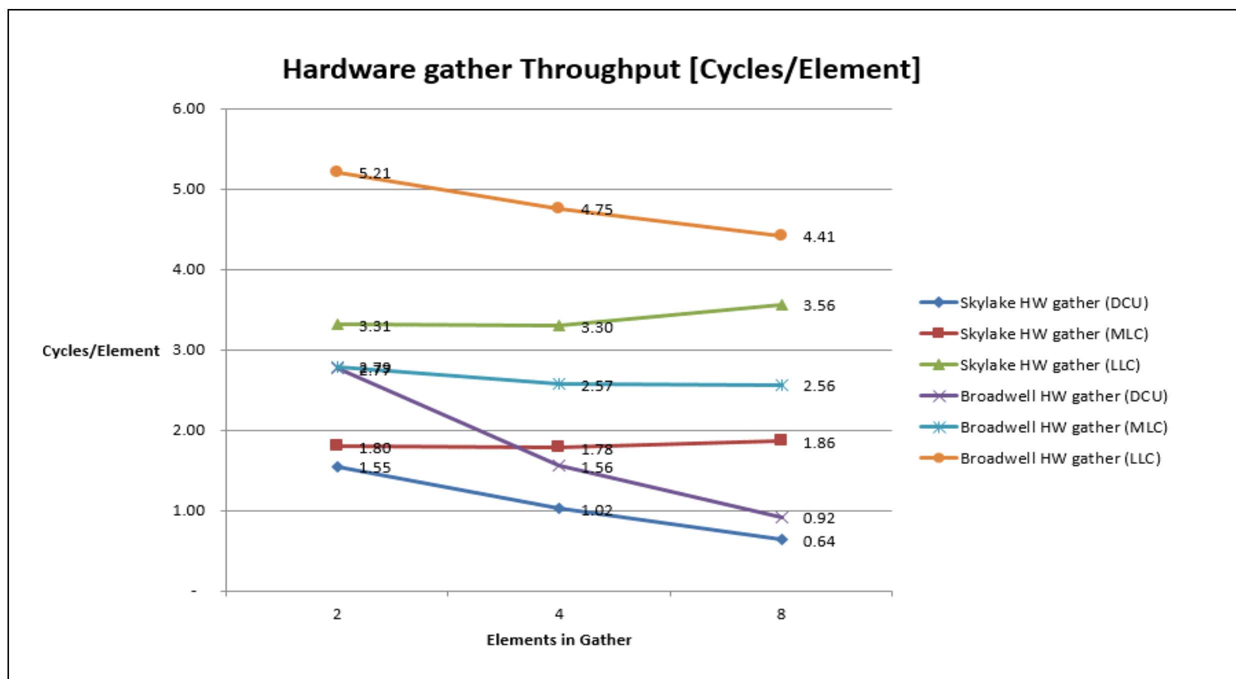


Figure 11-4. Throughput Comparison of Gather Instructions

Example 11-45 gives the asm sequence of software implementation that is equivalent to the VPGATHERD instruction. This can be used to compare the trade-off of using a hardware gather instruction or software gather sequence based on inserting an individual element.

Example 11-45. Software AVX Sequence Equivalent to Full-Mask VPGATHERD

```
mov eax, [rdi]                        // load index0
vmovd xmm0, [rsi+4*rax]               // load element0
mov eax, [rdi+4]                      // load index1
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x1  // load element1
mov eax, [rdi+8]                      // load index2
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x2  // load element2
mov eax, [rdi+12]                     // load index3
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x3  // load element3
mov eax, [rdi+16]                     // load index4
vmovd xmm1, [rsi+4*rax]               // load element4
mov eax, [rdi+20]                     // load index5
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x1  // load element5
mov eax, [rdi+24]                     // load index6
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x2  // load element6
mov eax, [rdi+28]                     // load index7
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x3  // load element7
vinserti128 ymm0, ymm0, xmm1, 1       //result in ymm0
```

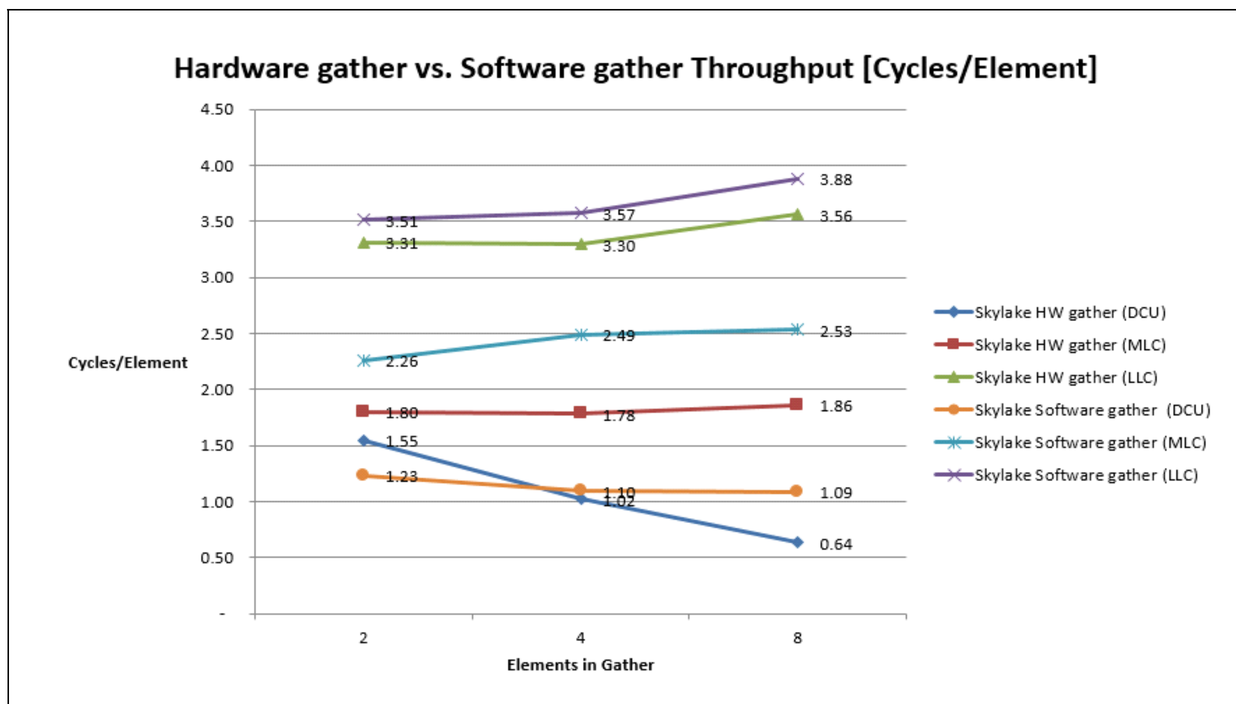


Figure 11-5. Comparison of HW GATHER Versus Software Sequence in Skylake Microarchitecture

Figure 11-5 compares per-element throughput using the VPGATHERD instruction versus a software gather sequence with Skylake microarchitecture as a function of cache locality of data supply. With the exception of using hardware GATHER on two data elements per instruction, the gather instruction out-performs the software sequence on Skylake microarchitecture.

If data supply locality is from memory, software sequences are likely to perform better than the hardware GATHER instruction.

11.16.4.1 Strided Loads

This section compares using the hardware GATHER instruction versus alternative implementations of handling Array of Structures (AOS) to Structure of Arrays (SOA) transformation. The code separates the real and imaginary elements in a complex array into two separate arrays.

C code:

```
for(int i=0;i<len;i++){
    Real_buffer[i] = Complex_buffer[i].real;
    Imaginary_buffer[i] = Complex_buffer[i].imag;
}
```

Example 11-46. AOS to SOA Transformation Alternatives

1: Scalar Code	2: AVX w/ VINSRT+VSHUFPS	3: AVX2 w/ VPGATHERD

loop:
lea eax, ptr [r10+r10*1]
movsxd rax, eax
inc r10d
mov r11d, dword ptr [rsi+rax*8]
mov dword ptr [rcx+rax*4], r11d
mov r11d, dword ptr [rsi+rax*8+0x4]
mov dword ptr [rdx+rax*4], r11d
mov r11d, dword ptr [rsi+rax*8+0x8]
mov dword ptr [rcx+rax*4+0x4], r11d
mov r11d, dword ptr [rsi+rax*8+0xc]
mov dword ptr [rdx+rax*4+0x4], r11d
cmp r10d, r8d
jl loop
	loop:
	vmovdqu xmm0, xmmword ptr [r10+rcx*8]
	vmovdqu xmm1, xmmword ptr [r10+rcx*8+0x10]
	vmovdqu xmm4, xmmword ptr [r10+rcx*8+0x40]
	vmovdqu xmm5, xmmword ptr [r10+rcx*8+0x50]
	vinserti128 ymm2, ymm0, xmmword ptr [r10+rcx*8+0x20], 0x1
	vinserti128 ymm3, ymm1, xmmword ptr [r10+rcx*8+0x30], 0x1
	vinserti128 ymm6, ymm4, xmmword ptr [r10+rcx*8+0x60], 0x1
	vinserti128 ymm7, ymm5, xmmword ptr [r10+rcx*8+0x70], 0x1
	add rcx, 0x10
	vshufps ymm0, ymm2, ymm3, 0x88
	vshufps ymm1, ymm2, ymm3, 0xdd
	vshufps ymm4, ymm6, ymm7, 0x88
	vshufps ymm5, ymm6, ymm7, 0xdd
	vmovups ymmword ptr [r9], ymm0
	vmovups ymmword ptr [r8], ymm1
	vmovups ymmword ptr [r9+0x20], ymm4
	vmovups ymmword ptr [r8+0x20],ymm5
		loop:
		lea r11, ptr [r10+rcx*8]
		vpxor ymm5, ymm5, ymm5
		add rcx, 0x8
		vpxor ymm6, ymm6, ymm6
		vmovdqa ymm3, ymm0
		vmovdqa ymm4, ymm0
		vpgatherdd ymm5, ymmword ptr [r11+ymm2*4], ymm3
		vpgatherdd ymm6, ymmword ptr [r11+ymm1*4], ymm4
		vmovdqu ymmword ptr [r9], ymm5
		vmovdqu ymmword ptr [r8],ymm6
		add r9, 0x20
		add r8, 0x20
		cmp rcx, rsi
		jl loop

Example 11-46. AOS to SOA Transformation Alternatives (Contd.)

1: Scalar Code	2: AVX w/ VINSRT+VSHUFPS	3: AVX2 w/ VPGATHERD

	add r9, 0x40
	add r8, 0x40
	cmp rcx, rsi
	jl loop	

With strided access patterns, an AVX software sequence can load and shuffle on multiple elements and is the more optimal technique.

Table 11-10. Comparison of AOS to SOA with Strided Access Pattern

Microarchitecture	Scalar	VPGATHERD	AVX VINSRTF128/VSHUFFLEPS
Broadwell	1X	1.7X	4.8X
Skylake	1X	2.7X	4.9X

11.16.4.2 Adjacent Loads

This section compares using the hardware GATHER instruction versus alternative implementations of handling a variant situation of AOS to SOA transformation. In this case, AOS data are not loaded sequentially but via an index array.

C code:

```
for(int i=0;i<len;i++){
    Real_buffer[i] = Complex_buffer[Index_buffer[i]].real;
    Imaginary_buffer[i] = Complex_buffer[Index_buffer[i]].imag;
}
```

Example 11-47. Non-Strided AOS to SOA

AVX2 GATHERPD	AVX VINSRTF128 /UNPACK

loop:
vmovdqu ymm1, ymmword ptr [rsi+rdx*4]
vpaddd ymm3, ymm1, ymm1
vxorpd ymm5, ymm5, ymm5
vmovdqa ymm2, ymm0
vxorpd ymm6, ymm6, ymm6
vmovdqa ymm4, ymm0
vxorpd ymm10, ymm10, ymm10
vmovdqa ymm7, ymm0
vxorpd ymm11, ymm11, ymm11
vmovdqa ymm9, ymm0
vextracti128 xmm8, ymm3, 0x1
vgatherdpd ymm6, ymmword ptr[r8+xmm8*8],ymm4
vgatherdpd ymm5, ymmword ptr[r8+xmm3*8],ymm2
vmovupd ymmword ptr [rax+rdx*8], ymm10
vmovupd ymmword ptr [rax+rdx*8+0x20], ymm11
add rdx, 0x8
cmp rdx, r11
jb loop
	loop:
	movsxd r10, dword ptr [rdx+rsi*4]
	shl r10, 0x4
	movsxd r11, dword ptr [rdx+rsi*4+0x8]
	shl r11, 0x4
	vmovupd xmm0, xmmword ptr [r9+r10*1]
	movsxd r10, dword ptr [rdx+rsi*4+0x4]
	shl r10, 0x4
	vinsertf128 ymm2, ymm0, xmmword ptr [r9+r11*1], 0x1
	vmovupd xmm1, xmmword ptr [r9+r10*1]
	movsxd r10, dword ptr [rdx+rsi*4+0xc]
	shl r10, 0x4
	vinsertf128 ymm3, ymm1, xmmword ptr [r9+r10*1], 0x1
	movsxd r10, dword ptr [rdx+rsi*4+0x10]
	shl r10, 0x4
	vunpcklpd ymm4, ymm2, ymm3
	vunpckhpd ymm5, ymm2, ymm3
	vmovupd ymmword ptr [rcx], ymm4

Example 11-47. Non-Strided AOS to SOA (Contd.)

AVX2 GATHERPD	AVX VINSRTF128 /UNPACK

	vmovupd xmm6, xmmword ptr [r9+r10*1]
	vmovupd ymmword ptr [rax], ymm5
	movsxd r10, dword ptr [rdx+rsi*4+0x18]
	shl r10, 0x4
	vinsertf128 ymm8, ymm6, xmmword ptr [r9+r10*1], 0x1
	movsxd r10, dword ptr [rdx+rsi*4+0x14]
	shl r10, 0x4
	vmovupd xmm7, xmmword ptr [r9+r10*1]
	movsxd r10, dword ptr [rdx+rsi*4+0x1c]
	add rsi, 0x8
	shl r10, 0x4
	vinsertf128 ymm9, ymm7, xmmword ptr [r9+r10*1], 0x1
	vunpcklpd ymm10, ymm8, ymm9
	vunpckhpd ymm11, ymm8, ymm9
	vmovupd ymmword ptr [rcx+0x20], ymm10
	add rcx, 0x40
	vmovupd ymmword ptr [rax+0x20], ymm11
	add rax, 0x40
	cmp rsi, r8
	jl loop

With non-strided, regular access pattern of AOS to SOA, an AVX software sequence that uses VINSERTF128 and interleaved packing of multiple elements can be more optimal.

Table 11-11. Comparison of Indexed AOS to SOA Transformation

Microarchitecture	VPGATHERPD	AVX VINSRTF128/VUNPCK*
Broadwell	1X	1.4X
Skylake	1.3X	1.7X

11.16.5 AVX2 Conversion Remedy to MMX Instruction Throughput Limitation

In processors based on the Skylake microarchitecture, the functionality of the MMX instruction set is unchanged from prior generations. But many MMX instructions are constrained to execute to one port with half the instruction throughput relative to prior microarchitectures. The MMX instructions with throughput constraints include:

- PADDS[B/W], PADDUS[B/W], PSUBS[B/W], PSUBUS[B/W].
- PCMPGT[B/W/D], PCMPEQ[B/W/D].
- PMAX[UB/SW], PMIN[UB/SW].
- PAVG[B/W], PABS[B/W/D], PSIGN[B/W/D].

To overcome the reduction of MMX instruction throughput, conversion of asm and intrinsic code to use AVX2 instruction will provide significant performance improvements. Example 11-48 shows the asm sequence using AVX2 versus MMX equivalent. In Skylake microarchitecture, the MMX code shown in Example 11-48 will execute at approximately half the speed relative to the Broadwell microarchitecture. This is due to PMAXSW/PMINSW throughput being reduced by half with the single-port restriction. When the same task is implemented with the equivalent AVX2 sequence, the performance of the AVX2 code on Skylake microarchitecture will be ~3.9X of the MMX code executing on the Broadwell microarchitecture.

Example 11-48. Conversion to Throughput-Reduced MMX sequence to AVX2 Alternative

MMX Code	AVX2 Code

mov rax, pIn
mov rbx, pOut
mov r8, len
xor rcx, rcx
mov rcx, 8
movq mm0, [rax]
movq mm1, [rax + 8]
movq mm2, mm0
movq mm3, mm1
cmp rcx, r8
jge end
loop:
movq mm4, [rax + 2*rcx]
movq mm5, [rax + 2*rcx + 8]
 	
pmaxsw mm0, mm4
pmaxsw mm1, mm5
pminsw mm2, mm4
pminsw mm3, mm5
add rcx, 8
cmp rcx, r8
jl loop
 	
end:
//Reduction
pmaxsw mm0, mm1
pshufw mm1, mm0, 0xE
pmaxsw mm0, mm1
pshufw mm1, mm0, 1
pmaxsw mm0, mm1
 	
pminsw mm2, mm3
pshufw mm3, mm2, 0xE
pminsw mm2, mm3
pshufw mm3, mm2, 1
pminsw mm2, mm3
 	
movd eax, mm0
mov WORD PTR [rbx], ax
movd eax, mm2
mov WORD PTR [rbx + 2], ax
emms
	mov rax, pIn
	mov rbx, pOut
	mov r8, len
	xor rcx, rcx
	mov rcx, 32
	vmovdqu ymm0, [rax]
	vmovdqu ymm1, [rax + 32]
	vmovdqu ymm2, ymm0
	vmovdqu ymm3, ymm1
	cmp rcx, r8
	jge end
	loop:
	vmovdqu ymm4, ymmword ptr [rax + 2*rcx]
	vmovdqu ymm5, ymmword ptr [rax + 2*rcx + 32]
	vpmaxsw ymm0, ymm0, ymm4
	vpmaxsw ymm1, ymm1, ymm5
	vpminsw ymm2, ymm2, ymm4
	vpminsw ymm3, ymm3, ymm5
	add rcx, 32
	cmp rcx, r8
	jl loop
	end:
	//Reduction
	vpmaxsw ymm0, ymm0, ymm1
	vextracti128 xmm1, ymm0, 1
	vpmaxsw xmm0, xmm0, xmm1
	vpshufd xmm1, xmm0, 0xe
	vpmaxsw xmm0, xmm0, xmm1
	vpshuflw xmm1, xmm0, 0xe
	vpmaxsw xmm0, xmm0, xmm1
	vpshuflw xmm1, xmm0, 1
	vpmaxsw xmm0, xmm0, xmm1
	vmovd eax, xmm0
	mov WORD PTR [rbx], ax
	vpminsw ymm2, ymm2, ymm3
	vextracti128 xmm1, ymm2, 1
	vpminsw xmm2, xmm2, xmm1
	vpshufd xmm1, xmm2, 0xe
	vpminsw xmm2, xmm2, xmm1
	vpshuflw xmm1, xmm2, 0xe
	vpminsw xmm2, xmm2, xmm1
	vpshuflw xmm1, xmm2, 1
	vpminsw xmm2, xmm2, xmm1
	vmovd eax, xmm2
	mov WORD PTR [rbx + 2], ax

CHAPTER 12
INTEL® TSX RECOMMENDATIONS

12.1 INTRODUCTION

Intel® Transactional Synchronization Extensions (Intel TSX) aim to improve the performance of lock-protected critical sections while maintaining the lock-based programming model.

Intel TSX allows the processor to determine dynamically whether threads need to serialize through lock-protected critical sections, and to perform serialization only when required. This lets hardware expose and exploit concurrency hidden in an application due to dynamically unnecessary synchronization through a technique known as lock elision.

With lock elision, the hardware executes the programmer-specified critical sections (also referred to as **transactional regions**) transactionally. In such an execution, the lock variable is only read within the transactional region; it is not written to (and therefore not acquired), with the expectation that the lock variable remains unchanged after the transactional region, thus exposing concurrency.

If the transactional execution completes successfully, then the hardware ensures that all memory operations performed within the transactional region will appear to have occurred instantaneously when viewed from other logical processors. A processor makes architectural updates performed within the region visible to other logical processors only on a successful commit, a process referred to as an **atomic commit**. Any updates performed within the transactional region are made visible to other logical processors only on an atomic commit.

Since a successful transactional execution ensures an atomic commit, the processor can execute the programmer-specified code section optimistically without synchronization. If synchronization was unnecessary for that specific execution, execution can commit without any cross-thread serialization.

If the transactional execution is unsuccessful, the processor cannot commit the updates atomically. When this happens, the processor will roll back the execution, a process referred to as a **transactional abort**. On a transactional abort, the processor will discard all updates performed in the region, restore architectural state to appear as if the optimistic execution never occurred, and resume execution non-transactionally. Depending on the policy in place, lock elision may be retried or the lock may be explicitly acquired to ensure forward progress.

Intel TSX provides two software interfaces to programmers:

- **Hardware Lock Elision** (HLE) is a legacy compatible instruction set extension (comprising of the XACQUIRE and XRELEASE prefixes).
- **Restricted Transactional Memory** (RTM) is a new instruction set interface (comprising of the XBEGIN and XEND instructions).

Programmers who would like to run Intel TSX enabled software on legacy hardware would use the HLE interface to implement lock elision. On the other hand, programmers who do not have legacy hardware requirements and who deal with more complex locking primitives would use the RTM interface of Intel TSX to implement lock elision. In the latter case when using new instructions, the programmer must always provide a non-transactional path (which would have code to eventually acquire the lock being elided) to execute following a transactional abort and must not rely on the transactional execution alone.

In addition, Intel TSX also provides the XTEST instruction to test whether a logical processor is executing transactionally, and the XABORT instruction to abort a transactional region.

A processor can perform a transactional abort for numerous reasons. A primary cause is due to conflicting data accesses between the transactionally executing logical processor and another logical processor. Such conflicting accesses may prevent a successful transactional execution. Memory addresses read from within a transactional region constitute the **read-set** of the transactional region and addresses written to within the transactional region constitute the **write-set** of the transactional region. Intel TSX maintains the read- and write-sets at the granularity of a cache line. For lock elision using RTM, the address of the lock being elided must be added to the read-set to ensure correct behavior of a transactionally executing thread in the presence of another thread that explicitly acquires the lock.

A conflicting data access occurs if another logical processor either reads a location that is part of the transactional region's write-set or writes a location that is a part of either the read- or write-set of the transactional region. We refer to this as a **data conflict**. Since Intel TSX detects data conflicts at the granularity of a cache line, unrelated data locations placed in the same cache line will be detected as conflicts. Transactional aborts may also occur due to limited transactional resources. For example, the amount of data accessed in the region may exceed an implementation-specific capacity. Some instructions, such as CPUID and IO instructions, may always cause a transactional execution to abort in the implementation.

Details of the Intel TSX interface can be found in Chapter 15 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The rest of this chapter provides guidelines for software developers to use the Intel TSX instructions. The guidelines focus on the use of the Intel TSX instructions to implement lock elision to enable concurrency of lock-protected critical sections, whether through the use of prefix hints as with HLE or through the use of new instructions as with RTM. Programmers may find other usages of the Intel TSX instructions beyond lock elision; those usages are not covered here.

In the sections below, we use the term **lock elision** to refer to either an **HLE**-based or an **RTM**-based implementation that elides locks.

12.1.1 Optimization Outline

This rest of this chapter describes the recommended approach for optimization and tuning of multi-threaded applications to use the Intel TSX instructions for lock elision. The focus of Intel TSX is to improve application performance (See Section 12.2) instead of synthetic micro-kernels that tend to overlook how real applications behave after acquiring a lock. We also discuss how to enable a synchronization library for lock elision using Intel TSX (See Section 12.3). We then discuss how to use the performance monitoring infrastructure for Intel TSX effectively (See Section 12.4) and present some performance guidelines for the first implementation (See Section 12.5).

The recommended guideline is to enable elision for all critical section locks and then identify problematic critical sections. Such a "bottoms-up" approach simplifies the evaluation and tuning of the resulting application and allows the programmer to focus on relevant critical sections.

Additional resources for TSX tuning are available at http://www.intel.com/software/tsx.

12.2 APPLICATION-LEVEL TUNING AND OPTIMIZATIONS

Applications typically use **synchronization libraries** to implement the lock acquire and lock release functions associated with critical sections. The simplest way to enable these applications to take advantage of Intel TSX-based lock elision is to use an Intel TSX-enabled synchronization library. Existing libraries may be already enabled to take advantage of the Intel TSX instructions (see Section 12.2.1). If an off-the-shelf, TSX-enabled library is not yet available, Section 12.3 discusses how to extend a locking library to use the Intel TSX instructions if it has not already been enabled. TSX-enabled synchronization libraries can be interchangeably used with conventional synchronization libraries.

While applications using these libraries can use Intel TSX without application modification, some basic tuning and profiling can improve performance by increasing the commit rate of transactional execution and by lowering the wasted execution cycles due to transactional aborts. The recommended first step for tuning is to use a profiling tool (see Section 12.4) to characterize the transactional behavior of the application. The profiling tool uses the performance monitoring and sampling capabilities implemented in the hardware to provide detailed information about the transactional behavior of the application. The tool uses capabilities provided by the processor such as performance monitoring counters and the Precise Event Based Sampling (PEBS) mechanism, see chapter 18 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Applications using an Intel TSX-enabled synchronization library should have the same functional behavior as if they were using a conventional synchronization library. However, because Intel TSX

changes latencies and can make cross-thread synchronization faster than before, latent bugs in the code may be exposed.

12.2.1 Existing TSX-enabled Locking Libraries

This section summarizes off-the-shelf locking libraries that are already TSX-enabled for lock elision. The list is non-exhaustive and represents a snap shot as of the first half of 2015. Not all libraries mentioned here may be completely tuned.

12.2.1.1 Libraries allowing lock elision for unmodified programs

- On Linux, GNU glibc 2.18 added support for lock elision of pthread mutexes of PTHREAD_MUTEX_DEFAULT type. Glibc 2.19 added support for elision of read/write mutexes. Whether elision is enabled. depends whether the --enable-lock-elision=yes parameter was set at compilation time of the library.
- Java JDK 8u20 or later support adaptive elision for synchronized sections when the -XX:+UseRTM-Locking option is enabled.
- Intel Composer XE 2013 SP1 or later supports lock elision for OpenMP omp_lock_t. Use “export KMP_LOCK_KIND=adaptive” to enable lock elision.

12.2.1.2 Libraries requiring program modifications

- Intel Thread Building Blocks (TBB) 4.2 supports elision with the speculative_spin_rw_mutex. The program needs to be modified to use this new lock type.
- gcc 4.8 and later supports TSX acceleration of its software transactional memory implementation.
- Concurrency Kit supports lock elision of spinlocks with its ck_elide wrappers.
- DPDK library supports lock elision of spin locks and read-write locks (through lock/unlock calls with “_tm” suffix).

12.2.2 Initial Checks

A couple of simple sanity checks can save tuning effort later on; specifically, using a good library implementation and dealing with statistics collection inside critical sections.

- Use a good Intel TSX enabled synchronization library. The application should directly be using the TSX-enabled synchronization library. When the application implements its own custom library built on top of an Intel TSX-enabled library, it still may be missing opportunities to identify transactional regions. See Section 3 on how to enable the synchronization library for Intel TSX.
- Avoid collecting statistics inside critical sections. Critical sections (and sometimes the synchronization library itself) may employ shared global statistics counters. Such counters will cause data conflicts and transactional aborts. Applications often have flags to disable such statistics collection. Disabling such statistics in the initial tuning phase will help focus on inherent data conflicts.

12.2.3 Run and Profile the Application

Visualizing synchronization-related thread interactions in multi-threaded applications is often difficult. The first step should be to run the application with an Intel TSX-enabled synchronization library and measure performance. Next, the profiling tool should be used to understand the result. First we should determine how much of the application is actually employing transactional execution, by using a profiling tool to measure the percentage of the application cycles spent in transactional execution (See Section 12.4).

Numerous causes may contribute to a low percentage of transactional execution cycles:

- The application may not be making noticeable use of critical-section based synchronization. In this case, lock elision is not going to provide benefits.
- The application's synchronization library may not use Intel TSX for all its primitives. This can occur if the application uses internal custom functions and libraries for some of the critical section locks. These lock implementations need to be identified and modified for elision (See Section 12.4.2).
- The application may be employing higher level locking constructs (referred to as meta-locks in this document) different from the one provided by the elision-enabled synchronization libraries. In these cases, the construct needs to be identified and enabled for elision (See Section 12.3.7)
- A program may be using LOCK-prefixed instructions for usages other than critical sections. TSX will not help with these typically, unless the algorithms are adapted to be transactional. Details on such non-locking usage are beyond the scope of this guide.

In the "bottom-up" approach of Intel TSX performance tuning, the methodology can be modularized into the following tasks:

- Identify all locks.
- Run the unmodified program with a TSX synchronization library eliding all locks.
- Use a profiling tool to measure transactional execution.
- Address causes of transactional aborts if necessary.

12.2.4 Minimize Transactional Aborts

Data conflicts are detected through the cache coherence protocol. Data conflicts cause transactional aborts. In the initial implementation, the thread that detects the data conflict will transactionally abort.

If an HLE-based transactional execution experiences a transactional abort, then in the current implementation, the hardware will restart at the XACQUIRE prefixed instruction that initiated HLE execution but will ignore the XACQUIRE prefix. This results in the re-execution without lock elision and the lock is explicitly acquired. If an RTM-based transactional execution experiences a transactional abort, then in the current implementation, the hardware will restart at the instruction address provided by the operation of the XBEGIN instruction.

The initial TSX implementation supports a limited form of nesting. RTM supports a nesting level of 7. HLE supports a nesting level of 1. This is an implementation specific number that may change in subsequent implementations of the same generation of processor families.

The Chapter 15 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* also describes the various causes for transactional aborts in detail. Details of Intel TSX instructions and prefixes can be found in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.*

The profiling tool can use performance monitoring to compute cycles that were spent in transactional execution that subsequently aborted. It is important to note that not all transactional aborts cause performance loss. The execution may otherwise have stalled due to waiting on a lock that had been acquired by another thread, and the transactional execution may also have a data prefetching effect.

The profiling tool can use PEBS to identify the top aborted transactional regions and provide information on the relative costs (see Section 12.4). We next discuss common causes for transactional aborts and provide mitigation strategies.

Tuning Suggestion 4. *Use a profiling tool to identify the transactional aborts that contribute most to any performance loss.*

The broad categories for transactional abort causes include:

- Aborts due to conflicting data accesses.
- Aborts due to conflicts on the lock variable.
- Aborts due to exceeding resource buffering.
- Aborts due to HLE interface specific constraints.
- Miscellaneous aborts as described in Chapter 8 of the *Intel® Architecture Instruction Set Extensions Programming Reference.*

12.2.4.1 Transactional Aborts due to Data Conflicts

A data conflict occurs if another logical processor either reads a location that is part of the transactional region's write-set or writes a location that is a part of either the read- or write-set of the transactional region. In the initial implementation, data conflicts are detected through the cache coherence protocol that operates at the granularity of a cache line.

We now discuss various sources of data conflicts that can cause transactional aborts. Some are avoidable while others are inherently present in the application.

Conflicts due to False Sharing

False sharing occurs when unrelated variables map to the same cache line (64 bytes) and are independently written by different threads. In this case, although the addresses of the unrelated variables do not overlap, since the hardware checks data conflicts at cache-line granularity, these unrelated variables appear to have the same address and this causes unnecessary transactional aborts.

Note that negative effects of false sharing are not unique to Intel TSX. The cache coherence protocol is moving the cache line around the system with high overhead. Good software practice already recommends against placing unrelated variables on the same cache line when at least one of the variables is frequently written by different threads.

Tuning Suggestion 5. *Add padding to put the two conflicting variables in separate cache line.*

Tuning Suggestion 6. *Reorganize the data structure to minimize false sharing whenever possible.*

Conflicts due to True Sharing

These transactional aborts occur if the conflict data is actually shared and is not due to false sharing. Sometimes such conflicts can also be mitigated through software changes. We discuss how to address some of these conflicts next.

Conflicts due to Statistics Maintenance

Software may often use global statistics counters shared among multiple threads. Examples of such use include synchronization libraries that count the number of times a critical section lock is either successfully acquired or was found to be held. Other examples include a count in a global variable or in an object that is accessed by multiple threads. Such statistics contribute to transactional aborts. In such cases, one must first try to understand the use of such statistics.

Sometimes these statistics can be disabled or conditionally skipped as they do not affect program logic. For example, such statistics may be measuring the frequency of serialized execution of a critical section. Without lock elision, the statistic is updated inside the critical section as the execution is already serialized. However, if the lock has been elided, then counting the number of times the lock has been elided isn't particularly useful. The only time it matters is if the lock was not elided; in those situations, the software can use the statistics to track the level of serialization. The XTEST instruction can be used to update the statistics only when the execution is not eliding a lock (i.e., serialized). Sometimes these statistics are only useful during program development and can be disabled in production software.

In some cases these statistics cannot be disabled or skipped. The programmer can avoid unnecessary transactional aborts by maintaining these statistics per logical thread (while taking care to avoid false sharing). Such an approach requires results to be aggregated across all threads when read. This can also improve the performance of applications even without Intel TSX instructions by minimizing communication among various threads.

Other approaches include moving the statistic outside critical sections and using an atomic operation to update the statistic. This will reduce transactional aborts but may add additional overhead due to an additional atomic operation and will not reduce the communication overhead.

Tuning Suggestion 7. *Global statistics may also be sampled rather than being updated for every operation.*

Tuning Suggestion 8. *Avoid unnecessary statistics in critical sections.*

Tuning Suggestion 9. *Consider maintaining statistics in critical sections on a per-thread basis.*

The programmer will have to determine the best approach for reducing transactional aborts due to shared global statistics. Disabling all global statistics during initial testing can help identify whether they are a problem.

Conflicts due to Accounting in Data Structures

Another common source of data conflicts are accounting operations in data structures. For example, data structures may maintain a variable to track the number of entries present at any time. This has the same effect as a statistics counter and can cause unnecessary transactional aborts.

In some usages, it is possible to move the accounting update to outside the critical section using atomic updates (e.g., the number of entries to trigger heap reorganization).

In other scenarios, approaches may be adopted to reduce the window of time where data conflicts may occur (see Section 12.2.4.1 on Reducing the Window for Data Conflict).

Conflicts in Memory Allocators

Some critical sections perform memory allocations. It is recommended to use a thread-friendly memory allocation library that maintains its free list in thread local space and avoid false sharing of the allocated memory.

Conflict Reduction through Conditional Writes

A common software pattern involves updates to a shared variable or flag that only infrequently changes value. Such an operation (even with the same value) causes an update to the cache line, which may in turn result in the processor requesting write-permissions to the cache line. Such an operation will cause transactional aborts in other threads that are also accessing the shared variable. Software can avoid such data conflicts by performing the update only when necessary - not performing the store if the value doesn't change, see Example 12-1.

Example 12-1. Reduce Data Conflict with Conditional Updates

state = true; // updates every time
var |= flag;
	if (state != true) state = true;
	if (!(var & flag)) var |= flag;

Reducing the Window for Data Conflict

Sometimes the techniques described are insufficient to avoid transactional aborts due to frequent real data conflicts. In such cases, the goal should be to reduce the window of time where a data conflict can occur. To reduce this probability, one may move the actual conflicting memory access towards the end of the critical section.

12.2.4.2 Transactional Aborts due to Limited Transactional Resources

While an Intel TSX implementation provides sufficient resources for executing common transactional regions, implementation constraints and excessive data footprint for transactional regions may cause a transactional abort. The architecture provides neither a guarantee of the resources available for transactional execution nor that a transactional execution will ever succeed.

The processor tracks both the **read-set** addresses and the **write-set** addresses in the first level data cache (L1 cache) of the processor.

An eviction of a read set address may not always result in an immediate transactional abort since these lines may be tracked in an implementation-specific second level structure. In current implementations, the second level structure tracks evicted read-set addresses probabilistically. As a result, accesses from other threads may at times result in a false positive match thus causing an unnecessary transactional abort. The rate of such false conflicts is a function of the address stream from different threads and the precise hardware implementation. The Broadwell microarchitecture implementation has an improved

second level structure. The rate of false conflicts is expected to reduce further with future implementations.

The architecture does not provide any guarantee for buffering and software must not assume any such guarantee.

With Haswell, Broadwell and Skylake microarchitectures, the L1 data cache has an associativity of 8. This means that in this implementation, a transactional execution that writes to 9 distinct locations mapping to the same cache set will abort. However, due to microarchitectural implementations, this does not mean that fewer accesses to the same set are guaranteed to never abort.

Additionally, in configurations with Intel Hyper-Threading Technology, the L1 cache is shared between the two logical processors on the same core, so operations in a sibling logical processor of the same core can cause evictions and significantly reduce the effective read and write set sizes.

Use the profiler to identify transactional regions that frequently abort due to capacity limitations (see Section 12.4.4). Software should avoid accessing excessive data within such transactional regions. Since, in general, accessing large amounts of data takes time, such aborts result in an excessive wasted execution cycles.

Sometimes, the data footprint of the critical section can be reduced by changing the algorithm. For example, for a sorted array, a binary instead of a linear search could be used to reduce the number of addresses accessed within the critical section.

If the algorithm expects certain code paths in the transactional region to access excessive data it may force an early transactional abort (through the XABORT instruction) or transition into a non-transactional execution without aborting by first acquiring the elided locks (see Section 12.2.6).

Sometimes, capacity aborts may occur due to side effects of actions inside a transactional region. For example, if an application invokes a dynamic library function for the first time the software system has to invoke the dynamic linker to resolve the symbols. If this first time happens inside a transactional region, it may result in excessive data being accessed, and thus will typically cause an abort. These types of aborts happen only the first time such a function is invoked. If this happens often, it is likely due to transactional only path not used in a non-transactional execution.

12.2.4.3 Lock Elision Specific Transactional Aborts

In addition to conflicts on data, transactional aborts may also occur due to conflicts on the lock itself. This is necessary to detect a transactional execution and a non-transactional execution of the critical section overlap in time. When implementing lock elision through Intel TSX, the implementation adds the lock to the read set - this occurs automatically for HLE but must be explicitly done in the software library when using RTM for lock elision. This allows checking conflicts with other threads that explicitly acquire the lock. This is a natural part of a transactional execution that aborts and re-starts and eventually acquires the lock.

For lock elision with HLE and RTM, many observed aborts occur due to such secondary conflicts on the lock variable: an aborting transactional thread transitions to a regular non-transactional execution, and as part of the transition also explicitly acquires the lock. This lock acquisition causes other transactionally executing threads to abort as they must serialize behind the thread that just acquired the lock.

For RTM, the fallback handler can potentially reduce these secondary aborts by waiting for the lock to be free before trying to acquire the lock (see Section 12.3.5).

12.2.4.4 HLE Specific Transactional Aborts

Some transactional aborts only occur in HLE-based lock elision. They are described in subsequent sections.

Unsupported Lock Elision Patterns

For the transactional execution to commit successfully, the lock must satisfy certain properties and access to the lock must follow certain guidelines. An XRELEASE-prefixed instruction must restore the value of the elided lock to the value it had before the corresponding XACQUIRE-prefixed lock acquisition.

This allows hardware to elide locks safely without adding them to the write-set. Both the data size and data address of the lock release (XRELEASE-prefixed) instruction must match that of the lock acquire (XACQUIRE-prefixed) and the lock must not cross a cache line boundary. For example, an XACQUIRE-prefixed lock acquire to an address A followed by an XRELEASE-prefixed lock release to a different address B will abort since the addresses A and B do not match.

Unsupported Access to Lock Variables inside HLE regions

Typically, a lock variable can be read from inside an HLE region without aborting. However, certain uncommon types of accesses may cause transactional aborts. For example, performing an unaligned access or a partially overlapping access to an elided lock variable will cause a transactional abort. Software should be changed to perform properly aligned accesses to the elided lock variable.

Software should not write to the elided lock inside a transactional HLE region with any instruction other than an XRELEASE prefixed instruction, otherwise it will cause a transactional abort.

12.2.4.5 Miscellaneous Transactional Aborts

Programmers can use any instruction safely inside a transactional region and can use transactional regions at any privilege level. However, some instructions will always abort the transactional execution and cause execution to seamlessly and safely transition to a non-transactional path. Such transactional aborts will appear as Instruction Aborts in the PEBS record transactional abort status collected by the profiling tool (see Section 12.4).

The Intel SDM presents a comprehensive list of such instructions. Common examples include instructions that operate on the X87 and MMX architecture state, operations that update segment, control, and debug registers, IO instructions, and instructions that cause ring transitions, such as SYSENTER, SYSCALL, SYSEXIT, and SYSRET.

Programmers should use SSE/AVX instructions instead of X87/MMX instructions inside transactional regions. However, programmers must be careful when inter-mixing SSE and AVX operations inside a transactional region. Intermixing SSE instructions accessing XMM registers and AVX instructions accessing YMM registers may cause transactional regions to abort. The VZEROUPPER instruction may also cause an abort, and programmers should try to move the instruction to prior to the critical section.

Certain 32-bit calling conventions may use X87 state to pass or return arguments. Programmers should consider alternate calling conventions or inline the functions. Some types such as long double may use X87 instructions and should be avoided.

In addition to the instruction-based considerations, various runtime events may cause transactional execution to abort.

Asynchronous events (NMI, SMI, INTR, IPI, PMI, etc.) occurring during transactional execution may cause the transactional execution to abort and transition to a non-transactional execution. The rate of such aborts depends on the background state of the operating system. For example, operating systems with timer ticks generate interrupts that can cause transactional aborts.

Synchronous exception events (#BR, #PF, #DB, #BP/INT3, etc.) that occur during transactional execution may cause an execution not to commit transactionally, and require a non-transactional execution. These events are suppressed as if they had never occurred.

Page faults (#PF) typically occur most when a program starts up. Transactional regions will experience aborts at a higher rate during this period since pages are being mapped for the first time. These aborts will disappear as the program reaches a steady state behavior. However, for programs with very short run times, these aborts may appear to dominate. A similar behavior happens when large regions of memory were allocated in the recent past.

Memory accesses within a transactional region may require the processor to set the Accessed and Dirty flags of the referenced page table entry. These actions occur on the first access and write to the page, respectively. These operations will cause a transactional abort in the current implementation. A re-execution in non-transactional mode will cause these bits to be appropriately updated and subsequent transactional executions will typically not observe these transactional aborts. Although these transac-

tional aborts will show up as Instruction Aborts in the PEBS record transactional abort status, special attention isn't needed unless they occur frequently.

In addition to the above, implementation-specific conditions and background system activity may cause transactional aborts. Examples include aborts as a result of the caching hierarchy of the system, subtle interactions with processor micro-architecture implementations, and interrupts from system timers among others. Aborts due to such activity are expected to be fairly infrequent for typical Intel TSX usage for lock elision.

Tuning Suggestion 10. *Transactional regions during program startup may observe a higher abort rate than during steady state.*

Tuning Suggestion 11. *Operating system services may cause infrequent transactional aborts due to background activity.*

12.2.5 Using Transactional-Only Code Paths

With Intel TSX, programmers can write code that is only ever executed in a transactional region and the non-transactional fallback path may be different. This is possible with RTM (through the use of the fallback handler) and with HLE in conjunction with the XTEST instruction.

Care is required if the code executed during transactional execution is significantly different than the code executed when not in transactional execution. Certain events such as page faults (instruction and data) and operations on pages that modify the accessed and dirty bits may repeatedly abort a transactional execution. Thus programmers must ensure such operations are also performed in a non-transactional fallback path, otherwise the transactional region may never succeed. This is not a problem in general since with lock elision the transactional path and non-transactional path in the application is the same and the only differences are captured in the synchronization libraries.

The XTEST instruction can be used to skip over code sequences that are unnecessary during transactional execution and likely to lead to aborts. The XTEST instruction can also be used to implement optimizations such as skipping unwind code and other error handling code (such as deadlock detection) that is only required if the lock is actually acquired.

Tuning Suggestion 12. *Keep any transactional only code paths simple and inlined.*

Tuning Suggestion 13. *Minimize code paths that are only executed transactionally.*

12.2.6 Dealing with Transactional Regions or Paths that Abort at a High Rate

Some transactional regions abort at a high rate and the methods discussed so far are not effective in reducing the aborts. In such cases, the following options may be considered.

12.2.6.1 Transitioning to Non-Elided Execution without Aborting

Sometimes, a transactional abort is unavoidable. Examples include system calls, and IO operations. When these are required on a transactional code path, software using RTM for lock elision can transition to a non-elided execution by attempting to acquire the lock and if successful committing the transactional execution. A simplified example is shown in Example 12-2. The actual code may need to handle nesting, etc.

Example 12-2. Transition from Non-Elided Execution without Aborting

```
/* ... in RTM transaction, but the transactional execution will abort */
/* Acquire the lock without elision */

<original lock acquire code>
_xend();  /* Commit */

/* Do aborting operation */
```

12.2.6.2 Forcing an Early Abort

Programmers should try to insert a PAUSE or XABORT instruction early in paths that lead to aborts inside transactional regions. This will force a transactional abort early and minimize work that needs to be discarded.

12.2.6.3 Not Eliding Selected Locks

Sometimes if the application performance is lower with lock elision and the transactional abort reduction techniques have been exhausted, software can disable elision for the specific locks that have high and expensive transactional abort rates. This should always be validated with application level performance metrics, as even high abort rates may still result in a performance improvement.

12.3 DEVELOPING AN INTEL TSX ENABLED SYNCHRONIZATION LIBRARY

This section describes how to enable a synchronization library for lock elision using the Intel TSX instructions.

12.3.1 Adding HLE Prefixes

The programmer uses the XACQUIRE prefix in front of the instruction that is used to acquire the lock that is protecting the critical section. The programmer uses the XRELEASE prefix in front of the instruction that is used to release the lock protecting the critical section. This instruction will be a write to the lock. If the instruction is restoring the value of the lock to the value it had prior to the XACQUIRE prefixed lock acquire operation on the same lock, then the processor elides the external write request associated with the release of the lock, enabling concurrency in the absence of data conflicts.

12.3.2 Elision Friendly Critical Section Locks

The library itself shouldn't be a source of data conflicts. Common examples of such problems include:

- Conflicts on the lock owner field.
- Conflicts on lock-related statistics.

When using HLE for lock elision, programmers must add the elision capability to the existing code path (since the code path executed with and without elision is the same with HLE). The programmer should also check that the only write operation to a shared location is through the lock-acquire/lock-release instructions on the lock variable. Any other write operation to a shared location would typically manifest itself as a data conflict among two threads using the elision library to elide a common lock. A test running multiple threads looping through an empty critical section protected by a shared lock can quickly identify such situations.

12.3.3 Using HLE or RTM for Lock Elision

Software can use the CPUID information to determine whether the processor supports the HLE and RTM extensions. However, software can use the HLE prefixes (XACQUIRE and XRELEASE) without checking whether the processor supports HLE. Processors without HLE support ignore these prefixes and will execute the code without entering transactional execution. In contrast, software must check if the processor supports RTM before it uses the RTM instructions (XBEGIN, XEND, XABORT). These instructions will generate a #UD exception when used on a processor that does not support RTM. The XTEST instruction also requires a CPUID check to ensure either HLE or RTM is supported, else it will also generate a #UD exception. The CPUID information may be cached in some variable to avoid checking for CPUID repeatedly.

With HLE, if the eliding processor itself reads the value of the lock in the critical section, the value returned will appear as if the processor had acquired the lock; the read will return the non-elided value. This behavior makes an HLE execution functionally equivalent to an execution without the HLE prefixes.

The RTM interface allows programmers to write more complex synchronization algorithms and to control the retry policies following transactional aborts. The preferred way is to use the RTM-based locking implementation as a wrapper with multiple code paths within; one path exercising the RTM-based lock and the other exercising the non-RTM based lock (See Section 12.3.4). This typically does not require changes to the non-RTM based lock code. Performance may further be improved by using a try-once primitive, which allows the thread to re-attempt lock elision after the lock becomes free.

Since the RTM instructions do not have any explicit lock associated with the instructions, software using these instructions for lock elision must test the lock within the transactional region, and only if free should it continue executing transactionally. Further, the software may also define a policy to retry if the lock is not free.

In a subtle difference with HLE, if the code within the RTM-based critical section reads the lock, it will appear as if it is free and not acquired. So library functions used to return the value of locks must abort the transactional execution and return the value when executed non-transactionally (See Section 12.3.9). This situation does not exist with HLE because the HLE instructions have an explicit lock address associated with them and the hardware ensures the right value is returned.

User/Source Coding Rule 36. *When using RTM for implementing lock elision, always test for lock inside the transactional region.*

Tuning Suggestion 14. *Don't use an RTM wrapper if the lock variable is not readable in the wrapper.*

12.3.4 An example wrapper for lock elision using RTM

This section describes how to write a wrapper to implement lock elision using RTM instructions. The idea is to take the conventional lock implementation (without elision), add a wrapper around it, and then add a new path within the wrapper to implement elision. Thus, the wrapper provides separate code paths for the elided path and the non-elided paths. The non-elided lock-acquire path is executed only if the elided path was unsuccessful. Further, such an approach allows the non-elided path to remain unchanged. Such an approach works well for wide variety of locks, including ticket locks and read-write locks.

An example code sequence is shown in Example 12-3 (See Section 12.7 for a description of the intrinsics used).

Example 12-3. Exemplary Wrapper Using RTM for Lock/Unlock Primitives

```
void rtm_wrapped_lock(lock) {
  if (_xbegin() == _XBEGIN_STARTED) {
    if (lock is free)
      /* add lock to the read-set */
      return; /* Execute transactionally */
    _xabort(0xff);
    /* 0xff means the lock was not free */
  }
  /* come here following the transactional abort */
  original_locking_code(lock);
}
```

```
void rtm_wrapped_unlock(lock) {
  /* If lock is free, assume that the lock was elided */
  if (lock is free)
    _xend();   /* commit */
  else
    original_unlocking_code(lock);
}
```

In Example 12-3, _xabort() terminates the transactional execution if the lock was not free. One can use _xend() to achieve the same effect. However, the profiling tool can easily recognize the _xabort() operation along with the 0xff abort code (which is a software convention) and determine that this is the case where the lock was not available. If the _xend() were used, the profiling tool would be unable to distinguish this case from the case where a lock was successfully elided.

The example above is a simplified version showing a basic policy of retrying only once and not distinguishing between various causes for transactional aborts. A more sophisticated implementation may add heuristics to determine whether to try elision on a per-lock basis based on information about the causes of transactional aborts. It may also have code to switch back to re-attempting lock elision after blocking if the lock was not free. This may require small changes to the underlying synchronization library.

Sometimes programming errors can lead to a thread releasing a lock that is already free. This error may not manifest itself immediately. However, when such a lock release function is replaced with an RTM-enabled library using the wrapper described above, an XEND instruction will execute outside a transactional region. In this case, the hardware will signal a #GP exception. It is generally a good idea to fix the error in the original application. Alternatively, if the software wants to retain the original erroneous code path, then a XTEST can be used to guard the XEND.

12.3.5 Guidelines for the RTM fallback handler

The fallback handler for RTM provides the code path that is executed if the RTM-based transactional execution is unsuccessful. Since the Intel TSX architecture specification does not provide any guarantee that a transactional execution will ever succeed, the RTM fallback handler must have the capability to ensure forward progress; it should not simply keep retrying the transactional execution.

Tuning Suggestion 15. *When RTM is used for lock elision, forward progress is easily ensured by acquiring the lock.*

If the fallback handler explicitly acquires the lock, then all other transactionally executing threads eliding the same lock will abort and the execution serializes on the lock. This is achieved by ensuring that the lock is in the transactional region's read-set.

Software can use the abort information provided in the EAX register to develop heuristics as to when to retry the transactional execution and when to fallback and explicitly acquire the lock. For example, if the _XABORT_RETRY bit is clear, then retrying the transactional execution is likely to result in another abort. The fallback handler should distinguish this situation from cases where the lock was not free (for example, the _XABORT_EXPLICIT bit is set but the _XABORT_CODE()[1] returns a 0xff identifying the condition as a "lock busy" condition). In those cases, the fallback handler should eventually retry after waiting.

Performance may also be improved by retrying (after a delay) if the abort cause was a data conflict (_XABORT_CONFLICT) because such conditions are often transient. Such retries however should be limited and must not continually retry.

A very small number of retries for capacity aborts (_XABORT_CAPACITY) can be beneficial on configurations with Hyper Threading enabled. The L1 cache is a shared resource between HT threads and one thread may push data out of the other. On retry there is a reasonable chance to succeed. This requires ignoring the _XABORT_RETRY bit in the status code for this case. The _XABORT_RETRY bit should not be ignored for any other reason.g

Generally on higher core count and multi-socket systems the number of retries should be increased.

In general, if the lock was not free, then the fallback handler should wait until the lock is free prior to retrying the transactional execution. This helps to avoid situations where the execution may persistently stay in a non-transactional execution without lock elision. This can happen because the fallback handler never had an opportunity to try a transactional execution while the lock was free (See Section 12.3.8).

User/Source Coding Rule 37. *RTM abort handlers must provide a valid tested non transactional fallback path.*

Tuning Suggestion 16. *Lock Busy retries should wait for the lock to become free again.*

12.3.6 Implementing Elision-Friendly Locks using Intel TSX

This section discusses strategies for implementing elision friendly versions of common locking algorithms using the Intel TSX instructions. Similar approaches can be adopted for algorithms not covered in this section.

12.3.6.1 Implementing a Simple Spinlock using HLE

A spinlock is a simple yet very common locking algorithm. In this algorithm, a thread first checks to see if the lock is free and then attempts to acquire the lock through a LOCK-prefixed instruction. If not, the thread spins (using a read operation that typically completes from the local data cache holding the lock value) on the lock waiting for it to become free.

For this example, assume the lock is free when its value is zero, and held by some thread otherwise. The lock is released through a regular store instruction.

Example 12-4 uses the gcc 4.8+ **atomic intrinsics** which are similar to the C11 standard. The description here follows the recommended approach to implement a spin lock using gcc 4.8+ intrinsics. To enable HLE for this spin lock, the only change required would be the addition of the __ATOMIC_HLE_ACQUIRE and __ATOMIC_HLE_RELEASE flags. The rest of the code is the same as without using HLE.

1. _XABORT_CODE accesses the xabort status in the RTM abort code

Example 12-4. Spin Lock Example Using HLE in GCC 4.8 and Later

```
#include <immintrin.h> /* For _mm_pause() */
/* Lock initialized with 0 initially */
void hle_spin_lock(int *lock)
{
   while (__atomic_exchange_n(lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE) != 0)
   { int val;
      /* Wait for lock to become free again before retrying. */
      do {
         _mm_pause();  /* Abort speculation */
         __atomic_load_n(lock,  &val, __ATOMIC_CONSUME);
      } while (val == 1);
   }
}
```

```
void hle_spin_unlock(int *lock)
{
   __atomic_clear(lock, __ATOMIC_RELEASE|__ATOMIC_HLE_RELEASE);
}
```

The following shows the same example using intrinsics for the Windows C/C++ compilers (Microsoft Visual Studio 2012 and Intel C++ Compiler XE 13.0).

Example 12-5. Spin Lock Example Using HLE in Intel and Microsoft Compiler Intrinsic

```
#include <intrin.h> /* For _mm_pause() */
#include <imminitrin.h> /* For HLE intrinsics */
/* Lock initialized with 0 initially */
void hle_spin_lock(int *lock)
{
while (_InterlockedCompareExchange_HLEAcquire(&lock, 1, 0) != 0){
          /* Wait for lock to become free again before retrying speculation */
         do {
               _mm_pause();  /* Abort speculation */
               /* prevent compiler instruction reordering and wait-loop skipping,
                 no additional fence instructions are generated on IA */
               _ReadWriteBarrier();
         } while (lock == 1);
}
}
```

```
void hle_spin_unlock(int *lock)
{
_Store_HLERelease (lock, 0);
}
```

See Section 12.7 for an assembler implementation of an HLE spinlock.

12.3.6.2 Implementing Reader-Writer Locks using Intel TSX

Reader-Writer locks are common where the critical sections are mostly read-only. Such locks can avoid serializing access to the critical section for readers; however, they still require an atomic operation on a shared location (often through a LOCK prefixed XADD or CMPXCHG) and require communication among the multiple readers. Note that lock elision essentially makes all locks behave as reader-writer locks - except that, with lock elision readers and non-conflicting writers can proceed concurrently without communication.

RTM can be used to elide reader-writer locks through a wrapper approach as discussed earlier. The only difference being that, with reader-writer locks, the lock algorithm normally checks both the reader and the writer states to determine that the lock is free. When it is possible to place the reader and writer locking state on different cache lines, it is also possible to let transactional and non-transactional readers execute in parallel. The readers only need to check the writer state being free.

With HLE, the code path for the elided version and non-elided version should remain the same. Some reader-writer lock implementations use a lock to protect the reader/writer state instead of the actual critical section. In this case, the lock first needs to be changed to have a fast path with a single atomic operation. Beyond this, the path should not change the cache line with the lock variable. This can be done by combining the reader and writer counts into a single field, and then checking/updating it atomically with a LOCK- prefixed XADD or CMPXCHG instruction for the lock acquire and lock release functions. The HLE prefixes - XACQUIRE and XRELEASE - are placed on these LOCK-prefixed operations. Interestingly, this approach also improves the performance of reader-writer locks even without using Intel TSX. Alternatively, using an RTM wrapper can avoid changing lock structure since you can have different lock acquire paths for elided and non-elided versions in the synchronization library.

Tuning Suggestion 17. *For Read/Write locks elide the complete lock operation, not the building block locks.*

12.3.6.3 Implementing Ticket Locks using Intel TSX

Ticket locks are another common algorithm. A ticket lock is a variant of a spinlock where instead of spinning on a shared location and then racing to acquire the lock when the lock is free, threads use tickets to determine which thread can enter the critical section.

RTM can be used to elide ticket locks through a wrapper approach as discussed earlier (See Section 12.3.4).

Some ticket lock implementations assume an increasing ticket value and such locks do not meet HLE's requirement that the value of the lock following the lock release be the same as the value prior to the lock acquire.

Tuning Suggestion 18. *Use RTM to elide ticket locks.*

12.3.6.4 Implementing Queue-Based Locks using Intel TSX

In general, the idea of lock elision requires multiple threads to concurrently enter and try to commit a common critical section. The idea of fair locks requires threads to enter and release the critical section in a first-come first-served order. The two ideas may sometimes appear at odds, but the general objective is usually more flexible.

Queue-based locks are a form of fair locks where the threads construct a queue of lock requests. This includes different forms of ticket locks.

In some implementations the queue is formed through an initial LOCK-prefixed operation. For such implementations, the HLE XACQUIRE prefix can be added to this operation to enable lock elision. In the absence of any transactional aborts, the queue remains empty following the lock release. However, if a transactional abort occurs and the aborting thread acquires the lock explicitly (thus forming a queue), subsequent threads will add themselves to the queue, and when the lock is released, only a single thread will attempt lock elision as the other threads are not at the front of the queue. Further, if another thread arrives and adds itself to the queue, this may cause the transactionally executing thread to abort, and the execution remains in a non-eliding phase until the queue is drained.

This scenario only occurs with lock implementations that attempt lock elision as part of the queuing process. It does not apply to implementations that construct a queue only after an initial atomic operation, like an adaptive spinning-sleeping lock that elides the spinning phase but only queues for waiting after initial spinning failed. Such a problem also doesn't exist for implementations that use wrappers (such as those using RTM). In these implementations, the thread does not attempt lock elision as part of the queuing process.

Tuning Suggestion 19. *Use an RTM wrapper for locks that implement queuing as part of the initial atomic operation.*

12.3.7 Eliding Application-Specific Meta-Locks using Intel TSX

Some applications build their own locks, called meta-locks, using an underlying synchronization library. In this approach, the application uses a lock from the underlying synchronization library to protect the data of the meta-lock. It then updates the data and releases the lock. If you recall, a similar approach was taken for the reader-writer lock implementation discussed in Section 12.3.6.2.

The application executes the critical section while holding the meta-lock, and then uses a lock from the underlying synchronization library to protect the meta-lock while it is being released. In this sequence, eliding the lock from the underlying synchronization library isn't useful; the goal should be to elide the meta-lock itself and transactionally execute the application code itself instead of the code in the synchronization library. A profiling tool can be used to identify such critical sections. An RTM wrapper (similar to one discussed in Section 12.3.4) can be used to avoid the meta-lock during lock elision.

For illustration, assume the following as an example of a meta-lock implementation.

Example 12-6. A Meta Lock Example

```
void meta_lock(Metalock *metalock) {
  __lock(metalock->lock);
  /* modify meta lock state for lock */
    unlock(metalock->lock);
}
```

```
void meta_unlock(Metalock *metalock) {
  lock(metalock->lock);
  /* drop metalock state */
  unlock(metalock->lock);
}
```

```
meta_lock(metalock);
  /* critical section */
meta_unlock(metalock);
```

The above example can be transformed into the following code.

Example 12-7. A Meta Lock Example Using RTM

```
void rtm_meta_lock(Metalock *metalock) {
    if (_xbegin() == _XBEGIN_STARTED)
        if (meta_state_is_all_free(metalock))
          return;
        _xabort(0xff);
    }
    meta_lock(metalock);
}
void rtm_meta_unlock(Metalock *metalock) {
    if (meta_state_is_all_free(metalock))
      _xend();
    else
        meta_unlock(metalock);
}
```

```
rtm_meta_lock(metalock);
/* critical section */
rtm_meta_unlock(metalock);
```

Tuning Suggestion 20. *For meta-locking elide the full outer lock, not the building block locks.*

12.3.8 Avoiding Persistent Non-Elided Execution

A transactional abort eventually results in execution transitioning to a non-transactional state without lock elision. This ensures forward progress. However, under certain conditions and with some lock acquire algorithms, threads may remain in a persistent non-transactional execution without attempting lock elision for an extended duration. This will limit performance opportunities.

To understand such situations, consider the following example with a simple spin lock implementation using HLE (a similar scenario can also exist with RTM). The lock value of zero means the lock is free and a value of one means it is acquired by some thread.

The HLE-enabled lock-acquire sequence can be written as shown in Example 12-8.

Example 12-8. HLE-enabled Lock-Acquire/ Lock-Release Sequence

```
  mov eax,$1
Retry:
  XACQUIRE; xchg LockWord,eax
  cmp eax,$0   # Was zero so lock was acquired successfully
  jz Locked
SpinWait:
  cmp LockWord, $1
  jz SpinWait# Still one
  jmp Retry# It's free, try to claim
Locked:
```

Example 12-8. HLE-enabled Lock-Acquire/ Lock-Release Sequence (Contd.)

```
XRELEASE; mov LockWord,$0
```

If a thread is unable to perform lock elision, then it acquires the lock without elision. Assume another thread arrives to acquire the lock. It executes the "XACQUIRE; xchg lockWord, eax" instruction, elides the lock operation on the lock, and enters transactional execution. However the lock at this point was held by another thread causing this thread to enter the SpinWait loop while still executing transactionally. This spin occurs during transactional execution because hardware does not have the notion of a critical section lock - it only sees the instruction to implement the atomic operation on the lock variable. The hardware doesn't have the semantic knowledge that the lock was not free.

Now, if the thread that held the lock releases it, the write operation to the lock will cause the current transactional thread spinning on the location to transactionally abort (because of the conflict between the lock release operation and the read loop of the lock by the transactional thread). Once it has aborted, the thread will restart execution without lock elision. It is easy to see how this extends to all other threads - they spin transactionally but end up executing non-transactionally and without lock elision when they actually find the lock free. This will continue until no other threads are trying to acquire the lock. The threads have thus entered a persistent non-elided execution.

A simple fix for this includes using the pause instruction (which causes an abort) as part of the spin-wait loop. This is also the recommended approach to waiting on a lock to be released, even without Intel TSX. The pause instruction will force the spin-wait loop to occur non-transactionally, thus allowing the threads to try lock elision when the lock is released.

Example 12-9. A Spin Wait Example Using HLE

```
        mov eax,$1
Retry:
        XACQUIRE; xchg LockWord,eax
        cmp eax,$0# Was zero so we got it
        jz Locked
SpinWait:
        pause
cmp LockWord, $1
        jz SpinWait# Still one
        jmp Retry# It's free, try to claim
Locked:
```

Tuning Suggestion 21. *Always include a pause instruction in the wait loop of a HLE spinlock.*

12.3.9 Reading the Value of an Elided Lock in RTM-based libraries

Some synchronization libraries provide interfaces that read the value of a lock. Libraries implementing lock elision using RTM may be unable to reliably determine if the lock variable has been acquired by the thread performing the elision since the lock was only read but not written to inside the library.

Sometimes the library interface may be as simple as a test to check whether a lock is acquired thus providing a sanity check to the software. To ensure the correct value is provided to the function using an RTM-based library, the transactional execution must be aborted and the lock explicitly acquired. This can be achieved by forcing an abort through the XABORT instruction (using _xabort(0xfe)). The 0xfe code

can be used by the fallback handler to determine this situation and aid in optimizations in eliminating such a read. Alternatively, the _xtest() intrinsic can be used avoid unnecessary transactional aborts:

```
assert(is_locked(my_lock)) => assert(_xtest() || is_locked(my_lock))
```

A better primitive for an elided synchronization library would combine both - the lock being acquired or a lock elision in progress. For example:

```
bool is_atomic(lock) { return _xtest() || is_locked(lock); }
```

At other times, the lock variable may be read as part of a function with assumptions about behavior. An example is the **try-lock** interface to acquire a lock where a thread makes a single attempt to acquire the lock and returns a value indicating whether the lock was free or not. This is in contrast to a spin lock that continues to spin trying to acquire the lock. In general, this isn't a problem. But sometimes, software may make implicit assumptions about the actual value returned by a nested try-lock. With an RTM-based implementation, the value returned will be that of a free lock since the lock was elided. If software is making such implicit assumptions about the value, then the synchronization library can force a transactional abort through the XABORT instruction (using _xabort(0xfd)). This will however cause unnecessary aborts in some programs. Such implicit programming assumptions are not recommended. As such implicit programming assumptions are rare, it is recommended to not abort in the synchronization library in trylock.

12.3.10 Intermixing HLE and RTM

HLE and RTM provide two alternative software interfaces to a common transactional execution capability. The behavior when HLE and RTM are nested together-HLE inside RTM or RTM inside HLE-is implementation specific. For the first implementation of the 4th generation Intel Core Processor, intermixing causes a transactional abort. This behavior may change in subsequent processor implementations but the semantics of a transactional commit will be maintained.

In general, applications should avoid intermixing HLE and RTM as they are essentially achieving the end purpose of lock elision but through different software interfaces. However, library functions implementing lock elision may be unaware of the calling function and whether the calling function is invoking the library function while eliding locks using RTM or HLE.

Software can handle such conditions by using the _xtest() operation. For example, the library may check if it was invoked within a transactional region and if the lock is free. If the call was within a transactional region, the library may avoid starting a new transactional region. If the lock was not free, the library may return an indication through the _xabort(0xff) function. This does require the function that will be invoked on a release to recognize that the acquire operation was skipped.

Example 12-10 shows a conceptual sequence.

Example 12-10. A Conceptual Example of Intermixed HLE and RTM

```
// Lock Acquire sequence
// Use a function local or per-thread location
bool lock_in_transactional_region = false;
if (_xtest() && my lock is free) { /* Already in a transactional region*/
  lock_in_transactional_region = true;
} else {
  // acquire lock if free, else abort
}

// the lock release sequence
if (!lock_in_transactional_region) {
    // release lock
}
```

12.4 USING THE PERFORMANCE MONITORING SUPPORT FOR INTEL TSX

Application tuning using Intel TSX relies on performance counter-based profiling to understand transactional execution behavior and the causes of transactional aborts. Achieving good performance with Intel TSX often requires some tuning based on data from a profiling tool to minimize aborts. Using the performance counters is often preferable to instrumenting the application as it is usually less intrusive and easier. Section 18.10.5 and Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B* provides information about the various performance monitoring events supported in the current implementation.

In general, profiling can impact transactional execution as any profiling tool generates periodic interrupts to collect information, and the interrupt will cause a transactional abort. Hence, any profiling should try to minimize the impact of this in analysis. This is not an issue if one is profiling only transactional aborts.

Program startup tends to have a large number of events that occur only once. When profiling complex programs, skipping over the startup phase can significantly reduce any noise introduced by these events.

Profilers that support TSX tuning include Linux perf, Intel Performance Counter Monitor, and Intel VTune. See http://www.intel.com/software/tsx for references.

12.4.1 Measuring Transactional Success

The first step should be to measure the transactional success in an application. This is done with the Unhalted_Core_Cycles event programmed in three separate configurations with three counters:

1. Use the fixed cycles counter (IA32_FIXED_CTR0) to measure FixedCyclesCounter.
2. Configure IA32_PERFEVTSEL2 with the IN_TX and IN_TXCP filters set to measure CyclesInTxCP in IA32_PMC2.
3. Configure another MSR IA32_PERFEVTSELx (x= 0, 1, 3) with IN_TX filter to measure CyclesInTXOnly on the corresponding counter.

These cycle measurements should be set up to count and not sample frequently; sampling may cause additional transactional aborts. With these three values the total cycles, cycles spent in transactional execution, and cycles spent in transactional regions that eventually aborted can be computed:

CyclesTotal = FixedCycleCounter
%CyclesTransactionalAborted = ((CyclesInTxOnly - CyclesInTxCP) / CyclesTotal) * 100.0
%CyclesTransactional = (CyclesInTx / CyclesTotal) * 100.0
%CyclesNonTransactional = 100.0 - %CyclesTransactional

If CyclesTransactional is near zero then the application is either not using lock-based synchronization or not using a synchronization library enabled for lock elision through the Intel TSX instructions. In the latter case, the programmer should use an Intel TSX-enabled synchronization library (See Section 12.3).

If CyclesTransactionalAborted is small relative to CyclesTransactional, then the transactional success rate is high and additional tuning is not required.

If the CyclesTransactionalAborted is almost the same as CyclesTransactional (but not very small), then most transactional regions are aborting and lock elision is not going to be beneficial. The next step would be to identify the causes for transactional aborts and reduce them (See Section 12.2.4).

12.4.2 Finding locks to elide and verifying all locks are elided.

This step is useful if the cycles spent in transactional execution is low. This may be because few locks are being elided. The MEM_UOPS_RETIRED.LOCK_LOADS event should be counted and compared to the RTM_RETIRED.START or HLE_RETIRED.START events. If the number of lock loads is significantly higher than the number of transactional regions started, then one can usually assume that not all locks are marked for lock elision. The PEBS version of MEM_UOPS_RETIRED.LOCK_LOADS can be sampled to identify the missing locks. However, this technique isn't effective in immediately detecting missed opportunities with meta-locking (See Section 12.3.7). Additionally, a profile on the call graph of the

MEM_UOPS_RETIRED.LOCK_LOADS event often identifies the high level synchronization library that needs to be TSX-enabled to allow transactional execution of the application level critical sections.

12.4.3 Sampling Transactional Aborts

The hardware implementation defines PEBS precise events to sample transactional aborts - HLE_RETIRED.ABORTED for HLE and RTM_RETIRED.ABORTED for RTM. This allows programmers to perform precise profiling of all transactional aborts in the execution. The test should be run with PEBS enabled and sampled to identify the code location where the transactional aborts are occurring. The PEBS handler (a part of the profiling tool) uses the EventingIP field in the PEBS record to report the correct code location of the transactional aborts.

As a next step, the most common transactional aborts should be examined and addressed. Sampling transactional aborts does not cause any additional aborts.

12.4.4 Classifying Aborts using a Profiling Tool

The PEBS record generated as a result of profiling transactional aborts contains additional information on the cause of the transactional abort in the TX Abort Information field. The lower 32 bits of the TX Abort Information, called Cycles_Last_TX, also provides the cycles spent in the last transactional region prior to the abort. This approximately captures the cost of a transactional abort.

RelativeCostOfAbortForIP = SUM(Cycles_Last_TX_For_IP)

Not all transactional aborts are equal - some don't contribute to performance degradation while the more expensive ones can have significant impact. The programmer can use this information to decide which transactional aborts to focus on first.

For more details on the PEBS record see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B* Section 18.10.5.1

The profiling tool should display the abort cost to the user to classify the abort.

Tuning Suggestion 22. *The aborts with the highest cost should be examined first.*

Tuning Suggestion 23. *The TX Abort Information has additional information about the transactional abort.*

If the PEBS record **Instruction_Abort** bit (bit 34) is set, then the cause of the transactional abort can be directly associated with an instruction. For these aborts, the PEBS record captures the instruction address that was the source of the transactional abort. Exceptions, like page faults (including those that would normally terminate the program and those that fault in the working set of the program at startup) also show up as in this category.

If the PEBS record **Non_Instruction_Abort** bit (bit 35) is set, then the abort may not have been caused by the instruction reported by the instruction address in the PEBS record. An example of such an abort is one due to a data conflict with other threads. In this case, the **Data_Conflict** bit (bit 37) is also set. Another example is when transactional aborts occur due to capacity limitations for transactional write- and read-sets. This is captured by the Capacity_Write (bit 38) and the Capacity_Read (bit 39) fields.

Aborts due to data conflicts may occur at arbitrary instructions within the transactional region. Hence it is useful to concentrate on conflict causes in the whole critical section. Instead of relying on the **EventingIP** reported by PEBS for the abort, one should focus on the return IP (IP of the abort code) in conjunction with the call graphs. The return IP typically points into the synchronization library, unless the lock is inlined. The caller identifies the critical section.

For capacity it can be also useful to concentrate on the whole critical section (profiling for ReturnIP) as the whole critical section needs to be changed to access less memory.

Tuning Suggestion 24. *Instruction aborts should be analyzed early, but only when they are costly and happen after program startup.*

Tuning Suggestion 25. *For data conflicts or capacity aborts, concentrate on the whole critical section, not just the instruction address reported at the time of the abort.*

Tuning Suggestion 26. *The profiler should support displaying the ReturnIP with callgraph for non-Instruction abort events, but display the EventingRIP for instruction abort events.*

Tuning Suggestion 27. *The PEBS TX Abort Information bits should be all displayed by the profiling tool.*

12.4.5 XABORT Arguments for RTM fallback handlers

If the XABORT instruction is used to abort an RTM-based transactional region, the instruction operand is passed to the fallback handler through the EAX register. This information is also provided by the PEBS-based profiling tool for RTM. A profiling tool can use this information to classify various XABORT-based transactional aborts. Defining a convention can be also helpful to write sophisticated fallback handlers. The following table presents the convention used in this document:

Table 12-1. RTM Abort Status Definition

XABORT Code	Description
0xff	XABORT-based abort because lock was not free when tested (Section 12.3.4)
0xfe	XABORT-based abort because lock tested for the value of the elided lock (Section 12.3.9)
0xfd	XABORT-based abort during a nested try lock (Section 12.3.9)
0xfc: 0xf0	Reserved

Tuning Suggestion 28. *The profiling tool should display the abort code to the user for RTM aborts.*

12.4.6 Call Graphs for Transactional Aborts

The profiling tool generates interrupts to collect performance monitoring information. Such interrupts will cause transactional aborts. This means a profiling tool can only collect information after a transactional abort happened and the tool cannot see any function calls on the stack that only happened inside the transactional region; the only view of the call graph it has was the one at the beginning of the transactional execution. When a transactional abort is sampled with PEBS the RIP field contains the instruction pointer after the abort and the EventingIP field contains the instruction pointer within the transactional region at the time of the abort. The same also applies for sampling non-abort events, as any sampling causes transactional aborts.

Depending on the type of abort, it can be useful to profile for either ReturnIP or EventingIP. The stack callgraph collected by the profiling tool is always associated with the ReturnIP. When it is combined with the EventingIP, it may appear noncontiguous (the EventingIP may not be associated with the lowest level caller), as any function calls inside the transactional region are not included. When the function calls inside the transactional region are required to understand the abort cause, Last Branch Records (LBRs, See Section 25) or the SDE software emulation (see Section 12.4.8) can be used.

Tuning Suggestion 29. *The profiler should have options to display ReturnIP and EventingIP.*

Tuning Suggestion 30. *The stack callgraph is always associated with the ReturnIP and may appear noncontiguous with the EventingIP.*

Tuning Suggestion 31. *To see function calls inside the transactional region use LBRs or SDE.*

12.4.7 Last Branch Records and Transactional Aborts

The Last Branch Records (see section 17.4 in Volume 3 of the Intel Software Developer's Manual) provide information about transactional execution and aborts. Regular LBR usage is compatible with Intel TSX. Using LBRs can be useful to provide context inside the transaction, as the normal call graph is not visible. The Icall filter can be used to approximate a call graph. However, the LBR Call Graph Stack facility (Section 17.8 in Volume 3 of the Intel Software Developer's Manual) is not compatible with Intel TSX and may provide incomplete information.

Tuning Suggestion 32. *The PEBS profiling handler should support sampling LBRs on abort and report them to the user.*

12.4.8 Profiling and Testing Intel TSX Software using the Intel SDE

The Intel Software Development Emulator (Intel SDE) tool [http://software.intel.com/en-us/articles/intel-software-development-emulator] enables software development for planned instruction set extensions before they appear in hardware. The tool can also be used for extended testing, debugging and analysis of software that take advantage of the new instructions.

Programmers can use a number of Intel SDE capabilities for functional testing, profiling and debugging programs using the Intel TSX instructions. The tool can provide insight into common transactional aborts and additional profiling capability not available directly on hardware. Programmers should not use the tool to derive runtimes and absolute performance characteristics as those are a function of the inherently high overheads of the emulation the tool performs.

As described previously in Section 12.4.4, hardware reports the precise address of the instruction that caused an abort, unless the abort is due to either a data conflict or a resource limitation. The tool can provide the precise address of such an instruction and additional information about the instruction. The tool can further map this back to the application source code, providing the instruction address, source file names, line number, the call stacks, and the data address information the instruction was operating on. For victim transactions (aborted due to a conflict) the tool can also output source code locations where conflicting memory accesses have been executed.

This is achieved through the tool options:

```
-tsx -hle_enabled 1 -rtm-mode full -tsx_stats 1 -tsx_stats_call_stack 1
```

The fallback handler can use the contents of the EAX register to determine causes of aborts. The SDE tool can force a transactional abort with a specific EAX register value provided as an emulator parameter. This allows developers to test their fallback handler code with different EAX values. In this mode, every RTM-based transactional execution will immediately abort with the EAX register value being that provided as the parameter. This is quite effective in functionally testing for corner cases where a transactional execution aborts due to unresolved page faults or other similar operations (EAX = 0).

This is achieved through the tool options:

```
-tsx -rtm-mode abort -rtm_abort_reason EAX.
```

Intel SDE has instruction and memory access logging features which are useful for debugging capacity aborts. With the log data from Intel SDE, one can diagnose cache set population to determine if there is non-uniform cache set usage causing capacity overflows. A refined log data may be used to further diagnose the source of the aborts. The logging feature is enabled with the following options:

```
-tsx_debug_log 3 -tsx_log_inst 1 -tsx_log_file 1
```

Additionally Intel SDE allows to use a standard debugger (gdb and Microsoft Visual Studio) to perform functional debugging inside transactions.

12.4.9 HLE Specific Performance Monitoring Events

The Intel TSX Performance Events also include HLE-specific transactional abort conditions. These events track aborts due to causes listed in Section 12.2.4.4. These aborts often occur due to issues in synchronization library implementations. When a synchronization library is initially enabled for Intel TSX, it is useful to measure these events and improve the library until these counts are negligible.

TX_MEM.ABORT_HLE_STORE_TO_ELIDED_LOCK counts the number of transactional aborts due to a store operation without the XRELEASE prefix operating on an elided lock in the elision buffer. This is often because the library is missing the XRELEASE prefix on the lock release instruction.

TX_MEM.ABORT_ELISION_BUFFER_NOT_EMPTY counts the number of transactional aborts that occur because an XRELEASE prefixed lock release instruction that was committing the transactional execution finds the elision buffer with an elided lock. This typically occurs for code sequences where an XRELEASE occurs on a lock that wasn't elided and hence wasn't in the elision buffer.

TX_MEM.ABORT_HLE_ELISION_BUFFER_MISMATCH counts the number of transactional aborts because the XRELEASE lock does not satisfy the address and value requirements for elision in the elision buffer. This occurs for example if the value being written by the XRELEASE operation is different from the value that was read by the earlier XACQUIRE operation to the same lock.

TX_MEM.ABORT_HLE_ELISION_UNSUPPORTED_ALIGNMENT counts the number of transactional aborts if the lock in the elision buffer was accessed by a read in the transactional region but the read could not be serviced. This typically occurs if the access was not properly aligned, or had a partial overlap, or the read operation's linear address was different than the elided locks but the physical address was the same. These are fairly rare events.

12.4.10 Computing Useful Metrics for Intel TSX

We now provide formulas to compute useful metrics with the performance events. While some of the counts are available as their own events, it can sometimes be useful to do a derivation with limited counters.

The following calculates the number of times a HLE or RTM transactional execution was started. This combines all nested regions into one region for counting purposes.

#HLE Regions Started: HLE_RETIRED.COMMIT + HLE_RETIRED.ABORTED
#RTM Regions Started: RTM_RETIRED.COMMIT + RTM_RETIRED.ABORTED

The following calculates the percentage of HLE or RTM transactional executions that aborted.

%AbortedHLE = 100.0 * (HLE_RETIRED.ABORTED/HLE_RETIRED.START)
%AbortedRTM = 100.0 * (RTM_RETIRED.ABORTED/RTM_RETIRED.START)

The following calculates the average number of cycles spent in a transactional region (See Section 12.4.1 for CyclesInTX computation).

AvgCyclesInHLE = CyclesInTX/HLE_RETIRED_START
AvgCyclesInRTM = CyclesInTX/RTM_RETIRED.START
AvgCyclesInTX=CyclesInTX/(HLE_RETIRED.START + RTM_RETIRED.START)

The following calculates the percentage of HLE or RTM transactional executions that aborted due to a data conflict.

%AbortedHLEDataConflict = TX_MEM.ABORT_CONFLICT/HLE_RETIRED.START;
%AbortedRTMDataConflict = TX_MEM.ABORT_CONFLICT / RTM_RETIRED.START;
%AbortedTXDataConlict= TX_MEM.ABORT_CONFLICT / (HLE_RETIRED.START+RTM_RETIRED.START);

The following calculates the number of HLE or RTM transactional executions that aborted due to limited resources for transactional stores.

%AbortedTXStoreResource = TX_MEM.ABORT_CAPACITY_WRITE

On processors based on the Broadwell and Skylake microarchitectures, the event "TX_MEM.ABORT_CAPACITY_WRITE" is replaced by TX_MEM.ABORT_CAPACITY that counts aborts due to either read or write.

The following calculates the total number of HLE or RTM transactional executions that aborted due to resource limitations. The distinction occurs because transactional reads that are evicted from the L1 data cache may not immediately cause an abort.

%AbortedHLEResource = HLE_RETIRED.ABORTED_MISC1 - TX_MEM.ABORT_CONFLICT
%AbortedRTMResource = RTM_RETIRED.ABORTED_MISC1- TX_MEM.ABORT_CONFLICT
%AbortedTXResource = (HLE_RETIRED.ABORTED_MISC1+RTM_RETIRED.ABORTED_MISC5) - TX_EM.ABORT_CONFLICT

For HLE, HLE_RETIRED.ABORTED_MISC1 may include some additional contributions from the events discussed in Section 12.4.9. For accurate results the lock library should be tuned first to minimize them.

Note that HLE_RETIRED.ABORTED_MISC1 is also known with the more descriptive name HLE_RETIRED.ABORTED_MIEM. Similarly, RTM_RETIRED.ABORTED_MISC1 is also known as RTM_RETIRED.ABORTED_MEM.

12.5 PERFORMANCE GUIDELINES

The 4th generation Intel Core Processor is the first implementation that support Intel TSX. Transactional execution incurs some implementation dependent overheads. Performance will improve in subsequent microarchitecture generations. The first TSX implementation is oriented towards typical usage of critical sections in applications. As a result, these overheads are amortized and do not normally manifest themselves at an application level performance.

However, some guidelines are relevant to keep in mind:

Tuning Suggestion 33. *Intel TSX is designed for critical sections and thus the latency profiles of the XBEGIN/XEND instructions and XACQUIRE/XRELEASE prefixes are intended to match the LOCK prefixed instructions. These instructions should not be expected to have the latency of a regular load operation.*

There is an additional implementation-specific overhead associated with executing a transactional region. This consists of a mostly fixed cost in addition to a variable dynamic component. The overhead is largely independent of the size and memory foot print of the critical section. The additional overhead is typically amortized and hidden behind the out-of-order execution of the microarchitecture. However, on the 4th generation Intel Core Processor implementation, certain sequences may appear to exacerbate the overhead. This is particularly true if the critical section is very small and appear in tight loops (for example something typically done in microbenchmarks). Realistic applications do not normally exhibit such behavior.

The overhead is amortized in larger critical sections but will be exposed in very small critical sections. One simple approach to reduce perceived overhead is to perform an access to the transactional cache lines early in the critical section

The overhead of commits is reduced with processors based on the Broadwell microarchitecture.

12.6 DEBUGGING GUIDELINES

Using Intel TSX to implement Lock Elision does not change application semantics - all architectural state updated during an aborted transactional execution is automatically discarded by the hardware. Care must be taken if new code paths are added to the application and these paths are exercised only under transactional execution (See Section 12.2.5).

However, lock elision may change the timing relationships among different threads since it requires communication among threads only when required by data conflicts. Hence, locks may appear to execute much faster than normal. Such timing changes may expose latent bugs in an application. Exposure of such latent bugs is not unique to Intel TSX and can be expected with every new hardware generation.

Code instrumentation is a common technique while debugging multi-threaded software. As is the case with debugging timing related issues, care must be taken when instrumenting code to not perturb timing significantly and to not cause unnecessary aborts. A per thread buffer can be utilized to trace execution and log events of interests. The RDTSC instruction can be used to obtain a timestamp. The buffer should be printed outside the critical section.

Transactional aborts discard all memory state updated within the transactional region. This information cannot be traced without instrumentation support. Issues within transactional regions will show up in a profiling tool as a transactional abort and the Last Branch Record information can be used to reconstruct the control flow. On processors that support Intel® Processor Trace, the trace log allows reconstructing the full trace of the control flow inside transactions. The trace also contains markers indicating transaction start, commit and abort.

The regular assert() function would cause a transactional abort and its output information would not make it out of the transactional region. When using the RTM instructions, the assert functionality can be enhanced to end the transactional execution, make side effects visible, and terminate the program through the assert function. For example:

```
assert(x) => if (!(x)) { while (_xtest()) _xend();  assert(0); }
```

12.7 COMMON INTRINSICS FOR INTEL TSX

Recent assemblers (GNU binutils version 2.23, Microsoft Visual Studio 2012) include support for the Intel TSX instructions. On older tool chains it is possible to use the instructions as byte values.

12.7.1 RTM C intrinsics

Recent C/C++ compilers (gcc 4.8, Microsoft Visual Studio 2012, Intel C++ Compiler XE 13.0) support RTM intrinsics in the **immintrin.h** header file. RTM is a new instruction set and should be only used after check the RTM feature flag using CPUID instruction (See Chapter 8 of the *Intel® Architecture Instruction Set Extensions Programming Reference*).

_xbegin()

_xbegin() starts the transactional region and returns _XBEGIN_STARTED when in the transactional region, otherwise the abort code. It is important to check _xbegin() against _XBEGIN_STARTED which is not zero. Zero is a valid abort code. When the value is not _XBEGIN_STARTED the return code contains various status bits and an optional 8bit constant passed by _xabort().

Valid status bits are:

- _XABORT_EXPLICIT: Abort caused by _xabort(). _XABORT_CODE(status) contains the value passed to _xabort().
- _XABORT_RETRY: When this bit is set retrying the transactional region has a chance to commit. If not set retrying will likely not succeed.
- _XABORT_CAPACITY: The abort is related to a capacity overflow.
- _XABORT_DEBUG: The abort happened due to a debug trap.
- _XABORT_NESTED: The abort happened in a nested transaction.

_xend()

_xend() commits the transaction.

_xtest()

_xtest() returns true when the code is currently executing in a transaction. It can be also used with HLE.

_xabort()

_xabort(constant) aborts the current transaction. Constant must be a constant and can be only 8bits. The constant is contained in the status code returned by _xbegin() and can be accessed with _XABORT_CODE() when the _XABORT_EXPLICIT flag is set. See Section 4.5 for a recommended convention.

On gcc 4.8 and later compilers the -mrtm flag needs to be used to enable these intrinsics.

12.7.1.1 Emulated RTM intrinsics on older gcc compatible compilers

On older gcc compatible compilers that do not support the RTM intrinsics in immintrin.h, Example 12-11 shows the inline assembler equivalents that can be used.

Example 12-11. Emulated RTM intrinsic for Older GCC compilers

```
/* Not needed on newer toolchains that support this interface in immintrin.h */
#define _XBEGIN_STARTED       (~0u)
#define _XABORT_EXPLICIT      (1 << 0)
#define _XABORT_RETRY         (1 << 1)
#define _XABORT_CONFLICT      (1 << 2)
#define _XABORT_CAPACITY      (1 << 3)
#define _XABORT_DEBUG         (1 << 4)
#define _XABORT_NESTED        (1 << 5)
#define _XABORT_CODE(x)       (((x) >> 24) & 0xff)

#define __force_inline __attribute__((__always_inline__)) inline

static __force_inline int _xbegin(void)
{
    int ret = _XBEGIN_STARTED;
    asm volatile(".byte 0xc7,0xf8 ; .long 0" : "+a" (ret) :: "memory");
    return ret;
}

static __force_inline void _xend(void)
{
    asm volatile(".byte 0x0f,0x01,0xd5" ::: "memory");
}

static __force_inline void _xabort(const unsigned int status)
{
    asm volatile(".byte 0xc6,0xf8,%P0" :: "i" (status) : "memory");
}

static __force_inline int _xtest(void)
{
    unsigned char out;
    asm volatile(".byte 0x0f,0x01,0xd6 ; setnz %0" : "=r" (out) :: "memory");
    return out;
}
```

12.7.2 HLE intrinsics on gcc and other Linux compatible compilers

On Linux and compatible systems HLE is implemented as an extension to the gcc 4.8 and an older form of the C11 atomic primitives. HLE XACQUIRE can be used by setting the __ATOMIC_HLE_ACQUIRE flag to the memory model argument. HLE XRELEASE can be used with __ATOMIC_HLE_RELEASE.

For __ATOMIC_HLE_ACQUIRE the memory model must be __ATOMIC_ACQUIRE or stronger, for __ATOMIC_HLE_RELEASE __ATOMIC_RELEASE or stronger. For operations with a failure memory model (like __atomic_compare_exchange_n) the HLE flag is only supported on the non-failure memory model.

HLE is only supported on atomic operations that can be directly translated into IA atomic instructions. It is not supported with:

- 8 byte values on 32bit targets.

- 16 byte values.
- Fetch-op or op-fetch other than add/sub when the result is accessed.
- __atomic_store and __atomic_clear only support __ATOMIC_HLE_RELEASE.

12.7.2.1 Generating HLE intrinsics with gcc4.8

Due to a compiler bug in some versions of gcc 4.8 the -O2 or higher optimization level must be used to generate HLE hints using the atomic intrinsics.

12.7.2.2 C++11 atomic support

gcc 4.8 has support for the C++11 <atomic> header. The memory models defined there are extended with HLE flags similar to the C atomic interface. Two new flags __memory_order_hle_acquire and __memory_order_hle_release are defined. The constraints listed for the C atomic intrinsics apply. Example 12-12 shows a C++ example of an HLE intrinsic.

Example 12-12. C++ Example of HLE Intrinsic

```
#include <atomic>
#include <immintrin.h>
using namespace std;
atomic_flag lock;
for (;;) {
   if (!lock.test_and_test(memory_order_acquire|__memory_order_hle_acquire) {
      // Critical section with HLE lock elision
      lock.clear(memory_order_release|__memory_order_hle_release);
    break;
   } else {
     // Lock not acquired. Wait for lock and retry.
     while (lock.load())
        _mm_pause();   // abort transactional region on lock busy
   }
}
```

12.7.2.3 Emulating HLE intrinsics with older gcc-compatible compilers

For older compilers that do not support these intrinsics inline assembler can be used. For example to emulate __atomic_exchange_n(&lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE), see Example 12-13.

Example 12-13. Emulated HLE Intrinsic with Older GCC compiler

```
#define XACQUIRE ".byte 0xf2; " /* For older assemblers not supporting XACQUIRE */
#define XRELEASE ".byte 0xf3; "
static inline int hle_acquire_xchg(int *lock, int val)
{
   asm volatile(XACQUIRE "xchg %0,%1" : "+r" (val), "+m" (*lock) :: "memory");
   return val;
}
```

Example 12-13. Emulated HLE Intrinsic with Older GCC compiler (Contd.)

```
static void hle_release_store(int *lock, int val)
{
   asm volatile(XRELEASE "mov %0,%1" : "r" (val), "+m" (*lock) :: "memory");
}
```

12.7.3 HLE intrinsics on Windows C/C++ compilers

Windows C/C++ compilers (Microsoft Visual Studio 2012 and Intel C++ Compiler XE 13.0) provide versions of certain atomic intrinsic with HLE prefixes; see Example 12-14.

Example 12-14. HLE Intrinsic Supported by Intel and Microsoft Compilers

```
Atomic compare-and-exchange operations:

long _InterlockedCompareExchange_HLEAcquire(long volatile *Destination, long Exchange, long Comparand);
__int64 _InterlockedCompareExchange64_HLEAcquire(__int64 volatile *Destination, __int64 Exchange, __int64
Comparand);
void * _InterlockedCompareExchangePointer_HLEAcquire(void * volatile     *Destination, void * Exchange, void *
Comparand);
long _InterlockedCompareExchange_HLERelease(long volatile *Destination, long Exchange, long Comparand);
__int64 _InterlockedCompareExchange64_HLERelease(__int64 volatile *Destination, __int64 Exchange, __int64
Comparand);
void * _InterlockedCompareExchangePointer_HLERelease(void * volatile *Destination, void * Exchange, void *
Comparand);
```

```
Atomic addition:

long _InterlockedExchangeAdd_HLEAcquire(long volatile *Addend, long Value);
__int64 _InterlockedExchangeAdd64_HLEAcquire(__int64 volatile *Addend, __int64 Value);
long _InterlockedExchangeAdd_HLERelease(long volatile *Addend, long Value);
__int64 _InterlockedExchangeAdd64_HLERelease(__int64 volatile *Addend, __int64 Value);


Intrinsics for HLE prefixed stores:

void _Store_HLERelease(long volatile *Destination, long Value);
void _Store64_HLERelease(__int64 volatile *Destination, __int64 Value);
void _StorePointer_HLERelease(void * volatile *Destination, void * Value);
```

Please consult the compiler documentation for further information on these intrinsics.

CHAPTER 13
POWER OPTIMIZATION FOR MOBILE USAGES

13.1 OVERVIEW

Mobile computing allows computers to operate anywhere, anytime. Battery life is a key factor in delivering this benefit. Mobile applications require software optimization that considers both performance and power consumption. This chapter provides background on power saving techniques in mobile processors[1] and makes recommendations that developers can leverage to provide longer battery life.

A microprocessor consumes power while actively executing instructions and doing useful work. It also consumes power in inactive states (when halted). When a processor is active, its power consumption is referred to as active power. When a processor is halted, its power consumption is referred to as static power.

ACPI 3.0 (ACPI stands for Advanced Configuration and Power Interface) provides a standard that enables intelligent power management and consumption. It does this by allowing devices to be turned on when they are needed and by allowing control of processor speed (depending on application requirements). The standard defines a number of P-states to facilitate management of active power consumption; and several C-state types[2] to facilitate management of static power consumption.

Pentium M, Intel Core Solo, Intel Core Duo processors, and processors based on Intel Core microarchitecture implement features designed to enable the reduction of active power and static power consumption. These include:

- Enhanced Intel SpeedStep® Technology enables operating system (OS) to program a processor to transition to lower frequency and/or voltage levels while executing a workload.
- Support for various activity states (for example: Sleep states, ACPI C-states) to reduces static power consumption by turning off power to sub-systems in the processor.

Enhanced Intel SpeedStep Technology provides low-latency transitions between operating points that support P-state usages. In general, a high-numbered P-state operates at a lower frequency to reduce active power consumption. High-numbered C-state types correspond to more aggressive static power reduction. The trade-off is that transitions out of higher-numbered C-states have longer latency.

13.2 MOBILE USAGE SCENARIOS

In mobile usage models, heavy loads occur in bursts while working on battery power. Most productivity, web, and streaming workloads require modest performance investments. Enhanced Intel SpeedStep Technology provides an opportunity for an OS to implement policies that track the level of performance history and adapt the processor's frequency and voltage. If demand changes in the last 300 ms[3], the technology allows the OS to optimize the target P-state by selecting the lowest possible frequency to meet demand.

Consider, for example, an application that changes processor utilization from 100% to a lower utilization and then jumps back to 100%. The diagram in Figure 13-1 shows how the OS changes processor frequency to accommodate demand and adapt power consumption. The interaction between the OS

1. For Intel® Centrino® mobile technology and Intel® Centrino® Duo mobile technology, only processor-related techniques are covered in this manual.
2. ACPI 3.0 specification defines four C-state types, known as C0, C1, C2, C3. Microprocessors supporting the ACPI standard implement processor-specific states that map to each ACPI C-state type.
3. This chapter uses numerical values representing time constants (300 ms, 100 ms, etc.) on power management decisions as examples to illustrate the order of magnitude or relative magnitude. Actual values vary by implementation and may vary between product releases from the same vendor.

power management policy and performance history is described below.

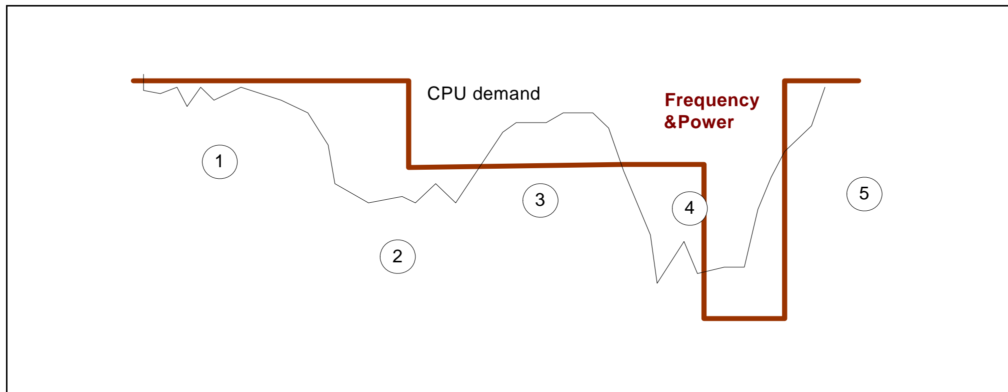


Figure 13-1. Performance History and State Transitions

1. Demand is high and the processor works at its highest possible frequency (P0).
2. Demand decreases, which the OS recognizes after some delay; the OS sets the processor to a lower frequency (P1).
3. The processor decreases frequency and processor utilization increases to the most effective level, 80-90% of the highest possible frequency. The same amount of work is performed at a lower frequency.
4. Demand decreases and the OS sets the processor to the lowest frequency, sometimes called Low Frequency Mode (LFM).
5. Demand increases and the OS restores the processor to the highest frequency.

13.2.1 Intelligent Energy Efficient Software

With recent advances in power technology and wide range of computing scenarios demanded by end users, intelligent balance between power consumption and performance becomes more and more important. Energy efficient software plays a key role in exploring the latest hardware power savings offered by current generation architecture. Poorly-written code can prevent a system from taking advantage of new hardware features and serving the dynamic needs of end users.

A mobile platform consists of various components such as a CPU, LCD, HDD, DVD, and chipsets, which individually contribute to the power drain of the notebook. Understanding the power contribution of each major component in the platform provides a better view on the total power usage, provides guidance on optimizing power consumption, and may help software to adjust dynamic balance of power budgets between some components.

The following are a few general guidelines for energy efficient software:

- Application should leverage modern OS facility to select appropriate operating frequency instead of setting processor frequency by itself. The latter is likely to have negative impact on both power consumption and performance.
- When your application is waiting for user input or another event to happen, let your application use services that are optimized to go to idle mode quickly. The idle behavior can have a big impact on power consumption. When an application knows it will be operating in a mostly idle context, reduce the frequency of application events that wake up the processor, avoid periodic polling, and reduce the number of services that are active in memory.
- Build context awareness into applications to extend battery life further and optimal user experience.
- Architect an awareness of power consumption and/or dynamic power policy into your application for contextual usages and optimal end user experiences. Specific detail may vary across OSes. For Microsoft Windows OS consult

http://www.microsoft.com/whdc/system/pnppwr/powermgmt/PMpolicy_Windows.mspx#. For Linux, consult http://www.lesswatts.org.

- Characterize your application's power consumption. There are various techniques available to measure the power consumption of a platform:
  - — Use hardware instrumentation such as Fluke NetDAQ*. This provides power measurements for each component such as CPU, HDD, and memory.
  - — Use C-state residency counters. See Chapter 35, "Model-Specific Registers (MSRs)" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.
  - — Study parameters such as CPU usage, kernel time, and time interrupt rate, to gain insight into the behavior of the software, which can then be related to platform power consumption if the hardware instrumentation is not available.

Section 13.5 provide some examples on how to relate performance with power consumption and techniques for optimizing software.

13.3 ACPI C-STATES

When computational demands are less than 100%, part of the time the processor is doing useful work and the rest of the time it is idle. For example, the processor could be waiting on an application time-out set by a Sleep() function, waiting for a web server response, or waiting for a user mouse click. Figure 13-2 illustrates the relationship between active and idle time.

When an application moves to a wait state, the OS issues a HLT instruction and the processor enters a halted state in which it waits for the next interrupt. The interrupt may be a periodic timer interrupt or an interrupt that signals an event.

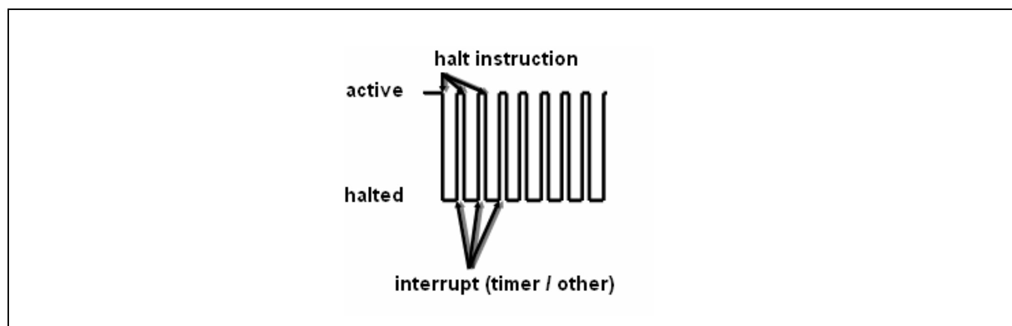


Figure 13-2. Active Time Versus Halted Time of a Processor

As shown in the illustration of Figure 13-2, the processor is in either active or idle (halted) state. ACPI defines four C-state types (C0, C1, C2 and C3). Processor-specific C states can be mapped to an ACPI C-state type via ACPI standard mechanisms. The C-state types are divided into two categories: active (C0), in which the processor consumes full power; and idle (C1-3), in which the processor is idle and may consume significantly less power.

The index of a C-state type designates the depth of sleep. Higher numbers indicate a deeper sleep state and lower power consumption. They also require more time to wake up (higher exit latency).

C-state types are described below:

- C0 — The processor is active and performing computations and executing instructions.
- C1 — This is the lowest-latency idle state, which has very low exit latency. In the C1 power state, the processor is able to maintain the context of the system caches.
- C2 — This level has improved power savings over the C1 state. The main improvements are provided at the platform level.

- C3 — This level provides greater power savings than C1 or C2. In C3, the processor stops clock generating and snooping activity. It also allows system memory to enter self-refresh mode.

The basic technique to implement OS power management policy to reduce static power consumption is by evaluating processor idle durations and initiating transitions to higher-numbered C-state types. This is similar to the technique of reducing active power consumption by evaluating processor utilization and initiating P-state transitions. The OS looks at history within a time window and then sets a target C-state type for the next time window, as illustrated in Figure 13-3:

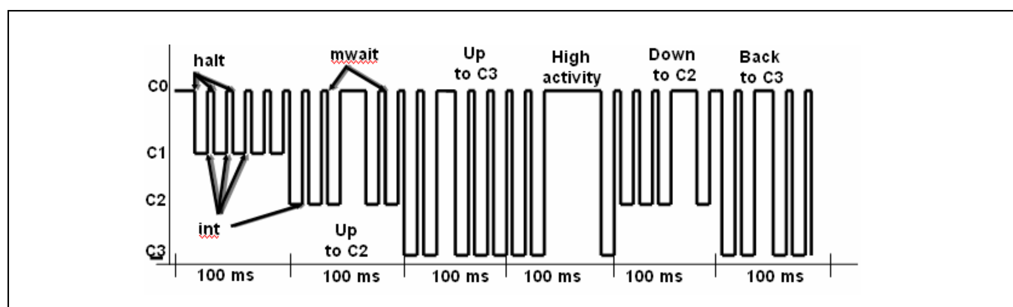


Figure 13-3. Application of C-states to Idle Time

Consider that a processor is in lowest frequency (LFM- low frequency mode) and utilization is low. During the first time slice window (Figure 13-3 shows an example that uses 100 ms time slice for C-state decisions), processor utilization is low and the OS decides to go to C2 for the next time slice. After the second time slice, processor utilization is still low and the OS decides to go into C3.

13.3.1 Processor-Specific C4 and Deep C4 States

The Pentium M, Intel Core Solo, Intel Core Duo processors, and processors based on Intel Core microarchitecture[4] provide additional processor-specific C-states (and associated sub C-states) that can be mapped to ACPI C3 state type. The processor-specific C states and sub C-states are accessible using MWAIT extensions and can be discovered using CPUID. One of the processor-specific state to reduce static power consumption is referred to as C4 state. C4 provides power savings in the following manner:

- The voltage of the processor is reduced to the lowest possible level that still allows the L2 cache to maintain its state.
- In an Intel Core Solo, Intel Core Duo processor or a processor based on Intel Core microarchitecture, after staying in C4 for an extended time, the processor may enter into a Deep C4 state to save additional static power.

The processor reduces voltage to the minimum level required to safely maintain processor context. Although exiting from a deep C4 state may require warming the cache, the performance penalty may be low enough such that the benefit of longer battery life outweighs the latency of the deep C4 state.

13.3.2 Processor-Specific Deep C-States and Intel® Turbo Boost Technology

Processors based on Intel microarchitecture code name Nehalem implements several processor-specific C-states.

4. Pentium M processor can be detected by CPUID signature with family 6, model 9 or 13; Intel Core Solo and Intel Core Duo processor has CPUID signature with family 6, model 14; processors based on Intel Core microarchitecture has CPUID signature with family 6, model 15.

Table 13-1. ACPI C-State Type Mappings to Processor Specific C-State for Mobile Processors Based on Intel Microarchitecture Code Name Nehalem

ACPI C-State Type	Processor-Specific C-State
C0	C0
C1	C1
C2	C3
C3	C7

The processor-specific deep C-states are implementation dependent. Generally, the low power C-states (higher numbered C-states) have higher exit latencies. For example, when the cores are already in C7, the last level cache (L3) is flushed. The processor support auto-demotion of OS request to deep C-states (C3/C7) and demote to C1/C3 state to support flexible power-performance settings.

In addition to low-power, deep C-states, Intel Turbo Boost Technology can opportunistically boost performance in normal state (C0) by mapping p1 state to the processor's qualified high-frequency mode operation. Headroom in the system's TDP can be converted to an even higher frequency than P1 state target. When the operating system requests P0 state, the processor sets core frequencies between P1 to P0 range. A P0 state with only one core busy, achieves the maximum possible Intel Turbo Boost Technology frequency, whereas when the processor is running two to four cores the frequency is constrained by processor limitations. Under normal conditions the frequency does not go below P1, even when all cores are running.

13.3.3 Processor-Specific Deep C-States for Intel® Microarchitecture Code Name Sandy Bridge

Processors based on Intel microarchitecture code name Sandy Bridge implements several processor-specific C-states.

Table 13-2. ACPI C-State Type Mappings to Processor Specific C-State of Intel Microarchitecture Code Name Sandy Bridge

ACPI C-State Type	Processor-Specific C-State
C0	C0
C1	C1
C2	C3
C3	C6/C7

The microarchitectural behavior of processor-specific deep C-states are implementation dependent. The following summarizes some of their key power-saving and intelligent responsive characteristics:

- For mobile platforms, while the cores are already in C7, the last level cache (L3) is flushed.
- Auto-demotion: The processor can demote OS requests to a target C-state (core C6/C7 or C3 state) to a numerically lower C-state (core C3 or C1 state) in the following cases:
  - When history indicates that C6/C7 or C3 states are less energy efficient than C3 or C1 states.
  - When history indicates that a deeper sleep state may impact performance.
  - Energy inefficiency or performance degradation can occur due to the deeper C-state transition overhead occurring too frequently. Intel microarchitecture code name Sandy Bridge has an enhanced algorithm that improves power gain from this feature.
- Un-demotion: An OS request to a deeper C-state can be demoted by auto-demotion, resulting in C1 or C3 states. After long residency in the demoted state, the hardware returns control back to the OS.

The expectation is that in this case, the OS will repeat the deeper C-state request and hardware un-demotion will enter into the OS-requested deeper C state.

13.3.4 Intel® Turbo Boost Technology 2.0

Intel® Turbo Boost Technology 2.0 is a second generation enhancement of Intel® Turbo Boost Technology. The latter can opportunistically boost the processor core's frequency to a higher frequency above the qualified frequency depending on the TDP headroom.

The TDP of Intel Core processors based on Intel microarchitecture code name Sandy Bridge include budgets for the processor core and processor graphic sub-system. Intel® Turbo Boost Technology 2.0 allows more opportunity to convert the thermal and power budget headroom to boost the processor core frequency and/or operating frequency of the processor graphic sub-system.

Energy consumption by the processor cores and/or by the processor graphic unit can be measured using a set of MSR interface[5]. Operating system requirements to support Intel Turbo Boost Technology, to use hints to optimize performance and energy bias in turbo mode operation, are described in Chapter 14, "Power and Thermal Management" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

13.4 GUIDELINES FOR EXTENDING BATTERY LIFE

Follow the guidelines below to optimize to conserve battery life and adapt for mobile computing usage:

- Adopt a power management scheme to provide just-enough (not the highest) performance to achieve desired features or experiences.
- Avoid using spin loops.
- Reduce the amount of work the application performs while operating on a battery.
- Take advantage of hardware power conservation features using ACPI C3 state type and coordinate processor cores in the same physical processor.
- Implement transitions to and from system sleep states (S1-S4) correctly.
- Allow the processor to operate at a higher-numbered P-state (lower frequency but higher efficiency in performance-per-watt) when demand for processor performance is low.
- Allow the processor to enter higher-numbered ACPI C-state type (deeper, low-power states) when user demand for processor activity is infrequent.

13.4.1 Adjust Performance to Meet Quality of Features

When a system is battery powered, applications can extend battery life by reducing the performance or quality of features, turning off background activities, or both. Implementing such options in an application increases the processor idle time. Processor power consumption when idle is significantly lower than when active, resulting in longer battery life.

Example of techniques to use are:

- Reducing the quality/color depth/resolution of video and audio playback.
- Turning off automatic spell check and grammar correction.
- Turning off or reducing the frequency of logging activities.
- Consolidating disk operations over time to prevent unnecessary spin-up of the hard drive.
- Reducing the amount or quality of visual animations.

---

5. Generally, energy measurements and power management decisions based on these MSR interfaces should operate within the same processor family/model and refrain from extrapolating across different family/models or unsupported environmental conditions.

- Turning off, or significantly reducing file scanning or indexing activities.
- Postponing possible activities until AC power is present.

Performance/quality/battery life trade-offs may vary during a single session, which makes implementation more complex. An application may need to implement an option page to enable the user to optimize settings for user's needs (see Figure 13-4).

To be battery-power-aware, an application may use appropriate OS APIs. For Windows XP, these include:

- GetSystemPowerStatus — Retrieves system power information. This status indicates whether the system is running on AC or DC (battery) power, whether the battery is currently charging, and how much battery life remains.
- GetActivePwrScheme — Retrieves the active power scheme (current system power scheme) index. An application can use this API to ensure that system is running best power scheme.Avoid Using Spin Loops.

Spin loops are used to wait for short intervals of time or for synchronization. The main advantage of a spin loop is immediate response time. Using the PeekMessage() in Windows API has the same advantage for immediate response (but is rarely needed in current multitasking operating systems).

However, spin loops and PeekMessage() in message loops require the constant attention of the processor, preventing it from entering lower power states. Use them sparingly and replace them with the appropriate API when possible. For example:

- When an application needs to wait for more then a few milliseconds, it should avoid using spin loops and use the Windows synchronization APIs, such as WaitForSingleObject().
- When an immediate response is not necessary, an application should avoid using PeekMessage(). Use WaitMessage() to suspend the thread until a message is in the queue.

Intel® Mobile Platform Software Development Kit[6] provides a rich set of APIs for mobile software to manage and optimize power consumption of mobile processor and other components in the platform.

13.4.2 Reducing Amount of Work

When a processor is in the C0 state, the amount of energy a processor consumes from the battery is proportional to the amount of time the processor executes an active workload. The most obvious technique to conserve power is to reduce the number of cycles it takes to complete a workload (usually that equates to reducing the number of instructions that the processor needs to execute, or optimizing application performance).

Optimizing an application starts with having efficient algorithms and then improving them using Intel software development tools, such as Intel VTune Performance Analyzers, Intel compilers, and Intel Performance Libraries.

See Chapter 3 through Chapter 7 for more information about performance optimization to reduce the time to complete application workloads.

13.4.3 Platform-Level Optimizations

Applications can save power at the platform level by using devices appropriately and redistributing the workload. The following techniques do not impact performance and may provide additional power conservation:

- Read ahead from CD/DVD data and cache it in memory or hard disk to allow the DVD drive to stop spinning.
- Switch off unused devices.
- When developing a network-intensive application, take advantage of opportunities to conserve power. For example, switch to LAN from WLAN whenever both are connected.

---

6. Evaluation copy may be downloaded at http://www.intel.com/cd/software/products/asmo-na/eng/219691.htm

- Send data over WLAN in large chunks to allow the WiFi card to enter low power mode in between consecutive packets. The saving is based on the fact that after every send/receive operation, the WiFi card remains in high power mode for up to several seconds, depending on the power saving mode. (Although the purpose keeping the WiFI in high power mode is to enable a quick wake up).
- Avoid frequent disk access. Each disk access forces the device to spin up and stay in high power mode for some period after the last access. Buffer small disk reads and writes to RAM to consolidate disk operations over time. Use the GetDevicePowerState() Windows API to test disk state and delay the disk access if it is not spinning.

13.4.4 Handling Sleep State Transitions

In some cases, transitioning to a sleep state may harm an application. For example, suppose an application is in the middle of using a file on the network when the system enters suspend mode. Upon resuming, the network connection may not be available and information could be lost.

An application may improve its behavior in such situations by becoming aware of sleep state transitions. It can do this by using the WM_POWERBROADCAST message. This message contains all the necessary information for an application to react appropriately.

Here are some examples of an application reaction to sleep mode transitions:

- Saving state/data prior to the sleep transition and restoring state/data after the wake up transition.
- Closing all open system resource handles such as files and I/O devices (this should include duplicated handles).
- Disconnecting all communication links prior to the sleep transition and re-establishing all communication links upon waking up.
- Synchronizing all remote activity, such as like writing back to remote files or to remote databases, upon waking up.
- Stopping any ongoing user activity, such as streaming video, or a file download, prior to the sleep transition and resuming the user activity after the wake up transition.

Recommendation: *Appropriately handling the suspend event enables more robust, better performing applications.*

13.4.5 Using Enhanced Intel SpeedStep® Technology

Use Enhanced Intel SpeedStep Technology to adjust the processor to operate at a lower frequency and save energy. The basic idea is to divide computations into smaller pieces and use OS power management policy to effect a transition to higher P-states.

Typically, an OS uses a time constant on the order of 10s to 100s of milliseconds[7] to detect demand on processor workload. For example, consider an application that requires only 50% of processor resources to reach a required quality of service (QOS). The scheduling of tasks occurs in such a way that the processor needs to stay in P0 state (highest frequency to deliver highest performance) for 0.5 seconds and may then goes to sleep for 0.5 seconds. The demand pattern then alternates.

Thus the processor demand switches between 0 and 100% every 0.5 seconds, resulting in an average of 50% of processor resources. As a result, the frequency switches accordingly between the highest and lowest frequency. The power consumption also switches in the same manner, resulting in an average power usage represented by the equation Paverage = (Pmax+Pmin)/2.

Figure 13-4 illustrates the chronological profiles of coarse-grain (> 300 ms) task scheduling and its effect on operating frequency and power consumption.

7. The actual number may vary by OS and by OS release.

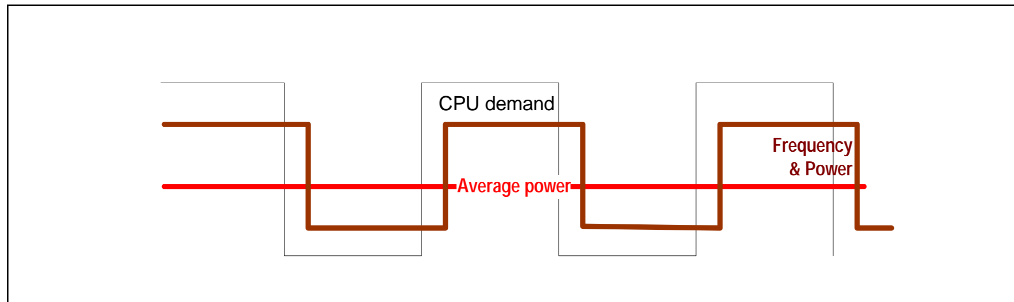


Figure 13-4. Profiles of Coarse Task Scheduling and Power Consumption

The same application can be written in such a way that work units are divided into smaller granularity, but scheduling of each work unit and Sleep() occurring at more frequent intervals (e.g. 100 ms) to deliver the same QOS (operating at full performance 50% of the time). In this scenario, the OS observes that the workload does not require full performance for each 300 ms sampling. Its power management policy may then commence to lower the processor's frequency and voltage while maintaining the level of QOS.

The relationship between active power consumption, frequency and voltage is expressed by the equation:

$$Power = \alpha * C * V^2 * F$$

In the equation: 'V' is core voltage, 'F' is operating frequency, and 'α' is the activity factor. Typically, the quality of service for 100% performance at 50% duty cycle can be met by 50% performance at 100% duty cycle. Because the slope of frequency scaling efficiency of most workloads will be less than one, reducing the core frequency to 50% can achieve more than 50% of the original performance level. At the same time, reducing the core frequency to 50% allows for a significant reduction of the core voltage.

Because executing instructions at higher P-state (lower power state) takes less energy per instruction than at P0 state, Energy savings relative to the half of the duty cycle in P0 state (Pmax /2) more than compensate for the increase of the half of the duty cycle relative to inactive power consumption (Pmin /2). The non-linear relationship between power consumption to frequency and voltage means that changing the task unit to finer granularity will deliver substantial energy savings. This optimization is possible when processor demand is low (such as with media streaming, playing a DVD, or running less resource intensive applications like a word processor, email or web browsing).

An additional positive effect of continuously operating at a lower frequency is that frequent changes in power draw (from low to high in our case) and battery current eventually harm the battery. They accelerate its deterioration.

When the lowest possible operating point (highest P-state) is reached, there is no need for dividing computations. Instead, use longer idle periods to allow the processor to enter a deeper low power mode.

13.4.6 Enabling Intel® Enhanced Deeper Sleep

In typical mobile computing usages, the processor is idle most of the time. Conserving battery life must address reducing static power consumption.

Typical OS power management policy periodically evaluates opportunities to reduce static power consumption by moving to lower-power C-states. Generally, the longer a processor stays idle, OS power management policy directs the processor into deeper low-power C-states.

After an application reaches the lowest possible P-state, it should consolidate computations in larger chunks to enable the processor to enter deeper C-States between computations. This technique utilizes the fact that the decision to change frequency is made based on a larger window of time than the period to decide to enter deep sleep. If the processor is to enter a processor-specific C4 state to take advantage of aggressive static power reduction features, the decision should be based on:

- Whether the QOS can be maintained in spite of the fact that the processor will be in a low-power, long-exit-latency state for a long period.
- Whether the interval in which the processor stays in C4 is long enough to amortize the longer exit latency of this low-power C state.

Eventually, if the interval is large enough, the processor will be able to enter deeper sleep and save a considerable amount of power. The following guidelines can help applications take advantage of Intel® Enhanced Deeper Sleep:

- Avoid setting higher interrupt rates. Shorter periods between interrupts may keep OSes from entering lower power states. This is because transition to/from a deep C-state consumes power, in addition to a latency penalty. In some cases, the overhead may outweigh power savings.
- Avoid polling hardware. In a ACPI C3 type state, the processor may stop snooping and each bus activity (including DMA and bus mastering) requires moving the processor to a lower-numbered C-state type. The lower-numbered state type is usually C2, but may even be C0. The situation is significantly improved in the Intel Core Solo processor (compared to previous generations of the Pentium M processors), but polling will likely prevent the processor from entering into highest-numbered, processor-specific C-state.

13.4.7 Multicore Considerations

Multicore processors deserves some special considerations when planning power savings. The dual-core architecture in Intel Core Duo processor and mobile processors based on Intel Core microarchitecture provide additional potential for power savings for multi-threaded applications.

13.4.7.1 Enhanced Intel SpeedStep® Technology

Using domain-composition, a single-threaded application can be transformed to take advantage of multi-core processors. A transformation into two domain threads means that each thread will execute roughly half of the original number of instructions. Dual core architecture enables running two threads simultaneously, each thread using dedicated resources in the processor core. In an application that is targeted for the mobile usages, this instruction count reduction for each thread enables the physical processor to operate at lower frequency relative to a single-threaded version. This in turn enables the processor to operate at a lower voltage, saving battery life.

Note that the OS views each logical processor or core in a physical processor as a separate entity and computes CPU utilization independently for each logical processor or core. On demand, the OS will choose to run at the highest frequency available in a physical package. As a result, a physical processor with two cores will often work at a higher frequency than it needs to satisfy the target QOS.

For example if one thread requires 60% of single-threaded execution cycles and the other thread requires 40% of the cycles, the OS power management may direct the physical processor to run at 60% of its maximum frequency.

However, it may be possible to divide work equally between threads so that each of them require 50% of execution cycles. As a result, both cores should be able to operate at 50% of the maximum frequency (as opposed to 60%). This will allow the physical processor to work at a lower voltage, saving power.

So, while planning and tuning your application, make threads as symmetric as possible in order to operate at the lowest possible frequency-voltage point.

13.4.7.2 Thread Migration Considerations

Interaction of OS scheduling and multicore unaware power management policy may create some situations of performance anomaly for multi-threaded applications. The problem can arise for multithreading application that allow threads to migrate freely.

When one full-speed thread is migrated from one core to another core that has idled for a period of time, an OS without a multicore-aware P-state coordination policy may mistakenly decide that each core demands only 50% of processor resources (based on idle history). The processor frequency may be

reduced by such multicore unaware P-state coordination, resulting in a performance anomaly. See Figure 13-5.

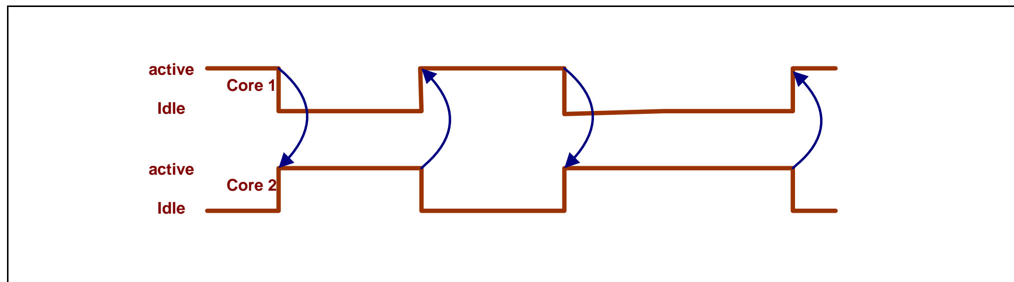


Figure 13-5. Thread Migration in a Multicore Processor

Software applications have a couple of choices to prevent this from happening:

- Thread affinity management — A multi-threaded application can enumerate processor topology and assign processor affinity to application threads to prevent thread migration. This can work around the issue of OS lacking multicore aware P-state coordination policy.
- Upgrade to an OS with multicore aware P-state coordination policy — Some newer OS releases may include multicore aware P-state coordination policy. The reader should consult with specific OS vendors.

13.4.7.3 Multicore Considerations for C-States

There are two issues that impact C-states on multicore processors.

Multicore-unaware C-state Coordination May Not Fully Realize Power Savings

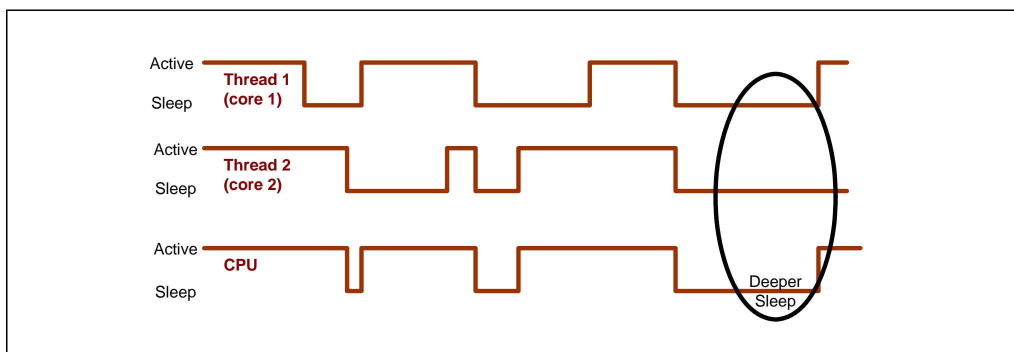


Figure 13-6. Progression to Deeper Sleep

When each core in a multicore processor meets the requirements necessary to enter a different C-state type, multicore-unaware hardware coordination causes the physical processor to enter the lowest possible C-state type (lower-numbered C state has less power saving). For example, if Core 1 meets the requirement to be in ACPI C1 and Core 2 meets requirement for ACPI C3, multicore-unaware OS coordination takes the physical processor to ACPI C1. See Figure 13-6.

Enabling Both Cores to Take Advantage of Intel Enhanced Deeper Sleep.

To best utilize processor-specific C-state (e.g., Intel Enhanced Deeper Sleep) to conserve battery life in multithreaded applications, a multi-threaded application should synchronize threads to work simultaneously and sleep simultaneously using OS synchronization primitives. By keeping the package in a fully

idle state longer (satisfying ACPI C3 requirement), the physical processor can transparently take advantage of processor-specific Deep C4 state if it is available.

Multi-threaded applications need to identify and correct load-imbalances of its threaded execution before implementing coordinated thread synchronization. Identifying thread imbalance can be accomplished using performance monitoring events. Intel Core Duo processor provides an event for this purpose. The event (Serial_Execution_Cycle) increments under the following conditions:

- Core actively executing code in C0 state.
- Second core in physical processor in idle state (C1-C4).

This event enables software developers to find code that is executing serially, by comparing Serial_Execution_Cycle and Unhalted_Ref_Cycles. Changing sections of serialized code to execute into two parallel threads enables coordinated thread synchronization to achieve better power savings.

Although Serial_Execution_Cycle is available only on Intel Core Duo processors, application thread with load-imbalance situations usually remains the same for symmetric application threads and on symmetrically configured multicore processors, irrespective of differences in their underlying microarchitecture. For this reason, the technique to identify load-imbalance situations can be applied to multi-threaded applications in general, and not specific to Intel Core Duo processors.

13.5 TUNING SOFTWARE FOR INTELLIGENT POWER CONSUMPTION

This section describes some techniques for tuning software for balance of both power and performance. Most of the power optimization techniques are generic. The last sub section (Section 13.5.8) describes features specific to Intel microarchitecture code name Sandy Bridge. Explore these features to optimize software for performance and corresponding power benefits.

13.5.1 Reduction of Active Cycles

Finishing the task quicker by reducing the amount of active cycles, then transfer control to the system idle loop will take advantage of modern operating system's power saving optimizations.

Reduction of active cycles can be achieved in several ways, from applying performance-oriented coding techniques discussed in Chapter 3, vectorization using SSE and/or AVX, to multi-threading.

13.5.1.1 Multi-threading to reduce Active Cycles

If given a task of some fixed amount of computational work that has thread-level parallelism, one can apply data-decomposition for multi-threading. The amount of reduction in active cycles will depend on the degree of parallelism. Similar principle can also apply to function-decomposition situations.

A balanced multi-threading implementation is more likely to achieve more optimal results in intelligent efficient performance and power saving benefits. Choosing the right synchronization primitives also has significant impact on both power and performance.

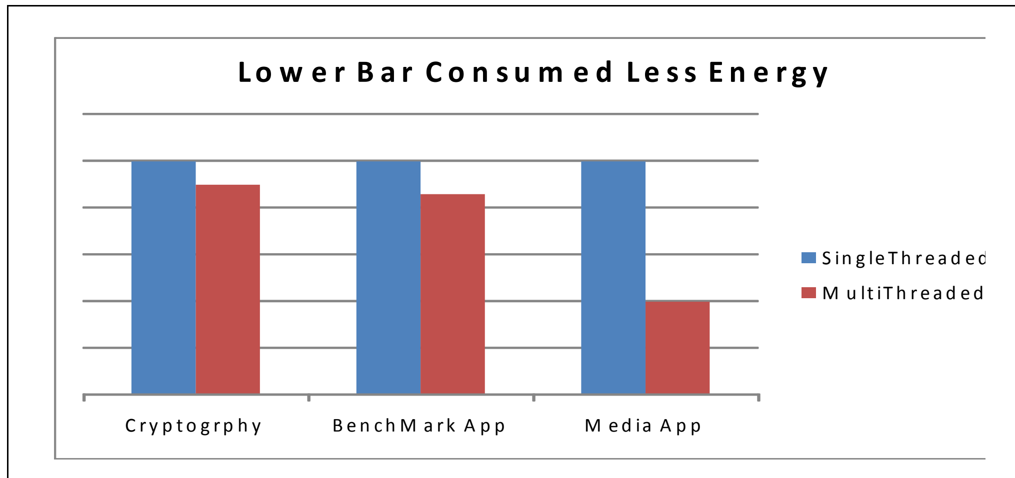


Figure 13-7. Energy Saving due to Performance Optimization

Figure 13-7 above shows the result of a study that compares processor energy consumption of single threaded workloads with their corresponding performance-optimized implementations, using three sets of applications across different application domains. In this particular study, optimization effort in application 1 (Cryptography) achieved 2X gain in performance alone. At the same time, its energy consumption reduced about 12%. In application 3 (a media application), performance optimization efforts including multi-threading and other techniques achieved 10X performance gain. Its energy consumption reduced about 60%.

13.5.1.2 Vectorization

Use SIMD instructions can reduce the path length of completing a given computational task, often reducing active cycles. Code that performs the same operation on multiple independent data elements is a good candidate for vectorization. Vectorization techniques are typically applied to applications with loops with elements that can be processed in single instruction. Typically, the slight power increase per unit time of using SIMD instructions are compensated by much greater reduction of active cycles. The net effect is improved energy consumption.

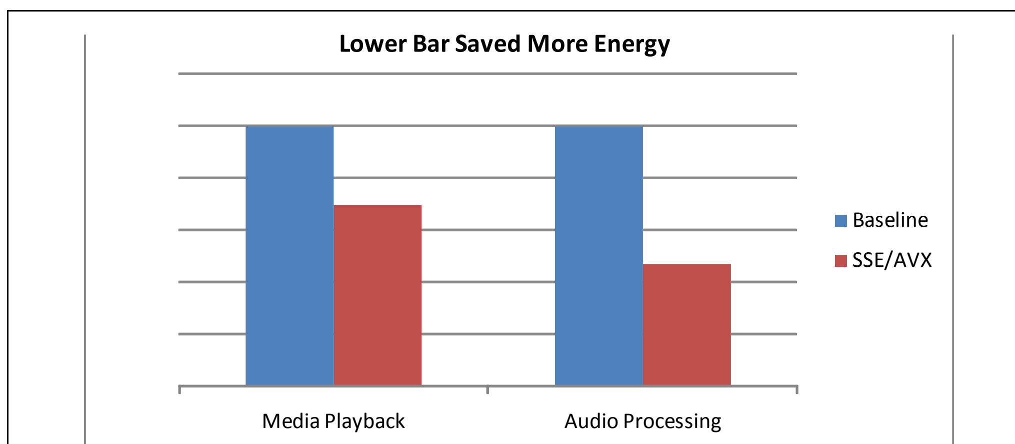


Figure 13-8. Energy Saving due to Vectorization

Figure 13-7 shows the result of a study on the energy saving effect due to vectorization. A media play-back workload achieved 2.15X speedup due to using SSE2 and SSE4 instruction sets. Another audio processing workload increased performance to ~5X by using Intel AVX instruction sets. At the same time, the latter also had better energy saving.

13.5.2 PAUSE and Sleep(0) Loop Optimization

In multi-threading implementation, a popular construct in thread synchronization and for yielding scheduling quanta to another thread waiting to carry out its task is to sit in a loop and issuing SLEEP(0).

These are typically called “sleep loops”, see Example 13-1. It should be noted that a SwitchToThread call can also be used. The “sleep loop” is common in locking algorithms and thread pools as the threads are waiting on work.

Example 13-1. Unoptimized Sleep Loop

```
while(!acquire_lock())
{    Sleep( 0 );      }
do_work();
release_lock();
```

This construct of sitting in a tight loop and calling Sleep() service with a parameter of 0 is actually a polling loop with side effects:

- Each call to Sleep() experiences the expensive cost of a context switch, which can be 10000+ cycles.
- It also suffers the cost of ring 3 to ring 0 transitions, which can be 1000+ cycles.
- When there is no other thread waiting to take possession of control, this sleep loop behaves to the OS as a highly active task demanding CPU resource, preventing the OS to put the CPU into a low-power state.

Example 13-2. Power Consumption Friendly Sleep Loop Using PAUSE

```
if (!acquire_lock())
{ /* Spin on pause max_spin_count times before backing off to sleep */
    for(int j = 0; j < max_spin_count; ++j)
    {      /* intrinsic for PAUSE instruction*/
        _mm_pause();
        if (read_volatile_lock())
        {
            if (acquire_lock())    goto PROTECTED_CODE;
        }
    }
    /* Pause loop didn't work, sleep now */
    Sleep(0);
    goto ATTEMPT_AGAIN;
}
PROTECTED_CODE:
do_work();
release_lock();
```

Example 13-2 shows the technique of using PAUSE instruction to make the sleep loop power friendly.

By slowing down the “spin-wait” with the PAUSE instruction, the multi-threading software gains:

- Performance by facilitating the waiting tasks to acquire resources more easily from a busy wait.

- Power-savings by both using fewer parts of the pipeline while spinning.
- Elimination of great majority of unnecessarily executed instructions caused by the overhead of a Sleep(0) call.

In one case study, this technique achieved 4.3x of performance gain, which translated to 21% power savings at the processor and 13% power savings at platform level.

13.5.3 Spin-Wait Loops

Use the PAUSE instruction in all spin wait loops. The PAUSE instruction de-pipelines the spin-wait loop to prevent it from consuming execution resources excessively and consuming power needlessly.

When executing a spin-wait loop, the processor can suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation and flushes the core processor's pipeline.

The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation and prevent the pipeline flush. However, you should try to keep spin-wait loops with PAUSE short.

13.5.4 Using Event Driven Service Instead of Polling in Code

Consistently polling for devices or state changes can cause the platform to wake up and consume more power. Minimize polling whenever possible and use an event driven framework if available. If an OS provides notification services for various device state changes, such as transition from AC to battery, use them instead of polling for device state changes. Using this new event notification framework reduces the overhead for the code to poll the status of the power source, because the code can get notifications asynchronously when status changes happen.

13.5.5 Reducing Interrupt Rate

High interrupt rate may have two consequences that impact processor power and performance:

- It prevents the processor package and its cores from going into deeper sleep states (C-states), which means that the system does not enable the hardware to utilize the power saving features.
- It limits the frequency to which Intel Turbo Boost Technology 2.0 can reach, and therefore the performance of other applications running on the processor degrades.

If a user session and/or an application experiences a rate of thousands of interrupts per second, it would have inhibited the processor to achieve intelligent balance between performance and saving power.

To avoid this situation minimize sporadic wakeups. Schedule all periodic activities of an application or driver into one wakeup period and reduce the interrupt rate to the minimum required.

Many media applications set a very high timer tick rate (1ms). Where possible, use the operating system default timer tick rate. If high granularity is absolutely necessary make sure the software resets the timer tick rate when the task finishes.

For more information, see the article at http://software.intel.com/en-us/articles/cpu-power-utilization-on-intel-architectures/.

13.5.6 Reducing Privileged Time

Applications spending significant time in privileged mode lead to excessive energy use due to various reasons. Some examples are: high system call rate and IO bottlenecks. You can use Windows Perfmon to get an estimate of privileged mode time.

A high system call rate, as measured by system calls per second, indicates that the software is causing frequent kernel mode transitions. That is, the application jumps from Ring3 - user mode to Ring0 - kernel mode, frequently. A very common example of this is using an expensive synchronization call such as the

Win32 API WaitForSingleObject(). This is a very important synchronization API, especially for inter-process communication. However, it enters kernel mode irrespective of whether the lock is achieved or not. For multi-threaded code with no or a short period contention on the lock, you can use EnterCriticalSection with a spin count. The advantage of this API over WaitForSingleObject() is that it does not enter kernel mode unless there is a contention on the lock. Hence, when there is no contention, EnterCriticalSection with spin count is much cheaper to use and reduces the time spent in privilege mode.

Studies were done by taking a small test application which has four active threads on an Intel microarchitecture code name Sandy Bridge-based system. The locks in the test case were implemented by using WaitForSingleObject and EnterCriticalSection. There was no contention on the lock, so each thread achieved the lock at the first attempt. As shown in the graph below, when there is no contention, using WaitForSingleObject() has negative impact on both power and performance as compared to using EnterCriticalSection().

As indicated in the following graph, using WaitForSingleObject() on an un-contended lock uses more power. Using EnterCriticalSection() provides a 10x performance gain and 60% energy reduction.

For more information see: http://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures/

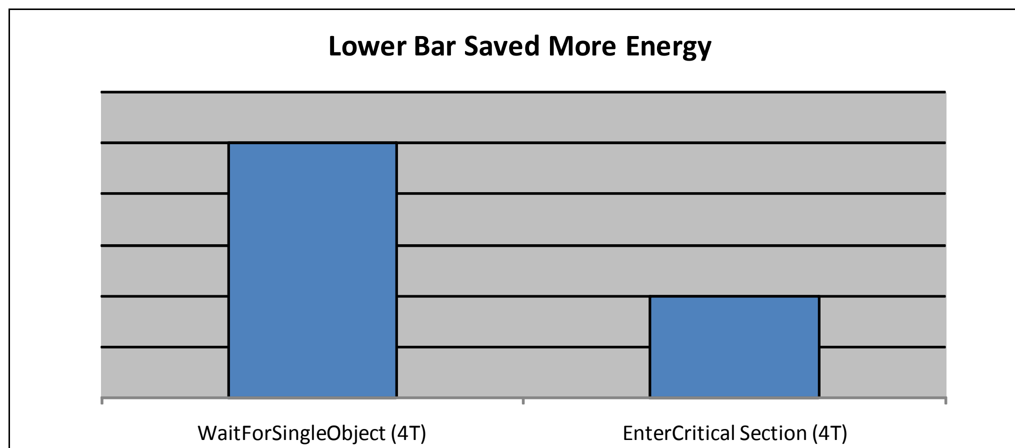


Figure 13-9. Energy Saving Comparison of Synchronization Primitives

13.5.7 Setting Context Awareness in the Code

Context awareness provides a way for software to save power when energy resources are limited. The following are some examples of context awareness that you can implement to conserve power:

- When running a game on laptop on battery power, change the frame rate to 60FPS instead of running uncapped.
- Dim the display when running on battery and not in use.
- Provide easy options for the end user to turn off devices such as wireless when not connected to network

Applications can do these changes transparently when running the game on battery power, or provide hints to users how to extend battery life. For either case, the application needs to be context aware to identify when battery power is in use as opposed to AC power.

A study done with two games running at different frame rates. The blue bar represents baseline default frame rate (uncapped) for these games. The brown line represents games running at 60FPS and the yellow line represents games running at 30FPS. This study shows that capping frame rate can help reduce power consumption

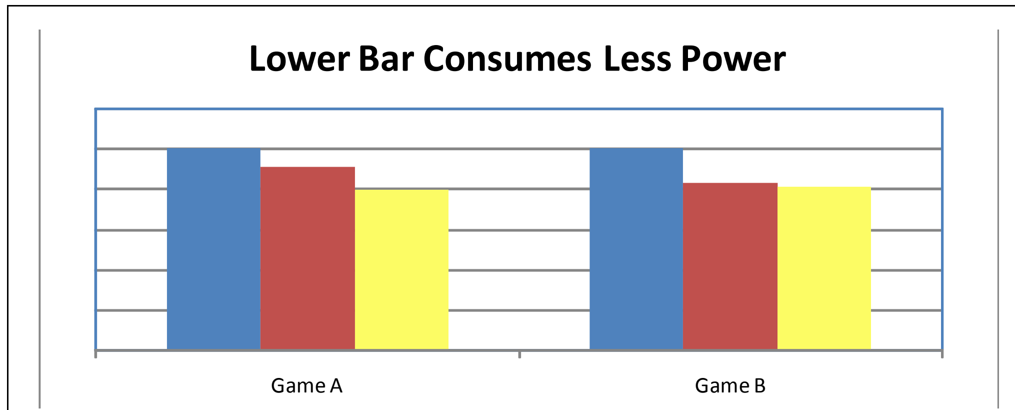


Figure 13-10. Power Saving Comparison of Power-Source-Aware Frame Rate Configurations

13.5.8 Saving Energy by Optimizing for Performance

As general rule, performance optimizations that reduce the number of cycles the CPU is busy running code also save energy consumption. Here are some examples of optimizing performance for specific microarchitectural features that produced energy savings.

The additional load port feature of Intel microarchitecture code name Sandy Bridge can save cycles, as demonstrated in Section 3.6.1. As a result the load port feature also saves power. For example, in an experiment with the kernel in Section 3.6.1.3, a sample application running on an engineering system with and without a bank conflict, the version without the bank conflict utilizes the second load port and provided a performance improvement along with 25mWHr energy savings.

Another features, the Decoded ICache and the LSD, cache micro-ops, hence eliminating power consumption by the decoders. For example, using the code alignment technique of arranging no more than three unconditional branches within an aligned 32 byte chunk (see Section 3.4.2.5) for switch statement, where we see 1.23x speedup, helps provide 1.12x power saving as well compared to the aligned version of dense unconditional branches in 32-byte chunk that can not fit the decoded ICache.

Using vectorization with Intel AVX allows processing of several elements in one cycle, hence reducing the overall processing cycles and saving energy. An example for the energy saving is shown in Figure 13-8.

13.6 PROCESSOR SPECIFIC POWER MANAGEMENT OPTIMIZATION FOR SYSTEM SOFTWARE

This section covers power management information that are processor specific. Specifically, they apply to second generation Intel Core processors based on Intel microarchitecture code name Sandy Bridge. These processors have CPUID DisplayFamily_DisplayModel signature of 06_2AH. These processor-specific capability may help system software to optimize/balance responsiveness and power consumption of the processor.

13.6.1 Power Management Recommendation of Processor-Specific Inactive State Configurations

Programming ACPI's _CST object with exit latency values appropriate to various inactive states will help OS power management to deliver optimal power reduction. Intel recommended values are model-specific.

Table 13-3 and Table 13-4 list Package C-State entry/exit latency for processors with CPUID DisplayFamily_DisplayModel signature of 06_2AH, and for two configurations of voltage regulator slew

rate capabilities. Table 13-3 applies to slow VR configuration, and Table 13-4 applies to fast VR configuration. For each configuration, the VR device can operate in either a fast interrupt break mode enabled or slow interrupt break mode, depending on the setting of MSR_POWER_CTL.[bit 4]. These C-Sate entry/exit latency are not processor specifications but estimates derived from empirical measurements. There may be some situations exit latency from a core is higher than those listed in Table 13-3 and Table 13-4.

Table 13-3. C-State Total Processor Exit Latency for Client Systems (Core+ Package Exit Latency) with Slow VR

C-State[1]	Typical Exit Latency [2]	Worst Case Exit Latency
	MSR_POWER_CTL MSR.[4] =0	MSR_POWER_CTL MSR.[4] =1
C1	1 µs	1 µs
C3	156 µs	80 µs
C6	181 µs	104 µs
C7	199 µs	109 µs

NOTES:

1. These C-State Entry/Exit Latencies are Intel estimates only and not processor specifications.
2. It is assumed that package is in C0 when one of the core is active.
3. Fast interrupt break mode is enabled if MSR_POWER_CTL.[4] = 1.
4. A device that connect to PCH may result in latencies equivalent to that of a slow interrupt break mode.

Table 13-4. C-State Total Processor Exit Latency for Client Systems (Core+ Package Exit Latency) with Fast VR

C-State[1]	Typical Worst Case Exit Latency Time (All Skus)[2]	
	MSR_POWER_CTL MSR.[4] =0	MSR_POWER_CTL MSR.[4] =1
C1	1 µs	1 µs
C3	156 µs	80 µs
C6	181 µs	104 µs
C7	199 µs	109 µs

NOTES:

1. These C-State Entry/Exit Latencies are Intel estimates only and not processor specifications.
2. It is assumed that package is in C0 when one of the core is active.
3. If the package is in a deeper C-states, the exit latency of Local APIC timer wake up depends on the typical core level exit latency; If the package is in C0, it may vary between typical or worst case of the respective core-level exit latency.

Table 13-5 lists Core-only C-State entry/exit latency for processors with CPUID DisplayFamily_DisplayModel signature of 06_2AH, and for two configurations of voltage regulator slew rate capabilities. Core-only exit latency is not affected by MSR_POWER_CTL.[4].

Table 13-5. C-State Core-Only Exit Latency for Client Systems with Slow VR

C-State[1]	Typical Worst Case Exit Latency Time (All Skus)[2]	
C1	1 µs	1 µs
C3	21 µs	240 µs
C6	46 µs	250 µs
C7	46 µs	250 µs

NOTES:

1. These C-State Entry/Exit Latencies are Intel estimates only and not processor specifications.

2. A slow VR device refers to a device with ramp time of 10 mv/µs in fast mode and 2.5 mv/µs in slow mode.

13.6.1.1 Balancing Power Management and Responsiveness of Inactive To Active State Transitions

MSR_PKGC3_IRTL, MSR_PKGC6_IRTL, MSR_PKGC7_IRTL provide processor-specific interfaces for system software to balance power consumption and system responsiveness. System software may change budgeted values from a package inactive states to C0 during runtime to accommodate system specific requirements. For example, more aggressive timings when on battery vs. on AC power.

The exit latency is greatly impacted by the VR swing rate. Table 13-5 specifies the total interrupt response times per state (including the core component) for a "fast" exit rate (the default recommended configuration in the PCH and CPU for all events except the internal HPET and CPU timers).

Selecting a "slow" rate by the BIOS (disable the fast break event method in the POWER_CTL MSR bit 4) for various events from the PCIE will require extending the above budget respectively. Otherwise CPU may select a shallower PKG_Cstate to still meet the budget at a much slower VR swing rate.

Selecting a "slow" exit rate for various PCH-connected devices (PCH BIOS setting) will not be visible to the above latency calculation mechanism and thus result in the CPU not meeting the required latency goals.

Table 13-6. POWER_CTL MSR in Next Generation Intel Processor (Intel® Microarchitecture Code Name Sandy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1FCH	508	MSR_POWER_CTL	Core	Power Control Register
		3:0		Reserved.
		4		**FAST_Brk_Int_En.** When set to 1, enables the voltage regulator for fast slope for PCI-E interrupt wakeup events.
		63:5		Reserved.

CHAPTER 14
INTEL® ATOM™ MICROARCHITECTURE AND SOFTWARE OPTIMIZATION

14.1 OVERVIEW

45 nm Intel Atom processors introduced Intel Atom microarchitecture. The same microarchitecture also used in 32 nm Intel Atom processors. This chapter covers a brief overview the Intel Atom microarchitecture, and specific coding techniques for software whose primary targets are processors based on the Intel Atom microarchitecture. The key features of Intel Atom processors to support low power consumption and efficient performance include:

- Enhanced Intel SpeedStep® Technology enables operating system (OS) to program a processor to transition to lower frequency and/or voltage levels while executing a workload.
- Support deep power down technology to reduces static power consumption by turning off power to cache and other sub-systems in the processor.
- Intel Hyper-Threading Technology providing two logical processor for multi-tasking and multi-threading workloads.
- Support Single-instruction multiple-data extensions up to SSE3 and SSSE3.
- Support for Intel 64 and IA-32 architecture.

The Intel Atom microarchitecture is designed to support the general performance requirements of modern workloads within the power-consumption envelop of small form-factor and/or thermally-constrained environments.

14.2 INTEL® ATOM™ MICROARCHITECTURE

Intel Atom microarchitecture achieves efficient performance and low power operation with a two-issue wide, in-order pipeline that support Hyper-Threading Technology. The in-order pipeline differs from out-of-order pipelines by treating an IA-32 instruction with a memory operand as a single pipeline operation instead of multiple micro-operations.

The basic block diagram of the Intel Atom microarchitecture pipeline is shown in Figure 14-1.

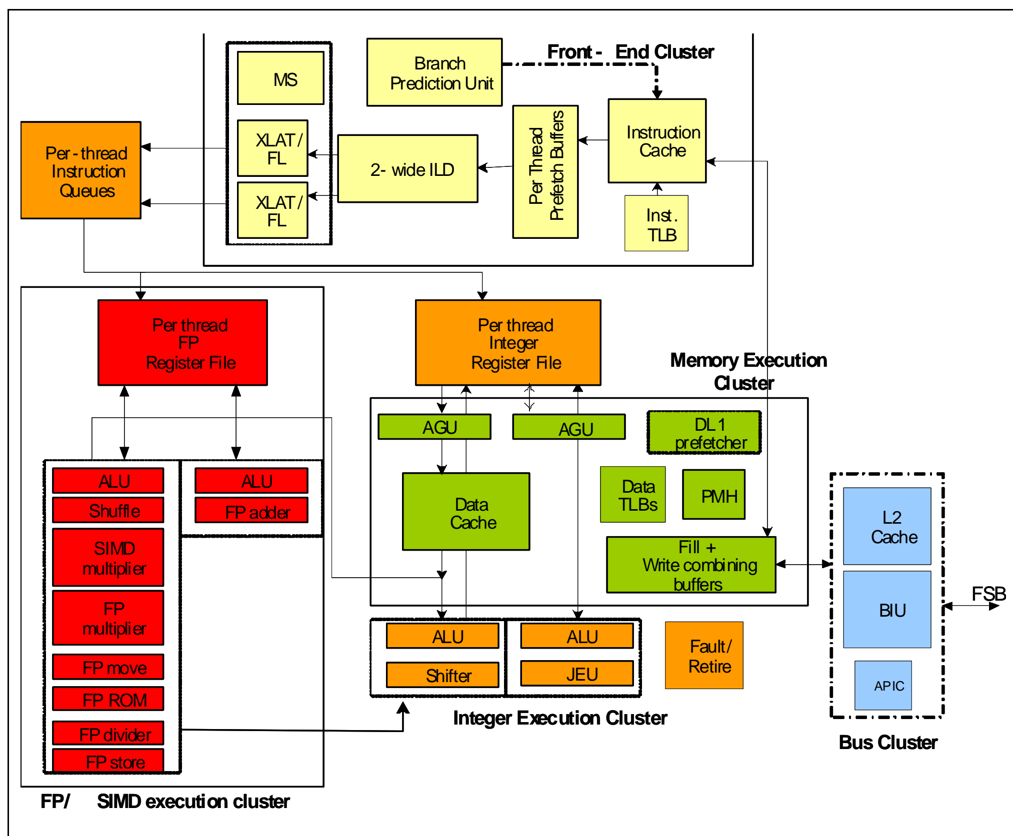


Figure 14-1. Intel Atom Microarchitecture Pipeline

The front end features a power-optimized pipeline, including:

- 32KB, 8-way set associative, first-level instruction cache.
- Branch prediction units and ITLB.
- Two instruction decoders, each can decode up to one instruction per cycle.

The front end can deliver up to two instructions per cycle to the instruction queue for scheduling. The scheduler can issue up to two instructions per cycle to the integer or SIMD/FP execution clusters via two issue ports.

Each of the two issue ports can dispatch an instruction per cycle to the integer cluster or the SIMD/FP cluster to execute. The port-bindings of the integer and SIMD/FP clusters have the following features:

- Integer execution cluster:
  - Port 0: ALU0, Shift/Rotate unit, Load/Store.
  - Port 1: ALU1, Bit processing unit, jump unite and LEA.
  - Effective “load-to-use” latency of 0 cycle.
- SIMD/FP execution cluster:
  - Port 0: SIMD ALU, Shuffle unit, SIMD/FP multiply unit, Divide unit, (support IMUL, IDIV).
  - Port 1: SIMD ALU, FP Adder.
  - The two SIMD ALUs and the shuffle unit in the SIMD/FP cluster are 128-bit wide, but 64-bit integer SIMD computation is restricted to port 0 only.
  - FP adder can execute ADDPS/SUBPS in 128-bit data path, data path for other FP add operations are 64-bit wide.

— Safe Instruction Recognition algorithm for FP/SIMD execution allow younger, short-latency integer instruction to execute without being blocked by older FP/SIMD instruction that might cause exception.

— FP multiply pipe also supports memory loads.

— FP ADD instructions with memory load reference can use both ports to dispatch.

The memory execution sub-system (MEU) can support 48-bit linear address for Intel 64 Architecture, either 32-bit or 36-bit physical addressing modes. The MEU provides:

- 24KB first level data cache.
- Hardware prefetching for L1 data cache.
- Two levels of DTLB for 4KByte and larger paging structure.
- Hardware pagewalker to service DTLB and ITLB misses.
- Two address generation units (port 0 supports loads and stores, port 1 supports LEA and stack operations).
- Store-forwarding support for integer operations.
- 8 write combining buffers.

The bus logic sub-system provides:

- 512KB, 8-way set associative, unified L2 cache.
- Hardware prefetching for L2 and interface logic to the front side bus.

14.2.1 Hyper-Threading Technology Support in Intel® Atom™ Microarchitecture

The instruction queue is statically partitioned for scheduling instruction execution from two threads. The scheduler is able to pick one instruction from either thread and dispatch to either of port 0 or port 1 for execution. The hardware makes selection choice on fetching/decoding/dispatching instructions between two threads based on criteria of fairness as well as each thread's readiness to make forward progress.

14.3 CODING RECOMMENDATIONS FOR INTEL® ATOM™ MICROARCHITECTURE

Instruction scheduling heuristics and coding techniques that apply to out-of-order microarchitectures may not deliver optimal performance on an in-order microarchitecture. Likewise instruction scheduling heuristics and coding techniques for an in-order pipeline like Intel Atom microarchitecture may not achieve optimal performance on out-of-order microarchitectures. This section covers specific coding recommendations for software whose primary deployment targets are processors based on Intel Atom microarchitecture.

14.3.1 Optimization for Front End of Intel® Atom™ Microarchitecture

The two decoders in the front end of Intel Atom microarchitecture can handle most instructions in the Intel 64 and IA-32 architecture. Some instructions dealing with complicated operations require the use of an MSROM in the front end. Instructions that go through the two decoders generally can be decoded by either decoder unit of the front end in most cases. Instructions the must use the MSROM or conditions that cause the front end to re-arrange decoder assignments will experience a delay in the front end.

Software can use specific performance monitoring events to detect instruction sequences and/or conditions that cause front end to re-arrange decoder assignment.

Assembly/Compiler Coding Rule 1. (MH impact, ML generality) *For Intel Atom processors, minimize the presence of complex instructions requiring MSROM to take advantage the optimal decode bandwidth provided by the two decode units.*

Using the performance monitoring events "MACRO_INSTS.NON_CISC_DECODED" and "MACRO_INSTS.CISC_DECODED" can be used to evaluate the percentage instructions in a workload that required MSROM.

Assembly/Compiler Coding Rule 2. (M impact, H generality) *For Intel Atom processors, keeping the instruction working set footprint small will help the front end to take advantage the optimal decode bandwidth provided by the two decode units.*

Assembly/Compiler Coding Rule 3. (MH impact, ML generality) *For Intel Atom processors, avoiding back-to-back X87 instructions will help the front end to take advantage the optimal decode bandwidth provided by the two decode units.*

Using the performance monitoring events "DECODE_RESTRICTION" can count the number of occurrences in a workload that encountered delays causing reduction of decode throughput.

In general the front end restrictions are not typical a performance limiter until the retired "cycle per instruction" becomes less than unity (maximum theoretical retirement throughput corresponds to CPI of 0.5). To reach CPI below unity, it is important to generate instruction sequences that go through the front end as instruction pairs decodes in parallel by the two decoders. After the front end, the scheduler and execution hardware do not need to dispatch the decode pairings through port 0 and port 1 in the same order.

The decoders cannot decode past a jump instruction, so jumps should be paired as the second instruction in a decoder-optimized pairing. The front end can only handle one X87 instruction per cycle, and only decoder unit 0 can request a transfer to use MSROM. Instructions that are longer than 8 bytes or having more than three prefixes will results in a MSROM transfer, experiencing two cycles of delay in the front end.

Instruction lengths and alignment can impact decode throughput. The prefetching buffers inside the front end imposes a throughput limit that if the number of bytes being decoded in any 7-cycle window exceeds 48 bytes, the front end will experience a delay to wait for a buffer. Additionally, every time an instruction pair crosses 16 byte boundary, it requires the front end buffer to be held on for at least one more cycle. So instruction alignment crossing 16 byte boundary is highly problematic.

Instruction alignment can be improved using a combination of an ignore prefix and an instruction.

Example 14-1. Instruction Pairing and Alignment to Optimize Decode Throughput on Intel® Atom™ Microarchitecture

Address	Instruction Bytes	Disassembly
7FFFFDF0	0F594301	mulps xmm0, [ebx+ 01h]
7FFFFDF4	8341FFFF	add dword ptr [ecx-01h], -1
7FFFFDF8	83C2FF	add edx, , -1
7FFFFDFB	64	; FS prefix override is ignored, improves code alignment
7FFFFDFC	F20f58E4	add xmm4, xmm4
7FFFFE00	0F594B11	mulps xmm1, [ebx+ 11h]
7FFFFE04	8369EFFF	sub dword ptr [ecx- 11h], -1
7FFFFE08	83EAFF	sub edx, -1
7FFFFE0B	64	; FS prefix override is ignored, improves code alignment
7FFFFE0C	F20F58ED	addsd xmm5, xmm5
7FFFFE10	0F595301	mulps xmm2, [ebx +1]

Example 14-1. Instruction Pairing and Alignment to Optimize Decode Throughput on Intel® Atom™ Microarchitecture

7FFFFE14	8341DFFF	add dword ptr [ecx-21H], -1
7FFFFE18	83C2FF	add edx, -1
7FFFFE1B	64	; FS prefix override is ignored, improves code alignment
7FFFFE1C	F20F58F6	addssd xmm6, xmm6
7FFFFE20	0F595B11	mulps xmm3, [ebx+ 11h]
7FFFFE24	8369CFFF	sub dword ptr [ecx- 31h], -1
7FFFFE28	83EAFF	sub edx, -1

When a small loop contains some long-latency operation inside, loop unrolling may be considered as a technique to find adjacent instruction that could be paired with the long-latency instruction to enable that adjacent instruction to make forward progress. However, loop unrolling must also be evaluated on its impact to increased code size and pressure to the branch target buffer.

The performance monitoring event "BACLEARS" can provide a means to evaluate whether loop unrolling is helping or hurting front end performance. Another event "ICACHE_MISSES" can help evaluate if loop unrolling is increasing the instruction footprint.

Branch predictors in Intel Atom processor do not distinguish different branch types. Sometimes mixing different branch types can cause confusion in the branch prediction hardware.

The performance monitoring event "BR_MISSP_TYPE_RETIRED" can provide a means to evaluate branch prediction issues due to branch types.

14.3.2 Optimizing the Execution Core

This section covers several items that can help software use the two-issue-wide execution core to make forward progress with two instructions more frequently.

14.3.2.1 Integer Instruction Selection

In an in-order machine, instruction selection and pairing can have an impact on the machine's ability to discover instruction-level-parallelism for instructions that have data ready to execute. Some examples are:

- **EFLAG**: The consumer instruction of any EFLAG flag bit can not be issued in the same cycle as the producer instruction of the EFLAG register. For example, ADD could modify the carry bit, so it is a producer; JC (or ADC) reads the carry bit and is a consumer.
  - — Conditional jumps are able to issue in the following cycle after the consumer.
  - — A consumer instruction of other EFLAG bits must wait one cycle to issue after the producer (two cycle delay).

Assembly/Compiler Coding Rule 4. (M impact, H generality) *For Intel Atom processors, place a MOV instruction between a flag producer instruction and a flag consumer instruction that would have incurred a two-cycle delay. This will prevent partial flag dependency.*

- **Long-latency Integer Instructions**: They will block shorter latency instruction on the same thread from issuing (required by program order). Additionally, they will also block shorter-latency instruction on both threads for one cycle to resolve writeback resource.
- **Common Destination**: Two instructions that produce results to the same destination can not issue in the same cycle.
- **Expensive Instructions**: Some instructions have special requirements and become expensive in consuming hardware resources for an extended period during execution. It may be delayed in execution until it is the oldest in the instruction queue; it may delay the issuing of other younger

instructions. Examples of these include FDIV, instructions requiring execution units from both ports, etc.

14.3.2.2 Address Generation

The hardware optimizes the general case of instruction ready to execute must have data ready, and address generation precedes data being ready. If address generation encounters a dependency that needs data from another instruction, this dependency in address generation will incur a delay of 3 cycles.

The address generation unit (AGU) may be used directly in three situations that affect execution throughput of the two-wide machine. The situations are:

- **Implicit ESP updates**: When the ESP register is not used as the destination of an instruction (explicit ESP updates), an implicit ESP update will occur with instructions like PUSH, POP, CALL, RETURN. Mixing explicit ESP updates and implicit ESP updates will also lead to dependency between address generation and data execution.
- **LEA**: The LEA instruction uses the AGU instead of the ALU. If one of the source register of LEA must come from an execution unit. This dependency will also cause a 3 cycle delay. Thus, LEA should not be used in the technique of adding two values and produce the result in a third register. LEA should be used for address computation.
- **Integer-FP/SIMD transfer**: Instructions that transfer integer data to the FP/SIMD side of the machine also uses AGU. Examples of these instructions include MOVD, PINSRW. If one of the source register of these instructions depends on the result of an execution unit, this dependency will also cause a delay of 3 cycles.

Example 14-2. Alternative to Prevent AGU and Execution Unit Dependency

```
a) Three cycle delay when using LEA in ternary operations
        mov eax, 0x01
        lea eax, 0x8000[eax+ebp]; values in eax comes from execution of previous instruction
        ; 3 cycle delay due to lea and execution dependency

b) Dependency handled in execution, avoiding AGU and execution dependency
        mov eax, 0x01
        add eax, 0x8000
        add eax, ebp
```

Assembly/Compiler Coding Rule 5. (MH impact, H generality) *For Intel Atom processors, LEA should be used for address manipulation; but software should avoid the following situations which creates dependencies from ALU to AGU: an ALU instruction (instead of LEA) for address manipulation or ESP updates; a LEA for ternary addition or non-destructive writes which do not feed address generation. Alternatively, hoist producer instruction more than 3 cycles above the consumer instruction that uses the AGU.*

14.3.2.3 Integer Multiply

Integer multiply instruction takes several cycles to execute. They are pipelined such that an integer multiply instruction and another long-latency instruction can make forward progress in the execution phase. However, integer multiply instructions will block other single-cycle integer instructions from issuing due to requirement of program order.

Assembly/Compiler Coding Rule 6. (M impact, M generality) *For Intel Atom processors, sequence an independent FP or integer multiply after an integer multiply instruction to take advantage of pipelined IMUL execution.*

Example 14-3. Pipeling Instruction Execution in Integer Computation

```
a) Multi-cycle Imul instruction can block 1-cycle integer instruction
        imul eax, eax
        add ecx, ecx ; 1 cycle int instruction blocked by imul for 4 cycles
        imul ebx, ebx ; instruction blocked by in-orer issue

b) Back-to-back issue of independent imul are pipelined
        imul eax, eax
        imul ebx, ebx ; 2nd imul can issue 1 cycle later
        add ecx, ecx ; 1 cycle int instruction blocked by imul
```

14.3.2.4 Integer Shift Instructions

Integer shift instructions that encodes shift count in the immediate byte have one-cycle latency. In contrast, shift instructions using shift count in the ECX register may need to wait for the register count are updated. Thus shift instruction using register count has 3-cycle latency.

Assembly/Compiler Coding Rule 7. (M impact, M generality) *For Intel Atom processors, hoist the producer instruction for the implicit register count of an integer shift instruction before the shift instruction by at least two cycles.*

14.3.2.5 Partial Register Access

Although partial register access does not cause additional delay, the in-order hardware tracks dependency on the full register. Thus 8-bit registers like AL and AH are not treated as independent registers. Additionally some instructions like LEA, vanilla loads, and pop are slower when the input is smaller than 4 bytes.

Assembly/Compiler Coding Rule 8. (M impact, MH generality) *For Intel Atom processors, LEA, simple loads and POP are slower if the input is smaller than 4 bytes.*

14.3.2.6 FP/SIMD Instruction Selection

Table 14-1 summarizes the characteristics of various execution units in Intel Atom microarchitecture that are likely used most frequently by software.

Table 14-1. Instruction Latency/Throughput Summary of Intel® Atom™ Microarchitecture

Instruction Category	Latency (cycles)	Throughput	# of Execution Unit
SIMD Integer ALU			
128-bit ALU/logical/move	1	1	2
64-bit ALU/logical/move	1	1	2
SIMD Integer Shift			
128-bit	1	1	1

Table 14-1. Instruction Latency/Throughput Summary of Intel® Atom™ Microarchitecture (Contd.)

64-bit	1	1	1
SIMD Shuffle			
128-bit	1	1	1
64-bit	1	1	1
SIMD Integer Multiply			
128-bit	5	2	1
64-bit	4	1	1
FP Adder			
X87 Ops (FADD)	5	1	1
Scalar SIMD (addsd, addss)	5	1	1
Packed single (addps)	5	1	1
Packed double (addpd)	6	5	1
FP Multiplier			
X87 Ops (FMUL)	5	2	1
Scalar single (mulss)	4	1	1
Scalar double (mulsd)	5	2	1
Packed single (mulps)	5	2	1
Packed double (mulpd)	9	9	1
IMUL			
IMUL r32, r/m32	5	1	1
IMUL r12, r/m16	6	1	1

SIMD/FP instruction selection generally should favor shorter latency first, then favor faster throughput alternatives whenever possible. Note that packed double-precision instructions are not pipelined, using two scalar double-precision instead can achieve higher performance in the execution cluster.

Assembly/Compiler Coding Rule 9. (MH impact, H generality) *For Intel Atom processors, prefer SIMD instructions operating on XMM register over X87 instructions using FP stack. Use Packed single-precision instructions where possible. Replace packed double-precision instruction with scalar double-precision instructions.*

Assembly/Compiler Coding Rule 10. (M impact, ML generality) *For Intel Atom processors, library software performing sophisticated math operations like transcendental functions should use SIMD instructions operating on XMM register instead of native X87 instructions.*

Assembly/Compiler Coding Rule 11. (M impact, M generality) *For Intel Atom processors, enable DAZ and FTZ whenever possible.*

Several performance monitoring events may be useful for SIMD/FP instruction selection tuning: “SIMD_INST_RETIRED.{PACKED_SINGLE, SCALAR_SINGLE, PACKED_DOUBLE, SCALAR_DOUBLE}” can be used to determine the instruction selection in the program. “FP_ASSIST” and “SIR” can be used to see if floating exceptions (or false alarms) are impacting program performance.

The latency and throughput of divide instructions vary with input values and data size. Intel Atom micro-architecture implements a radix-2 based divider unit. So, divide/sqrt latency will be significantly longer than other FP operations. The issue throughput rate of divide/sqrt will be correspondingly lower. The divide unit is shared between two logical processors, so software should consider all alternatives to using the divide instructions.

Assembly/Compiler Coding Rule 12. (H impact, L generality) *For Intel Atom processors, use divide instruction only when it is absolutely necessary, and pay attention to use the smallest data size operand.*

The performance monitoring events “DIV” and “CYCLES_DIV_BUSY” can be used to see if the divides are a bottleneck in the program.

FP operations generally have longer latency than integer instructions. Writeback of results from FP operation generally occur later in the pipe stages than integer pipeline. Consequently, if an instruction has dependency on the result of some FP operation, there will be a two-cycle delay. Examples of these type of instructions are FP-to-integer conversions CVTxx2xx, MOVD from XMM to general purpose registers.

In situations where software needs to do computation with consecutive groups 4 single-precision data elements, PALIGNR+MOVAPS is preferred over MOVUPS. Loading 4 data elements with unconstrained array index *k*, such as MOVUPS xmm1, _pArray[k], where the memory address _pArray is aligned on 16-byte boundary, will periodically causing cache line split, incurring a 14-cycle delay.

The optimal approach is for each k that is not a multiple of 4, round down k to multiples of 4 with j = 4*(k/4), do a MOVAPS MOVAPS xmm1, _pArray[j] and MOVAPS xmm1, _pArray[j+4], and use PALIGNR to splice together the four data elements needed for computation.

Assembly/Compiler Coding Rule 13. (MH impact, M generality) *For Intel Atom processors, prefer a sequence MOVAPS+PALIGN over MOVUPS. Similarly, MOVDQA+PALIGNR is preferred over MOVDQU.*

14.3.3 Optimizing Memory Access

This section covers several items that can help software optimize the performance of the memory sub-system.

Memory access to system memory of cache access that encounter certain hazards can cause the memory access to become an expensive operation, blocking short-latency instructions to issue even when they have data ready to execute.

The performance monitoring events “REISSUE” can be used to assess the impact of re-issued memory instructions in the program.

14.3.3.1 Store Forwarding

In a few limited situations, Intel Atom microarchitecture can forward data from a preceding store operation to a subsequent load instruction. The situations are:

- Store-forwarding is supported only in the integer pipeline, and does not apply to FP nor SIMD data. Furthermore, the following conditions must be met:
  a. The store and load operations must be of the same size and to the same address.
  b. Data size larger than 8 bytes do not forward from a store operation.
- When data forwarding proceeds, data is forwarded base on the least significant 12 bits of the address. So software must avoid the address aliasing situation of storing to an address and then loading from another address that aliases in the lowest 12-bits with the store address.

14.3.3.2 First-level Data Cache

Intel Atom microarchitecture handles each 64-byte cache line of the first-level data cache in 16 4-byte chunks. This implementation characteristic has a performance impact to data alignment and some data access patterns.

Assembly/Compiler Coding Rule 14. (MH impact, H generality) *For Intel Atom processors, ensure data are aligned in memory to its natural size. For example, 4-byte data should be aligned to 4-byte boundary, etc. Additionally, smaller access (less than 4 bytes) within a chunk may experience delay if they touch different bytes.*

14.3.3.3 Segment Base

In Intel Atom microarchitecture, the address generation unit assumes that the segment base will be 0 by default. Non-zero segment base will cause load and store operations to experience a delay.

- If the segment base isn't aligned to a cache line boundary, the max throughput of memory operations is reduced to one very 9 cycles.

If the segment base is non-zero but cache line aligned the penalty varies by segment base.

- DS will have a max throughput of one every two cycles.
- FS, and GS will have a max throughput of one every two cycles. However, FS and GS are anticipated to be used only with non-zero bases and therefore have a max throughput of one every two cycles even if the segment base is zero.
- ES:
  — If used as the implicit segment base for the destination of string operation, will have a max throughput of one every two cycles for non-zero but cacheline aligned bases.
  — Otherwise, only do one operation every nine cycles.
- CS and SS will always have a max throughput of one every nine cycles if its segment base is non-zero but cache line aligned.

Assembly/Compiler Coding Rule 15. (H impact, ML generality) *For Intel Atom processors, use segments with base set to 0 whenever possible; avoid non-zero segment base address that is not aligned to cache line boundary at all cost.*

Assembly/Compiler Coding Rule 16. (H impact, L generality) *For Intel Atom processors, when using non-zero segment bases, Use DS, FS, GS; string operation should use implicit ES.*

Assembly/Compiler Coding Rule 17. (M impact, ML generality) *For Intel Atom processors, favor using ES, DS, SS over FS, GS with zero segment base.*

14.3.3.4 String Moves

Using MOVS/STOS instruction and REP prefix on Intel Atom processor should recognize the following items:

- For small count values, using REP prefix is less efficient than not using REP prefix. This is because the hardware does have small REP count optimization.
- For small count values, using REP prefix is less efficient than not using REP prefix. This is because the hardware does have small REP count optimization.

- For large count values, using REP prefix will be less efficient than using 16-byte SIMD instructions.
- Incrementing address in loop iterations should favor LEA instruction over explicit ADD instruction.
- If data footprint is such that memory operation is accessing L2, use of software prefetch to bring data to L1 can avoid memory operation from being re-issued.
- If string/memory operation is accessing system memory, using non-temporal hints of streaming store instructions can avoid cache pollution.

Example 14-4. Memory Copy of 64-byte

```
T1:     prefetcht0 [eax+edx+0x80] ; prefetch ahead by two iterations
        movdqa  xmm0, [eax+ edx] ; load data from source (in L1 by prefetch)
        movdqa  xmm1, [eax+ edx+0x10]
        movdqa  xmm2, [eax+ edx+0x20]
        movdqa  xmm3, [eax+ edx+0x30]
        movdqa  [ebx+ edx], xmm0; store data to destination
        movdqa  [ebx+ edx+0x10], xmm1
        movdqa  [ebx+ edx+0x30], xmm2
        movdqa  [ebx+ edx+0x30], xmm3
        lea     edx, 0x40 ; use LEA to adjust offset address for next iteration
        dec     ecx
        jnz     T1
```

14.3.3.5 Parameter Passing

Due to the limited situations of load-to-store forwarding support in Intel Atom microarchitecture, parameter passing via the stack places restrictions on optimal usage by the callee function. For example, “bool” and “char” data usually are pushed onto the stack as 32-bit data, a callee function that reads “bool” or “char” data off the stack will face store-forwarding delay and causing the memory operation to be re-issued.

Compiler should recognize this limitation and generate prolog for callee function to read 32-bit data instead of smaller sizes.

Assembly/Compiler Coding Rule 18. (MH impact, M generality) *For Intel Atom processors, “bool” and “char” value should be passed onto and read off the stack as 32-bit data.*

14.3.3.6 Function Calls

In Intel Atom microarchitecture, using PUSH/POP instructions to manage stack space and address adjustment between function calls/returns will be more optimal than using ENTER/LEAVE alternatives. This is because PUSH/POP will not need MSROM flows and stack pointer address update is done at AGU.

When a callee function need to return to the caller, the callee could issue POP instruction to restore data and restore the stack pointer from the EBP.

Assembly/Compiler Coding Rule 19. (MH impact, M generality) *For Intel Atom processors, favor register form of PUSH/POP and avoid using LEAVE; Use LEA to adjust ESP instead of ADD/SUB.*

14.3.3.7 Optimization of Multiply/Add Dependent Chains

Computations of dependent multiply and add operations can illustrate the usage of several coding techniques to optimize for the front end and in-order execution pipeline of the Intel Atom microarchitecture.

Example 14-5a shows a code sequence that may be used on out-of-order microarchitectures. This sequence is far from optimal on Intel Atom microarchitecture. The full latency of multiply and add operations are exposed and it is not very successful at taking advantage of the two-issue pipeline.

Example 14-5b shows an improved code sequence that takes advantage of the two-issue in-order pipeline of Intel Atom microarchitecture. Because the dependency between multiply and add operations are present, the exposure of latency are only partially covered.

Example 14-5. Examples of Dependent Multiply and Add Computation

a) Instruction sequence that encounters stalls

```
; accumulator xmm2 initialized
Top:    movaps xmm0, [esi] ; vector stored in 16-byte aligned memory
        movaps xmm1, [edi] ; vector stored in 16-byte aligned memory
        mulps xmm0, xmm1
        addps xmm2, xmm0 ; dependency and branch exposes latency of mul and add
        add esi, 16 ;
        add edi, 16
        sub ecx, 1
        jnz top
```

b) Improved instruction sequence to increase execution throughput

```
; accumulator xmm4 initialized
Top:    movaps xmm0, [esi] ; vector stored in 16-byte aligned memory
        lea esi, [esi+16] ; can schedule in parallel with load
        mulps xmm0, [edi] ;
        lea edi, [edi+16] ; can schedule in parallel with multiply
        addps xmm4, xmm0 ; latency exposures partially covered by independent instructions
        dec ecx ;
        jnz top
```

c) Improving instruction sequence further by unrolling and interleaving

```
; accumulator xmm0, xmm1, xmm2, xmm3 initialized
Top:    movaps xmm0, [esi] ; vector stored in 16-byte aligned memory
        lea esi, [esi+16] ; can schedule in parallel with load
        mulps xmm0, [edi] ;
        lea edi, [edi+16] ; can schedule in parallel with multiply
        addps xmm5, xmm1 ; dependent multiply hoisted by unrolling and interleaving
        movaps xmm1, [esi] ; vector stored in 16-byte aligned memory
        lea esi, [esi+16] ; can schedule in parallel with load
        mulps xmm1, [edi] ;
        lea edi, [edi+16] ; can schedule in parallel with multiply
        addps xmm6, xmm2 ; dependent multiply hoisted by unrolling and interleaving
```

(continue)

Example 14-5. Examples of Dependent Multiply and Add Computation (Contd.)

```
        movaps xmm2, [esi] ; vector stored in 16-byte aligned memory
        lea esi, [esi+16] ; can schedule in parallel with load
        mulps xmm2, [edi] ;
        lea edi, [edi+16] ; can schedule in parallel with multiply
        addps xmm7, xmm3 ; dependent multiply hoisted by unrolling and interleaving
        movaps xmm3, [esi] ; vector stored in 16-byte aligned memory
        lea esi, [esi+16] ; can schedule in parallel with load
        mulps xmm3, [edi] ;
        lea edi, [edi+16] ; can schedule in parallel with multiply
        addps xmm4, xmm0 ; dependent multiply hoisted by unrolling and interleaving
        sub ecx, 4;
        jnz top
        ; sum up accumulators xmm0, xmm1, xmm2, xmm3 to reduce dependency inside the loop
```

Example 14-5c illustrates a technique that increases instruction-level parallelism and further reduces latency exposures of the multiply and add operations. By unrolling four times, each ADDPS instruction can be hoisted far from its dependent producer instruction MULPS. Using an interleaving technique, non-dependent ADDPS and MULPS can be placed in close proximity. Because the hardware that executes MULPS and ADDPS is pipelined, the associated latency can be covered much more effectively by this technique relative to Example 14-5b.

14.3.3.8 Position Independent Code

Position independent code often needs to obtain the value of the instruction pointer. Example 14-5a show one technique to put the value of IP into the ECX register by issuing a CALL without a matching RET. Example 14-5b show an alternative technique to put the value of IP into the ECX register using a matched pair of CALL/RET.

Example 14-6. Instruction Pointer Query Techniques

```
a) Using call without return to obtain IP
        call _label; return address pushed is the IP of next instruction
_label:
        pop ECX; IP of this instruction is now put into ECX
```

```
b) Using matched call/ret pair

        call _lblcx;
        ... ; ECX now contains IP of this instruction
        ...
_lblcx
        mov ecx, [esp];
        ret
```

14.4 INSTRUCTION LATENCY

This section lists the port-binding and latency information of Intel Atom microarchitecture. The port-binding information for each instruction may show one of 3 situations:

- 'Single digit' - the specific port that must be issued.
- (0, 1) - either port 0 or port 1.
- 'B' - both ports are required.

In the "Instruction" column:

- If different operand syntax of the same instruction have the same port-binding and latency, operand syntax is omitted.
- When different operand syntax may produce different latency or port binding, the operand syntax is listed; but instruction syntax of different operand sizes may be compacted and abbreviated with a footnote.

Instruction that required decoder assistance from MSROM are marked in the "Comment" column (should be used minimally if more decode-efficient alternatives are available).

Table 14-2. Intel® Atom™ Microarchitecture Instructions Latency Data

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
ADD/AND/CMP/OR/SUB/XOR/TEST[1] (E)AX/AL, imm;	(0, 1)	1	0.5
ADD/AND/CMP/OR/SUB/XOR[2] mem, Imm8; ADD/AND/CMP/OR/SUB/XOR/TEST[4] mem, imm; TEST m8, imm8	0	1	1
ADD/AND/CMP/OR/SUB/XOR/TEST[2] mem, reg; ADD/AND/CMP/OR/SUB/XOR[2] reg, mem;	0	1	1
ADD/AND/CMP/OR/SUB/XOR[2] reg, Imm8; ADD/AND/CMP/OR/SUB/XOR[4] reg, imm	(0, 1)	1	0.5
ADDPD/ADDSUBPD/MAXPD/MAXPS/MINPD/MINPS/SUBPD xmm, mem	B	7	6
ADDPD/ADDSUBPD/MAXPD/MAXPS/MINPD/MINPS/SUBPD xmm, xmm	B	6	5
ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/SUBSS xmm, mem	B	5	1
ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/SUBSS xmm, xmm	1	5	1
ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS xmm, mem	0	1	1
ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS xmm, xmm	(0, 1)	1	1
BSF/BSR r16, m16	B	17	16
BSF/BSR[3] reg, mem	B	16	15
BSF/BSR[4] reg, reg	B	16	15
BT m16, imm8; BT[3] mem, imm8	(0, 1)	2; 1	1
BT m16, r16; BT[3] mem, reg	B	10, 9	8
BT[4] reg, imm8; BT[4] reg, reg	1	1	1
BTC m16, imm8; BTC[3] mem, imm8	B	3; 2	2
BTC/BTR/BTS m16; r16	B	12	11
BTC/BTR/BTS[3] mem, reg	B	11	10
BTC/BTR/BTS[4] reg, imm8; BTC/BTR/BTS[4] reg, reg	1	1	1
CALL mem	(0, 1)	2	2

Table 14-2. Intel® Atom™ Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
CALL reg; CALL rel16; CALL rel32	B	1	1
CMOV[4] reg, mem; MOV[1] (E)AX/AL, MOFFS; MOV[2] mem, imm	0	1	1
CMOV[4] reg, reg; MOV[2] reg, imm; MOV[2] reg, reg; ; SETcc r8	(0, 1)	1	0.5
CMPPD/CMPPS xmm, mem, imm; CVTTPS2DQ xmm, mem	B	7	6
CMPPD/CMPPS xmm, xmm, imm; CVTTPS2DQ xmm, xmm	B	6	5
CMPSD/CMPSS xmm, mem, imm	B	5	1
CMPSD/CMPSS xmm, xmm, imm	1	5	1
(U)COMISD/(U)COMISS xmm, mem;	B	10	9
(U)COMISD/(U)COMISS xmm, xmm;	B	9	8
CVTDQ2PD/CVTPD2DQ/CVTPD2PS xmm, mem	B	8	7
CVTDQ2PD/CVTPD2DQ/CVTPD2PS xmm, xmm	B	7	6
CVTDQ2PS/CVTSD2SS/CVTSI2SS/CVTSS2SD xmm, mem	B	7	6
CVTDQ2PS/CVTSD2SS/CVTSS2SD xmm, xmm	B	6	5
CVT(T)PD2PI mm, mem; CVTPI2PD xmm, mem	B	8	7
CVT(T)PD2PI mm, xmm; CVTPI2PD xmm, mm	B	7	6
CVTPI2PS/CVTSI2SD xmm, mem;	B	5	4
CVTPI2PS xmm, mm;	1	5	1
CVTPS2DQ xmm, mem;	B	7	6
CVTPS2DQ xmm, xmm;	B	6	5
CVT(T)PS2PI mm, mem;	B	5	5
CVT(T)PS2PI mm, xmm;	1	5	1
CVT(T)SD2SI[3] reg, mem; CVT(T)SS2SI r32, mem	B	9	8
CVT(T)SD2SI[3] reg, xmm; CVT(T)SS2SI r32, xmm	B	8	7
CVTSI2SD xmm, r32; CVTSI2SS xmm, r32	B	7; 6	5
CVTSI2SD xmm, r64; CVTSI2SS xmm, r64	B	6; 7	5
CVT(T)SS2SI r64, mem; RCPPS xmm, mem	B	10	9
CVT(T)SS2SI r64, xmm; RCPPS xmm, xmm	B	9	8
CVTTPD2DQ xmm, mem	B	8	7
CVTTPD2DQ xmm, xmm	B	7	6
DEC/INC[2] mem; MASKMOVQ; MOVAPD/MOVAPS mem, xmm	0	1	1
DEC/INC[2] reg; FLD ST; FST/FSTP ST; MOVDQ2Q mm, xmm	(0, 1)	1	0.5
DIVPD; DIVPS	B	125; 70	124; 69
DIVSD; DIVSS	B	62; 34	61; 33
EMMS; LDMXCSR	B	5	4
FABS/FCHS/FXCH; MOVQ2DQ xmm, mm; MOVSX/MOVZX r16, r16	(0, 1)	1	0.5
FADD/FSUB/FSUBR[3] mem	B	5	4

Table 14-2. Intel® Atom™ Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
FADD/FADDP/FSUB/FSUBP/FSUBR/FSUBRP ST;	1	5	1
FCMOV	B	6	5
FCOM/FCOMP[3] mem	B	1	1
FCOM/FCOMP/FCOMPP/FUCOM/FUCOMP ST; FTST	1	1	1
FCOMI/FCOMIP/FUCOMI/FUCOMIP ST	B	9	8
FDIV/FSQRT[3] mem; FDIV/FSQRT ST	0	25-65	24-64
FIADD/FIMUL[5] mem	B	11	10
FICOM/FICOMP mem	B	7	6
FILD[4] mem	B	5	4
FLD[3] mem; FXAM; MOVAPD/MOVAPS/MOVD xmm, mem	0	1	1
FLDCW	B	5	4
FMUL/FMULP ST; FMUL[3] mem	0	5	1
FNSTSW AX; FNSTSW m16	B	10; 14	9; 13
FST/FSTP[3] mem	B	2	1
HADDPD/HADDPS/HSUBPD/HSUBPS xmm, mem	B	9	8
HADDPD/HADDPS/HSUBPD/HSUBPS xmm, xmm	B	8	7
IDIV r/m8; IDIV r/m16; IDIV r/m32; IDIV r/m64;	B	33;42;57;197	32;41;56;196
IMUL/MUL[6] EAX/AL, mem; IMUL/MUL AX, m16	B	7; 8	6; 7
IMUL/MUL[7] AX/AL, reg; IMUL/MUL EAX, r32	B	7; 6	6; 5
IMUL m16, imm8/imm16; IMUL r16, m16	B	7;	6
IMUL r/m32, imm8/imm32; IMUL r32, r/m32	0	5	1
IMUL r/m64, imm8/imm32;	B	14	13
IMUL r16, r16; IMUL r16, imm8/imm16	B	6	5
IMUL r64, r/m64; IMUL/MUL RAX, r/m64	B	11; 12	10; 11
JCC[1]; JMP[4] reg; JMP[1]	1	1	1
JCXZ; JECXZ; JRCXZ	B	4	1
JMP mem[4];	B	2	1
LDDQU; MOVDQU/MOVUPD/MOVUPS xmm, mem;	B	3	2
LEA r16, mem; MASKMOVDQU; SETcc m8	(0, 1)	2	1
LEA, reg, mem	1	1	1
LEAVE;	B	2;	2
MAXSD/MAXSS/MINSD/MINSS xmm, mem	B	5	1
MAXSD/MAXSS/MINSD/MINSS xmm, xmm	1	5	1
MOV[2] MOFFS, (E)AX/AL; MOV[2] reg, mem; MOV[2] mem, reg	0	1	1
MOVD mem[3], mm; MOVD xmm, reg[3]; MOVD mm, mem[3]	0	1	1

Table 14-2. Intel® Atom™ Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
MOVD reg[3], mm; MOVD reg[3], xmm; PMOVMSK reg[3], mm	0	3	1
MOVDQA/MOVQ xmm, mem; MOVDQA/MOVD mem, xmm;	0	1	1
MOVDQA/MOVDQU/MOVUPD xmm, xmm; MOVQ mm, mm	(0, 1)	1	0.5
MOVDQU/MOVUPD/MOVUPS mem, xmm;	B	2	2
MOVHLPS;MOVLHPS;MOVHPD/MOVHPS/MOVLPD/MOVLPS	0	1	1
MOVMSKPD/MOVSKPS/PMOVMSKB reg[3], xmm	0	3	1
MOVNTI[3] mem, reg; MOVNTPD/MOVNTPS; MOVNTQ	0	1	1
MOVQ mem, mm; MOVQ mm, mem; MOVDDUP	0	1	1
MOVSD/MOVSS xmm, xmm; MOVSXD[5] reg, reg	(0, 1)	1	0.5
MOVSD/MOVSS xmm, mem; PALIGNR	0	1	1
MOVSD/MOVSS mem, xmm; PINSRW	0	1	1
MOVSHDUP/MOVSLDUP xmm, mem	0	1	1
MOVSHDUP/MOVSLDUP/MOVUPS xmm, xmm	(0, 1)	1	0.5
MOVSX/MOVZX r16, m8; MOVSX/MOVZX r16, r8	0	3; 2	1
MOVSX/MOVZX reg[3], r/m8; MOVSX/MOVZX reg[3], r/m16	0	1	1
MOVSXD[5] reg, mem; MOVSXD r64, r/m32	0	1	1
MULPS/MULSD xmm, mem; MULSS xmm, mem;	0	5; 4	2
MULPS/MULSD xmm, xmm; MULSS xmm, xmm	0	5; 4	2
MULPD	B	5; 4	2
NEG/NOT[2] mem; PREFETCHNTA; PREFETCHTx	0	10	9
NEG/NOT[2] reg; NOP	(0, 1)	1	0.5
PABSB/D/W mm, mem; PABSB/D/W xmm, mem	0	1	1
PABSB/D/W mm, mm; PABSB/D/W xmm, xmm	(0, 1)	1	0.5
PACKSSDW/WB mm, mem; PACKSSDW/WB xmm, mem	0	1	1
PACKSSDW/WB mm, mm; PACKSSDW/WB xmm, xmm	0	1	1
PACKUSWB mm, mem; PACKUSWB xmm, mem	0	1	1
PACKUSWB mm, mm; PACKUSWB xmm, xmm	0	1	1
PADDB/D/W/Q mm, mem; PADDB/D/W/Q xmm, mem	0	1	1
PADDB/D/W/Q mm, mm; PADDB/D/W/Q xmm, xmm	(0, 1)	1	0.5
PADDSB/W mm, mem; PADDSB/W xmm, mem	0	1	1
PADDSB/W mm, mm; PADDSB/W xmm, xmm	(0, 1)	1	0.5
PADDUSB/W mm, mem; PADDUSB/W xmm, mem	0	1	1
PADDUSB/W mm, mm; PADDUSB/W xmm, xmm	(0, 1)	1	0.5
PAND/PANDN/POR/PXOR mm, mem; PAND/PANDN/POR/PXOR xmm, mem	0	1	1
PAND/PANDN/POR/PXOR mm, mm; PAND/PANDN/POR/PXOR xmm, xmm	(0, 1)	1	0.5
PAVGB/W mm, mem; PAVGB/W xmm, mem	0	1	1

Table 14-2. Intel® Atom™ Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
PAVGB/W mm, mm; PAVGB/W xmm, xmm	(0, 1)	1	0.5
PCMPEQB/D/W mm, mem; PCMPEQB/D/W xmm, mem	0	1	1
PCMPEQB/D/W mm, mm; PCMPEQB/D/W xmm, xmm	(0, 1)	1	0.5
PCMPGTB/D/W mm, mem; PCMPGTB/D/W xmm, mem	0	1	1
PCMPGTB/D/W mm, mm; PCMPGTB/D/W xmm, xmm	(0, 1)	1	0.5
PEXTRW;	B	4	1
PHADDD/PHSUBD mm, mem; PHADDD/PHSUBD xmm, mem	B	4	3
PHADDD/PHSUBD mm, mm; PHADDD/PHSUBD xmm, xmm	B	3	2
PHADDW/PHADDSW mm, mem; PHADDW/PHADDSW xmm, mem	B	6; 8	5;7
PHADDW/PHADDSW mm, mm; PHADDW/PHADDSW xmm, xmm	B	5; 7	M
PHSUBW/PHSUBSW mm, mem; PHSUBW/PHSUBSW xmm, mem	B	6; 8	M
PHSUBW/PHSUBSW mm, mm; PHSUBW/PHSUBSW xmm, xmm	B	5; 7	M
PMADDUBSW/PMADDWD/PMULHRSW/PSADBW mm, mm; PMADDUBSW/PMADDWD/PMULHRSW/PSADBW mm, mem	0	4	1
PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, xmm; PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, mem	0	5	1
PMAXSW/UB mm, mem; PMAXSW/UB xmm, mem	0	1	1
PMAXSW/UB mm, mm; PMAXSW/UB xmm, xmm	(0, 1)	1	0.5
PMINSW/UB mm, mem; PMINSW/UB xmm, mem	0	1	1
PMINSW/UB mm, mm; PMINSW/UB xmm, xmm	(0, 1)	1	0.5
PMULHUW/PMULHW/PMULLW/PMULUDQ mm, mm; PMULHUW/PMULHW/PMULLW/PMULUDQ mm, mem	0	4	1
PMULHUW/PMULHW/PMULLW/PMULUDQ xmm, xmm; PMULHUW/PMULHW/PMULLW/PMULUDQ xmm, mem	0	5	1
POP mem[5]; PSLLD/Q/W mm, mem; PSLLD/Q/W xmm, mem	B	3	2
POP r16; PUSH mem[4]; PSLLD/Q/W mm, mm; PSLLD/Q/W xmm, xmm	B	2	1
POP reg[3]; PUSH reg[4]; PUSH imm	B	1	1
POPA ; POPAD	B	9	8
PSHUFB mm, mem; PSHUFD; PSHUFHW; PSHUFLW; PSHUFW	0	1	1
PSHUFB mm, mm; PSLLD/Q/W mm, imm; PSLLD/Q/W xmm, imm	0	1	1
PSHUFB xmm, mem	B	5	4
PSHUFB xmm, xmm	B	4	3
PSIGNB/D/W mm, mem; PSIGNB/D/W xmm, mem	0	1	1
PSIGNB/D/W mm, mm; PSIGNB/D/W xmm, xmm	(0, 1)	1	0.5
PSRAD/W mm, imm; PSRAD/W xmm, imm;	0	1	1
PSRLD/Q/W mm, mem; PSRLD/Q/W xmm, mem	B	3	2
PSRLD/Q/W mm, mm; PSRLD/Q/W xmm, xmm	B	2	1

Table 14-2. Intel® Atom™ Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
PSRLD/Q/W mm, imm; PSRLD/Q/W xmm, imm;	0	1	1
PSLLDQ/PSRLDQ xmm, imm; SHUFPD/SHUFPS	0	1	1
PSUBB/D/W/Q mm, mem; PSUBB/D/W/Q xmm, mem	0	1	1
PSUBB/D/W/Q mm, mm; PSUBB/D/W/Q xmm, xmm	(0, 1)	1	0.5
PSUBSB/W mm, mem; PSUBSB/W xmm, mem	0	1	1
PSUBSB/W mm, mm; PSUBSB/W xmm, xmm	(0, 1)	1	0.5
PSUBUSB/W mm, mem; PSUBUSB/W xmm, mem	0	1	1
PSUBUSB/W mm, mm; PSUBUSB/W xmm, xmm	(0, 1)	1	0.5
PUNPCKHBW/DQ/WD; PUNPCKLBW/DQ/WD	0	1	1
PUNPCKHQDQ; PUNPCKLQDQ	0	1	1
PUSHA ; PUSHAD	B	8	7
RCL mem[2], 1; RCL reg[2], 1	0	1	1
RCL m8, CL; RCL m16, CL; RCL mem[3], CL;	B	18;16; 14	17;15;13
RCL m8, imm; RCL m16, imm; RCL mem[3], imm;	B	18; 17; 14	17;16;13
RCL r8, CL; RCL r16, CL; RCL reg[3], CL;	B	17; 16; 14	16;15;14
RCL r8, imm; RCL r16, imm; RCL reg[3], imm;	B	18;16; 14	17;15;13
RCPSS	0	4	1
RCR mem[2], 1; RCR reg[2], 1	B	7; 5	6;4
RCR m8, CL; RCR m16, CL; RCR mem[3], CL;	B	15; 13; 12	14;12;11
RCR m8, imm; RCR m16, imm; RCR mem[3], imm;	B	16,;14; 12	15;13;11
RCR r8, CL; RCR r16, CL; RCR reg[3], CL;	B	14; 13; 12	13;12;11
RCR r8, imm; RCR r16, imm; RCR reg[3], imm;	B	15, 14, 12	14;13;11
RET imm16	B	1	1
RET (far)	B	79	
ROL; ROR; SAL; SAR; SHL; SHR	0	1	1
SETcc		1	1
SHLD[8] mem, reg, imm; SHLD r64, r64, imm; SHLD m64, r64, CL	B	11	10
SHLD m32, r32; SHLD r32, r32	B	4; 2	3; 1
SHLD m16, r16, CL; SHLD r16, r16, imm; SHLD r64, r64, CL	B	10	9
SHLD r16, r16, CL; SHRD m64, r64; SHRD r64, r64, imm	B	9	8
SHRD m32, r32; SHRD r32, r32	B	4; 2	3; 1
SHRD m16, r16; SHRD r16, r16	B	6	5
SHRD r64, r64, CL	B	8	7
STMXCSR	B	15	14

Table 14-2. Intel® Atom™ Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
TEST[2] reg, reg; TEST[4] reg, imm	(0, 1)	1	0.5
UNPCKHPD; UNPCKHPS; UNPCKLPD, UNPCKLPS	0	1	1

Notes on operand size (osize) and address size (asize):
1. osize = 8, 16, 32 or asize = 8, 16, 32
2. osize = 8, 16, 32, 64
3. osize = 32, 64
4. osize = 16, 32, 64 or asize = 16, 32, 64
5. osize = 16, 32
6. osize = 8, 32
7. osize = 8, 16
8. osize = 16, 64

CHAPTER 15
SILVERMONT MICROARCHITECTURE AND SOFTWARE OPTIMIZATION

15.1 OVERVIEW

This chapter covers a brief overview the Silvermont microarchitecture, and specific coding techniques for software targeting Intel® Atom™ processors based on the Silvermont microarchitecture (see Table 35-1 in CHAPTER 35 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*). The software optimization recommendations in this chapter will focus on issues unique to the Silvermont microarchitecture that should be considered in addition to the generic x86 coding style.

15.1.1 Intel Atom Processor Family Based on the Silvermont Microarchitecture

The Intel Atom processor E3000 and C2000 Series are based on the Silvermont microarchitecture. The Silvermont microarchitecture spans a wide range of computing devices using Intel's 22 nm process technology. In addition to support for Intel 64 and IA-32 architecture, major enhancements of the Silvermont microarchitecture include:

- Out-of-order execution for integer instructions and de-coupled ordering between non-integer and memory instructions. In contrast, the previous generation Intel Atom microarchitecture (see CHAPTER 14) was strictly in-order with limited ability to exploit available instruction-level parallelism.
- Non-blocking memory instructions allowing multiple (8) outstanding misses. In previous generation processors, problems in a single memory instruction (e.g. cache miss) caused all subsequent instructions to stall until the problem was resolved. The new microarchitecture allows up to 8 unique outstanding references.
- Modular system design with two cores sharing an L2 cache connected to a new integrated memory controller using a point-to-point interface instead of the Front-Side Bus
- Instruction set enhancements to include SSE 4.1, SSE 4.2, AESNI and PCLMULQDQ, making it compatible with the 32 nm first generation Intel® Core™ processor.

15.2 SILVERMONT MICROARCHITECTURE

The block diagram for the Silvermont microarchitecture is depicted in Figure 15-1. While the memory and execute clusters were significantly redesigned for improved single thread performance, the primary focus is still a highly efficient design in a small form factor power envelope. Each pipeline is accompanied with a dedicated scheduling queue called a reservation station. While floating-point and memory instructions schedule from their respective queues in program order, integer execution instructions schedule from their respective queues out of order.

Integer instructions can be scheduled from their queues out of order in contrast to in-order execution in previous generations. Out of order scheduling allows these instructions to tolerate stalls caused by unavailable (re)sources. Memory instructions must generate their addresses (AGEN) in-order and schedule from the scheduling queue in-order but they may complete out-of-order.

Non-integer instructions (including SIMD integer, SIMD floating-point, and x87 floating-point) also schedule from their respective scheduling queue in program order. However, these separate scheduling queues allow their execution to be decoupled from instructions in other scheduling queues.

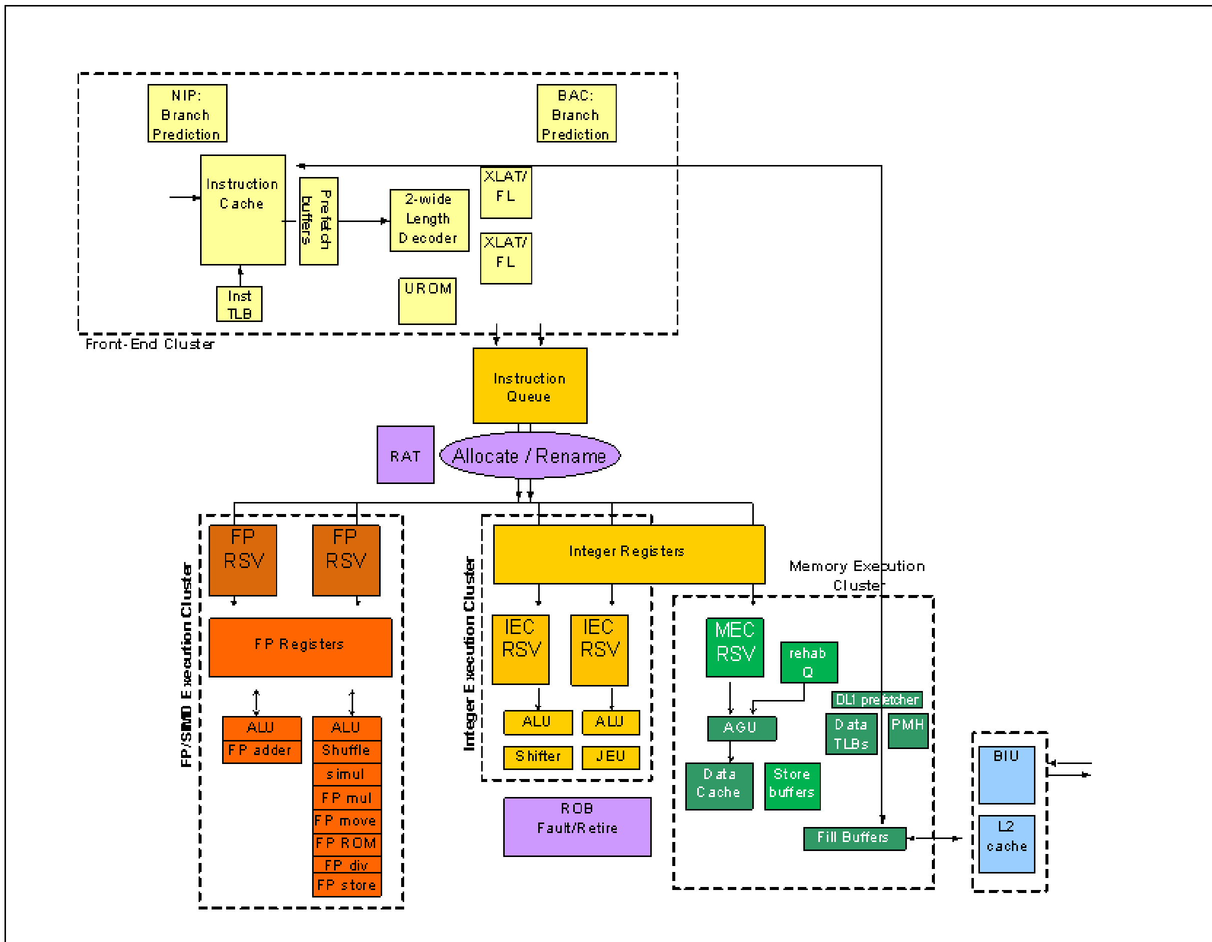


Figure 15-1. Silvermont Microarchitecture Pipeline

The design of the microarchitecture takes into account maximizing platform performance of multiple form factors (e.g. phones, tablets, to micro-servers) and minimizing the power and area cost due to out of order scheduling (i.e. maximizing performance/power/cost efficiency). Intel Hyper-Threading Technology is not supported in favor of a multi-core architecture with a shared L2 cache. The rest of this section will cover some of the cluster-level features in more detail.

The front end cluster (FEC), shown in yellow in Figure 15-1, features a power optimized 2-wide decode pipeline. FEC is responsible for fetching and decoding instructions from instruction memory. FEC utilizes predecode hints from the icache to avoid costly on-the-fly instruction length determination. The front end contains a Branch Target Buffer (BTB), plus advanced branch predictor hardware.

The front end is connected to the execution units through the Allocation, Renaming and Retirement (ARR) cluster (lavender color in Figure 15-1). ARR receives uops from the FEC and is responsible for resource checks. The Register Alias Table (RAT) renames the logical registers to the physical registers. The Reorder Buffer (ROB) puts the operations back into program order and completes (retires) them. It also stops execution at interrupts, exceptions and assists and runs program control over microcode.

Scheduling in the Silvermont microarchitecture is distributed, so after renaming, uops are sent to various clusters (IEC: integer execution cluster; MEC: memory execution cluster; FPC: floating-point cluster) for scheduling (shown as RSV for FP, IEC, and MEC in Figure 15-1).

There are 2 sets of reservation stations for FPC and IEC (one for each port) and a single set of reservation stations for MEC. Each reservation station is responsible for receiving up to 2 ops from the ARR cluster in a cycle and selecting one ready op for dispatching to execution as soon as the op becomes ready.

To support the distributed reservation station concept, load-op and load-op-store macro-instructions requiring integer execution must be split into a memory sub-op that is sent to the MEC and resides in the memory reservation station and an integer execution sub-op that is sent to the integer reservation station. The IEC schedulers pick the oldest ready instruction from each of its RSVs while the MEC and the FPC schedulers only look at the oldest instruction in their respective RSVs. Even though the MEC and FPC clusters employ in-order schedulers, a younger instruction from a particular FPC RSV can execute before an older instruction in the other FPC RSV for example (or the IEC or MEC RSVs).

Each execution port has specific functional units available. Table 15-1 shows the mapping of functional units to ports for IEC (the orange units in Figure 15-1), MEC (the green units in Figure 15-1), and the FPC (the red units in Figure 15-1). Compared to the previous Intel Atom Microarchitecture, the Silvermont microarchitecture adds an integer multiply unit (IMUL) in IEC.

Table 15-1. Function Unit Mapping of the Silvermont Microarchitecture

	Port 0	Port 1
IEC	ALU0, Shift/Rotate Unit, LEA with no index	ALU1, Bit processing unit, Jump unit, IMUL, POPCNT, CRC32, LEA[1]
FPC	SIMD ALU, SIMD shift/Shuffle unit, SIMD FP mul/div/cvt unit, STTNI/AESNI/PCLMULQDQ unit, RCP/RSQRT unit, F2I convert unit	SIMD ALU, SIMD FPadd unit, F2I convert unit
MEC	Load/Store	

NOTES:

1. LEAs with valid index and displacement are split into multiple UOPs and use both ports. LEAs with valid index execute on port 1.

The Memory Execution Cluster (MEC) (shown in green in Figure 15-1) can support both 32-bit and 36-bit physical addressing modes. The Silvermont microarchitecture has a 2 level Data TLB hierarchy with support for both large (2MB or 4MB) and small page structures. A small micro TLB (referred to as uTLB) is backed up by a larger 2nd level TLB (referred to as DTLB). A hardware page walker services misses from both the Instruction and Data TLBs.

The MEC also owns the MEC RSV, which is responsible for scheduling of all loads and stores. Load and store instructions go through addresses generation phase in program order to avoid on-the-fly memory ordering later in the pipeline. Therefore, an unknown address will stall younger memory instructions. Memory operations that incur problems (e.g. uTLB misses, unavailable resources, etc.) are put in a separate queue called the RehabQ. This allows younger instructions (that do not incur problems) to continue execution rather than stalling all younger instructions. The problematic instruction is later reissued from the RehabQ when the problem is resolved. Note that load misses are not considered problematic as the Silvermont microarchitecture features a non-blocking data cache that can sustain 8 outstanding misses.

The Bus Cluster (BIU) includes the second-level cache (L2) and is responsible for all communication with components outside the processor core. The L2 cache supports up to 1MB with an optimized latency less than the previous Intel Atom Microarchitecture. The Front-Side Bus from earlier Intel Atom processors has been replaced by an intra-die interconnect (IDI) fabric connecting to a newly optimized memory controller. The BIU also houses the L2 data prefetcher.

The new core level multi-processing (or CMP) system configuration features two processor cores making requests to a single BIU, which will handle the multiplexing between cores. This basic CMP module can be replicated to create a quad-core configuration, or one core chopped off to create a single-core configuration.

15.2.1 Integer Pipeline

Load pipeline stages are no longer inlined with the rest of the integer pipeline. As a result, non-load ops can reach execute faster, and the branch misprediction penalty is effectively 3 cycles less compared to earlier Intel Atom processors. Front end pipe stages are the same as earlier Intel Atom processors (3 cycles for fetch, 3 cycles for decode). ARR pipestages perform out-of-order allocation and register renaming, split the uop into parts if necessary, and send them to the distributed reservation stations. RSV stage is where the distributed reservation station performs its scheduling. The execution pipelines are very similar to earlier Intel Atom processors. When all parts of a uop are marked as finished, the ROB handles final completion in-order.

15.2.2 Floating-Point Pipeline

Compared to the INT pipeline, the FP pipeline is longer. The execution stages can vary between one and five depending on the instruction. Like other Intel microarchitectures, the Silvermont microarchitecture needs to limit the number of FP assists (when certain floating-point operations cannot be handled natively by the execution pipeline, and must be performed by microcode) to the bare minimum to achieve high performance. To do this the processor should be run with exceptions masked and the DAZ (denormal as zero) and FTZ (flush to zero) flags set whenever possible.

As mentioned, while each FPC RSV schedules instructions in-order, the RSVs can get out of order with respect to each other.

15.3 CODING RECOMMENDATIONS FOR SILVERMONT MICROARCHITECTURE

15.3.1 Optimizing The Front End

15.3.1.1 Instruction Decoder

Some instructions are too complex to decode into a single uop and require a lookup in the microcode sequencer ROM (MSROM) to determine the uops that accomplish the task of the instruction. For determining which instructions require an MSROM lookup, see the instruction latency/bandwidth table in Appendix A.

In the Silvermont microarchitecture, much fewer instructions require MSROM involvement. Key instructions such as packed double precision SIMD instructions and unaligned loads/stores are no longer microcoded.

It is still advisable to avoid ucode flows where possible.

Tuning Suggestion 1. *Use the perfmon counter MS_DECODED.MS_ENTRY to find the number of instructions that need the MSROM (the count will include any assist or fault that occurred).*

Assembly/Compiler Coding Rule 1. (M impact, M generality) *Try to keep the I-footprint small to get the best reuse of the predecode bits.*

Tuning Suggestion 2. *Use the perfmon counter DECODE_RESTRICTION.PREDECODE_WRONG to count the number of times that a decode restriction reduced instruction decode throughput because predecoded bits are incorrect.*

15.3.1.2 Front End High IPC Considerations

In general front end restrictions are not typically a performance limiter until you reach higher (>1) Instructions Per Cycle (IPC) levels.

The decode restrictions that must be followed to get two instructions per cycle through the decoders include:

- MSROM instructions should be avoided if possible. A good example is the memory form of PUSH and CALL. It will often be better to perform a load into a register and then perform the register version of PUSH and CALL.
- The total length of the instruction pair that will be decoded together should be less than 16 bytes with the length of the first instruction being limited to at most 8 bytes. The Silvermont microarchitecture can only decode one instruction per cycle if the instruction exceeds 8 bytes.
- Instructions should have no more than three prefixes and escape bytes. There is a 3 cycle penalty when the escape/prefix count exceeds 3.
- The Silvermont microarchitecture cannot decode 2 branches in the same cycle. For example a predicted not taken conditional jump on decoder 0 and an unconditional jump on decoder 1 will trigger a 3 cycle penalty for the unconditional jump.

Unlike the previous generation, the Silvermont microarchitecture can decode two x87 instructions in the same cycle without incurring a 2-cycle penalty. Branch decoder restrictions are also relaxed. In earlier Intel Atom processors, decoding past a conditional or indirect branch in decoder 0 resulted in a 2-cycle penalty. The Silvermont microarchitecture can decode past conditional and indirect branch instructions in decoder 0. However, if the next instruction (on decoder 1) is also a branch, there is a 3-cycle penalty for the second branch instruction.

Assembly/Compiler Coding Rule 2. (MH impact, H generality) *Minimize the use of instructions that have the following characteristics to achieve more than one instruction per cycle throughput: (i) using the MSROM, (ii) more than 3 escape/prefix bytes, (iii) more than 8 bytes long, or (iv) have back to back branches.*

An example where a slowdown occurs due to too many prefixes is when a REX prefix is in use for the PCLMULQDQ instruction. For instance:

```
PCLMULQDQ 66 0F 3A 44 C7 01   pclmulqdq xmm0, xmm7, 0x1
```

Note the 66, and 0F 3A, before the opcode byte 44 are required prefixes. If any of the XMM registers, XMM8-15, is referenced, an additional REX prefix will be required. For instance:

```
PCLMULQDQ 66 41 0F 3A 44 C0 01 pclmulqdq xmm0, xmm8, 0x1
```

(note the 41, in between the 66 and the 0F 3A).

This 4th prefix causes a decoding delay of 3 cycles. Also, it forces the instruction to be decoded on decoder 0. This penalty is even larger if the instruction starts out on decoder 1, since it takes an extra 3 clocks to steer it to decoder 0 (for a total of 6 cycle penalty for the decoder). Therefore, when hand writing high performance assembly, it would be wise to be aware of these cases. It would be beneficial to pre-align these cases to decoder 0 if they occur infrequently using a taken branch target or MS entry point as a decoder 0 alignment vehicle. NOP insertion should be used only as a last resort as NOP instructions consume resources in other parts of the pipeline. Similar alignment is necessary for MS entry points which suffer the additional 3 cycle penalty if they align originally to decoder 1.

Another important performance consideration from a front end standpoint is branch prediction. For 64-bit applications, branch prediction performance can be negatively impacted when the target of a branch is more than 4GB away from the branch. This is more likely to happen when the application is split into shared libraries. Developers can build statically to improve the locality in their code. Building with LTO should further improve performance.

15.3.1.3 Loop Unrolling and Loop Stream Detector

The Silvermont microarchitecture includes a Loop Stream Detector (LSD) that provides the back end with uops that are already decoded. This provides performance and power benefits. When the LSD is engaged, front end decode restrictions, such as number of prefix/escape bytes and instruction length, no longer apply.

One way to reduce the overhead of loop maintenance code and increase the amount of independent work in a loop is software loop unrolling. Unfortunately care must be taken on where it is utilized because loop unrolling has both positive and negative performance effects. The negative performance effects are caused by the increased code size and increased BTB and register pressure. Furthermore, loop unrolling

can increase the loop size beyond the limits of the LSD. Care must be taken to keep the loop size under 29 instructions for the loop to fit in the LSD.

User/Source Coding Rule 1. (M impact, M generality) *Keep per-iteration instruction count below 29 when considering loop unrolling technique on short loops with high iteration count.*

Tuning Suggestion 3. *Use the BACLEARS.ANY perfmon counter to see if the loop unrolling is causing too much pressure. Use the ICACHE.MISSES perfmon counter to see if loop unrolling is having an excessive negative effect on the instruction footprint.*

15.3.2 Optimizing The Execution Core

15.3.2.1 Scheduling

The Silvermont microarchitecture is less sensitive to instruction ordering than its predecessors due to the introduction of out-of-order execution for integer instructions. FP instructions have their own reservation stations but still execute in order with respect to each other. Memory instructions also issue in order but with the addition of the Rehab Queue, they can complete out of order and memory system delays are no longer blocking.

Tuning Suggestion 4. *Use the perfmon counter NO_ALLOC_CYCLE.ANY as an indicator of performance bottlenecks in the back end. This includes delays in the memory system, and execution delays.*

15.3.2.2 Address Generation

Address generation limitations from the previous Intel Atom Microarchitecture have been eliminated from the Silvermont microarchitecture. As such, using LEA or ADD instructions to generate addresses are equally effective. In earlier Intel Atom generations, using an LEA was preferable.

The rule of thumb for ADDs and LEAs is that it is justified to use LEA with a valid index and/or displacement for non-destructive destination purposes (especially useful for stack offset cases), or to use a SCALE. Otherwise, ADD(s) are preferable.

15.3.2.3 FP Multiply-Accumulate-Store Execution

FP arithmetic instructions executing on different ports can execute out-of-order with respect to each other in the Silvermont microarchitecture. As a result, in unrolled loops with multiplication results feeding into add instructions which in turn produce results for store instructions, grouping the store instructions at the end of the loop will improve performance. This allows it to overlap the execution of the multiplies and the adds. Consider the example shown in Example 15-1.

Example 15-1. Unrolled Loop Executes In-Order Due to Multiply-Store Port Conflict

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
mulps, xmm1, xmm1	EX1	EX2	EX3	EX4	EX5													
addps xmm1, xmm1						EX1	EX2	EX3										
movaps mem, xmm1									EX1									
mulps, xmm2, xmm2										EX1	EX2	EX3	EX4	EX5				
addps xmm2, xmm2															EX1	EX2	EX3	
movaps mem, xmm2																		EX1

Due to the data dependence, the add instructions cannot start executing until the corresponding multiply instruction is executed. Because multiplies and stores use the same port, they have to execute in program order. This means the second multiply instruction cannot start execution even though it is independent from the first multiply and add instructions. If you group the store instructions together at the end of the loop as shown below, the second multiply instruction can execute in parallel with the first multiply instruction (note the 1 cycle bubble when multiplies are overlapped).

Example 15-2. Grouping Store Instructions Eliminates Bubbles and Improves IPC

Instruction	1	2	3	4	5	6	7	8	9	10	11
mulps, xmm1, xmm1	EX1	EX2	EX3	EX4	EX5						
addps xmm1, xmm1						EX1	EX2	EX3			
mulps, xmm2, xmm2		bubble	EX1	EX2	EX3	EX4	EX5				
addps xmm2, xmm2								EX1	EX2	EX3	
movaps mem, xmm1									EX1		
movaps mem, xmm2											EX1

15.3.2.4 Integer Multiply Execution

There is a dedicated integer multiplier in the Silvermont microarchitecture. Table 15-2 shows the latency and the number of uops for different forms of mul/imul instructions.

Table 15-2. Integer Multiply Operation Latency

Input Size	8 bit		16 bit		32 bit		64 bit	
Output Size/Latency	Size	Lat	Size	Lat	Size	Lat	Size	Lat
imul/mul reg	16	5[u]	32	5[u]	64	4[u]	128	7[u]
imul/mul reg, reg			16	4[u]	32	3	64	4
imul/mul reg, reg, imm8			16	4[u]	32	3	64	4
imul/mul reg, reg, imm16/32			16	4[u]	32	3	64	4
u: ucode flow								

The multiply forms with microcode flows should be avoided. In addition, looking at the table shows that the best performance will result from executing one of imul, r32, rm32 variants. These multiplies are fully pipelined (1 result per cycle) while the imul r64, r/m64 variants have a throughput of 1 result every 4 cycles.

15.3.2.5 Zeroing Idioms

XOR / PXOR / XORPS / XORPD instructions are commonly used to force register values to zero when the source and the destination register are the same (e.g. XOR eax, eax).

This method of zeroing is preferred by compilers instead of the equivalent MOV eax, 0x0 instructions as the MOV encoding is larger than the XOR in code bytes.

The Silvermont microarchitecture has special hardware support to recognize these cases and mark both the sources as valid in the architectural register file. This helps the XOR execute faster since any value XORed with itself will accomplish the necessary zeroing.

The logic will also support PXOR, XORPS, and XORPD idioms.

Note that 64 bit operand sizes are not supported by this idiom. 64 bit register zeroing for the lower architectural registers can still be done however by simply avoiding the REX.W bit. To zero out rax, for example, specify XOR eax, eax and not XOR rax, rax. Similarly, to zero out r8, use XOR r8d, r8d and not XOR r8, r8.

15.3.2.6 Flags usage

Many instructions have an implicit data result that is captured in a flags register. These results can be consumed by a variety of instructions such as conditional moves (cmovs), branches and even a variety of logic/arithmetic operations (such as rcl). The most common instructions used in computing branch conditions are compare instructions (CMP). Branches dependent on the CMP instruction can execute in the next cycle. The same is true for branch instructions dependent on ADD or SUB instructions.

In the Silvermont microarchitecture, INC and DEC instructions require an additional uop to merge the flags as they are partial flag writers. As a result, a branch instruction depending on an INC or a DEC instruction incurs a 1 cycle penalty. Note that this penalty only applies to branches that are directly dependent on the INC or DEC instruction.

Assembly/Compiler Coding Rule 3. (M impact, M generality) *Use CMP/ADD/SUB instructions to compute branch conditions instead of INC/DEC instructions whenever possible.*

15.3.2.7 Instruction Selection

Table 15-3 summarizes the latency for floating-point and SIMD integer operations in the Silvermont microarchitecture. The throughput column indicates the number of instructions that can start execution each cycle (e.g. ¼ indicates 1 instruction can start executing every 4 cycles).

Table 15-3. Floating-Point and SIMD Integer Latency

	Latency	Throughput*
SIMD integer ALU		
128-bit ALU/logical/move	1	2/1
64-bit ALU/logical/move	1	2/1
SIMD integer shift		
128-bit	1	1/1
64-bit	1	1/1
SIMD shuffle		
128-bit	1	1/1
64-bit	1	1/1
SIMD integer multiplier		
128-bit	5	1/2
64-bit	4	1/1
FP Adder		
x87 (fadd)	3	1/1
scalar (addsd, addss)	3	1/1
packed (addpd, addps)	4	1/2
FP Multiplier		
x87 (fmul)	5	1/2
scalar single-precision (mulss)	4	1/1
scalar double-precision (mulsd)	5	1/2
packed single-precision (mulps)	5	1/2
packed double-precision (mulpd)	7	1/4
Converts		
CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPS2DQ, CVTPS2PD, CVTTPD2DQ, CVTPD2PI, CVTPS2DQ	5	1/2
CVTPI2PS, CVTPS2PI, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSI2SS, CVTSS2SD, CVTSS2SI, CVTTPS2PI, CVTTSD2SI, CVTTSS2SI	4	1/1
FP Divider		

Table 15-3. Floating-Point and SIMD Integer Latency (Contd.)

	Latency	Throughput*
x87 fdiv (extended-precision)	39	1/39
x87 fdiv (double-precision)	34	1/34
x87 fdiv (single-precision)	19	1/19
scalar single-precision (divss)	19	1/17
scalar double-precision (divsd)	34	1/32
packed single-precision (divps)	39	1/39
packed double-precision (divpd)	69	1/69
* Throughput notation “m/n” is ‘m’ ops can be dispatched every ‘n’ cycle.		

Note that scalar SSE single precision multiples are one cycle faster than most FP operations. From inspection of the table you can also see that packed SSE doubles have a slightly larger latency and smaller throughput compared to their scalar counterparts.

Assembly/Compiler Coding Rule 4. (M impact, M generality) *Favor SSE floating-point instructions over x87 floating point instructions.*

Assembly/Compiler Coding Rule 5. (MH impact, M generality) *Run with exceptions masked and the DAZ and FTZ flags set (whenever possible).*

Tuning Suggestion 5. *Use the perfmon counters MACHINE_CLEARS.FP_ASSIST to see if floating exceptions are impacting program performance.*

Latency of integer division operations can vary profoundly on the input value and data sizes (this is common across many Intel microarchitectures). Table 15-4 and Table 15-5 show the latency range for divide instructions.

Table 15-4. Unsigned Integer Division Operation Latency

	Dividend	Divisor	Quotient	Remainder	Cycles
DIV r8	AX	r8	AL	AH	25
DIV r16	DX:AX	r16	AX	DX	26-30
DIV r32	EDX:EAX	r32	EAX	EDX	26-38
DIV r64	RDX:RAX	r64	RAX	RDX	38-123

Table 15-5. Signed Integer Division Operation Latency

	Dividend	Divisor	Quotient	Remainder	Cycles
IDIV r8	AX	r8	AL	AH	34
IDIV r16	DX:AX	r16	AX	DX	35-40
IDIV r32	EDX:EAX	r32	EAX	EDX	35-47
IDIV r64	RDX:RAX	r64	RAX	RDX	49-135

User/Source Coding Rule 2. (M impact, L generality) *Use divides only when really needed and take care to use the correct data size and sign so that you get the most efficient execution.*

Tuning Suggestion 6. *Use the perfmon counter CYCLES_DIV_BUSY.ANY to see if the divides are a bottleneck in the program.*

If one needs unaligned groups of packed singles where the whole array is aligned, the use of PALIGNR is recommend over MOVUPS. For instance, load A[x+y+3:x+y] where x and y are loop variables; it is better to calculate x+y, round down to a multiple of 4 and use a MOVAPS and PALIGNR to get the elements (rather than a MOVUPS at x+y). While this may look longer, the integer operations can execute in parallel to FP ones. This will also avoid the periodic MOVUPS that splits a line at the cost of approximately 6 cycles.

User/Source Coding Rule 3. (M impact, M generality) *Use PALIGNR when stepping through packed single elements*

15.3.3 Optimizing Memory Accesses

15.3.3.1 Memory Reissue/Sleep causes

The various situations in which the memory cluster can trigger a memory instruction to be placed in the RehabQ include:

- Load blocks.
- Load/store splits.
- Locks.
- TLB misses.
- Unknown addresses.
- Too many stores.

Tuning Suggestion 7. *Use the perfmon counters REHABQ to assess the impact that reissues on application performance.*

15.3.3.2 Store Forwarding

Forwarding is significantly improved in the Silvermont microarchitecture compared to its predecessors. A store instruction will forward its data to a receiving load instruction if the following are true:

- The forwarding store and the receiving load start at the same address.
- The receiving load is smaller than or equal to the forwarding store in terms of width.
- The forwarding store or the receiving load do not incur cache line splits.

If one (or more) of these conditions is not satisfied, the load is blocked and put into the RehabQ to reissue again.

To eliminate/avoid store forwarding problems, use the guidelines below (in order of preference):

- Use registers instead of memory.
- Hoist the store as early as possible (stores happen later in the pipeline than loads, so the store needs to be hoisted many instructions earlier than the load).

In the Silvermont microarchitecture, successful forwarding incurs an additional 3 cycles compared to previous Intel Atom processors (i.e. if the store executes at cycle n, the load will execute at cycle n+3).

15.3.3.3 PrefetchW Instruction

The Silvermont microarchitecture supports the PrefetchW instruction (0f 0d /1). This instruction is a hint to the hardware to prefetch the specified line into the cache with a read-for-ownership request. This can allow later stores to that line to complete faster than they would if the line was not prefetched or was prefetched with a different instruction. All prefetch instructions may cause performance loss if misused. Care should be used to ensure that prefetch instructions, including PrefetchW, actually improve performance. The instruction opcode 0f 0d /0 continues to be a NOP. It does not prefetch the indicated line.

15.3.3.4 Cache Line Splits and Alignment

Cache line splits cause load and store instructions to operate at reduced bandwidth. As a result, they should be avoided where possible.

Tuning Suggestion 8. *Use the REHABQ.ST_SPLIT and REHABQ.LD_SPLIT perfmon counters to locate splits, and to count the number of split operations.*

While aligned accesses are preferred, the Silvermont microarchitecture has hardware support for unaligned references. As such, MOVUPS/MOVUPD/MOVDQU instructions are all single UOP instructions in contrast to previous generation Intel Atom processors.

15.3.3.5 Segment Base

For simplicity, the AGU in the Silvermont microarchitecture assumes that the segment base will be zero. However, while studies have shown that this is overwhelmingly true, there are times when a non-zero segment base (NZB) must be used. When using NZBs, keep the segment base cache line (0x40) aligned if at all possible. NZB address generation involves a 1 cycle penalty.

15.3.3.6 Copy and String Copy

Compilers typically provide libraries with memcpy/memset routines that provide good performance while managing code size and alignment issues.

Memcpy and memset operation can be accomplished using REP MOVS/STOS instructions with length of operation decomposed for optimized byte/dword granular operations and alignment considerations. This usually provides a decent copy/set solution for the general case. The REP MOVS/STOS instructions have a fixed overhead. REP STOS should be able to cope with line splits for long strings; but REP MOVS cannot due to the complexity of the possible alignment matches between source and destination.

For specific copy/set needs, macro code sequence using SIMD instruction can provide modest gains (on the order of a dozen clocks or so), depending on the alignment, buffer length, and cache residency of the buffers. Large memory copies with cache line splits are a notable exception to this rule, where careful macrocode might avoid the cache lines splits and substantially improve on REP MOV.

Processors based on the Silvermont microarchitecture support the Enhanced REP MOVSB and STOSB operation (ERMSB) feature. REP string operations using MOVSB and STOSB can provide the smallest code size with both flexible and high performance REP string operations for software in common situations like memory copy and set operations. Processors that provide enhanced MOVSB/STOSB operations are enumerated by the CPUID feature flag: CPUID:(EAX=7H, ECX=0H):EBX.ERMSB[bit 9] = 1.

Software wishing to have a simple default string copy or store routine that will work well on a range of implementations (including future implementations) should consider using REP MOVSB or REP STOSB on implementations that support ERMSB. Although these instructions may not be as fast on a specific implementation as a more specialized copy/store routine, such specialized routines may not perform as well on future processors and may not take advantage of future enhancements. REP MOVSB and REP STOSB will continue to perform reasonably well on future processors.

15.4 INSTRUCTION LATENCY

This section lists the port-binding and latency information of Silvermont microarchitecture. Instructions that required decoder assistance from MSROM are marked in the "Comment" column (instructions marked with 'Y' should be used minimally if more decode-efficient alternatives are available). Throughput value listed as "n/m", where 'm' ops can be dispatched every 'n' cycle.

Table 15-6. Silvermont Microarchitecture Instructions Latency and Throughput

Instruction	Throughput	Latency	MSROM
DisplayFamily_DisplayModel	**06_37H, 06_4AH, 06_4DH**	**06_37H, 06_4AH, 06_4DH**	**06_37H, 06_4AH, 06_4DH**
ADC/SBB r32, imm8	2	2	N
ADC/SBB r32, r32	2	2	N
ADD/AND/CMP/OR/SUB/XOR/TEST r32, r32	0.5	1	N
ADDPD/ADDSUBPD/MAXPD/MINPD/SUBPD xmm, xmm	2	4	N
ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/SUBSS xmm, xmm	1	3	N
MAXPS/MAXSD/MAXSS/MINPS/MINSD/MINSS xmm, xmm	1	3	N
ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS xmm, xmm	0.5	1	N
AESDEC/AESDECLAST/AESENC/AESENCLAST/AESIMC/AESKEYGEN xmm, xmm	5	8	Y
BLENDVPD/BLENDVPS xmm, xmm	4	4	Y
BSF/BSR r32, r32	10	10	Y
BSWAP r32	1	1	N
BT/BTC/BTR/BTS r32, r32	1	1	N
CBW	4	4	Y
CDQ/CLC/CMC	1	1	N
CMOVxx r32; r32	1	2	N
CMPPD xmm, xmm, imm	2	4	N
CMPSD/CMPPS/CMPSS xmm, xmm, imm	1	3	N
CMPXCHG r32, r32	6	6	Y
(U)COMISD/(U)COMISS xmm, xmm;	1	1	N
CPUID	60	60	Y
CRC32 r32, r32	1	3	N
CVTDQ2PD/CVTDQ2PS/CVTPD2DQ/CVTPD2PS xmm, xmm	2	5	N
CVT(T)PD2PI/CVT(T)PI2PD xmm, xmm	2	2	N
CVT(T)PS2DQ/CVTPS2PD xmm, xmm;	2	5	N
CVT(T)SD2SS/CVTSS2SD xmm, xmm	1	4	N
CVTSI2SD xmm, r32	1	1	N
DEC/INC r32	1	1	N
DIV r8	25	25	Y
DIV r16	26-30	26-30	Y
DIV r32	26-38	26-38	Y
DIV r64	38-123	38-123	Y
DIVPD	27-69	27-69	Y
DIVPS	27-39	27-39	Y
DIVSD	11-32	13-34	N
DIVSS	11-17	13-19	N

Table 15-6. Silvermont Microarchitecture Instructions Latency and Throughput (Contd.)

Instruction	Throughput	Latency	MSROM
DisplayFamily_DisplayModel	**06_37H, 06_4AH, 06_4DH**	**06_37H, 06_4AH, 06_4DH**	**06_37H, 06_4AH, 06_4DH**
DPPD xmm, xmm, imm	8	12	Y
DPPS xmm, xmm, imm	12	15	Y
EMMS	10	10	Y
EXTRACTPS	4	4	Y
F2XM1	88	88	Y
FABS/FCHS/FCOM/FXCH	1	1	N
FADD/FSUB	1	3	N
FCOS	168	168	Y
FDECSTP/FINCSTP	0.5	1	N
FDIV	39	39	Y
FLDZ	277	277	Y
FMUL	1	5	N
FPATAN/FYL2X/FYL2XP1	296	296	Y
FPTAN/FSINCOS	281	281	Y
FRNDINT	25	25	Y
FSCALE	74	74	Y
FSIN	150	150	Y
FSQRT	40	40	Y
HADDPD/HSUBPD xmm, xmm	5	6	Y
HADDPS/HSUBPS xmm, xmm	6	6	Y
IDIV r8	34	34	Y
IDIV r16	35-40	35-40	Y
IDIV r32	35-47	35-47	Y
IDIV r64	49-135	49-135	Y
IMUL r32, r32	1	3	N
INSERTPS	1	1	N
MASKMOVDQU	5	5	Y
MOVAPD/MOVAPS/MOVDQA/MOVDQU/MOVUPD/MOVUPS xmm, xmm;	0.5	1	N
MOVD r32, xmm; MOVD xmm, r32	1	1	N
MOVDDUP/MOVHLPS/MOVLHPS/MOVSHDUP/MOVSLDUP	1	1	N
MOVDQ2Q/MOVQ/MOVQ2DQ	0.5	1	N
MOVSD/MOVSS xmm, xmm;	0.5	1	N
MPSADBW	5	7	Y
MULPD	4	7	Y
MULPS; MULSD	2	5	N
MULSS	1	4	N

Table 15-6. Silvermont Microarchitecture Instructions Latency and Throughput (Contd.)

Instruction	Throughput	Latency	MSROM
DisplayFamily_DisplayModel	**06_37H, 06_4AH, 06_4DH**	**06_37H, 06_4AH, 06_4DH**	**06_37H, 06_4AH, 06_4DH**
NEG/NOT r32	0.5	1	N
PACKSSDW/WB xmm, xmm; PACKUSWB xmm, xmm	1	1	N
PABSB/D/W xmm, xmm	0.5	1	N
PADDB/D/W xmm, xmm; PSUBB/D/W xmm, xmm	0.5	1	N
PADDQ/PSUBQ/PCMPEQQ xmm, xmm	4	4	Y
PADDSB/W; PADDUSB/W; PSUBSB/W; PSUBUSB/W	0.5	1	N
PALIGNR xmm, xmm	1	1	N
PAND/PANDN/POR/PXOR xmm, xmm	0.5	1	N
PAVGB/W xmm, xmm	0.5	1	N
PBLENDVB xmm, xmm	0.5	1	N
PCLMULQDQ xmm, xmm, imm	10	10	Y
PCMPEQB/D/W xmm, xmm	0.5	1	N
PCMPESTRI xmm, xmm, imm	21	21	Y
PCMPESTRM xmm, xmm, imm	17	17	Y
PCMPGTB/D/W xmm, xmm	0.5	1	N
PCMPGTQ/PHMINPOSUW xmm, xmm	2	5	N
PCMPISTRI xmm, xmm, imm	17	17	Y
PCMPISTRM xmm, xmm, imm	13	13	Y
PEXTRB/WD r32, xmm, imm	4	4	Y
PINSRB/WD xmm, r32, imm	1	1	N
PHADDD/PHSUBD xmm, xmm	6	6	Y
PHADDW/PHADDSW xmm, xmm	9	9	Y
PHSUBW/PHSUBSW xmm, xmm	9	9	Y
PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, xmm	2	5	N
PMAXSB/W/D xmm, xmm; PMAXUB/W/D xmm, xmm	0.5	1	N
PMINSB/W/D xmm, xmm; PMINUB/W/D xmm, xmm	0.5	1	N
PMOVMSKB r32, xmm	1	1	N
PMOVSXBW/BD/BQ/WD/WQ/DQ xmm, xmm	1	1	N
PMOVZXBW/BD/BQ/WD/WQ/DQ xmm, xmm	1	1	N
PMULDQ/PMULUDQ xmm, xmm	2	5	N
PMULHUW/PMULHW/PMULLW xmm, xmm	2	5	N
PMULLD xmm, xmm	11	11	Y
POPCNT r32, r32	1	3	N
PSHUFB xmm, xmm	5	5	Y
PSHUFD xmm, mem, imm	1	1	N
PSHUFHW; PSHUFLW; PSHUFW	1	1	N

Table 15-6. Silvermont Microarchitecture Instructions Latency and Throughput (Contd.)

Instruction	Throughput	Latency	MSROM
DisplayFamily_DisplayModel	**06_37H, 06_4AH, 06_4DH**	**06_37H, 06_4AH, 06_4DH**	**06_37H, 06_4AH, 06_4DH**
PSIGNB/D/W xmm, xmm	1	1	N
PSLLDQ/PSRLDQ xmm, imm; SHUFPD/SHUFPS	1	1	N
PSLLD/Q/W xmm, xmm	2	2	N
PSRAD/W xmm, imm;	1	1	N
PSRAD/W xmm, xmm;	2	2	N
PSRLD/Q/W xmm, imm;	1	1	N
PSRLD/Q/W xmm, xmm	2	2	N
PUNPCKHBW/DQ/WD; PUNPCKLBW/DQ/WD	1	1	N
PUNPCKHQDQ; PUNPCKLQDQ	1	1	N
RCPPS/RSQRTPS	8	9	Y
RCPSS/RSQRTSS	1	4	N
RDTSC	30	30	Y
ROUNDPD/PS	2	5	N
ROUNDSD/SS	1	4	N
ROL; ROR; SAL; SAR; SHL; SHR	1	1	N
SHLD/SHRD r32, r32, imm	2	2	N
SHLD/SHRD r32, r32, CL	4	4	Y
SHUFPD/SHUFPS xmm, xmm, imm	1	1	N
SQRTPD/PS	26	26	N
SQRTSD/SS	11	13	N
TEST r32, r32	0.5	1	N
UNPCKHPD; UNPCKHPS; UNPCKLPD, UNPCKLPS	1	1	N
XADD r32, r32	5	5	Y
XCHG r32, r32	5	5	Y

APPENDIX A
APPLICATION PERFORMANCE TOOLS

Intel offers an array of application performance tools that are optimized to take advantage of the Intel architecture (IA)-based processors. This appendix introduces these tools and explains their capabilities for developing the most efficient programs without having to write assembly code.

The following performance tools are available:

- **Compilers**
  - — Intel® C++ Compiler: a high-performance, optimized C and C++ cross compiler with the capability of offloading compute-intensive code to Intel® Many Integrated Core Architecture (Intel® MIC Architecture) as well as Intel® HD Graphics, and executing on multiple execution units by using Intel® Cilk™ parallel extensions.
  - — Intel® Fortran Compiler: a high-performance, optimized Fortran compiler.
- **Performance Libraries** — a set of software libraries optimized for Intel architecture processors.
  - — Intel® Integrated Performance Primitives (Intel® IPP): performance building blocks to boost embedded system performance.
  - — Intel® Math Kernel Library (Intel® MKL): a set of highly optimized linear algebra, Fast Fourier Transform (FFT), vector math, and statistics functions.
  - — Intel® Threading Building Blocks (Intel® TBB): a C and C++ template library for creating high performance, scalable parallel applications.
  - — Intel® Data Analytics Acceleration Library (Intel® DAAL): C++ and Java API library of optimized analytics building blocks for all data analysis stages, from data acquisition to data mining and machine learning. Essential for engineering high performance Big Data applications.
- **Performance Profilers** — performance profilers collect, analyze, and display software performance data for tuning CPU, GPU, threading, vectorization and MPI parallelism from the system-wide view down to a specific line of code.
  - — Intel® VTune™ Amplifier XE: performance profiler.
  - — Intel® Graphics Performance Analyzers (Intel® GPA) - a set of performance analyzers for graphics applications.
  - — Intel® Advisor: vectorization optimization and thread prototyping.
  - — Intel® Trace Analyzer and Collector: MPI communications performance profiler and correctness checker.
- **Debuggers**
  - — Intel® Inspector: memory and thread debugger.
  - — Intel® Application Debugger.
  - — Intel® JTAG Debugger.
  - — Intel® System Debugger.
- **Cluster Tools**
  - — Intel® MPI Library: high-performance MPI library.
  - — Intel® MPI Benchmarks: a set of MPI kernel tests to verify the performance of your cluster or MPI implementation.

The performance tools listed above can be found in the following product suites.

- **Intel® Parallel Studio XE**
  - — Intel® Media Server Studio.
  - — Intel® Systems Studio.

A.1 COMPILERS

Intel compilers support several general optimization settings, including /O1, /O2, /O3, and /fast. Each of them enables a number of specific optimization options. In most cases, /O2 is recommended over /O1 because the /O2 option enables function expansion, which helps programs that have many calls to small functions. The /O1 may sometimes be preferred when code size is a concern. The /O2 option is on by default.

The /Od (-O0 on Linux) option disables all optimizations. The /O3 option enables more aggressive optimizations, most of which are effective only in conjunction with processor-specific optimizations described below.

The /fast option maximizes speed across the entire program. For most Intel 64 and IA-32 processors, the "/fast" option is equivalent to "/O3 /Qipo /Qprec-div- /fp:fast=2 /QxHost" on Windows*, "-ipo -O3 -no-prec-div -static -fp-model fast=2 -xHost" on Linux, and "-ipo -mdynamic-no-pic -O3 -no-prec-div -fp-model fast=2 -xHost" on OS X*.

All the command-line options are described in Intel® C++ Compiler documentation.

A.1.1 Recommended Optimization Settings for Intel® 64 and IA-32 Processors

Table A-1 lists some examples of recommended compiler options for generating code for Intel processors. Table A-1 also applies to code targeted to run in compatibility mode on an Intel 64 processor, but does not apply to running in 64-bit mode. For an up-to-date list see this article: https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations/.

Table A-1. Recommended Processor Optimization Options

Need	Recommendation	Comments
Best performance on Intel processors utilizing Intel® AVX2 instructions.	• /QxCORE-AVX2 (-x CORE-AVX2 on Linux and Mac OS)	• Single code path.
Best performance on Intel processors utilizing Intel® AVX2 instructions.	• /QaxCORE-AVX2 (-ax CORE-AVX2 on Linux and Mac OS)	• Multiple code paths are generated. • Be sure to validate your application on all systems where it may be deployed.
Best performance on Intel processors utilizing Intel SSE4.2 instructions.	• /QxSSE4.2 (-xSSE4.2 on Linux and Mac OS)	• Single code path.
Best performance on Intel processors utilizing Intel SSE4.2 instructions.	• /QaxSSE4.2 (-axSSE4.2 on Linux and Mac OS)	• Multiple code paths are generated. • Be sure to validate your application on all systems where it may be deployed.

A.1.2 Vectorization and Loop Optimization

The Intel C++ and Fortran Compiler's vectorization feature can detect sequential data access by the same instruction and transforms the code to use Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3 and Intel SSE4, depending on the target processor platform. The vectorizer supports the following features:

- Multiple data types: Float/double, char/short/int/long (both signed and unsigned), _Complex float/double are supported.
- Step by step diagnostics: Through the /Qopt-report /Qopt-report-phase (-qopt-report -qopt-report-phase on Linux and Mac OS) switch, the vectorizer can identify, line-by-line and variable-by-variable, what code was vectorized, what code was not vectorized, and more importantly, why it was not vectorized. This feedback gives the developer the information necessary to slightly adjust or restructure code, with dependency directives and restrict keywords, to allow vectorization to occur.

- Advanced dynamic data-alignment strategies: Alignment strategies include loop peeling and loop unrolling. Loop peeling can generate aligned loads, enabling faster application performance. Loop unrolling matches the prefetch of a full cache line and allows better scheduling.
- Portable code: By using appropriate Intel compiler switches to take advantage new processor features, developers can avoid the need to rewrite source code.

The processor-specific vectorizer switch options are: /Qx<CODE> and /Qax<CODE> (-x<CODE> and -xa<CODE> on Linux and Mac OS). The compiler provides a number of other vectorizer switch options that allow you to control vectorization. The latter switches require one of these switches to be on. The default is off.

A.1.2.1 Multithreading with OpenMP*

Both the Intel C++ and Fortran Compilers support shared memory parallelism using OpenMP compiler directives, library functions and environment variables. OpenMP directives are activated by the compiler switch /Qopenmp (-openmp on Linux and Mac OS). The available directives are described in the Compiler User's Guides available with the Intel C++ and Fortran Compilers. For information about the OpenMP standard, see http://www.openmp.org.

A.1.2.2 Automatic Multithreading

Both the Intel C++ and Fortran Compilers can generate multithreaded code automatically for simple loops with no dependencies. This is activated by the compiler switch /Qparallel (-parallel in Linux and Mac OS).

A.1.3 Inline Expansion of Library Functions (/Oi, /Oi-)

The compiler inlines a number of standard C, C++, and math library functions by default. This usually results in faster execution. Sometimes, however, inline expansion of library functions can cause unexpected results. For explanation, see the Intel C++ Compiler documentation.

A.1.4 Interprocedural and Profile-Guided Optimizations

The following are two methods to improve the performance of your code based on its unique profile and procedural dependencies.

A.1.4.1 Interprocedural Optimization (IPO)

You can use the /Qip (-ip in Linux and Mac OS) option to analyze your code and apply optimizations between procedures within each source file. Use multifile IPO with /Qipo (-ipo in Linux and Mac OS) to enable the optimizations between procedures in separate source files.

A.1.4.2 Profile-Guided Optimization (PGO)

Creates an instrumented program from your source code and special code from the compiler. Each time this instrumented code is executed, the compiler generates a dynamic information file. When you compile a second time, the dynamic information files are merged into a summary file. Using the profile information in this file, the compiler attempts to optimize the execution of the most heavily travelled paths in the program.

Profile-guided optimization is particularly beneficial for the Pentium 4 and Intel Xeon processor family. It greatly enhances the optimization decisions the compiler makes regarding instruction cache utilization and memory paging. Also, because PGO uses execution-time information to guide the optimizations, branch-prediction can be significantly enhanced by reordering branches and basic blocks to keep the most commonly used paths in the microarchitecture pipeline, as well as generating the appropriate branch-hints for the processor.

When you use PGO, consider the following guidelines:

- Minimize the changes to your program after instrumented execution and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated.

NOTE

The compiler issues a warning that the dynamic information corresponds to a modified function.

- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.

For more on code optimization options, see the Intel C++ Compiler documentation.

A.1.5 Intel® Cilk Plus

Intel Cilk Plus is an Intel C/C++ compiler extension with only 3 keywords that simplifies implementing simple loop and task parallel applications. It offers superior functionality by combining vectorization features with high-level loop-type data parallelism and tasking.

A.2 PERFORMANCE LIBRARIES

The Intel Performance Libraries implement a number of optimizations that are discussed throughout this manual. Examples include architecture-specific tuning such as loop unrolling, instruction pairing and scheduling; and memory management with explicit and implicit data prefetching and cache tuning.

The Libraries take advantage of the parallelism in the SIMD instructions using MMX technology, Intel Streaming SIMD Extensions (Intel SSE), Intel Streaming SIMD Extensions 2 (Intel SSE2), and Intel Streaming SIMD Extensions 3 (Intel SSE3). These techniques improve the performance of computationally intensive algorithms and deliver hand coded performance in a high level language development environment.

For performance sensitive applications, the Intel Performance Libraries free the application developer from the time consuming task of assembly-level programming for a multitude of frequently used functions. The time required for prototyping and implementing new application features is substantially reduced and most important, the time to market is substantially improved. Finally, applications developed with the Intel Performance Libraries benefit from new architectural features of future generations of Intel processors simply by relinking the application with upgraded versions of the libraries.

The library set includes the Intel Integrated Performance Primitives (Intel IPP), Intel Math Kernel Library (Intel MKL) and Intel Threading Building Blocks (Intel TBB).

A.2.1 Intel® Integrated Performance Primitives (Intel® IPP)

Intel Integrated Performance Primitives for Linux and Windows: IPP is a cross-platform software library which provides a range of library functions for video decode/encode, audio decode/encode, image color conversion, computer vision, data compression, string processing, signal processing, image processing, JPEG decode/encode, speech recognition, speech decode/encode, cryptography plus math support routines for such processing capabilities.

These ready-to-use functions are highly optimized using Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) instruction sets. With a single API across the range of platforms, the users can have platform compatibility and reduced cost of development.

A.2.2 Intel® Math Kernel Library (Intel® MKL)

The Intel Math Kernel Library for Linux, Windows and OS X: MKL is composed of highly optimized mathematical functions for engineering, scientific and financial applications requiring high performance on Intel platforms. The functional areas of the library include linear algebra consisting of LAPACK and BLAS, Discrete Fourier Transforms (DFT), vector transcendental functions (vector math library/VML) and vector statistical functions (VSL). Intel MKL is optimized for the latest features and capabilities of the Intel® Itanium® , Intel® Xeon®, Intel® Pentium® 4, and Intel® Core2 Duo processor-based systems. Special attention has been paid to optimizing multi-threaded performance for the new Quad-Core Intel® Xeon® processor 5300 series.

A.2.3 Intel® Threading Building Blocks (Intel® TBB)

Intel TBB is a C++ template library for creating reliable, portable, and scalable parallel applications. Use Intel TBB for a simple and rapid way of developing robust task-based parallel applications that scale to available processor cores, are compatible with multiple environments, and are easier to maintain.

Intel TBB is validated and commercially supported on Windows, Linux and OS X* platforms. It is also available on FreeBSD*, IA Solaris*, XBox* 360, and PowerPC-based systems via the open source community.

A.2.4 Benefits Summary

The overall benefits the libraries provide to the application developers are as follows:

- **Time-to-Market** — Low-level building block functions that support rapid application development, improving time to market.
- **Performance** — Highly-optimized routines with a C interface that give Assembly-level performance in a C/C++ development environment (Intel MKL also supports a Fortran interface).
- **Platform tuned** — Processor-specific optimizations that yield the best performance for each Intel processor.
- **Compatibility** — Processor-specific optimizations with a single application programming interface (API) to reduce development costs while providing optimum performance.
- **Threaded application support** — Applications can be threaded with the assurance that the Intel MKL and Intel IPP functions are safe for use in a threaded environment.

A.3 PERFORMANCE PROFILERS

Intel® serial and parallel processing profiling tools locate performance bottlenecks without recompilation and with very low overhead, and provide quick access to scaling information for faster and improved decision making. The profiling tools enable evaluation of all sizes of Intel® processor based systems, from embedded systems through supercomputers, to help you improve application performance.

A.3.1 Intel® VTune™ Amplifier XE

Intel® VTune™ Amplifier XE[1] is a powerful threading and performance optimization tool for Windows and Linux. Use the VTune Amplifier to fine-tune for optimal performance, ensuring cores are fully exploited and new processor capabilities are supported to the fullest.

The sections that follow briefly describe the major features of the VTune Amplifier. For more details on these features, run the VTune Amplifier and see the online documentation.

1 For additional information, see: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/?wapkw=vtune

A.3.1.1 Hardware Event-Based Sampling Analysis

VTune Amplifier introduces a set of microarchitecture analysis types based on the event-based sampling data collection and targeted for the Intel® Core™ 2 processor family, processors based on the Intel processors. Depending on the analysis type, the VTune Amplifier monitors a set of hardware events and displays collected data as raw event count (for example, cache misses, clock ticks, and instructions retired) and as performance metrics. Each metric is an event ratio with its own threshold values. As soon as the performance of a program unit per metric exceeds the threshold, the VTune Amplifier marks this value as a performance issue (in pink) and provides recommendations how to fix it.

See Chapter 19, "Performance Monitoring Events," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B* for event lists available on various Intel processors.

A.3.1.2 Algorithm Analysis

VTune Amplifier introduces a set of algorithm analysis types based on the user-mode sampling and tracing collection:

- **Basic Hotspots** analysis that helps understand the application execution flow and identify sections of code that took a long time to execute (hotspots). A large number of samples collected at a specific process, thread, or module can imply high processor utilization and potential performance bottlenecks. Some hotspots can be removed, while other hotspots are fundamental to the application functionality and cannot be removed. VTune Amplifier creates a list of functions in your application ordered by the amount of time spent in a function. It also detects the call stacks for each of these functions so you can see how the hot functions are called.
- **Locks and Waits** analysis that helps identify the cause of the ineffective processor utilization. One of the most common problems is threads waiting too long on synchronization objects (locks). Performance suffers when waits occur while cores are under-utilized. During the Locks and Waits analysis you can estimate the impact each synchronization object introduces to the application and understand how long the application was required to wait on each synchronization object, or in blocking APIs, such as sleep and blocking I/O.
- **Concurrency** analysis that helps identify hotspot functions where processor utilization is poor. When cores are idle at a hotspot, you have an opportunity to improve performance by getting those cores working for you.

A.3.1.3 Platform Analysis

You may enable the VTune Amplifier to collect platform-wide metrics for applications that use a Graphics Processing Unit (GPU) for rendering, video processing, and computations. Use the CPU/GPU Concurrency analysis as a starting point to understand the code execution on the various CPU and GPU cores in your system and identify whether your target application is GPU or CPU bound.

A.4 THREAD AND MEMORY CHECKERS

Intel® tools combine threading and memory error checking into one powerful error checking tool to help increase the reliability, security, and accuracy of your applications.

A.4.1 Intel® Inspector

Intel® Inspector provides thread debugging analysis for higher performing parallel applications (find: data races, deadlocks, thread and sync APIs used and memory accesses between threads) and memory checking analysis for serial and parallel applications (find: memory leaks and memory corruption, memory allocation and deallocation API mismatches, and inconsistent memory API usage).

Intel® Inspector enhances developer productivity and facilitates application reliability by effectively finding crucial memory and threading defects early in the development cycle. It gives detailed insights into application memory and threading behavior to improve application reliability. The powerful thread checker and debugger makes it easier to find latent errors on the executed code path. It also finds inter-

mittent and non-deterministic errors, even if the error-causing timing scenario does not happen. In addition, developers can test their code more often, without the need to use special test builds or compilers.

A.5 VECTORIZATION ASSISTANT

A.5.1 Intel® Advisor

The Intel Advisor is the vectorization assistant and threading prototyping tool that simplifies threading, parallelizing, and vectorizing your source code by identifying those areas in your applications where vectorization and/or threading parallelism would have the greatest impact.

A.6 CLUSTER TOOLS

The Intel Parallel Studio XE, Cluster Edition helps you develop, analyze and optimize performance of parallel applications for clusters using IA-32, IA-64, and Intel® 64 architectures. The Cluster Edition includes the following tools for developing code for clusters: Intel® Trace Analyzer and Collector, Intel MPI Library, and Intel MPI Benchmarks.

A.6.1 Intel® Trace Analyzer and Collector

The Intel® Trace Analyzer and Collector[2] helps to provide information critical to understanding and optimizing application performance on clusters by quickly finding performance bottlenecks in MPI communication. It supports Intel® architecture-based cluster systems, features a high degree of compatibility with current standards, and includes trace file idealization and comparison, counter data displays, performance assistant and an MPI correctness checking library. Analyze MPI performance, speed up parallel application runs, locate hotspots and bottlenecks, and compare trace files with graphics providing extensively detailed analysis and aligned timelines.

A.6.1.1 MPI Performance Snapshot

The MPI Performance Snapshot (MPS) is a scalable lightweight performance tool for MPI applications. It collects a variety of MPI application statistics (such as communication, activity, and load balance) and presents it in an easy-to-read format. MPS combines lightweight statistics from the Intel® MPI Library with OS and hardware-level counters to provide you with high-level overview of your application. The tool is provided as part of the Intel® Trace Analyzer and Collector installation.

A.6.2 Intel® MPI Library

The Intel MPI Library is a multi-fabric message passing library that implements the Message Passing Interface, v2 (MPI-2) specification. It provides a standard library across Intel® platforms. The Intel MPI Library supports multiple hardware fabrics including InfiniBand, Myrinet*, and Intel® True Scale Fabric. Intel® MPI Library covers all your configurations by providing an accelerated universal, multi-fabric layer for fast interconnects via the Direct Access Programming Library (DAPL) methodology. Develop MPI code independent of the fabric, knowing it will run efficiently on whatever fabric is chosen by the user at runtime.

Intel MPI Library dynamically establishes the connection, but only when needed, which reduces the memory footprint. It also automatically chooses the fastest transport available. The fallback to sockets at job startup avoids the chance of execution failure even if the interconnect selection fails. This is especially helpful for batch computing. Any products developed with Intel MPI Library are assured run time

2 Intel® Trace Analyzer and Collector is only available as part of Intel® Cluster Studio or Intel® Cluster Studio XE.

compatibility since your users can download Intel's free runtime environment kit. Application performance can also be increased via the large message bandwidth advantage from the optional use of DAPL inside a multi-core or SMP node.

A.6.3 Intel® MPI Benchmarks

The Intel MPI Benchmarks will help enable an easy performance comparison of MPI functions and patterns, the benchmark features improvements in usability, application performance, and interoperability.

A.7 INTEL® ACADEMIC COMMUNITY

You can find information on classroom training offered by the Intel Academic Community at http://software.intel.com/en-us/articles/intel-academic-community/. Find general information for developers at http://software.intel.com/en-us/.

APPENDIX B
USING PERFORMANCE MONITORING EVENTS

Performance monitoring events provide facilities to characterize the interaction between programmed sequences of instructions and microarchitectural sub-systems. Performance monitoring events are described in Chapter 18 and Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.*

The first part of this chapter provides information on the Top-Down Microarchitecture Analysis Method (TMAM) for analyzing performance bottlenecks when tuning for Intel microarchitectures. Section B.1 presents a generalized formalism that can adapt to several recent Intel microarchitectures. Instantiation of TMAM for the Skylake microarchitecture and examples are included.

The rest of this chapter covers information that is useful for previous generations of Intel microarchitectures.

B.1 TOP-DOWN ANALYSIS METHOD

This section describes the Top-down Microarchitecture Analysis Method (TMAM) for identifying performance bottlenecks in out-of-order cores. The general hierarchical framework and the spirit of the hierarchical technique can apply to many out-of-order microarchitectures.

TMAM simplifies cycle-accounting (the process of identifying costs of performance bottlenecks, also often called CPI breakdown) using microarchitecture-independent metrics organized in one simple hierarchy.

Figure B-1 depicts the hierarchical approach to classify performance bottlenecks correlating to major functional blocks of modern out-of-order microarchitectures. Using TMAM, the high-learning curve associated with each microarchitecture-generation is replaced by a structured drill-down that quickly guides you to true performance limiters. This enables analyzing performance without learning all the microarchitecture details.

The advantage of this top-down hierarchical framework is a structured approach to drill down and guide you toward the likely area of microarchitecture to investigate. Weights are assigned to nodes in the tree to enable you to focus your analysis efforts on issues that indeed matter and disregard insignificant issues.

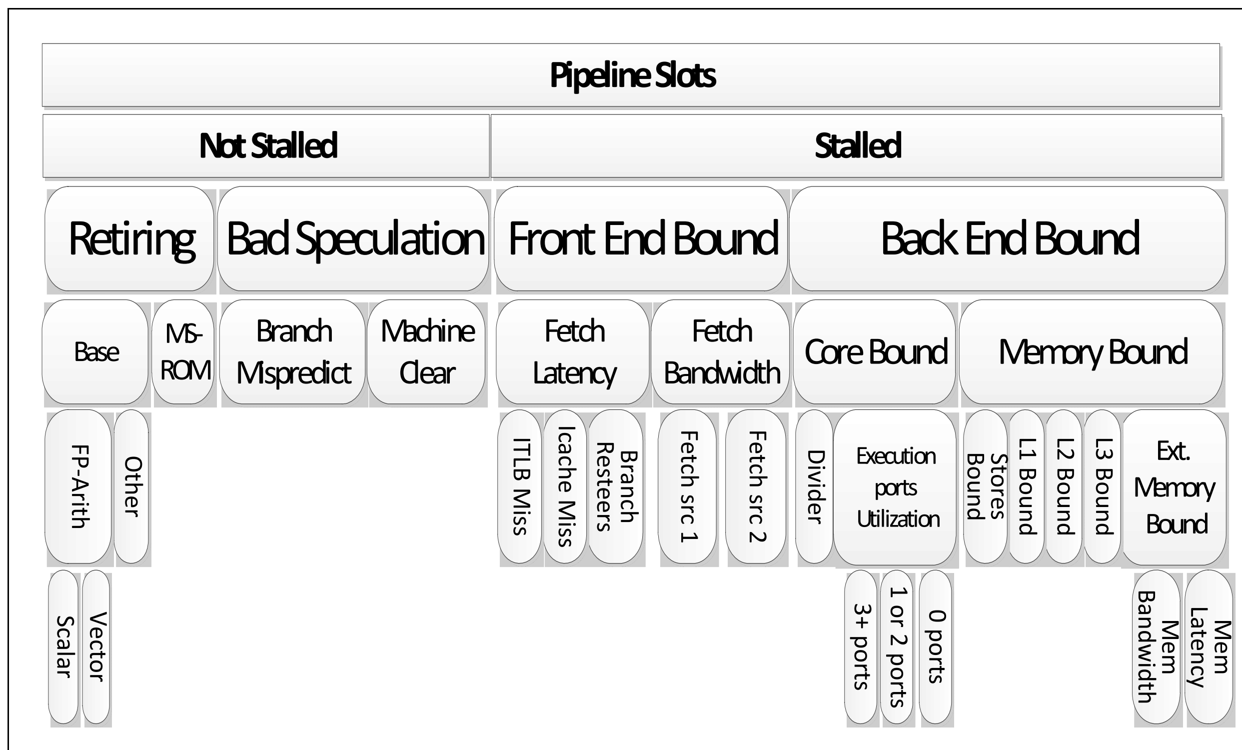


Figure B-1. General TMAM Hierarchy for Out-of-Order Microarchitectures

For example, if an application is significantly hurt by instruction fetch issues; TMAM categorizes it as Front End Bound at the uppermost level of the tree. A user/tool can drill down and focus only on the front end sub-tree. The drill down is recursively performed until a tree-leaf is reached. A leaf can point to a specific stall of the workload, or it can denote a subset of issues with a common micro-architectural symptom which are likely to limit the application's performance.

TMAM was first developed in conjunction with the performance monitoring capability of the Sandy Bridge microarchitecture. The methodology is refined with subsequent generations to support multiple microarchitecture generations and enhanced by subsequent PMU capabilities.

B.1.1 Top-Level

At the top-level, TMAM classifies pipeline-slots into four main categories:

- Front End Bound.
- Back End Bound.
- Bad Speculation.
- Retiring.

The latter two denote non-stalled slots while the former two denote stalls, as illustrated in Figure B-1. Figure B-2 depicts a simple decision tree to start the drill down process.

- If a slot is utilized by some operation, it will be classified as Retiring or Bad Speculation, depending on whether it eventually gets retired (committed).
- Un-utilized slots are classified as Back End Bound if the back-end portion of the pipeline is unable to accept more operations (a.k.a. back-end stall), or
    - — Front End Bound: indicating there were no operations (uops) delivered while there was no back-end stall.

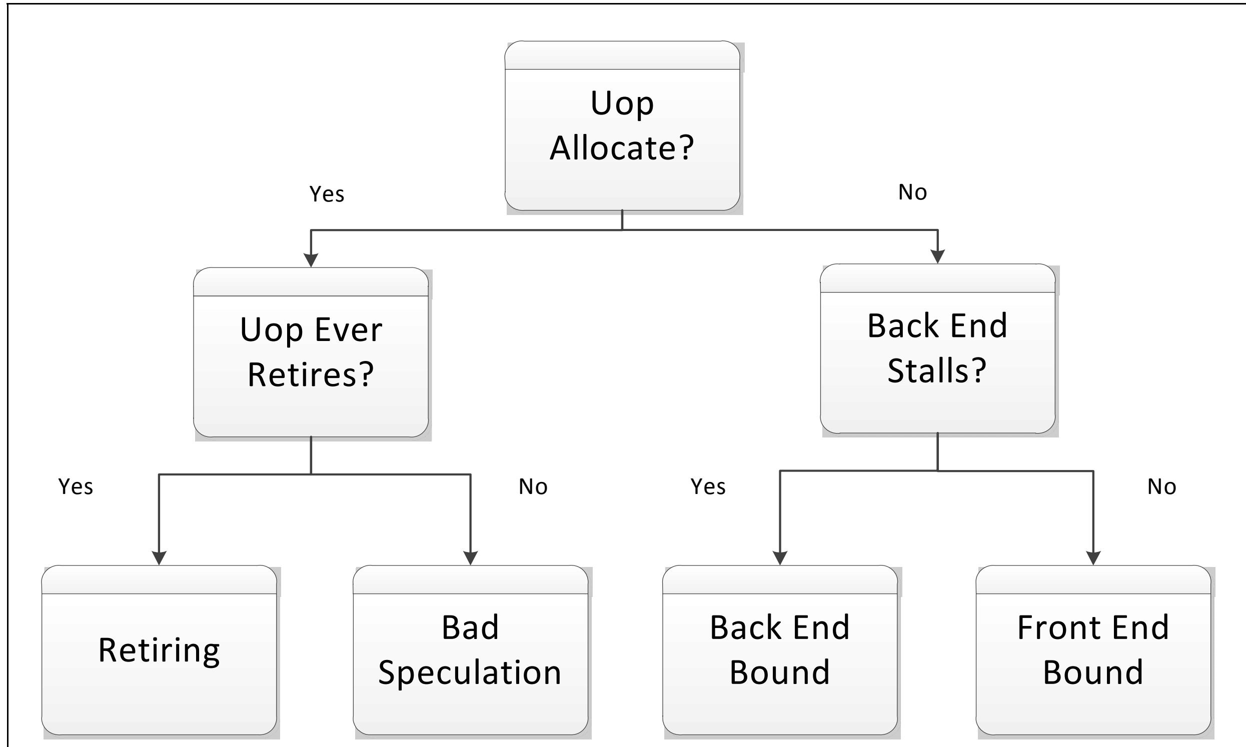


Figure B-2. TMAM's Top Level Drill Down Flowchart

The single entry point of division at a pipeline's issue-stage (allocation-stage) makes the four categories additive to the total possible slots. The classification at slots granularity (sub-cycle) makes the break-down very accurate and robust for superscalar cores, which is a necessity at the top-level.

Retiring denotes slots utilized by "good operations". Ideally, you want to see all slots attributed here since it correlates with Instructions Per Cycle (IPC). Nevertheless, a high Retiring fraction does not necessary mean there is no room for speedup.

Bad Speculation denotes slots wasted due to all aspects of incorrect speculations. It includes: (a) slots of operations that do not eventually retire, and (b) slots where the issue pipeline was blocked due to recovery from earlier mis-speculations. Note there is a third portion covered by Branch_Resteers1. This category can be split per type of speculation. For example, Branch Mispredicts and Machine Clears cover control-flow and data mis-speculation, respectively.

Front End Bound denotes when the pipeline's front end under-supplies the back end. Front end is the portion of the pipeline responsible for delivering operations to be executed later by the back end. This category is further classified into Fetch Latency (for example, ICache or ITLB misses) and Fetch Band-width (for example, sub-optimal decoding).

Back End Bound denotes remaining stalled slots due to lack of required back-end resources to accept new operations. It is split into: Memory Bound which reflects execution stalls due to the memory subsystem, and Core Bound which reflects either pressure on the execution units (compute bound) or lack of Instructions-Level-Parallelism (ILP).

Following sections provide more details on these categories and nodes in next levels of the hierarchy.

B.1.2 Front End Bound

Front end denotes the portion of the pipeline where the branch predictor predicts the next address to fetch, streams of code bytes are fetched from ICache, parsed into instructions, and decoded into micro-

ops that can be executed later by the back end. **Front End Bound** denotes when the front end of the processor core under-supplies the back end. That is, there were fetch bubbles when the back end was ready to accept uops (micro-ops).

Dealing with front-end issues is tricky without TMAM, as they occur at the very beginning of the long and buffered pipeline. This often means transient issues will not dominate the actual performance and you should investigate these issues only when Front End Bound is flagged at the top-level. In many cases the front-end supply bandwidth can dominate the performance, especially when high IPC applies. This has led to the addition of dedicated units to hide the fetch pipeline latency and sustain required bandwidth such as the Loop Stream Detector (LSD), as well as Decoded I-cache (DSB).

TMAM further distinguishes between latency and bandwidth front-end stalls:

- An ICache miss is classified under **Fetch Latency**.
- Inefficiency in the instruction decoders is classified under **Fetch Bandwidth**.

Note that these metrics are defined in the top-down approach: **Fetch Latency** accounts for cases that lead to fetch starvation (the symptom of no uop delivery) regardless of what has caused that. Familiar i-cache and i-TLB misses fit here, but not only these. **Branch Resteers** accounts for fetch delays following pipeline flushes. Pipeline flushes can be caused by clear events such as branch misprediction or memory nukes. **Branch Resteers** are tightly coupled with Bad Speculation.

The methodology further classifies bandwidth issues per fetch-unit, inserting uops to the Micro-Op-Queue (see Figure 2-1). Instruction decoders are used to translate commonly-used x86 instructions into micro-ops that the rest of machine understands; that would be one fetch unit. Some x86 instructions require sophisticated micro-op flows, like CPUID, relying on the MSROM to supply the long micro-op flows; that would be the 2nd fetch unit, and so on. Different fetch units may have different supply bandwidths from one generation to another. Figure 2-1 provides additional details for the Skylake microarchitecture.

B.1.3 Back End Bound

Back End Bound reflects slots where no micro-ops are being delivered at the issue pipeline, due to lack of required resources for accepting them in the back end. Examples of performance issues attributed in this category include data-cache misses, or stalls due to the divider unit being overloaded.

Back End Bound is split into **Memory Bound** and **Core Bound**. This is achieved by breaking down back-end stalls based on execution units' occupation at every cycle. In order to sustain a maximum IPC, it is necessary to keep execution units busy. For example, in a four-slot-wide machine, if three or less micro-ops are executed in a steady state of some code, this would prevent it from achieving an optimal IPC of 4. These sub-optimal cycles are called ExecutionStalls.

B.1.4 Memory Bound

Memory Bound corresponds to execution stalls related to the cache and memory subsystems. These stalls usually manifest with execution units getting starved after a short while, like in the case of a load missing all caches. Many recent generations of Intel Core processors have three levels of cache hierarchy to hide latency of external memory. The first level has a data cache (L1D). L2 is the second level shared instruction and data cache, which is private to each core. L3 is shared among all the processors cores within a physical package.

The out-of-order scheduler can dispatch micro-ops into multiple execution units for execution. While these micro-ops were executing in-flight, some of the memory access latency exposure for data can be hidden by keeping the execution units busy with useful micro-ops that do not depend on pending memory accesses. Thus for common cases, the true penalty for a memory access is when the scheduler has nothing ready to feed the execution units. It is likely that further micro-ops are either waiting for the pending memory access, or depend on other unready micro-ops.

ExecutionStalls span several sub-categories, each associated with a particular cache level and depending on the demanded data being satisfied by the respective cache level. In some situations, an Execution-

Stall can experience significant delay, greater than the nominal latency of the corresponding cache level, while no demand-load is missing that cache level.

For example, L1D cache often has short latency which is comparable to ALU stalls (or waiting for completion of some commonly-used execution units like floating-point adds/multiplies or integer multiplies). Yet in certain scenarios, like a load blocked from forward data from an earlier store to an overlapping address, this load might suffer a high effective latency while eventually being satisfied by L1D. In such a scenario, the in-flight load will last for a long period without missing L1D. Hence, it gets tagged under L1 Bound. Load blocks due to 4K Aliasing is another scenario with the same symptom.

Additionally, ExecutionStalls related to store operations are treated in the **Stores Bound** category. Store operations are buffered and executed post-retirement due to memory ordering requirements. Typically, store operations have small impact on performance, but they cannot be completely neglected. TMAM defines Stores Bound as fraction of cycles with low execution ports utilization and high number of stores consuming resources needed for buffering the stores.

Data TLB misses is categorized under various Memory Bound sub-nodes. For example, if a TLB translation is satisfied by L1D, it is tagged under L1 Bound.

Lastly, a simple heuristic is used to distinguish **MEM Bandwidth** and **MEM Latency** under **Ext. Memory Bound**. The heuristic uses occupancy of requests pending on data return from the memory controller. Whenever the occupancy exceeds a high threshold, say 70% of max number of requests, the memory controller can serve simultaneously, TMAM flags this as potentially limited by the memory bandwidth. The remainder fraction will be attributed to memory latency.

B.1.5 Core Bound

Core Bound corresponds to pressure on the execution units or lack of Instructions-Level-Parallelism (ILP) in your program. Core bound stalls can manifest either with short execution starvation periods, or with sub-optimal execution port utilization, which makes it more difficult to identify. For example, a long latency divide operation might serialize execution, while pressure on an execution port that serves specific types of micro-ops might manifest as a small number of ports utilized in a cycle.

Core Bound issues often can be mitigated with better code generation. For example, a sequence of dependent arithmetic operations would be classified as Core Bound. A compiler may relieve this stall with better instruction scheduling. Vectorization can mitigate Core Bound issues as well.

B.1.6 Bad Speculation

Bad Speculation reflects slots wasted due to incorrect speculations. These include two portions:

- Slots used to issue micro-ops that do not eventually retire.
- Slots in which the issue pipeline was blocked due to recovery from earlier mis-speculations.

For example, micro-ops issued in the shadow of a mispredicted branch are accounted in this category. Note the third portion of a misprediction penalty deals with how quick is the fetch from the correct target. This is accounted in **Branch Resteers** as it may overlap with other front-end stalls.

Having Bad Speculation category at the Top-Level is a key principle in TMAM. It determines the fraction of the workload under analysis that is affected by incorrect execution paths, which in turn dictates the accuracy of observations listed in other categories. Furthermore, this permits nodes at lower levels to make use of some of the many traditional performance counter events, despite most of those counter events count speculatively. Hence, you should treat a high value in **Bad Speculation** as a "red flag" that needs to be investigated first, before looking at other categories. In other words, minimizing Bad Speculation not only improves utilization of the processor resources, but also increases confidence in metrics reported throughout the hierarchy.

TMAM further classifies **Bad Speculation** into **Branch Mispredict** and **Machine Clears** both with similar symptoms where the pipeline is flushed. Branch misprediction apply when the BPU incorrectly predicts the branch direction and/or target. Incorrect data speculation generated Memory Order Machine Clears (for example, due to memory disambiguation) is a subset of Machine Clears. The next steps to analyze these issues can be completely different. The first deals with how to make the program control

flow friendlier to the branch predictor, while the latter often points unexpected situations such as memory ordering machine clears or self modifying code.

B.1.7 Retiring

This category reflects slots utilized by “good micro-ops” – issued micro-ops that get retired expeditiously without performance bottlenecks. Ideally, we would want to see all slots attributed to the **Retiring** category; that is Retiring of 100% of every slots correspond to hitting the maximal micro-ops retired per cycle of the given microarchitecture. For example, assuming one instruction is decoded into one micro-op, Retiring of 50% in one slot means an IPC of 2 was achieved in a four-wide machine. In other words, maximizing the **Retiring** category increases the IPC of your program.

Nevertheless, a high Retiring value does not necessary mean there is no room for more performance. Microcode sequences such as Floating Point (FP) assists typically hurt performance and can be avoided. They are isolated under **MSROM** in order to bring it to your attention.

A high Retiring value for non-vectorized code may be a good hint to vectorize the code. Doing so essentially lets more operations to be completed by single instruction/micro-op; hence improve performance. TMAM further breaks down the **Retiring->Base** category into **FP Arith**. with **Scalar** and **Vector** operations distinction. For more details see Matrix-Multiply use-case2.

B.1.8 TMAM and Skylake Microarchitecture

The performance monitoring capabilities in the Skylake microarchitecture is significantly enhanced over prior generations. TMAM benefits directly from the enhancement in the breadth of available counter events and in Precise Event Based Sampling (PEBS) capabilities. Figure B-3 shows Skylake microarchitecture’s support for TMAM, where the boxes in green indicates PEBS events are available.

The Intel Vtune Performance Analyzer allows the user to apply TMAM on many Intel microarchitectures. The reader may wish to consult the white paper available at https://software.intel.com/en-us/articles/how-to-tune-applications-using-a-top-down-characterization-of-microarchitectural-issues, and the use cases in the white paper for additional details.

B.1.8.1 TMAM Examples

Section 11.15.1 describes techniques of optimizing floating-point calculations involving latency and throughput considerations of FP MUL, FP ADD and FMA instructions. There is no explicit performance counter events that can directly detect exposures of latency issues of FP_ADD and FP_MUL instructions.

TMAM may be used to figure out when this performance issue is likely to be a performance limiter.

If the primary bottleneck is Backend_Bound->Core_Bound->Ports_Utilization and there is a significant measure in the GFLOPS metric, it is possible that the user code is hitting this issue. The user may consider optimizations listed in Section 11.15.1.

Section 11.3.1 describes possible performance issues of executing SSE code while the upper YMM state is dirty in Skylake Microarchitecture. To detect the performance issue associated with partial register dependence and associated blend cost on SSE code execution, TMAM can be used to monitor the rate of mixture of SSE operation and AVX operation on performance-critical SSE code whose source code did not directly execute AVX instructions.

If the primary bottleneck is Backend_Bound->Core_Bound, and there is a significant measure in Vector-MixRate metric, it is possible that the presence of Vector operation with mis-matched vector width was due to the extra blend operation on the upper YMM registers.

The VectorMixRate metric requires the UOPS_ISSUED.VECTOR_WIDTH_MISMATCH event that is available in the Skylake Microarchitecture. This event count Uops inserted at issue-stage in order to preserve upper bits of vector registers. This event counts the number of Blend Uops issued by the Resource Allocation Table (RAT) to the reservation station (RS) in order to preserve upper bits of vector registers.

Additionally, the metric uses the UOPS_ISSUED.ANY, which is common in recent Intel microarchitectures, as the denominator. The UOPS_ISSUED.ANY event counts the total number of Uops that the RAT issues to RS.

The VectorMixRate metric gives the percentage of injected blend uops out of all uops issued. Usually a VectorMixRate over 5% is worth investigating.

VectorMixRate[%] = 100 * UOPS_ISSUED.VECTOR_WIDTH_MISMATCH / UOPS_ISSUED.ANY

Note the actual penalty may vary as it stems from the additional data-dependency on the destination register the injected blend operations add.

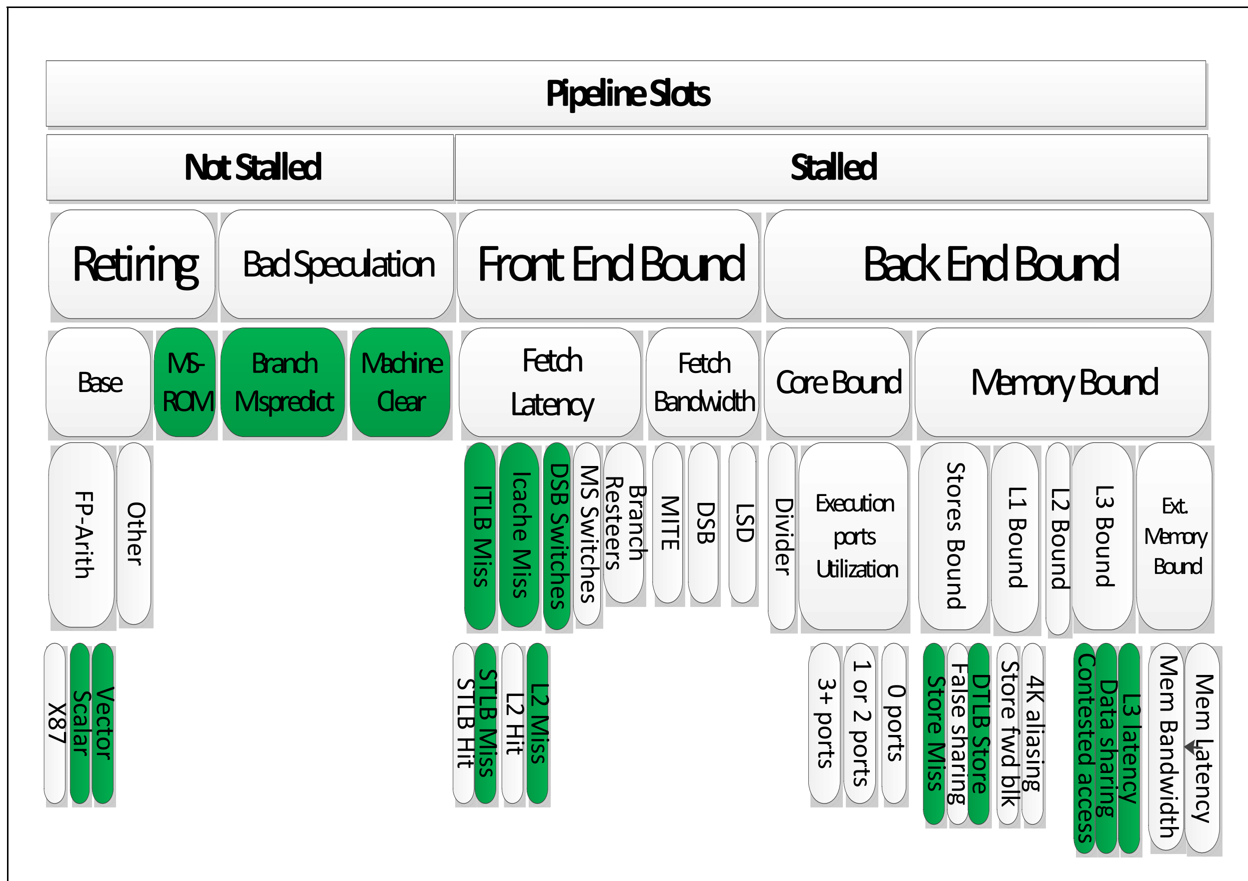


Figure B-3. TMAM Hierarchy Supported by Skylake Microarchitecture

B.2 INTEL® XEON® PROCESSOR 5500 SERIES

Intel Xeon processor 5500 series are based on the same microarchitecture as Intel Core i7 processors, see Section 2.5, "Intel® Microarchitecture Code Name Nehalem". In addition, Intel Xeon processor 5500 series support non-uniform memory access (NUMA) in platforms that have two physical processors, see Figure B-4. Figure B-4 illustrates 4 processor cores and an uncore sub-system in each physical processor. The uncore sub-system consists of L3, an integrated memory controller (IMC), and Intel QuickPath Interconnect (QPI) interfaces. The memory sub-system consists of three channels of DDR3 memory locally connected to each IMC. Access to physical memory connected to a non-local IMC is often described as a remote memory access.

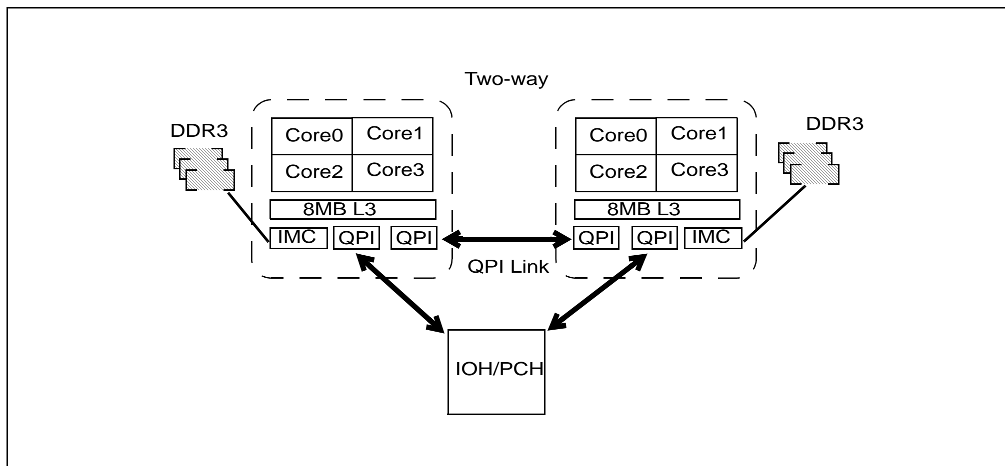


Figure B-4. System Topology Supported by Intel® Xeon® Processor 5500 Series

The performance monitoring events on Intel Xeon processor 5500 series can be used to analyze the interaction between software (code and data) and microarchitectural units hierarchically:

- Per-core PMU: Each processor core provides 4 programmable counters and 3 fixed counters. The programmable per-core counters can be configured to investigate front end/micro-op flow issues, stalls inside a processor core. Additionally, a subset of per-core PMU events support precise event-based sampling (PEBS). Load latency measurement facility is new in Intel Core i7 processor and Intel Xeon processor 5500.
- Uncore PMU: The uncore PMU provides 8 programmable counters and 1 fixed counter. The programmable per-core counters can be configured to characterize L3 and Intel QPI operations, local and remote data memory accesses.

The number and variety of performance counters and the breadth of programmable performance events available in Intel Xeon processor 5500 offer software tuning engineers the ability to analyze performance issues and achieve higher performance. Using performance events to analyze performance issues can be grouped into the following subjects:

- Cycle Accounting and Uop Flow.
- Stall Decomposition and Core Memory Access Events (non-PEBS).
- Precise Memory Access Events (PEBS).
- Precise Branch Events (PEBS, LBR).
- Core Memory Access Events (non-PEBS).
- Other Core Events (non-PEBS).
- Front End Issues.
- Uncore Events.

B.3 PERFORMANCE ANALYSIS TECHNIQUES FOR INTEL® XEON® PROCESSOR 5500 SERIES

The techniques covered in this chapter focuses on identifying opportunity to remove/reduce performance bottlenecks that are measurable at runtime. Compile-time and source-code level techniques are covered in other chapters in this document. Individual sub-sections describe specific techniques to identify tuning opportunity by examining various metrics that can be measured or derived directly from performance monitoring events.

B.3.1 Cycle Accounting and Uop Flow Analysis

The objectives, performance metrics and component events of the basic cycle accounting technique is summarized in Table B-1.

Table B-1. Cycle Accounting and Micro-ops Flow Recipe

	Summary
Objective	Identify code/basic block that had significant stalls
Method	Binary decomposition of cycles into “productive“ and “unproductive“ parts
PMU-Pipeline Focus	Micro-ops issued to execute
Event code/Umask	Event code B1H, Umask= 3FH for micro-op execution; Event code 3CH, Umak= 1, CMask=2 for counting total cycles
EvtSelc	Use CMask, Invert, Edge fields to count cycles and separate stalled vs. active cycles
Basic Equation	“Total Cycles“ = UOPS_EXECUTED.CORE_STALLS_CYCLES + UOPS_EXECUTED.CORE_ACTIVE_CYCLES
Metric	UOPS_EXECUTED.CORE_STALLS_CYCLES / UOPS_EXECUTED.CORE_STALLS_COUNT
Drill-down scope	Counting: Workload; Sampling: basic block
Variations	Port 0,1, 5 cycle counting for computational micro-ops execution.

Cycle accounting of executed micro-ops is an effective technique to identify stalled cycles for performance tuning. Within the microarchitecture pipeline, the meaning of micro-ops being “issued“, “dispatched“, “executed”, “retired” has precise meaning. This is illustrated in Figure B-5.

Cycles are divided into those where micro-ops are dispatched to the execution units and those where no micro-ops are dispatched, which are thought of as execution stalls.

“Total cycles” of execution for the code under test can be directly measured with CPU_CLK_UNHALTED.THREAD (event code 3CH, Umask= 1) and setting CMask = 2 and INV=1 in IA32_PERFEVTSELCn.

The signals used to count the memory access uops executed (ports 2, 3 and 4) are the only core events which cannot be counted per-logical processor. Thus, Event code B1H with Umask=3FH only counts on a per-core basis, and the total execution stall cycles can only be evaluated on a per core basis. If HT is disabled, this presents no difficulty to conduct per-thread analysis of micro-op flow cycle accounting.

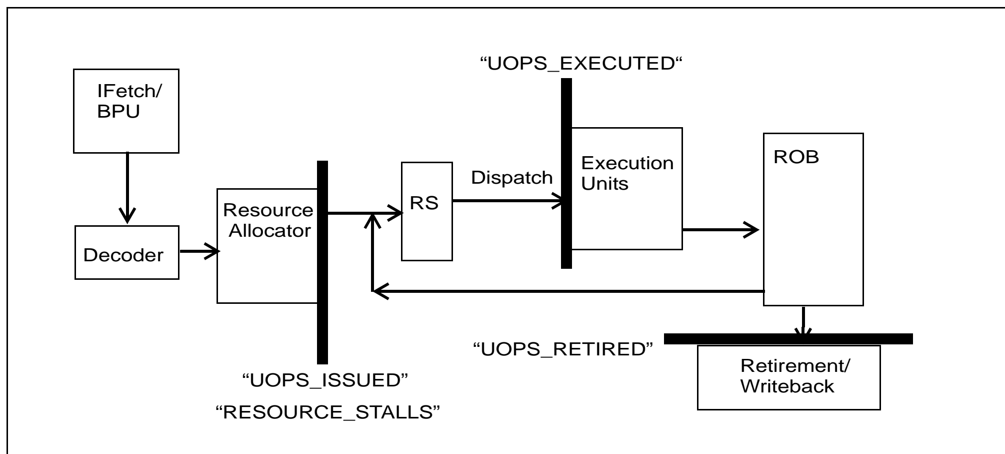


Figure B-5. PMU Specific Event Logic Within the Pipeline

The PMU signals to count uops_executed in port 0, 1, 5 can count on a per-thread basis even when HT is active. This provides an alternate cycle accounting technique when the workload under test interacts with HT.

The alternate metric is built from UOPS_EXECUTED.PORT015_STALL_CYCLES, using appropriate CMask, Inv, and Edge settings. Details of performance events are shown in Table B-2.

Table B-2. CMask/Inv/Edge/Thread Granularity of Events for Micro-op Flow

Event Name	Umask	Event Code	Cmask	Inv	Edge	All Thread
CPU_CLK_UNHALTED.TOTAL_CYCLES	0H	3CH	2	1	0	0
UOPS_EXECUTED.CORE_STALLS_CYCLES	3FH	B1H	1	1	0	1
UOPS_EXECUTED.CORE_STALLS_COUNT	3FH	B1H	1	1	!	1
UOPS_EXECUTED.CORE_ACTIVE_CYCLES	3FH	B1H	1	0	0	1
UOPS_EXECUTED.PORT015_STALLS_CYCLES	40H	B1H	1	1	0	0
UOPS_RETIRED.STALL_CYCLES	1H	C2H	1	1	0	0
UOPS_RETIRED.ACTIVE_CYCLES	1H	C2H	1	0	0	0

B.3.1.1 Cycle Drill Down and Branch Mispredictions

While executed micro-ops are considered productive from the perspective of execution units being subscribed, not all such micro-ops contribute to forward progress of the program. Branch mispredictions can introduce execution inefficiencies in OOO processor that are typically decomposed into three components:

- Wasted work associated with executing the uops of the incorrectly predicted path.
- Cycles lost when the pipeline is flushed of the incorrect uops.

- Cycles lost while waiting for the correct uops to arrive at the execution units.

In processors based on Intel microarchitecture code name Nehalem, there are no execution stalls associated with clearing the pipeline of mispredicted uops (component 2). These uops are simply removed from the pipeline without stalling executions or dispatch. This typically lowers the penalty for mispredicted branches. Further, the penalty associated with instruction starvation (component 3) can be measured.

The wasted work within executed uops are those uops that will never be retired. This is part of the cost associated with mispredicted branches. It can be found through monitoring the flow of uops through the pipeline. The uop flow can be measured at 3 points in Figure B-5, going into the RS with the event UOPS_ISSUED, going into the execution units with UOPS_EXECUTED and at retirement with UOPS_RETIRED. The differences of between the upstream measurements and at retirement measure the wasted work associated with these mispredicted uops.

As UOPS_EXECUTED must be measured per core, rather than per thread, the wasted work per core is evaluated as:

Wasted Work = UOPS_EXECUTED.PORT234_CORE + UOPS_EXECUTED.PORT015_All_Thread - UOPS_RETIRED.ANY_ALL_THREAD.

The ratio above can be converted to cycles by dividing the average issue rate of uops. The events above were designed to be used in this manner without corrections for micro fusion or macro fusion.

A “per thread” measurement can be made from the difference between the uops issued and uops retired as the latter two of the above events can be counted per thread. It over counts slightly, by the mispredicted uops that are eliminated in the RS before they can waste cycles being executed, but this is usually a small correction:

Wasted Work/thread = (UOPS_ISSUED.ANY + UOPS_ISSUED.FUSED) - UOPS_RETIRED.ANY.

Table B-3. Cycle Accounting of Wasted Work Due to Misprediction

	Summary
Objective	Evaluate uops that executed but not retired due to misprediction
Method	Examine uop flow differences between execution and retirement
PMU-Pipeline Focus	Micro-ops execute and retirement
Event code/Umask	Event code B1H, Umask= 3FH for micro-op execution; Event code C2H, Umask= 1, AllThread=1 for per-core counting
EvtSelc	Zero CMask, Invert, Edge fields to count uops
Basic Equation	“Wasted work“ = UOPS_EXECUTED.PORT234_CORE + UOPS_EXECUTED.PORT015_ALL_THREAD - UOPS_RETIRED.ANY_ALL_THREAD
Drill-down scope	Counting: Branch misprediction cost
Variations	Divide by average uop issue rate for cycle accounting. Set AllThread=0 to estimate per-thread cost.

The third component of the misprediction penalty, instruction starvation, occurs when the instructions associated with the correct path are far away from the core and execution is stalled due to lack of uops in the RAT. Because the two primary cause of uops not being issued are either front end starvation or resource not available in the back end. So we can explicitly measured at the output of the resource allocation as follows:

- Count the total number of cycles where no uops were issued to the OOO engine.

- Count the cycles where resources (RS, ROB entries, load buffer, store buffer, etc.) are not available for allocation.

If HT is not active, instruction starvation is simply the difference:

Instruction Starvation = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY.

When HT is enabled, the uop delivery to the RS alternates between the two threads. In an ideal case the above condition would then over count, as 50% of the issuing stall cycles may be delivering uops for the other thread. We can modify the expression by subtracting the cycles that the other thread is having uops issued.

> Instruction Starvation (per thread) = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY - UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD.

The per-thread expression above will over count somewhat because the resource_stall condition could exist on "this" thread while the other thread in the same core was issuing uops. An alternative might be:

> CPU_CLK_UNHALTED.THREAD - UOPS_ISSUED.CORE_CYCLES_ACTIVE-RESOURCE_STALLS.ANY.

The above technique is summarized in Table B-4.

Table B-4. Cycle Accounting of Instruction Starvation

	Summary
Objective	Evaluate cycles that uops issuing is starved after misprediction
Method	Examine cycle differences between uops issuing and resource allocation
PMU-Pipeline Focus	Micro-ops issue and resource allocation
Event code/Umask	Event code 0EH, Umak= 1, for uops issued. Event code A2H, Umask=1, for Resource allocation stall cycles
EvtSelc	Set CMask=1, Inv=1, fields to count uops issue stall cycles. Set CMask=1, Inv=0, fields to count uops issue active cycles. Use AllThread = 0 and AllThread=1 on two counter to evaluate contribution from the other thread for UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD
Basic Equation	"Instruction Starvation" (HT off) = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY;
Drill-down scope	Counting: Branch misprediction cost
Variations	Evaluate per-thread contribution with Instruction Starvation = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY - UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD

Details of performance events are shown in Table B-5.

Table B-5. CMask/Inv/Edge/Thread Granularity of Events for Micro-op Flow

Event Name	Umask	Event Code	Cmask	Inv	Edge	All Thread
UOPS_EXECUTED.PORT234_CORE	80H	B1H	0	0	0	1
UOPS_EXECUTED.PORT015_ALL_THREAD	40H	B1H	0	0	0	1
UOPS_RETIRED.ANY_ALL_THREAD	1H	C2H	0	0	0	1
RESOURCE_STALLS.ANY	1H	A2H	0	0	0	0
UOPS_ISSUED.ANY	1H	0EH	0	0	0	0
UOPS_ISSUED.STALL_CYCLES	1H	0EH	1	1	0	0
UOPS_ISSUED.ACTIVE_CYCLES	1H	0EH	1	0	0	0
UOPS_ISSUED.CORE_CYCLES_ACTIVE	1H	0EH	1	0	0	1

B.3.1.2 Basic Block Drill Down

The event INST_RETIRED.ANY (instructions retired) is commonly used to evaluate a cycles/instruction ratio (CPI). Another important usage is to determine the performance-critical basic blocks by evaluating basic block execution counts.

In a sampling tool (such as VTune Analyzer), the samples tend to cluster around certain IP values. This is true when using INST_RETIRED.ANY or cycle counting events. Disassembly listing based on the hot samples may associate some instructions with high sample counts and adjacent instructions with no samples.

Because all instructions within a basic block are retired exactly the same number of times by the very definition of a basic block. Drilling down the hot basic blocks will be more accurate by averaging the sample counts over the instructions of the basic block.

Basic Block Execution Count = Sum (Sample counts of instructions within basic block) * Sample_after_value / (number of instructions in basic block)

Inspection of disassembly listing to identify basic blocks associated with loop structure being a hot loop or not can be done systematically by adapting the technique above to evaluate the trip count of each loop construct. For a simple loop with no conditional branches, the trip count ends up being the ratio of the basic block execution count of the loop block to the basic block execution count of the block immediately before and/or after the loop block. Judicious use of averaging over multiple blocks can be used to improve the accuracy.

This will allow the user to identify loops with high trip counts to focus on tuning efforts. This technique can be implemented using fixed counters.

Chains of dependent long-latency instructions (fmul, fadd, imul, etc) can result in the dispatch being stalled while the outputs of the long latency instructions become available. In general there are no events that assist in counting such stalls with the exception of instructions using the divide/sqrt execution unit. In such cases, the event ARITH can be used to count both the occurrences of these instructions and the duration in cycles that they kept their execution units occupied. The event ARITH.CYCLES_DIV_BUSY counts the cycles that either the divide/sqrt execution unit was occupied.

B.3.2 Stall Cycle Decomposition and Core Memory Accesses

The decomposition of the stall cycles is accomplished through a standard approximation. It is assumed that the penalties occur sequentially for each performance impacting event. Consequently, the total loss of cycles available for useful work is then the number of events, N_i, times the average penalty for each type of event, P_i

Counted_Stall_Cycles = Sum (N_i* P_i)

This only accounts for the performance impacting events that are or can be counted with a PMU event. Ultimately there will be several sources of stalls that cannot be counted, however their total contribution can be estimated:

Unaccounted stall cycles = Stall_Cycles - Counted_Stall_Cycles =
UOPS_EXECUTED.CORE_STALLS_CYCLES - Sum (N_i* P_i)_both_threads

The unaccounted component can become negative as the sequential penalty model is overly simple and usually over counts the contributions of the individual microarchitectural issues.

As noted in Section B.3.1.1, UOPS_EXECUTED.CORE_STALL_CYCLES counts on a per core basis rather than on a per thread basis, the over counting can become severe. In such cases it may be preferable to use the port 0,1,5 uop stalls, as that can be done on a per thread basis:

Unaccounted stall cycles (per thread) = UOPS_EXECUTED.PORT015_THREADED_STALLS_CYCLES - Sum (N_i* P_i)

This unaccounted component is meant to represent the components that were either not counted due to lack of performance events or simply neglected during the data collection.

One can also choose to use the "retirement" point as the basis for stalls. The PEBS event, UOPS_RETIRED.STALL_CYCLES, has the advantage of being evaluated on a per thread basis and being having the HW capture the IP associated with the retiring uop. This means that the IP distribution will not be effected by STI/CLI deferral of interrupts in critical sections of OS kernels, thus producing a more accurate profile of OS activity.

B.3.2.1 Measuring Costs of Microarchitectural Conditions

Decomposition of stalled cycles in this manner should start by first focusing on conditions that carry large performance penalty, for example, events with penalties of greater than 10 cycles. Short penalty events (P < 5 cycles) can frequently be hidden by the combined actions of the OOO execution and the compiler. The OOO engine manages both types of situations in the instruction stream and strive to keep the execution units busy during stalls of either type due to instruction dependencies. Usually, the large penalty operations are dominated by memory access and the very long latency instructions for divide and sqrt.

The largest penalty events are associated with load operations that require a cacheline which is not in L1 or L2 of the cache hierarchy. Not only must we count how many occur, but we need to know what penalty to assign.

The standard approach to measuring latency is to measure the average number of cycles a request is in a queue:

Latency = Sum (CYCLES_Queue_entries_outstanding) /Queue_inserts

where "queue_inserts" refers to the total number of entries that caused the outstanding cycles in that queue. However, the penalty associated with each queue insert (i.e. cachemiss), is the latency divided by the average queue occupancy. This correction is needed to avoid over counting associated with overlapping penalties.

Avg_Queue_Depth= Sum (CYCLES_Queue_entries_outstanding) / Cycles_Queue_not_empty

The the penalty (cost) of each occurrence is

Penalty = Latency / Avg_Queue_Depth = Cycles_Queue_not_empty / Queue_inserts

An alternative way of thinking about this is to realize that the sum of all the penalties, for an event that occupies a queue for its duration, cannot exceed the time that the queue is not empty

Cycles_Queue_not_empty = Events * <Penalty>

The standard techniques described above are simple conceptually. In practice, the large amount of memory references in the workload and wide range of varying state/location-specific latencies made standard sampling techniques less practical. Using precise-event-based sampling (PEBS) is the preferred technique on processors based on Intel microarchitecture code name Nehalem.

The profiling the penalty by sampling (to localize the measurement in IP) is likely to have accuracy difficulties. Since the latencies for L2 misses can vary from 40 to 400 cycles, collecting the number of required samples will tend to be invasive.

The use of the precise latency event, that will be discussed later, provides a more accurate and flexible measurement technique when sampling is used. As each sample records both a load to use latency and a data source, the average latency per data source can be evaluated. Further as the PEBS hardware supports buffering the events without generating a PMI until the buffer is full, it is possible to make such an evaluation efficient without perturbing the workload intrusively.

A number of performance events in core PMU can be used to measure the costs of memory accesses that originated in the core and experienced delays due to various conditions, locality, or traffic due to cache coherence requirements. The latency of memory accesses vary, depending on locality of L3, DRAM attached to the local memory controller or remote controller, and cache coherency factors. Some examples of the approximate latency values are shown in Table B-6.

Table B-6. Approximate Latency of L2 Misses of Intel Xeon Processor 5500

Data Source	Latency
L3 hit, Line exclusive	~ 42 cycles
L3 Hit, Line shared	~ 63 cycles
L3 Hit, modified in another core	~ 73 cycles
Remote L3	100 - 150 cycles
Local DRAM	~ 50 ns
Remote DRAM	~ 90 ns

B.3.3 Core PMU Precise Events

The Precise Event Based Sampling (PEBS) mechanism enables the PMU to capture the architectural state and IP at the completion of the instruction that caused the event. This provides two significant benefit for profiling and tuning:

- The location of the eventing condition in the instruction space can be accurate profiled,
- Instruction arguments can be reconstructed in a post processing phase, using captured PEBS records of the register states.

The PEBS capability has been greatly expanded in processors based on Intel microarchitecture code name Nehalem, covering a large number of and more types of precise events.

The mechanism works by using the counter overflow to arm the PEBS data acquisition. Then on the next event, the data is captured and the interrupt is raised.

The captured IP value is sometimes referred to as IP +1, because at the completion of the instruction, the IP value is that of the next instruction.

By their very nature precise events must be “at-retirement” events. For the purposes of this discussion the precise events are divided into Memory Access events, associated with the retirement of loads and stores, and Execution Events, associated with the retirement of all instructions or specific non memory instructions (branches, FP assists, SSE uops).

B.3.3.1 Precise Memory Access Events

There are two important common properties to all precise memory access events:

- The exact instruction can be identified because the hardware captures the IP of the offending instruction. Of course the captured IP is that of the following instruction but one simply moves the samples up one instruction. This works even when the recorded IP points to the first instruction of a basic block because n such a case the offending instruction has to be the last instruction of the previous basic block, as branch instructions never load or store data, instruction arguments can be reconstructed in a post processing phase, using captured PEBS records of the register states.
- The PEBS buffer contains the values of all 16 general registers, R1-R16, where R1 is also called RAX. When coupled with the disassembly the address of the load or store can be reconstructed and used for data access profiling. The Intel® Performance Tuning Utility does exactly this, providing a wide variety of powerful analysis techniques

Precise memory access events mainly focus on loads as those are the events typically responsible for the very long duration execution stalls. They are broken down by the data source, thereby indicating the typical latency and the data locality in the intrinsically NUMA configurations. These precise load events are the only L2, L3 and DRAM access events that only count loads. All others will also include the L1D and/or L2 hardware prefetch requests. Many will also include RFO requests, both due to stores and to the hardware prefetchers.

All four general counters can be programmed to collect data for precise events. The ability to reconstruct the virtual addresses of the load and store instructions allows an analysis of the cacheline and page usage efficiency. Even though cachelines and pages are defined by physical address the lower order bits are identical, so the virtual address can be used.

As the PEBS mechanism captures the values of the register at completion of the instruction, one should be aware that pointer-chasing type of load operation will not be captured because it is not possible to infer the load instruction from the dereferenced address.

The basic PEBS memory access events falls into the following categories:

- MEM_INST_RETIRED: This category counts instruction retired which contain a load operation, it is selected by event code 0BH.
- MEM_LOAD_RETIRED: This category counts retired load instructions that experienced specific condition selected by the Umask value, the event code is 0CBH.
- MEM_UNCORE_RETIRED: This category counts memory instructions retired and received data from the uncore sub-system, it is selected by event code 0FH.
- MEM_STORE_RETIRED: This category counts instruction retired which contain a store operation, it is selected by event code 0CH.
- ITLE_MISS_RETIRED: This counts instruction retired which missed the ITLB, it is selected by event code 0C8H

Umask values and associated name suffixes for the above PEBS memory events are listed under the in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

The precise events listed above allow load driven cache misses to be identified by data source. This does not identify the "home" location of the cachelines with respect to the NUMA configuration. The exceptions to this statement are the events

MEM_UNCORE_RETIRED.LOCAL_DRAM and MEM_UNCORE_RETIRED.NON_LOCAL_DRAM. These can be used in conjunction with instrumented malloc invocations to identify the NUMA "home" for the critical contiguous buffers used in an application.

The sum of all the MEM_LOAD_RETIRED events will equal the MEM_INST_RETIRED.LOADS count.

A count of L1D misses can be achieved with the use of all the MEM_LOAD_RETIRED

events, except MEM_LOAD_RETIRED.L1D_HIT. It is better to use all of the individual MEM_LOAD_RETIRED events to do this, rather than the difference of MEM_INST_RETIRED.LOADS-MEM_LOAD_RETIRED.L1D_HIT because while the total counts of precise events will be correct, and they

will correctly identify instructions that caused the event in question, the distribution of the events may not be correct due to PEBS SHADOWING, discussed later in this section.

L1D_MISSES = MEM_LOAD_RETIRED.HIT_LFB + MEM_LOAD_RETIRED.L2_HIT + MEM_LOAD_RETIRED.L3_UNSHARED_HIT + MEM_LOAD_RETIRED.OTHER_CORE_HIT_HITM + MEM_LOAD_RETIRED.L3_MISS

The MEM_LOAD_RETIRED.L3_UNSHARED_HIT event merits some explanation. The inclusive L3 has a bit pattern to identify which core has a copy of the line. If the only bit set is for the requesting core (unshared hit) then the line can be returned from the L3 with no snooping of the other cores. If multiple bits are set, then the line is in a shared state and the copy in the L3 is current and can also be returned without snooping the other cores.

If the line is read for ownership (RFO) by another core, this will put the copy in the L3 into an exclusive state. If the line is then modified by that core and later evicted, the written back copy in the L3 will be in a modified state and snooping will not be required. MEM_LOAD_RETIRED.L3_UNSHARED_HIT counts all of these. The event should really have been called MEM_LOAD_RETIRED.L3_HIT_NO_SNOOP.

The event MEM_LOAD_RETIRED.L3_HIT_OTHER_CORE_HIT_HITM could have been named as MEM_LOAD_RETIRED.L3_HIT_SNOOP intuitively for similar reason.

When a modified line is retrieved from another socket it is also written back to memory. This causes remote HITM access to appear as coming from the home dram. The MEM_UNCORE_RETIRED.LOCAL_DRAM and MEM_UNCORE_RETIRED.REMOTE_DRAM evens thus also count the L3 misses satisfied by modified lines in the caches of the remote socket.

There is a difference in the behavior of MEM_LOAD_RETIRED.DTLB_MISSES with respect to that on Intel® Core™2 processors. Previously the event only counted the first miss to the page, as do the imprecise events. The event now counts all loads that result in a miss, thus it includes the secondary misses as well.

B.3.3.2 Load Latency Event

Intel Processors based on the Intel microarchitecture code name Nehalem provide support for "load-latency event", MEM_INST_RETIRED with event code 0BH and Umask value of 10H (LATENCY_ABOVE_THRESHOLD). This event samples loads, recording the number of cycles between the execution of the instruction and actual deliver of the data. If the measured latency is larger than the minimum latency programmed into MSR 0x3f6, bits 15:0, then the counter is incremented.

Counter overflow arms the PEBS mechanism and on the next event satisfying the latency threshold, the PMU writes the measured latency, the virtual or linear address, and the data source into a PEBS record format in the PEBS buffer. Because the virtual address is captured into a known location, the sampling driver could also execute a virtual to physical translation and capture the physical address. The physical address identifies the NUMA home location and in principle allows an analysis of the details of the cache occupancies.

Further, as the address is captured before retirement even the pointer chasing encoding "MOV RAX, [RAX+const]" have their addresses captured. Because the MSR_PEBS_LD_LAT_THRESHOLD MSR is required to specify the latency threshold value, only one minimum latency value can be sampled on a core during a given period. To enable this, the Intel performance tools restrict the programming of this event to counter 4 to simplify the scheduling. Table B-7 lists a few examples of event programming configurations used by the Intel® PTU and Vtune™ Performance Analyzer for the load latency events. Different threshold values for the minimum latencies are specified in MSR_PEBS_LD_LAT_THRESHOLD (address 0x3f6).

Table B-7. Load Latency Event Programming

Load Latency Precise Events	MSR 0x3F6	Umask	Event Code
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_4	4	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_8	8	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_10	16	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_20	32	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_40	64	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_80	128	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_100	256	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_200	512	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_8000	32768	10H	0BH

One of the three fields written to each PEBS record by the PEBS assist mechanism of the load latency event, encodes the data source locality information.

Table B-8. Data Source Encoding for Load Latency PEBS Record

Encoding	Description
0x0	Unknown L3 cache miss.
0x1	Minimal latency core cache hit. This request was satisfied by the L1 data cache.
0x2	Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway. The data is not yet in the data cache, but is located in a fill buffer that will soon be committed to cache.
0x3	This data request was satisfied by the L2.
0x4	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
0x5	L3 HIT (other core hit snoop). Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where no modified copies were found. (Clean).
0x6	L3 HIT (other core HITM). Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where modified copies were found. (HITM).
0x7	Reserved
0x8	L3 MISS (remote cache forwarding). Local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted).
0x9	Reserved.
0xA	L3 MISS (local DRMA go to S). Local home requests that missed the L3 cache and was serviced by local DRAM (go to shared state).
0xB	L3 MISS (remote DRMA go to S). Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to shared state).
0xC	L3 MISS (local DRMA go to E). Local home requests that missed the L3 cache and was serviced by local DRAM (go to exclusive state).

Table B-8. Data Source Encoding for Load Latency PEBS Record (Contd.)

Encoding	Description
0xD	L3 MISS (remote DRMA go to E). Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to exclusive state).
0xE	I/O, Request of input/output operation.
0xF	The request was to un-cacheable memory.

The latency event is the recommended method to measure the penalties for a cycle accounting decomposition. Each time a PMI is raised by this PEBS event a load to use latency and a data source for the cacheline is recorded in the PEBS buffer. The data source for the cacheline can be deduced from the low order 4 bits of the data source field and the table shown above. Thus an average latency for each of the 16 sources can be evaluated from the collected data. As only one minimum latency at a time can be collected it may be awkward to evaluate the latency for an MLC hit and a remote socket dram. A minimum latency of 32 cycles should give a reasonable distribution for all the offcore sources however. The Intel® PTU version 3.2 performance tool can display the latency distribution in the data profiling mode and allows sophisticated event filtering capabilities for this event.

B.3.3.3 Precise Execution Events

PEBS capability in core PMU goes beyond load and store instructions. Branches, near calls and conditional branches can all be counted with precise events, for both retired and mispredicted (and retired) branches of the type selected. For these events, the PEBS buffer will contain the target of the branch. If the Last Branch Record (LBR) is also captured then the location of the branch instruction can also be determined.

When the branch is taken the IP value in the PEBS buffer will also appear as the last target in the LBR. If the branch was not taken (conditional branches only) then it won't and the branch that was not taken and retired is the instruction before the IP in the PEBS buffer.

In the case of near calls retired, this means that Event Based Sampling (EBS) can be used to collect accurate function call counts. As this is the primary measurement for driving the decision to inline a function, this is an important improvement. In order to measure call counts, you must sample on calls. Any other trigger introduces a bias that cannot be guaranteed to be corrected properly.

The precise branch events can be found under event code C4H in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

There is one source of sampling artifact associated with precise events. It is due to the time delay between the PMU counter overflow and the arming of the PEBS hardware. During this period events cannot be detected due to the timing shadow. To illustrate the effect, consider a function call chain where a long duration function, "foo", which calls a chain of 3 very short duration functions, "foo1" calling "foo2" which calls "foo3", followed by a long duration function "foo4". If the durations of foo1, foo2 and foo3 are less than the shadow period the distribution of PEBS sampled calls will be severely distorted. For example:

- If the overflow occurs on the call to foo, the PEBS mechanism is armed by the time the call to foo1 is executed and samples will be taken showing the call to foo1 from foo.
- If the overflow occurs due to the call to foo1, foo2 or foo3 however, the PEBS mechanism will not be armed until execution is in the body of foo4. Thus the calls to foo2, foo3 and foo4 cannot appear as PEBS sampled calls.

Shadowing can effect the distribution of all PEBS events. It will also effect the distribution of basic block execution counts identified by using the combination of a branch retired event (PEBS or not) and the last entry in the LBR. If there were no delay between the PMU counter overflow and the LBR freeze, the last LBR entry could be used to sample taken retired branches and from that the basic block execution counts. All the instructions between the last taken branch and the previous target are executed once.

Such a sampling could be used to generate a "software" instruction retired event with uniform sampling, which in turn can be used to identify basic block execution counts. Unfortunately the shadowing causes the branches at the end of short basic blocks to not be the last entry in the LBR, distorting the measurement. Since all the instructions in a basic block are by definition executed the same number of times.

The shadowing effect on call counts and basic block execution counts can be alleviated to a large degree by averaging over the entries in the LBR. This will be discussed in the section on LBRs.

Typically, branches account for more than 10% of all instructions in a workload, loop optimization needs to focus on those loops with high tripcounts. For counted loops, it is very common for the induction variable to be compared to the tripcount in the termination condition evaluation. This is particularly true if the induction variable is used within the body of the loop, even in the face of heavy optimization. Thus a loop sequence of unrolled operation by eight times may resemble:

```
add     rcx, 8
cmp     rcx, rax
jnge    triad+0x27
```

In this case the two registers, rax and rcx are the tripcount and induction variable. If the PEBS buffer is captured for the conditional branches retired event, the average values of the two registers in the compare can be evaluated. The one with the larger average will be the tripcount. Thus the average, RMS, min and max can be evaluated and even a distribution of the recorded values.

B.3.3.4 Last Branch Record (LBR)

The LBR captures the source and target of each retired taken branch. Processors based on Intel microarchitecture code name Nehalem can track 16 pair of source/target addresses in a rotating buffer. Filtering of the branch instructions by types and privilege levels are permitted using a dedicated facility, MSR_LBR_SELECT. This means that the LBR mechanism can be programmed to capture branches occurring at ring0 or ring3 or both (default) privilege levels. Further the types of taken branches that are recorded can also be filtered. The list of filtering options that can be specified using MSR_LBR_SELECT is described in Chapter 17, “Debug, Branch Profile, TSC, and Quality of Service” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

The default is to capture all branches at all privilege levels (all bits zero). Another reasonable programming would set all bits to 1 except bit 1 (capture ring 3) and bit 3 (capture near calls) and bits 6 and 7. This would leave only ring 3 calls and unconditional jumps in the LBR. Such a programming would result in the LBR having the last 16 taken calls and unconditional jumps retired and their targets in the buffer.

A PMU sampling driver could then capture this restricted “call chain” with any event, thereby providing a “call tree” context. The inclusion of the unconditional jumps will unfortunately cause problems, particularly when there are if-else structures within loops.

In the case of frequent function calls at all levels, the inclusion of returns could be added to clarify the context. However this would reduce the call chain depth that could be captured. A fairly obvious usage would be to trigger the sampling on extremely long latency loads, to enrich the sample with accesses to heavily contended locked variables, and then capture the call chain to identify the context of the lock usage.

Call Counts and Function Arguments

If the LBRs are captured for PMIs triggered by the BR_INST_RETIRED.NEAR_CALL event, then the call count per calling function can be determined by simply using the last entry in LBR. As the PEBS IP will equal the last target IP in the LBR, it is the entry point of the calling function. Similarly, the last source in the LBR buffer was the call site from within the calling function. If the full PEBS record is captured as well, then for functions with limited numbers of arguments on 64-bit OS’s, you can sample both the call counts and the function arguments.

LBRs and Basic Block Execution Counts

Another interesting usage is to use the BR_INST_RETIRED.ALL_BRANCHES event and the LBRs with no filter to evaluate the execution rate of basic blocks. As the LBRs capture all taken branches, all the basic blocks between a branch IP (source) and the previous target in the LBR buffer were executed one time. Thus a simple way to evaluate the basic block execution counts for a given load module is to make a map of the starting locations of every basic block. Then for each sample triggered by the PEBS collection of BR_INST_RETIRED.ALL_BRANCHES, starting from the PEBS address (a target but perhaps for a not taken branch and thus not necessarily in the LBR buffer) and walking backwards through the LBRs until

finding an address not corresponding to the load module of interest, count all the basic blocks that were executed. Calling this value "number_of_basic_blocks", increment the execution counts for all of those blocks by 1/(number_of_basic_blocks). This technique also yields the taken and not taken rates for the active branches. All branch instructions between a source IP and the previous target IP (within the same module) were not taken, while the branches listed in the LBR were taken. This is illustrated in the graphics below.

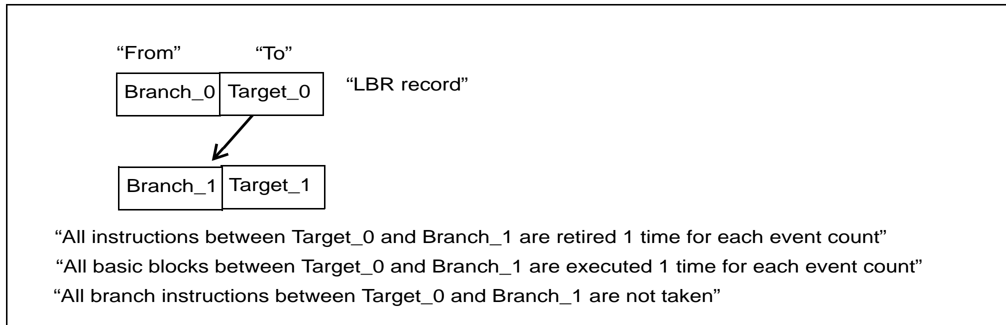


Figure B-6. LBR Records and Basic Blocks

The 16 sets LBR records can help rectify the artifact of PEBS samples aggregating disproportionately to certain instructions in the sampling process. The situation of skewed distribution of PEBS sample is illustrated below in Figure B-7.

Consider a number of basic blocks in the flow of normal execution, some basic block takes 20 cycles to execute, others taking 2 cycles, and shadowing takes 10 cycles. Each time an overflow condition occurs, the delay of PEBS being armed is at least 10 cycles. Once the PEBS is armed, PEBS record is captured on the next eventing condition. The skewed distribution of sampled instruction address using PEBS record will be skewed as shown in the middle of Figure B-7. In this conceptual example, we assume every branch in taken in these basic blocks.

In the skewed distribution of PEBS samples, the branch IP of the last basic block will be recorded 5 times as much as the least sampled branch IP address (the 2nd basic block).

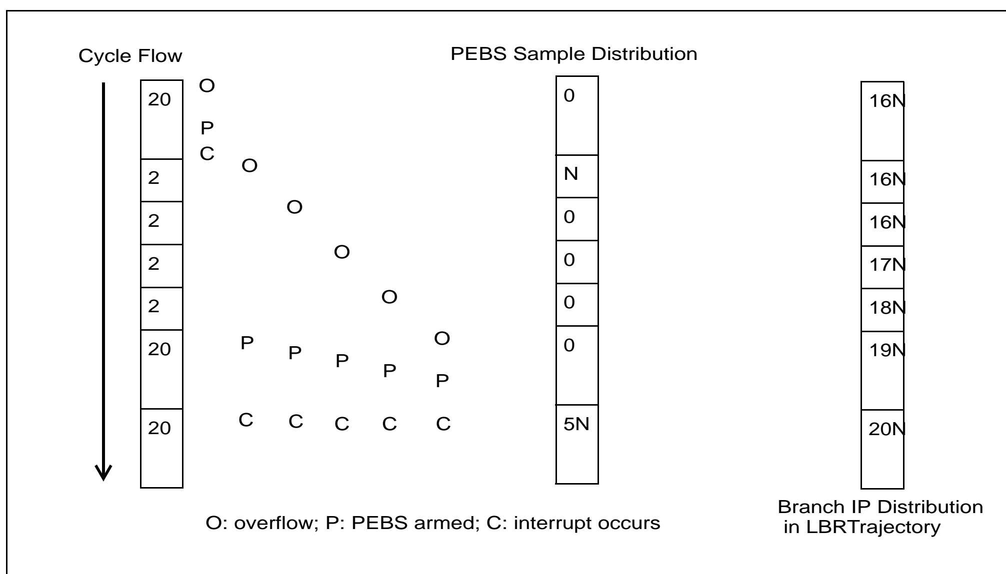


Figure B-7. Using LBR Records to Rectify Skewed Sample Distribution

This situation where some basic blocks would appear to never get samples and some have many times too many. Weighting each entry by 1/(num of basic blocks in the LBR trajectory), in this example would result in dividing the numbers in the right most table by 16. Thus we end up with far more accurate execution counts ((1.25-> 1.0) * N) in all of the basic blocks, even those that never directly caused a PEBS sample.

As on Intel® Core™2 processors there is a precise instructions retired event that can be used in a wide variety of ways. In addition there are precise events for uops_retired, various SSE instruction classes, FP assists. It should be noted that the FP assist events only detect x87 FP assists, not those involving SSE FP instructions. Detecting all assists will be discussed in the section on the pipeline Front End.

The instructions retired event has a few special uses. While its distribution is not uniform, the totals are correct. If the values recorded for all the instructions in a basic block are averaged, a measure of the basic block execution count can be extracted. The ratios of basic block executions can be used to estimate loop tripcounts when the counted loop technique discussed above cannot be applied.

The PEBS version (general counter) instructions retired event can further be used to profile OS execution accurately even in the face of STI/CLI semantics, because the PEBS interrupt then occurs after the critical section has completed, but the data was frozen correctly. If the cmask value is set to some very high value and the invert condition is applied, the result is always true, and the event will count core cycles (halted + unhalted).

Consequently both cycles and instructions retired can be accurately profiled. The UOPS_RETIRED.ANY event, which is also precise can also be used to profile Ring 0 execution and really gives a more accurate display of execution. The precise events available for this purpose are listed under event code C0H, C2H, C7H, F7H in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

B.3.3.5 Measuring Core Memory Access Latency

Drilling down performance issues associated with locality or cache coherence issues will require using performance monitoring events. In each processor core, there is a super queue that allocates entries to buffer requests of memory access traffic due to an L2 miss to the uncore sub-system. Table B-9 lists various performance events available in the core PMU that can drill down performance issues related to L2 misses.

Table B-9. Core PMU Events to Drill Down L2 Misses

Core PMU Events	Umask	Event Code
OFFCORE_REQUESTS.DEMAND.READ_DATA[1]	01H	B0H
OFFCORE_REQUESTS.DEMAND.READ_CODE[1]	02H	B0H
OFFCORE_REQUESTS.DEMAND.RFO[1]	04H	B0H
OFFCORE_REQUESTS.ANY.READ	08H	B0H
OFFCORE_REQUESTS.ANY.RFO	10H	B0H
OFFCORE_REQUESTS.UNCACHED_MEM	20H	B0H
OFFCORE_REQUESTS.L1D.WRITEBACK	40H	B0H
OFFCORE_REQUESTS.ANY	80H	B0H

NOTES:

1. The *DEMAND* events also include any requests made by the L1D cache hardware prefetchers.

Table B-10 lists various performance events available in the core PMU that can drill down performance issues related to super queue operation.

Table B-10. Core PMU Events for Super Queue Operation

Core PMU Events	Umask	Event Code
OFFCORE_REQUESTS_BUFFER_FULL	01H	B2H

Additionally, L2 misses can be drilled down further by data origin attributes and response attributes. The matrix to specify data origin and response type attributes is done by a dedicated MSR OFFCORE_RSP_0 at address 1A6H. See Table B-11 and Table B-12.

Table B-11. Core PMU Event to Drill Down OFFCore Responses

Core PMU Events	OFFCORE_RSP_0 MSR	Umask	Event Code
OFFCORE_RESPONSE	See Table B-12	01H	B7H

Table B-12. OFFCORE_RSP_0 MSR Programming

	Position	Description	Note
Request type	0	**Demand Data Rd = DCU reads (includes partials, DCU Prefetch)**	
	1	**Demand RFO = DCU RFOs**	
	2	**Demand Ifetch = IFU Fetches**	
	3	**Writeback = L2_EVICT/DCUWB**	
	4	**PF Data Rd = L2 Prefetcher Reads**	
	5	**PF RFO= L2 Prefetcher RFO**	
	6	**PF Ifetch= L2 Prefetcher Instruction fetches**	
	7	**Other**	Include non-temporal stores
	8	**L3_HIT_UNCORE_HIT**	exclusive line
	9	**L3_HIT_OTHER_CORE_HIT_SNP**	clean line
	10	**L3_HIT_OTHER_CORE_HITM**	modified line
	11	**L3_MISS_REMOTE_HIT_SCRUB**	Used by multiple cores
	12	**L3_MISS_REMOTE_FWD**	Clean line used by one core
	13	**L3_MISS_REMOTE_DRAM**	
	14	**L3_MISS_LOCAL_DRAM**	
	15	**Non-DRAM**	Non-DRAM requests

Although Table B-12 allows 2^16 combinations of setting in MSR_OFFCORE_RSP_0 in theory, it is more useful to consider combining the subsets of 8-bit values to specify "Request type" and "Response type". The more common 8-bit mask values are listed in Table B-13.

Table B-13. Common Request and Response Types for OFFCORE_RSP_0 MSR

Request Type	Mask	Response Type	Mask
ANY_DATA	xx11H	**ANY_CACHE_DRAM**	7FxxH
ANY_IFETCH	xx44H	**ANY_DRAM**	60xxH
ANY_REQUEST	xxFFH	**ANY_L3_MISS**	F8xxH
ANY_RFO	xx22H	**ANY_LOCATION**	FFxxH
CORE_WB	xx08H	**IO**	80xxH
DATA_IFETCH	xx77H	**L3_HIT_NO_OTHER_CORE**	01xxH
DATA_IN	xx33H	**L3_OTHER_CORE_HIT**	02xxH
DEMAND_DATA	xx03H	**L3_OTHER_CORE_HITM**	04xxH
DEMAND_DATA_RD	xx01H	**LOCAL_CACHE**	07xxH
DEMAND_IFETCH	xx04H	**LOCAL_CACHE_DRAM**	47xxH
DEMAND_RFO	xx02H	**LOCAL_DRAM**	40xxH
OTHER[1]	xx80H	**REMOTE_CACHE**	18xxH
PF_DATA	xx30H	**REMOTE_CACHE_DRAM**	38xxH
PF_DATA_RD	xx10H	**REMOTE_CACHE_HIT**	10xxH
PF_IFETCH	xx40H	**REMOTE_CACHE_HITM**	08xxH
PF_RFO	xx20H	**REMOTE-DRAM**	20xxH
PREFETCH	xx70H		

NOTES:

1. The PMU may report incorrect counts with setting MSR_OFFCORE_RSP_0 to the value of 4080H. Non-temporal stores to the local DRAM is not reported in the count.

B.3.3.6 Measuring Per-Core Bandwidth

Measuring the bandwidth of all memory traffic for an individual core is complicated, the core PMU and uncore PMU do provide capability to measure the important components of per-core bandwidth.

At the microarchitectural level, there is the buffering of L3 for writebacks/evictions from L2 (similarly to some degree with the non temporal writes). The eviction of modified lines from the L2 causes a write of the line back to the L3. The line in L3 is only written to memory when it is evicted from the L3 some time later (if at all). And L3 is part of the uncore sub-system, not part of the core.

The writebacks to memory due to eviction of modified lines from L3 cannot be associated with an individual core in the uncore PMU logic. The net result of this is that the total write bandwidth for all the cores can be measured with events in the uncore PMU. The read bandwidth and the non-temporal write bandwidth can be measured on a per core basis. In a system populated with two physical processor, the NUMA nature of memory bandwidth implies the measurement for those 2 components has to be divided into bandwidths for the core on a per-socket basis.

The per-socket read bandwidth can be measured with the events:

OFFCORE_RESPONSE_0.DATA_IFETCH.L3_MISS_LOCAL_DRAM.

OFFCORE_RESPONSE_0.DATA_IFETCH.L3_MISS_REMOTE_DRAM.

The total read bandwidth for all sockets can be measured with the event:

OFFCORE_RESPONSE_0.DATA_IFETCH.ANY_DRAM.

The per-socket non-temporal store bandwidth can be measured with the events:

OFFCORE_RESPONSE_0.OTHER.L3_MISS_LOCAL_CACHE_DRAM.

OFFCORE_RESPONSE_0.OTHER.L3_MISS_REMOTE_DRAM.

The total non-temporal store bandwidth can be measured with the event:

OFFCORE_RESPONSE_0.OTHER.ANY.CACHE_DRAM.

The use of "CACHE_DRAM" encoding is to work around the defect in the footnote of Table B-13. Note that none of the above includes the bandwidth associated with writebacks of modified cacheable lines.

B.3.3.7 Miscellaneous L1 and L2 Events for Cache Misses

In addition to the OFFCORE_RESPONSE_0 event and the precise events that will be discussed later, there are several other events that can be used as well. There are additional events that can be used to supplement the offcore_response_0 events, because the offcore_response_0 event code is supported on counter 0 only.

L2 misses can also be counted with the architecturally defined event LONGEST_LAT_CACHE_ACCESS, however as this event also includes requests due to the L1D and L2 hardware prefetchers, its utility may be limited. Some of the L2 access events can be used for both drilling down L2 accesses and L2 misses by type, in addition to the OFFCORE_REQUESTS events discussed earlier. The L2_RQSTS and L2_DATA_RQSTS events can be used to discern assorted access types. In all of the L2 access events the designation PREFETCH only refers to the L2 hardware prefetch. The designation DEMAND includes loads and requests due to the L1D hardware prefetchers.

The L2_LINES_IN and L2_LINES_OUT events have been arranged slightly differently than the equivalent events on Intel® Core™2 processors. The L2_LINES_OUT event can now be used to decompose the evicted lines by clean and dirty (i.e. a Writeback) and whether they were evicted by an L1D request or an L2 HW prefetch.

The event L2_TRANSACTIONS counts all interactions with the L2.

Writes and locked writes are counted with a combined event, L2_WRITE.

The details of the numerous derivatives of L2_RQSTS, L2_DATA_RQSTS, L2_LINES_IN, L2_LINES_OUT, L2_TRANSACTIONS, L2_WRITE, can be found under event codes24H, 26H, F1H, F2H, F0H, and 27H in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

B.3.3.8 TLB Misses

The next largest set of memory access delays are associated with the TLBs when linear-to-physical address translation is mapped with a finite number of entries in the TLBs. A miss in the first level TLBs results in a very small penalty that can usually be hidden by the OOO execution and compiler's scheduling. A miss in the shared TLB results in the Page Walker being invoked and this penalty can be noticeable in the execution.

The (non-PEBS) TLB miss events break down into three sets:

- DTLB misses and its derivatives are programmed with event code 49H.
- Load DTLB misses and its derivatives are programmed with event code 08H.
- ITLB misses and its derivatives are programmed with event code 85H.

Store DTLB misses can be evaluated from the difference of the DTLB misses and the Load DTLB misses. Each then has a set of sub events programmed with the umask value. The Umask details of the numerous derivatives of the above events are listed in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

B.3.3.9 L1 Data Cache

There are PMU events that can be used to analyze L1 data cache operations. These events can only be counted with the first 2 of the 4 general counters, i.e. IA32_PMC0 and IA32_PMC1. Most of the L1D events are self explanatory.

The total number of references to the L1D can be counted with L1D_ALL_REF, either just cacheable references or all. The cacheable references can be divided into loads and stores with L1D_CACHE_LOAD and L1D_CACHE.STORE. These events are further subdivided by MESI states through their Umask values, with the I state references indicating the cache misses.

The evictions of modified lines in the L1D result in writebacks to the L2. These are counted with the L1D_WB_L2 events. The umask values break these down by the MESI state of the version of the line in the L2.

The locked references can be counted also with the L1D_CACHE_LOCK events. Again these are broken down by MES states for the lines in L1D.

The total number of lines brought into L1D, the number that arrived in an M state and the number of modified lines that get evicted due to receiving a snoop are counted with the L1D event and its Umask variations.

The L1D events are listed under event codes28H, 40H, 41H, 42H, 43H, 48H, 4EH, 51H, 52H, 53H, 80H, and 83H in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

There are few cases of loads not being able to forward from active store buffers. The predominant situations have to do with larger loads overlapping smaller stores. There is not event that detects when this occurs. There is also a "false store forwarding" case where the addresses only match in the lower 12 address bits. This is sometimes referred to as 4K aliasing. This can be detected with the event "PARTIAL_ADDRESS_ALIAS" which has event code 07H and Umask 01H.

B.3.4 Front End Monitoring Events

Branch misprediction effects can sometimes be reduced through code changes and enhanced inlining. Most other front end performance limitations have to be dealt with by the code generation. The analysis of such issues is mostly of use by compiler developers.

B.3.4.1 Branch Mispredictions

In addition to branch retired events that was discussed in conjunction with PEBS in Section B.3.3.3. These are enhanced by use of the LBR to identify the branch location to go along with the target location captured in the PEBS buffer. Aside from those usage, many other PMU events (event code E6, E5, E0, 68, 69) associated with branch predictions are more relevant to hardware design than performance tuning.

Branch mispredictions are not in and of themselves an indication of a performance bottleneck. They have to be associated with dispatch stalls and the instruction starvation condition, UOPS_ISSUED:C1:I1 – RESOURCE_STALLS.ANY. Such stalls are likely to be associated with icache misses and ITLB misses. The precise ITLB miss event can be useful for such issues. The icache and ITLB miss events are listed under event code 80H, 81H, 82H, 85H, AEH.

B.3.4.2 Front End Code Generation Metrics

The remaining front end events are mostly of use in identifying when details of the code generation interact poorly with the instructions decoding and uop issue to the OOO engine. Examples are length

changing prefix issues associated with the use of 16 bit immediates, rob read port stalls, instruction alignment interfering with the loop detection and instruction decoding bandwidth limitations. The activity of the LSD is monitored using CMASK values on a signal monitoring activity. Some of these events are listed under event code 17H, 18H, 1EH, 1FH, 87H, A6H, A8H, D0H, D2H in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Some instructions (FSIN, FCOS, and other transcendental instructions) are decoded with the assistance of MS-ROM. Frequent occurrences of instructions that required assistance of MS-ROM to decode complex uop flows are opportunity to improve instruction selection to reduce such occurrences. The UOPS_DECODED.MS event can be used to identify code regions that could benefit from better instruction selection.

Other situations that can trigger this event are due to FP assists, like performing a numeric operation on denormalized FP values or QNaNs. In such cases the penalty is essentially the uops required for the assist plus the pipeline clearing required to ensure the correct state.

Consequently this situation has a very clear signature consisting of MACHINE_CLEAR.CYCLES and uops being inserted by the microcode sequencer, UOPS_DECODED.MS. The execution penalty being the sum of these two contributions. The event codes for these are listed under D1H and C3H.

B.3.5 Uncore Performance Monitoring Events

The uncore sub-system includes the L3, IMC and Intel QPI units in the diagram shown in Figure B-4. Within the uncore sub-system, the uncore PMU consists of eight general-purpose counters and one fixed counter. The fixed counter in uncore monitors the unhalted clock cycles in the uncore clock domain, which runs at a different frequency than the core.

The uncore cannot by itself generate a PMI interrupt. While the core PMU can raise PMI at a per-logical-processor specificity, the uncore PMU can cause PMI at a per-core specificity using the interrupt hardware in the processor core. When an uncore counter overflows, a bit pattern is used to specify which cores should be signaled to raise a PMI. The uncore PMU is unaware of the core, Processor ID or Thread ID that caused the event that overflowed a counter. Consequently the most reasonable approach for sampling on uncore events is to raise a PMI on all the logical processors in the package.

There are a wide variety of events that monitor queue occupancies and inserts. There are others that count cacheline transfers, dram paging policy statistics, snoop types and responses, and so on. The uncore is the only place the total bandwidth to memory can be measured. This will be discussed explicitly after all the uncore components and their events are described.

B.3.5.1 Global Queue Occupancy

Each processor core has a super queue that buffers requests of memory access traffic due to an L2 miss. The uncore has a global queue (GQ) to service transaction requests from the processor cores and buffers data traffic that arrive from L3, IMC, or Intel QPI links.

Within the GQ, there are 3 "trackers" in the GQ for three types of transactions:

- On-package read requests, its tracker queue has 32 entries.
- On-package writeback requests, its tracker queue has 16 entries.
- Requests that arrive from a "peer", its tracker queue has 12 entries.

A "peer" refers to any requests coming from the Intel® QuickPath Interconnect.

The occupancies, inserts, cycles full and cycles not empty for all three trackers can be monitored. Further as load requests go through a series of stages the occupancy and inserts associated with the stages can also be monitored, enabling a "cycle accounting" breakdown of the uncore memory accesses due to loads.

When a uncore counter is first programmed to monitor a queue occupancy, for any of the uncore queues, the queue must first be emptied. This is accomplished by the driver of the monitoring software tool issuing a bus lock. This only needs to be done when the counter is first programmed. From that point on

the counter will correctly reflect the state of the queue, so it can be repeatedly sampled for example without another bus lock being issued.

The uncore events that monitor GQ allocation (UNC_GQ_ALLOC) and GQ tracker occupancy (UNC_GQ_TRACKER_OCCUP) are listed under the event code 03H and 02H in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. The selection between the three trackers is specified from the Umask value. The mnemonic of these derivative events use the notation: "RT" signifying the read tracker, "WT", the write tracker and "PPT" the peer probe tracker.

Latency can measured by the average duration of the queue occupancy, if the occupancy stops as soon as the data has been delivered. Thus the ratio of UNC_GQ_TRACKER_OCCUP.X/UNC_GQ_ALLOC.X measures an average duration of queue occupancy, where 'X' represents a specific Umask value. The total occupancy period of the read tracker as measured by:

Total Read Period = UNC_GQ_TRACKER_OCCUP.RT/UNC_GQ_ALLOC.RT

Is longer than the data delivery latency due to it including time for extra bookkeeping and cleanup. The measurement:

LLC response Latency = UNC_GQ_TRACKER_OCCUP.RT_TO_LLC_RESP / UNC_GQ_ALLOC.RT_TO_LLC_RESP

is essentially a constant. It does not include the total time to snoop and retrieve a modified line from another core for example, just the time to scan the L3 and see if the line is or is not present in this socket.

An overall latency for an L3 hit is the weighted average of three terms:

- The latency of a simple hit, where the line has only been used by the core making the request.
- The latencies for accessing clean lines by multiple cores.
- The latencies for accessing dirty lines that have been accessed by multiple cores.

These three components of the L3 hit for loads can be decomposed using the derivative events of OFFCORE_RESPONSE:

- OFFCORE_RESPONSE_0.DEMAND_DATA.L3_HIT_NO_OTHER_CORE.
- OFFCORE_RESPONSE_0.DEMAND_DATA.L3_HIT_OTHER_CORE_HIT.
- OFFCORE_RESPONSE_0.DEMAND_DATA.L3_HIT_OTHER_CORE_HITM.

The event OFFCORE_RESPONSE_0.DEMAND_DATA.LOCAL_CACHE should be used as the denominator to obtain latencies. The individual latencies could have to be measured with microbenchmarks, but the use of the precise latency event will be far more effective as any bandwidth loading effects will be included.

The L3 miss component is the weighted average over three terms:

- The latencies of L3 hits in a cache on another socket (this is described in the previous paragraph).
- The latencies to local DRAM.
- The latencies to remote DRAM.

The local dram access and the remote socket access can be decomposed with more uncore events.

Miss to fill latency = UNC_GQ_TRACKER_OCCUP.RT_LLC_MISS / UNC_GQ_ALLOC.RT_LLC_MISS

The uncore GQ events using Umask value associated with *RTID* mnemonic allow the monitoring of a sub component of the Miss to fill latency associated with the communications between the GQ and the QHL.

There are uncore PMU events which monitor cycles when the three trackers are not empty (>= 1 entry) or full. These events are listed under the event code 00H and 01H in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Because the uncore PMU generally does not differentiate which processor core causes a particular eventing condition, the technique of dividing the latencies by the average queue occupancy in order to determine a penalty does not work for the uncore. Overlapping entries from different cores do not result in overlapping penalties and thus a reduction in stalled cycles. Each core suffers the full latency independently.

To evaluate the correction on a per core basis one needs the number of cycles there is an entry from the core in question. A *NOT_EMPTY_CORE_N type event would needed. There is no such event. Consequently, in the cycle decomposition one must use the full latency for the estimate of the penalty. As has been stated before it is best to use the PEBS latency event as the data sources are also collected with the latency for the individual sample.

The individual components of the read tracker, discussed above, can also be monitored as busy or full by setting the cmask value to 1 or 32 and applying it to the assorted read tracker occupancy events.

Table B-14. Uncore PMU Events for Occupancy Cycles

Uncore PMU Events	Cmask	Umask	Event Code
UNC_GQ_TRACKER_OCCUP.RT_L3_MISS_FULL	32	02H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_L3_RESP_FULL	32	04H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACCQUIRED_FULL	32	08H	02H
UNC_GQ_TRACKER_OCCUP.RT_L3_MISS_BUSY	1	02H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_L3_RESP_BUSY	1	04H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACCQUIRED_BUSY	1	08H	02H

B.3.5.2 Global Queue Port Events

The GQ data buffer traffic controls the flow of data to and from different sub-systems via separate ports:

- Core traffic: two ports handles data traffic, each port dedicated to a pair of processor cores.
- L3 traffic: one port service L3 data traffic.
- Intel QPI traffic: one service traffic to QPI logic.
- IMC traffic: one service data traffic to integrated memory controller.

The ports for L3 and core traffic transfer a fixed number of bits per cycle. However the Intel® QuickPath Interconnect protocols can result in either 8 or 16 bytes being transferred on the read Intel QPI and IMC ports. Consequently these events cannot be used to measure total data transfers and bandwidths.

The uncore PMU events that can distinguish traffic flow are listed under the event code 04H and 05H in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

B.3.5.3 Global Queue Snoop Events

Cacheline requests from the cores or from a remote package or the I/O Hub are handled by the GQ. When the uncore receives a cacheline request from one of the cores, the GQ first checks the L3 to see if the line is on the package. Because the L3 is inclusive, this answer can be quickly ascertained. If the line is in the L3 and was owned by the requesting core, data can be returned to the core from the L3 directly. If the line is being used by multiple cores, the GQ will snoop the other cores to see if there is a modified copy. If so the L3 is updated and the line is sent to the requesting core.

In the event of an L3 miss, the GQ must send out requests to the local memory controller (or over the Intel QPI links) for the line. A request through the Intel QPI to a remote L3 (or remote DRAM) must be made if data exists in a remote L3 or does not exist in local DRAM. As each physical package has its own local integrated memory controller the GQ must identify the "home" location of the requested cacheline from the physical address. If the address identifies home as being on the local package then the GQ makes a simultaneous request to the local memory controller. If home is identified as belonging to the remote package, the request sent over the Intel QPI will also access the remote IMC.

The GQ handles the snoop responses for the cacheline requests that come in from the Intel® QuickPath Interconnect. These snoop traffic correspond to the queue entries in the peer probe tracker.

The snoop responses are divided into requests for locally homed data and remotely homed data. If the line is in a modified state and the GQ is responding to a read request, the line also must be written back to memory. This would be a wasted effort for a response to a RFO as the line will just be modified again, so no Writeback is done for RFOs.

The snoop responses of local home events that can be monitored by an uncore PMU are listed under event code 06H in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. The snoop responses of remotely home events are listed under event code 07H.

Some related events count the MESI transitions in response to snoops from other caching agents (processors or IOH). Some of these rely on programming MSR so they can only be measured one at a time, as there is only one MSR. The Intel performance tools will schedule this correctly by restricting these events to a single general uncore counter.

B.3.5.4 L3 Events

Although the number of L3 hits and misses can be determined from the GQ tracker allocation events, Several uncore PMU event is simpler to use. They are listed under event code 08H and 09H in the uncore event list of Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

The MESI states breakdown of lines allocated and victimized can also be monitored with LINES_IN, LINES_OUT events in the uncore using event code 0AH and 0BH. Details are listed in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

B.3.6 Intel QuickPath Interconnect Home Logic (QHL)

When a data misses L3 and causing the GQ of the uncore to send out a transaction request, the Intel QPI fabric will fulfill the request either from the local DRAM controller or from a remote DRAM controller in another physical package. The GQ must identify the "home" location of the requested cacheline from the physical address. If the address identifies home as being on the local package then the GQ makes a simultaneous request to the local memory controller, the Integrated memory controller (IMC). If home is identified as belonging to the remote package, the request is sent to the Intel QPI first and then to access the remote IMC.

The Intel QPI logic and IMC are distinct units in the uncore sub-system. The Intel QPI logic distinguish the local IMC relative to remote IMC using the concept of "caching agent" and "home agent". Specifically, the Intel QPI protocol considers each socket as having a "caching agent": and a "home agent":

- Caching Agent is the GQ and L3 in the uncore (or an IOH if present).
- Home Agent is the IMC.

An L3 miss result in simultaneous queries for the line from all the Caching Agents and the Home agent (wherever it is).

QHL requests can be superseded when another source can supply the required line more quickly. L3 misses to locally homed lines, due to on package requests, are simultaneously directed to the QHL and Intel QPI. If a remote caching agent supplies the line first then the request to the QHL is sent a signal that the transaction is complete. If the remote caching agent returns a modified line in response to a read request then the data in dram must be updated with a writeback of the new version of the line.

There is a similar flow of control signals when the Intel QPI simultaneously sends a snoop request for a locally homed line to both the GQ and the QHL. If the L3 has the line, the QHL must be signaled that the transaction was completely by the L3/GQ. If the line in L3 (or the cores) was modified and the snoop request from the remote package was for a load, then a writeback must be completed by the QHL and the QHL forwards the line to the Intel QPI to complete the transaction.

Uncore PMU provides events for monitoring these cacheline access and writeback traffic in the uncore by using the QHL opcode matching capability. The opcode matching facility is described in Chapter 33, "Handling Boundary Conditions in a Virtual Machine Monitor" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*. The uncore PMU event that uses the opcode matching capability is listed under event code 35H. Several of the more useful setting to program QHL opcode matching is shown in Table B-15.

Table B-15. Common QHL Opcode Matching Facility Programming

Load Latency Precise Events	MSR 0x396	Umask	Event Code
UNC_ADDR_OPCODE_MATCH.IOH.NONE	0	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPFWDI	40001900_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPFWDS	40001A00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPIWB	40001D00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.NONE	0	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPFWDI	40001900_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPFWDS	40001A00_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPIWB	40001D00_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.NONE	0	4H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPFWDI	40001900_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPFWDS	40001A00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPIWB	40001D00_00000000	1H	35H

These predefined opcode match encodings can be used to monitor HITM accesses. It is the only event that allows profiling the code requesting HITM transfers.

The diagrams Figure B-8 through Figure B-15 show a series of Intel QPI protocol exchanges associated with Data Reads and Reads for Ownership (RFO), after an L3 miss, under a variety of combinations of the local home of the cacheline, and the MESI state in the remote cache. Of particular note are the cases where the data comes from the remote QHL even when the data was in the remote L3. These are the Read Data with the remote L3 having the line in an M state.

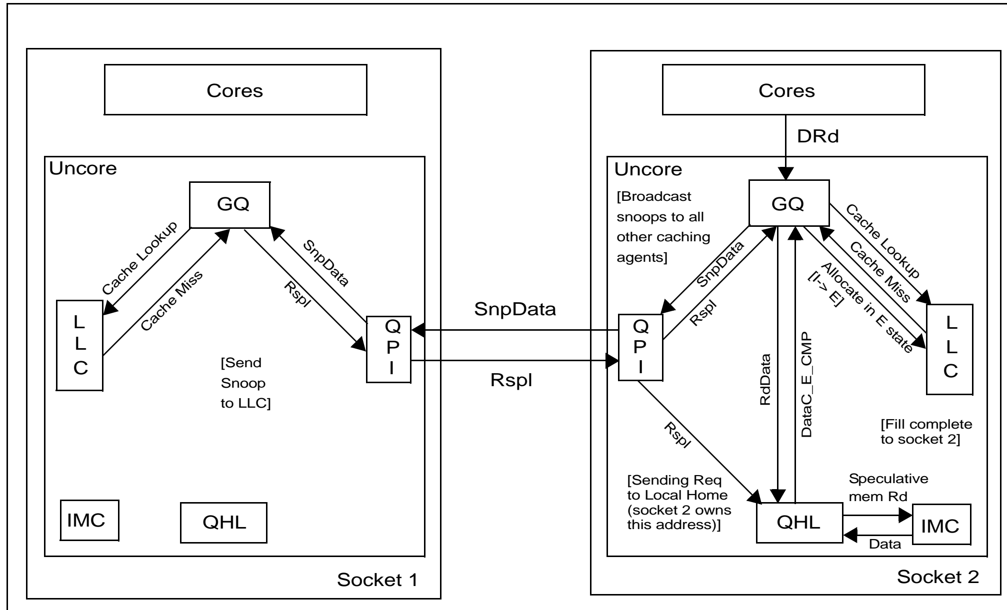


Figure B-8. RdData Request after LLC Miss to Local Home (Clean Rsp)

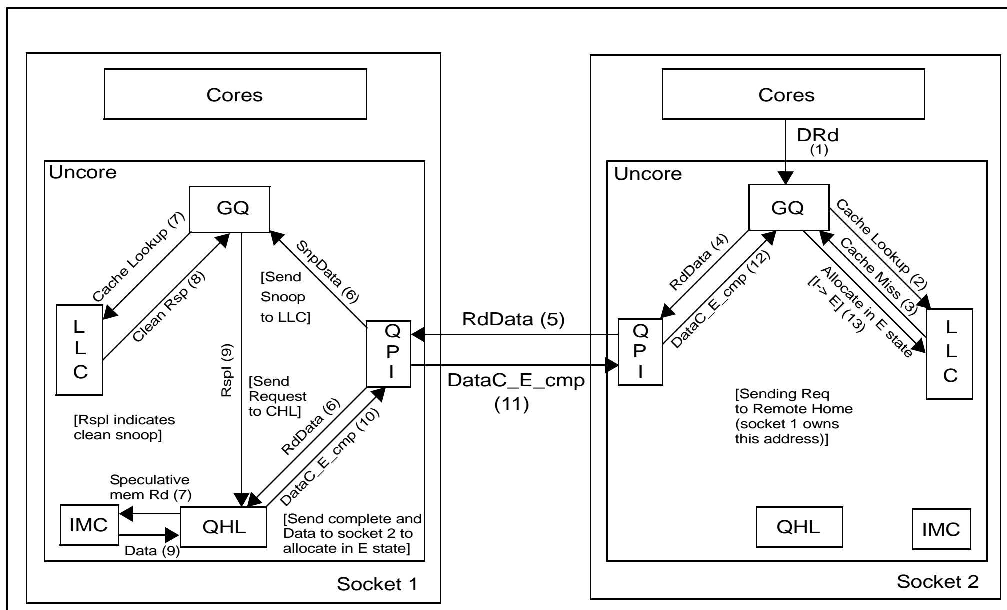


Figure B-9. RdData Request after LLC Miss to Remote Home (Clean Rsp)

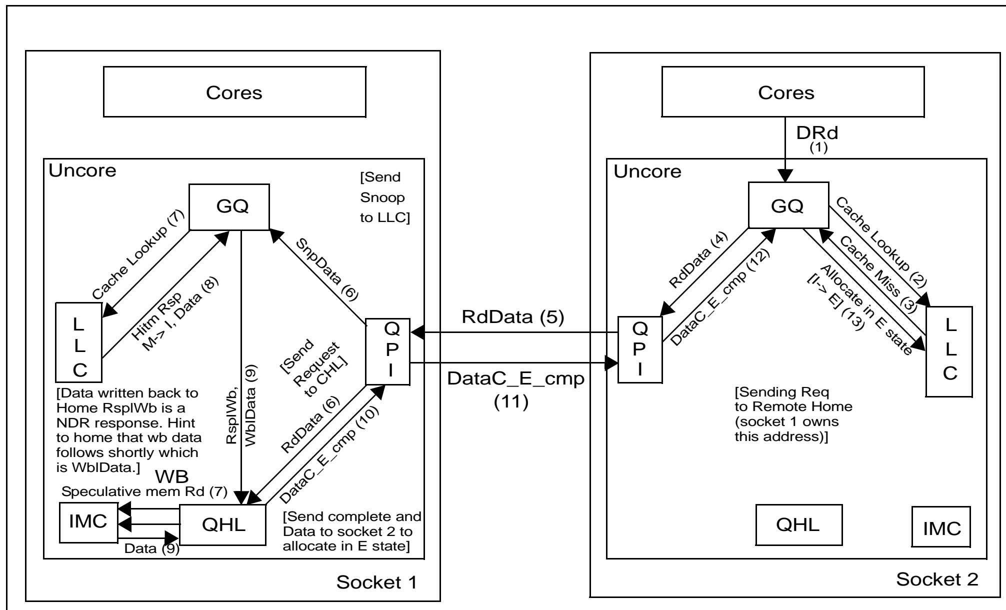


Figure B-10. RdData Request after LLC Miss to Remote Home (Hitm Response)

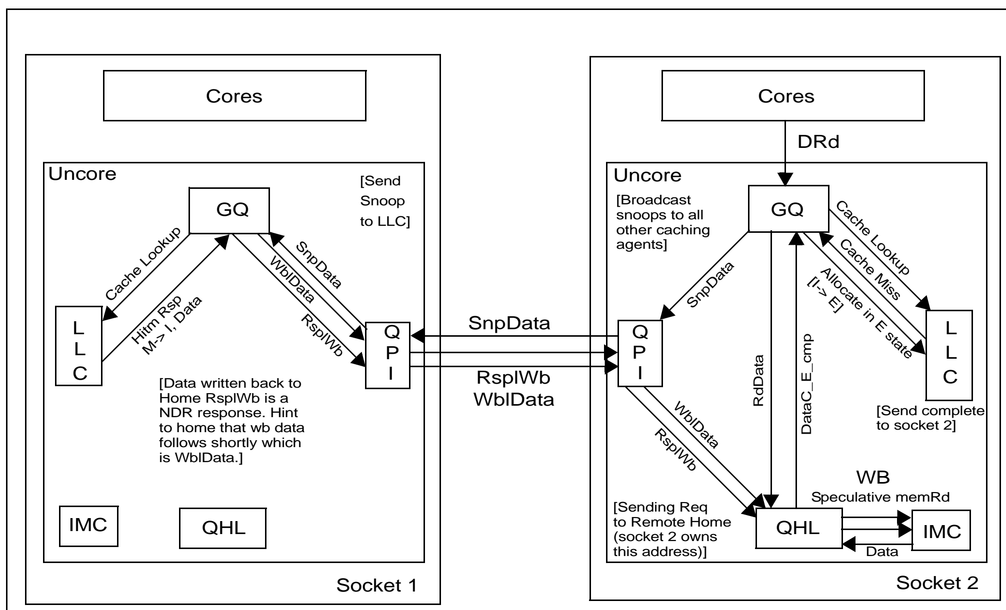


Figure B-11. RdData Request after LLC Miss to Local Home (Hitm Response)

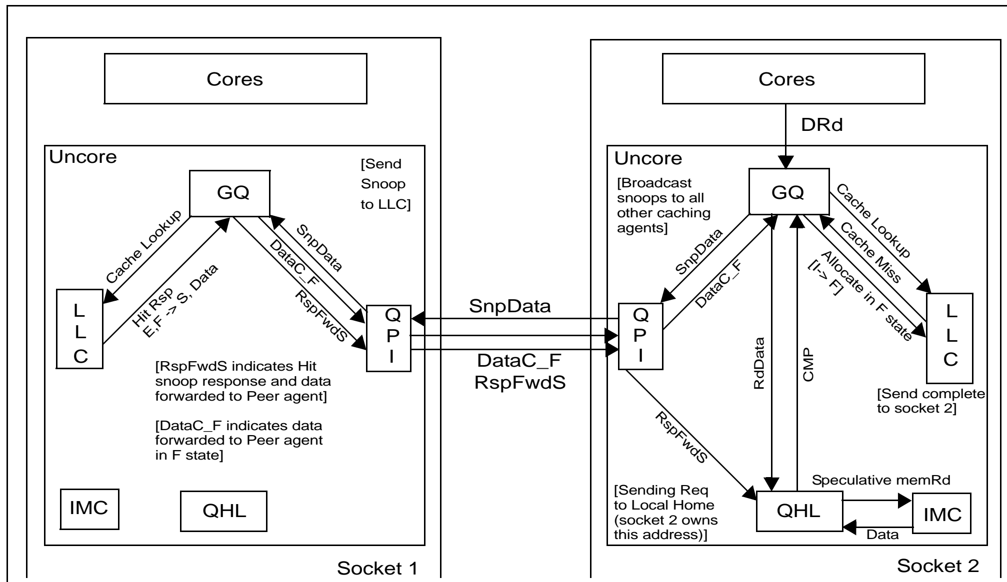


Figure B-12. RdData Request after LLC Miss to Local Home (Hit Response)

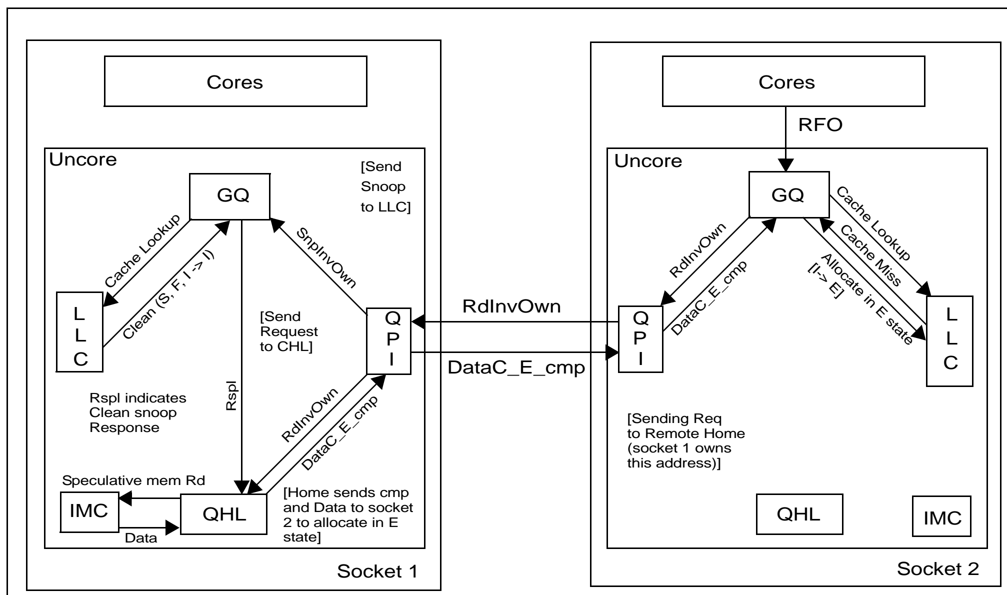


Figure B-13. RdInvOwn Request after LLC Miss to Remote Home (Clean Res)

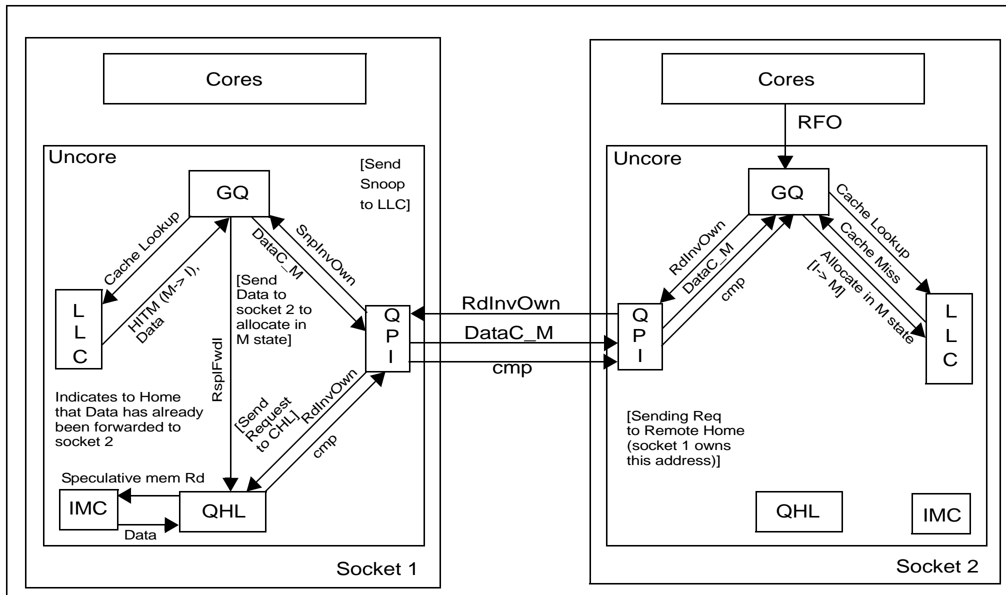


Figure B-14. RdInvOwn Request after LLC Miss to Remote Home (Hitm Res)

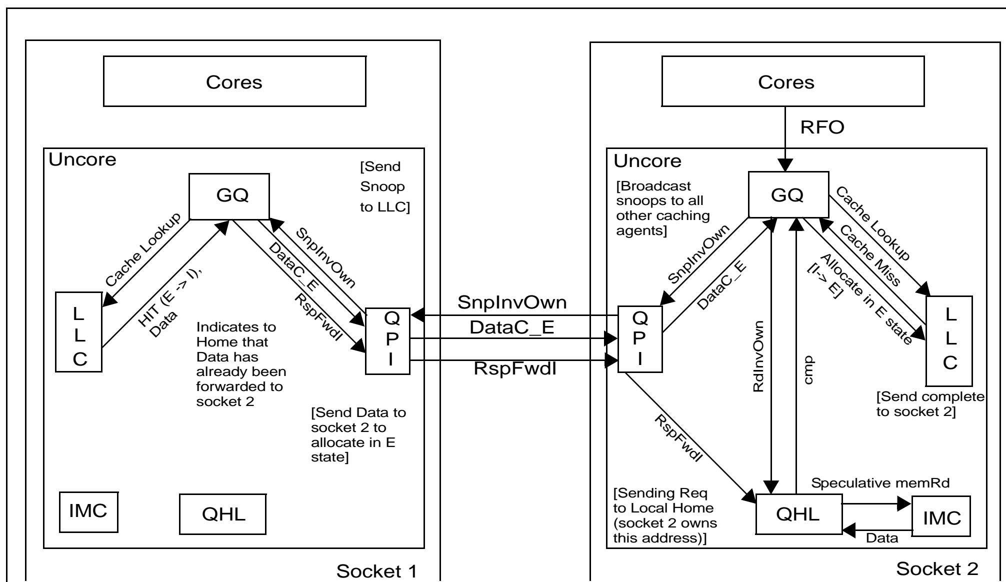


Figure B-15. RdInvOwn Request after LLC Miss to Local Home (Hit Res)

Whether the line is locally or remotely "homed" it has to be written back to dram before the originating GQ receives the line, so it always appears to come from a QHL. The RFO does not do this. However, when responding to a remote RFO (SnpInvOwn) and the line is in an S or F state, the cacheline gets invalidated and the line is sent from the QHL. The point is that the data source might not always be so obvious.

B.3.7 Measuring Bandwidth From the Uncore

Read bandwidth can be measured on a per core basis using events like OFFCORE_RESPONSE_0.DATA_IN.LOCAL_DRAM and OFFCORE_RESPONSE_0.DATA_IN.REMOTE_DRAM. The total bandwidth includes writes and these cannot

be monitored from the core as they are mostly caused by evictions of modified lines in the L3. Thus a line used and modified by one core can end up being written back to dram when it is evicted due to a read on another core doing some completely unrelated task. Modified cached lines and writebacks of uncached lines (e.g. written with non temporal streaming stores) are handled differently in the uncore and their writebacks increment various events in different ways.

All full lines written to DRAM are counted by the UNC_IMC_WRITES.FULL.* events. This includes the writebacks of modified cached lines and the writes of uncached lines, for example generated by non-temporal SSE stores. The uncached line writebacks from a remote socket will be counted by UNC_QHL_REQUESTS.REMOTE_WRITES. The uncached writebacks from the local cores are not counted by UNC_QHL_REQUESTS.LOCAL_WRITES, as this event only counts writebacks of locally cached lines.

The UNC_IMC_NORMAL_READS.* events only count the reads. The UNC_QHL_REQUESTS.LOCAL_READS and the UNC_QHL_REQUESTS.REMOTE_READS count the reads and the "InvtoE" transactions, which are issued for the uncacheable writes, eg USWC/UC writes. This allows the evaluation of the uncacheable writes, by computing the difference of UNC_QHL_REQUESTS.LOCAL_READS +

UNC_QHL_REQUESTS.REMOTE_READS – UNC_IMC_NORMAL_READS.ANY.

These uncore PMU events that are useful for bandwidth evaluation are listed under event code 20H, 2CH, 2FH in Chapter 19, "Performance Monitoring Events" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

B.4 PERFORMANCE TUNING TECHNIQUES FOR INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

This section covers various performance tuning techniques using performance monitoring events. Some techniques can be adapted in general to other microarchitectures, most of the performance events are specific to Intel microarchitecture code name Sandy Bridge.

B.4.1 Correlating Performance Bottleneck to Source Location

Performance analysis tools often sample events to identify hot spots of instruction pointer addresses to help programmers identify source locations of potential performance bottlenecks.

The sampling technique requires a service routine to respond to the performance monitoring interrupt (PMI) generated from an overflow condition of the performance counter. There is a finite delay between the performance monitoring event detection of the eventing condition relative to the capture of the instruction pointer address. This is known as "skid". In other words, the event skid is the distance between the instruction or instructions that caused the issue and the instruction where the event is tagged. There are a few things to note in general on skid:

- Precise events have a defined event skid of 1 instruction to the next instruction retired. In the case when the offending instruction is a branch, the event is tagged with the branch target, which can be separated from the branch instruction. Thus sampling with precise events is likely to have less noise in pin-pointing source locations of bottlenecks.
- Using a performance event with eventing condition that carries a larger performance impact generally has a shorter skid and vice versa. The following examples illustrate this rule:
  - A store forward block issue can cause a penalty of more than 10 cycles. Sampling a store forward block event almost always tags to the next couple of instructions after the blocked load.
  - On the other hand, sampling loads that forwarded successfully with no penalty will have much larger skids, and less helpful for performance tuning.
- The closer the eventing condition is to the retirement of the instruction, the shorter the skid. The events in the front end of the pipeline tend to tag to instructions further from the responsible instruction than events that are taken at execution or retirement.

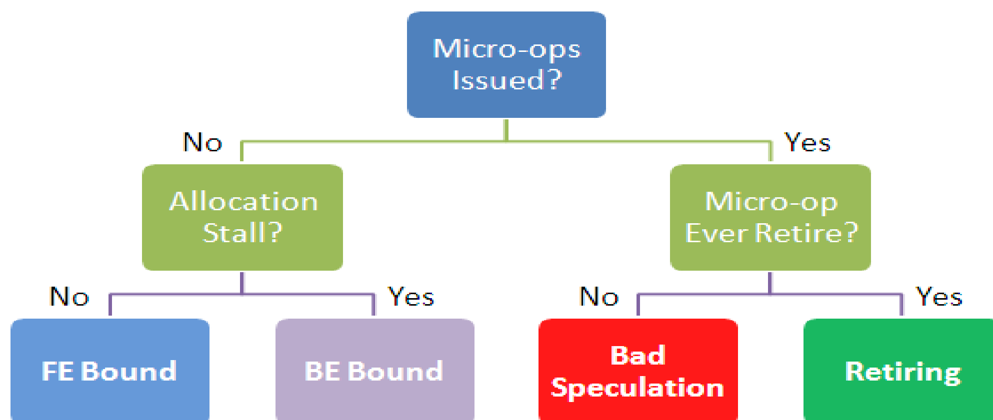
- Cycles counted with the event CPU_CLK_UNHALTED.THREAD often tag in greater counts on the instruction after larger bottlenecks in the pipeline. If cycles are accumulated on an instruction this is probably due to a bottleneck on the instruction at the previous instruction.
- It is very difficult to determine the source of issues with a low cost that occur in the front end. Front end events can also skid to IPs that precede the actual instructions that are causing the issue.

B.4.2 Hierarchical Top-Down Performance Characterization Methodology and Locating Performance Bottlenecks

Intel microarchitecture code name Sandy Bridge has introduced several performance events which help narrow down which portion of the microarhcitecture pipeline is stalled. This starts with a hierarchical approach to characterize a workload of where CPU cycles are spent in the microarchitecture pipelines. At the top level, there are 4 areas to attribute CPU cycles: which are described below. To determine what portion of the pipeline is stalled, the technique looks at a buffer that queues the micro-ops supplied by the front end and feeds the out-of-order back end (see Section 2.3.1). This buffer is called the micro-op queue. From the micro-op queue viewpoint, there may be four different types of stalls:

- Front end stalls - The front end is delivering less than four micro-ops per cycle when the back end of the pipeline is requesting micro-ops. When these stalls happen, the rename/allocate part of the OOO engine will starved. Thus, execution is said to be front end bound.
- Back end stalls – No micro-ops are being delivered from the micro-op queue due to lack of required resources for accepting more micro-ops in the back end of the pipeline. When these stalls happen, execution is said to be back end bound.
- Bad speculation - The pipeline performs speculative execution of instructions that never successfully retire. The most common case is a branch misprediction where the pipeline predicts a branch target in order to keep the pipeline full instead of waiting for the branch to execute. If the processor prediction is incorrect it has to flush the pipeline without retiring the speculated instructions.
- Retiring – The micro-op queue delivers micro-ops that eventually retire. In the common case, the micro-ops originate from the program code. One exception is with assists where the microcode sequencer generates micro-ops to deal with issues in the pipeline.

The following figure illustrates how the execution opportunities are logically divided.



It is possible to estimate the amount of execution slots spent in each category using the following formulas in conjunction with core PMU performance events in Intel microarchitecture code name Sandy Bridge:

%FE_Bound =

100 * (IDQ_UOPS_NOT_DELIVERED.CORE / **N**) ;

%Bad_Speculation =

100 * ((UOPS_ISSUED.ANY – UOPS_RETIRED.RETIRE_SLOTS + 4 * INT_MISC.RECOVERY_CYCLES) / **N**) ;

%Retiring = 100 * (UOPS_RETIRED.RETIRE_SLOTS/ **N**) ;
%BE_Bound = 100 * (1 – (**FE_Bound** + **Retiring** + **Bad_Speculation**)) ;

N represents total execution slots opportunities. Execution opportunities are the number of cycles multiplied by four.

- **N** = 4*CPU_CLK_UNHALTED.THREAD

The following sections explain the source for penalty cycles in three categories: back end stalls, front end stalls and bad speculation. They use formulas that can be applied to process, module, function, and instruction granularity.

B.4.2.1 Back End Bound Characterization

Once the %BE_Bound metric raises concern, a user may need to drill down to the next level of possible issues in the back end. Our methodology examines back end stalls based on execution unit occupation at every cycle. Naturally, optimal performance may be achieved when all execution resources are kept busy. Currently, this methodology splits **back end bound** issues into two categories: **memory bound** and **core bound**.

"Memory bound" corresponds to stalls related to the memory subsystem. For example, cache misses may eventually cause execution starvation. On the other hand, "core bound" which corresponds to stalls due to either the Execution- or OOO-clusters, is a bit trickier. These stalls can manifest either with execution starvation or non-optimal execution ports utilization. For example, a long latency divide operation may serialize the execution causing execution starvation for some period, while pressure on an execution port that serves specific types of uops, might manifest as small number of ports utilized in a cycle.

To calculate this, we use performance monitoring events at the execution units:

%BE_Bound_at_EXE =

(CYCLE_ACTIVITY.CYCLES_NO_EXECUTE + UOPS_EXECUTED.THREAD:c1 -
UOPS_EXECUTED.THREAD:c2) / CLOCKS

CYCLE_ACTIVITY.CYCLES_NO_EXECUTE counts complete starvation cycles where no uop is executed whatsoever.

UOPS_EXECUTED.THREAD:c1 and UOPS_EXECUTED.THREAD:c2 count cycles where at least 1- and 2-uops were executed in a cycle, respectively. Hence the event count difference measures the cycles when the OOO back end could execute only 1 uop.

The **%BE_Bound_at_EXE** metric is counted at execution unit pipestages so the number would not match the Backend_Bound ratio which is done at the allocation stage. However, redundancy is good here as one can use both counters to confirm the execution is indeed back end bound (both should be high).

B.4.2.2 Core Bound Characterization

A "back end bound" workload can be identified as "core bound" by the following metric:

%Core_Bound = %Backend_Bound_at_EXE - %Memory_Bound

The metric "%Memory_Bound" is described in Section B.4.2.3. Once a workload is identified as "core bound", the user may want to drill down into OOO or Execution related issues through their transitional targeted performance counter, like, for example, execution ports pressure, or use of FP-chained long-latency arithmetic operations.

B.4.2.3 Memory Bound Characterization

More primitive methods of characterizing performance issues in the memory pipeline tend to use naïve calculations to estimate the penalty of memory stalls. Usually the number of misses to a given cache level access is multiplied by a pre-defined latency for that cache level per the CPU specifications, in order to get an estimation for the penalty. While this might work for an in-order processor, it often over-estimates the contribution of memory accesses on CPU cycles for highly out-of-order processors, because memory accesses tend to overlap and the scheduler manages to hide a good portion of the latency. The

scheduler might be able to hide some of the memory access stalls by keeping the execution stalls busy with uops that do not require the memory access data. Thus penalty for a memory access is when the scheduler has nothing more ready to dispatch and the execution units get starved as a result. It is likely that further uops are either waiting for memory access data, or depend on other non-dispatched uops.

In Intel microarchitecture code name Ivy Bridge, a new performance monitoring event "CYCLE_ACTIVITY.STALLS_LDM_PENDING" is provided to estimate the exposure of memory accesses. We use it to define the "**memory bound**" metric. This event measures the cycles when there is a non-completed in-flight memory demand load coincident with execution starvation. Note we account only for demand load operations as uops do not typically wait for (direct) completion of stores or HW prefetches:

%Memory_Bound = CYCLE_ACTIVITY.STALLS_LDM_PENDING / CLOCKS

If a workload is memory bound, it is possible to further characterize its performance characteristic with respect to the contribution of the cache hierarchy and DRAM system memory.

L1 cache has typically the shortest latency which is comparable to ALU units' stalls that are the shortest among all stalls. Yet in certain cases, like loads blocked on older stores, a load might suffer high latency while eventually being satisfied by the L1. There are no fill-buffers allocated for L1 hits; instead we'll use the LDM stalls sub-event as it accounts for any non-completed load.

%L1 Bound = (CYCLE_ACTIVITY.STALLS_LDM_PENDING -CYCLE_ACTIVITY.STALLS_L1D_PENDING)/ CLOCKS

As explained above, L2 Bound is detected as:

%L2 Bound =(CYCLE_ACTIVITY.STALLS_L1D_PENDING - CYCLE_ACTIVITY.STALLS_L2_PENDING)/ CLOCKS

In principle, L3 Bound can be calculated similarly by subtracting out the L3 miss contribution. However an equivalent event to measure L3_PENDING is not available. Nevertheless, we can infer an estimate using L3_HIT and L3_MISS load count events in conjunction with a correction factor. This estimation could be tolerated as the latencies are longer on L3 and Memory. The correction factor MEM_L3_WEIGHT is approximately the external memory to L3 cache latency ratio. A factor of 7 can be used for the third generation Intel Core processor family. Note this correction factor has some dependency on CPU and Memory frequencies.

%L3 Bound = CYCLE_ACTIVITY.STALLS_L2_PENDING * L3_Hit_fraction / CLOCKS

Where L3_Hit_fraction is:

MEM_LOAD_UOPS_RETIRED.LLC_HIT / (MEM_LOAD_UOPS_RETIRED.LLC_HIT+ MEM_L3_WEIGHT*MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS)

To estimate the exposure of DRAM traffic on third generation Intel Core processors, the remainder of L2_PENDING is used for MEM Bound:

%MEM Bound = CYCLE_ACTIVITY.STALLS_L2_PENDING * L3_Miss_fraction / CLOCKS

Where L3_Miss_fraction is:

WEIGHT*MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS / (MEM_LOAD_UOPS_RETIRED.LLC_HIT+ WEIGHT*MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS)

Sometimes it is meaningful to refer to all memory stalls outside the core as Uncore Bound:

%Uncore Bound = CYCLE_ACTIVITY.STALLS_L2_PENDING / CLOCKS

B.4.3 Back End Stalls

Back end stalls have two main sources: memory sub-system stalls and execution stalls. As a first step to understanding the source of back end stalls, use the resource stall event.

Before putting micro-ops into the scheduler, the rename stage has to have certain resources allocated. When an application encounters a significant bottleneck at the back end of the pipeline, it runs out of these resources as the pipeline backs up. The RESOURCE_STALLS event tracks stall cycles when a resource could not be allocated. The event breaks up each resource into a separate sub-event so you can

track which resource is not available for allocation. Counting these events can help identifying the reason for issues in the back end of the pipeline.

The resource stall ratios described below can be accomplished at process, module, function and even instruction granularities with the cycles, counted by CPU_CLK_UNHALTED.THREAD, representing the penalty tagged at the same granularity.

Usages of Specific Events

RESOURCE_STALLS.ANY - Counts stall cycles that the rename stage is unable to put micro-ops into the scheduler, due to lack of resources that have to be allocated at this stage. The event skid tends to be low since it is close to the retirement of the blocking instruction. This event accounts for all stalls counted by other RESOURCE_STALL sub events and also includes the sub-events of RESOURCE_STALLS2. If this ratio is high, count the included sub-events to get a better isolation of the reason for the stall.

%RESOURCE.STALLS.COST =
    100 * RESOURCE_STALLS.ANY / CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS.SB - Occurs when a store micro-op is ready for allocation and all store buffer entries are in use, usually due to long latency stores in progress. Typically this event tags to the IP after the store instruction that is stalled at allocation.

%RESOURCE.STALLS.SB.COST =
    100 * RESOURCE_STALLS.SB / CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS.LB - Counts cycles in which a load micro-op is ready for allocation and all load buffer entries are taken, usually due to long latency loads in progress. In many cases the queue to the scheduler becomes full by micro-ops that depend on the long latency loads, before the load buffer gets full.

%RESOURCE.STALLS.LB.COST =
    100 * RESOURCE_STALLS.LB / CPU_CLK_UNHALTED.THREAD;

In the above cases the event RESOURCE_STALLS.RS will often count in parallel. The best methodology to further investigate loss in data locality is the high cache line replacement study described in Section B.4.4.2, concentrating on L1 DCache replacements first

RESOURCE_STALLS.RS - Scheduler slots are typically the first resource that runs out when the pipeline is backed up. However, this can be due to almost any bottleneck in the back end, including long latency loads and instructions backed up at the execute stage. Thus it is recommended to investigate other resource stalls, before digging into the stalls tagged to lack of scheduler entries. The skid tends to be low on this event.

%RESOURCE.STALLS.RS.COST =
    100 * RESOURCE_STALLS.RS/ CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS.ROB - Counts cycles when allocation stalls because all the reorder buffer (ROB) entries are taken. This event occurs less frequently than the RESOURCE_STALLS.RS and typically indicates that the pipeline is being backed up by a micro-op that is holding all younger micro-ops from retiring because they have to retire in order.

%RESOURCE.STALLS.ROB.COST =
    100 * RESOURCE_STALLS.ROB/ CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS2.BOB_FULL - Counts when allocation is stalled due to a branch micro-op that is ready for allocation, but the number of branches in progress in the processor has reached the limit.

%RESOURCE.STALLS.BOB.COST =
    100 * RESOURCE_STALLS2.BOB/ CPU_CLK_UNHALTED.THREAD;

B.4.4 Memory Sub-System Stalls

The following subsections discusses using specific performance monitoring events in Intel microarchitecture code name Sandy Bridge to identify stalls in the memory sub-systems.

B.4.4.1 Accounting for Load Latency

The breakdown of load operation locality can be accomplished at any granularity including process, module, function and instruction. When you find that a load instruction is a bottleneck, investigate it further with the precise load breakdown. If this does not explain the bottleneck, check for other issues which can impact loads.

You can use these events to estimate the costs of the load causing a bottleneck, and to obtain a percentage breakdown of memory hierarchy level. Not all tools provide support for precise event sampling. If the precise version (event name ends with a suffix PS) of these event is not supported in a given tool, you can use the non-precise version.

The precise load events tag the event to the next instruction retired (IP+1). See the load latency at each hierarchy level in Table 2-17.

Required events

MEM_LOAD_UOPS_RETIRED.L1_HIT_PS - Counts demand loads that hit the first level of the data cache, the L1 DCache. Demand loads are non speculative load micro-ops.

MEM_LOAD_UOPS_RETIRED.L2_HIT_PS - Counts demand loads that hit the 2nd level cache, the L2.

MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS - Counts demand loads that hit the 3rd level shared cache, the LLC.

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS - Counts demand loads that hit the 3rd level shared cache and are assumed to be present also in a cache of another core but the cache line was already evicted from there.

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS - Counts demand loads that hit a cache line in a cache of another core and the cache line has not been modified.

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS - Counts demand loads that hit a cache line in the cache of another core and the cache line has been written to by that other core. This event is important for many performance bottlenecks that can occur in multi-threaded applications, such as lock contention and false sharing.

MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS_PS - Counts demand loads that missed the LLC. This means that the load is usually satisfied from memory in client system.

MEM_LOAD_UOPS_RETIRED.HIT_LFB_PS - Counts demand loads that hit in the line fill buffer (LFB). A LFB entry is allocated every time a miss occurs in the L1 DCache. When a load hits at this location it means that a previous load, store or hardware prefetch has already missed in the L1 DCache and the data fetch is in progress. Therefore the cost of a hit in the LFB varies. This event may count cache-line split loads that miss in the L1 DCache but do not miss the LLC.

On 32-byte Intel AVX loads, all loads that miss in the L1 DCache show up as hits in the L1 DCache or hits in the LFB. They never show hits on any other level of memory hierarchy. Most loads arise from the line fill buffer (LFB) when Intel AVX loads miss in the L1 DCache.

Precise Load Breakdown

The percentage breakdown of each load source can be tagged at any granularity including a single IP, function, module, or process. This is particularly useful at a single instruction to determine the breakdown of where the load was found in the cache hierarchy. The following formula shows how to calculate the percentage of time a load was satisfied by the LLC. Similar formulas can be used for all other hierarchy levels.

%LocL3.HIT =
    100 * MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS / **$SumOf_PRECISE_LOADS;**

$SumOf_PRECISE_LOADS =
    MEM_LOAD_UOPS_RETIRED.HIT_LFB_PS +MEM_LOAD_UOPS_RETIRED.L1_HIT_PS +
    MEM_LOAD_UOPS_RETIRED.L2_HIT_PS +MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS +
    MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS +

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS +
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS +
MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS_PS;

Estimated Load Penalty

The formulas below help estimating to what degree loads from a certain memory hierarchy are responsible for a slowdown. The CPU_CLK_UNHALTED.THREAD programmable event represents the penalty in cycles tagged at the same granularity. At the instruction level, the cycle cost of an expensive load tends to only skid one IP, similar to the precise event. The calculations below apply to any granularity process, module, function or instruction, since the events are precise. Anything representing 10%, or higher, of the total clocks in a granularity of interest should be investigated.

If the code has highly dependent loads you can use the MEM_LOAD_UOPS_RETIRED.L1_HIT_PS event to determine if the loads are hit by the five cycle latency of the L1 DCache.

Estimated cost of L2 latency

%L2.COST =
12 * MEM_LOAD_UOPS_RETIRED.L2_HIT_PS / CPU_CLK_UNHALTED.THREAD;

Estimated cost of L3 hits

%L3.COST =
26 * MEM_LOAD_UOPS_RETIRED.L3_HIT_PS / CPU_CLK_UNHALTED.THREAD;

Estimated cost of hits in the cache of other cores

%HIT.COST =
43* MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS /
CPU_CLK_UNHALTED.THREAD;

Estimated cost of memory latency

%MEMORY.COST =
200 * MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS_PS /
CPU_CLK_UNHALTED.THREAD;

Actual memory latency can vary greatly depending on memory parameters. The amount of concurrent memory traffic often reduces the effect cost of a given memory hierarchy. Typically, the estimates above may be on the pessimistic side (like pointer-chasing situations).

Often, cache misses will manifest as delaying and bunching on the retirement of instructions. The precise loads breakdown can provide estimates of the distribution of hierarchy levels where the load is satisfied.

Given a significant impact from a particular cache level, the first step is to find where heavy cache line replacements are occurring in the code. This could coincide with your hot portions of code detected by the memory hierarchy breakdown, but often does not. For instance, regular traversal of a large data structure can unintentionally clear out levels of cache.

If hits of non modified or modified data in another core have high estimated cost and are hot at locations in the code, it can be due to locking, sharing or false sharing issues between threads.

If load latency in memory hierarchy levels further from the L1 DCache does not justify the amount of cycles spent on a load, try one of the following:

- Eliminate unneeded load operations such as spilling general purpose registers to XMM registers rather than memory.
- Continue searching for issues impacting load instructions described in Section B.4.4.4.

B.4.4.2 Cache-line Replacement Analysis

When an application has many cache misses, it is a good idea to determine where cache lines are being replaced at the highest frequency. The instructions responsible for high amount of cache replacements are not always where the application is spending the majority of its time, since replacements can be driven by the hardware prefetchers and store operations which in the common case do not hold up the pipeline. Typically traversing large arrays or data structures can cause heavy cache line replacements.

Required events:

L1D.REPLACEMENT - Replacements in the 1st level data cache.

L2_LINES_IN.ALL - Cache lines being brought into the L2 cache.

Usages of events:

Identifying the replacements that potentially cause performance loss can be done at process, module, and function level. Do it in two steps:

- Use the precise load breakdown to identify the memory hierarchy level at which loads are satisfied and cause the highest penalty.
- Identify, using the formulas below, which portion of code causes the majority of the replacements in the level below the one that satisfies these high penalty loads.

For example, if there is high penalty due to loads hitting the LLC, check the code which is causing replacements in the L2 and the L1. In the formulas below, the nominators are the replacements accounted for a module or function. The sum of the replacements in the denominators is the sum of all replacements in a cache level for all processes. This enables you to identify the portion of code that causes the majority of the replacements.

L1D Cache Replacements

%L1D.REPLACEMENT =
    L1D.REPLACEMENT / SumOverAllProcesses(L1D.REPLACEMENT);

L2 Cache Replacements

%L2.REPLACEMENT =
    L2_LINES_IN.ALL / SumOverAllProcesses(L2_LINES_IN.ALL);

B.4.4.3 Lock Contention Analysis

The amount of contention on locks is critical in scalability analysis of multi-threaded applications. A typical ring3 lock almost always results in the execution of an atomic instruction. An atomic instruction is either an XCHG instruction involving a memory address or one of the following instructions with memory destination and lock prefix: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR or XADD. Precise events enable you to get an idea of the contention on any lock. Many locking APIs start by an atomic instruction in ring3 and back off a contended lock by jumping into ring0. This means many locking APIs can be very costly in low contention scenarios. To estimate the amount of contention on a locked instruction, you can measure the number of times the cache line containing the memory destination of an atomic instruction is found modified in another core.

Required events:

MEM_UOPS_RETIRED.LOCK_LOADS_PS - Counts the number of atomic instructions which are retired with a precise skid of IP+1.

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS - Counts the occurrences that the load hits a modified cache line in another core. This event is important for many performance bottlenecks that can occur in multi-core systems, such as lock contention, and false sharing.

Usages of events:

The lock contention factor gives the percentage of locked operations executed that contend with another core and therefore have a high penalty. Usually a lock contention factor over 5% is worth investigating on a hot lock. A heavily contended lock may impact the performance of multiple threads.

%LOCK.CONTENTION =
    100 * MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS /
    MEM_UOPS_RETIRED.LOCK_LOAD_PS;

B.4.4.4 Other Memory Access Issues

Store Forwarding Blocked

When store forwarding is not possible the dependent loads are blocked. The average penalty for store forward block is 13 cycles. Since many cases of store forwarding blocked were fixed in prior architectures, the most common case in code today involves storing to a smaller memory space than an ensuing larger load.

Required events:

LD_BLOCKS.STORE_FORWARD - Counts the number of times a store forward opportunity is blocked due to the inability of the architecture to forward a small store to a larger load and some rare alignment cases.

Usages of Events:

Use the following formula to estimate the cost of the store forward block. The event LD_BLOCKS.STORE_FORWARD tends to be tagged to the next IP after the attempted load, so it is recommended to look at this issue at the instruction level. However it is possible to inspect the ratio at any granularity: process, module, function or IP.

```
%STORE.FORWARD.BLOCK.COST =
    100 *LD_BLOCKS.STORE_FORWARD * 13 / CPU_CLK_UNHALTED.THREAD;
```

After you find a load that is blocked from store forwarding, you need to find the location of the store. Typically, about 60% of all store forwarded blocked issue are caused by stores in the last 10 instructions executed prior to the load.

The most common case where we see store forward blocked is a small store that is unable to forward to a larger load. For example the code below generated writes to a byte pointer address and then reads from a four byte (dword) memory space:

```
    and     byte ptr [ebx],7f
    and     dword ptr [ebx], ecx
```

To fix a store forward block it's usually best to fix the store operation and not the load.

Cache Line Splits

Starting from the Intel microarchitecture code name Nehalem, the L1 DCache has split registers which enable it to handle loads and stores that span two cache lines in a faster manner. This puts the cost of split loads at about five cycles, as long as split registers are available, instead of the 20 cycles required in earlier microarchitectures. Handling of split stores handling is usually hidden, but if there are many of them they can stall allocation due to a full store buffer, or they can consume split registers that may be needed for handling split loads. You can still get solid quantifiable gains from eliminating cache line splits.

Required events:

MEM_UOPS_RETIRED.SPLIT_LOADS_PS - Counts the number of demand loads that span two cache lines. The event is precise.

MEM_UOPS_RETIRED.SPLIT_STORES_PS - Counts the number of stores that span two cache lines. The event is precise.

Usages of events:

Finding split loads is fairly easy because they usually tag the majority of their cost to the next IP which is executed. The ratio below can be used at any granularity: process, module, function, and IP after split.

```
%SPLIT.LOAD.COST =
    100 * MEM_UOPS_RETIRED.SPLIT_STORES_PS * 5 / CPU_CLK_UNHALTED.THREAD;
```

Split store penalty is more difficult to find using an estimated cost, because in typical cases stores do not push out the retirement of instructions. To detect significant amount of split stores divide their number by the total number of stores retired at that IP.

```
SPLIT.STORE.RATIO =
    MEM_UOPS_RETIRED.SPLIT_STORES_PS / MEM_UOPS_RETIRED.ANY_STORES_PS;
```

4k Aliasing

A 4k aliasing conflict between loads and stores causes a reissue on the load. Five cycles is used as an estimate in the model below.

Required Events:

LD_BLOCKS_PARTIAL.ADDRESS_ALIAS - Counts the number of loads that have partial address match with preceding stores, causing the load to be reissued.

Usages of events:

%4KALIAS.COST =

100 * LD_BLOCK_PARTIAL.ADDRESS_ALIAS * 5 / CPU_CLK_UNHALTED.THREAD;

Load and Store Address Translation

There are two levels of translation look-aside buffer (TLB) for linear to physical address translation. A miss in the DTLB, the first level TLB, that hits in the STLB, the second level TLB, incurs a seven cycle penalty.

Missing in the STLB requires the processor to walk through page table entries that contain the address translation. These walks have variable cost depending on the location of the page table entries. The walk duration is a fairly good estimate of the cost of STLB misses.

Required events:

DTLB_LOAD_MISSES.STLB_HIT - Counts loads that miss the DTLB and hit in the STLB. This event has a low skid and hence can be used at the IP level.

DTLB_LOAD_MISSES.WALK_DURATION - Duration of a page walks in cycles following STLB misses. Event skid is typically one instruction, enabling you to detect the issue at instruction, function, module or process granularities.

MEM_UOPS_RETIRED.STLB_MISS_LOADS_PS - Precise event for loads which have their translation miss the STLB. The event counts only the first load from a page that initiates the page walk.

Usage of events:

Cost of STLB hits on loads:

%STLB.HIT.COST =

100 * DTLB_LOAD_MISSES.STLB_HIT * 7/ CPU_CLK_UNHALTED.THREAD;

Cost of page walks:

%STLB.LOAD.MISS.WALK.COST =

100 * DTLB_LOAD_MISSES.WALK_DURATION / CPU_CLK_UNHALTED.THREAD;

Use the precise STLB miss event at the IP level to determine exactly which instruction and source line suffers from frequent STLB misses.

%STLB.LOAD.MISS =

100 * MEM_UOPS_RETIRED.STLB_MISS_LOADS_PS/
MEM_UOPS_RETIRED.ANY_LOADS_PS;

Large walk durations, of hundreds of cycles, are an indication that the page tables have been thrown out of the LLC. To determine the average cost of a page walk use the following ratio:

STLB.LOAD.MISS.AVGCOST =

DTLB_LOAD_MISSES.WALK_DURATION /
DTLB_LOAD_MISSES.WALK_COMPLETED;

To a lesser extent than loads, STLB misses on stores can be a bottleneck. If the store itself is a large bottleneck, cycles will tag to the next IP after the store.

%STLB.STORE.MISS =

100 * MEM_UOPS_RETIRED.STLB_MISS_STORES_PS/
MEM_UOPS_RETIRED.ANY_STORES_PS;

Reducing DTLB/STLB misses increases data locality. One may consider using an commercial-grade memory allocators to improve data locality. Compilers which offer profile guided optimizations may

reorder global variables to increase data locality, if the compiler can operate on the whole module. For issues with a large amount of time spent in page walks, server and HPC applications may be able to use large pages for data.

B.4.5 Execution Stalls

The following subsections discusses using specific performance monitoring events in Intel microarchitecture code name Sandy Bridge to identify stalls in the out-of-order engine.

B.4.5.1 Longer Instruction Latencies

Some microarchitectural changes manifested in longer latency for some legacy instructions in existing code. It is possible to detect some of these situations:

- Three-operand slow LEA instructions (see Section 3.5.1.3).
- Flags merge micro-op - These merges are primarily required by “shift cl” instructions (see Section 3.5.2.6).

These events tend to have a skid as high as a 10 instructions because the eventing condition is detected early in the pipeline.

Event usage:

To use this event effectively without being distracted by the event skid, you can use it to locate performance issue at the process, module and function granularities, but not at the instruction level. To identify issue at the instruction IP granularity, one can perform static analysis on the functions identified by this event. To estimate the contribution of these events to the code latency, divide them by the cycles at the same granularity. To estimate the overall impact, start with the total cycles due to these issues and if significant continue to search for the exact reason using the sub events.

Total cycles spent in the specified scenarios:

Flags Merge micro-op ratio:

```
%FLAGS.MERGE.UOP =
    100 * PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP_CYCLES /
    CPU_CLK_UNHALTED.THREAD;
```

Slow LEA instructions allocated:

```
%SLOW.LEA.WINDOW =
    100 * PARTIAL_RAT_STALLS.SLOW_LEA_WINDOW /
    CPU_CLK_UNHALTED.THREAD;
```

B.4.5.2 Assists

Assists usually involve the microcode sequencer that helps handle the assist. Determining the number of cycles where microcode is generated from the microcode sequencer is often a good methodology to determine the total cost of the assist. If the overcall cost of assists are high, a breakdown of assists into specific types will be useful.

Estimating the total cost of assists using microcode sequencer cycles:

```
%ASSISTS.COST =
    100 * IDQ.MS_CYCLES / CPU_CLK_UNHALTED.THREAD;
```

Floating-point assists:

Denormal inputs for X87 instructions require an FP assist, potentially costing hundreds of cycles.

```
%FP.ASSISTS =
    100 *FP_ASSIST.ANY / INST_RETIRED.ANY;
```

Transitions between Intel SSE and Intel AVX:

The transitions between SSE and AVX code are explained in detail in Section 11.3.1. The typical cost is about 75 cycles.

```
%AVX2SSE.TRANSITION.COST =
    75 * OTHER_ASSISTS.AVX_TO_SSE / CPU_CLK_UNHALTED.THREAD;
%SSE2AVX.TRANSITION.COST =
    75 * OTHER_ASSISTS.SSE_TO_AVX / CPU_CLK_UNHALTED.THREAD;
```

32-byte AVX store instructions that span two pages require an assist that costs roughly 150 cycles. A large amount of microcode tagged to the IP after a 32-byte AVX store is a good sign that an assist has occurred.

```
%AVX.STORE.ASSIST.COST =
    150 * OTHER_ASSISTS.AVX_STORE / CPU_CLK_UNHALTED.THREAD;
```

B.4.6 Bad Speculation

This section discusses mispredicted branch instructions resulting in a pipeline flush.

B.4.6.1 Branch Mispredicts

The largest challenge with mispredicted branches is finding the branch which caused them. Branch mispredictions incur penalty of about 20 cycles. The cost varies based upon the misprediction, and whether the correct path is found in the Decoded ICache or in the legacy decode pipeline.

Required Events:

BR_MISP_RETIRED.ALL_BRANCHES_PS is a precise event that counts branches that incorrectly predicted the branch target. Since this is a precise event that skids to the next instruction, it tags to the first instruction in the correct path after the branch misprediction. This study can be performed at the process, module, function or instruction granularity.

Usages of Events:

Use the following ratio to estimate the cost of mispredicted branches:

```
%BR.MISP.COST =
    20 * BR_MISP_RETIRED.ALL_BRANCHES_PS / CPU_CLK_UNHALTED.THREAD;
```

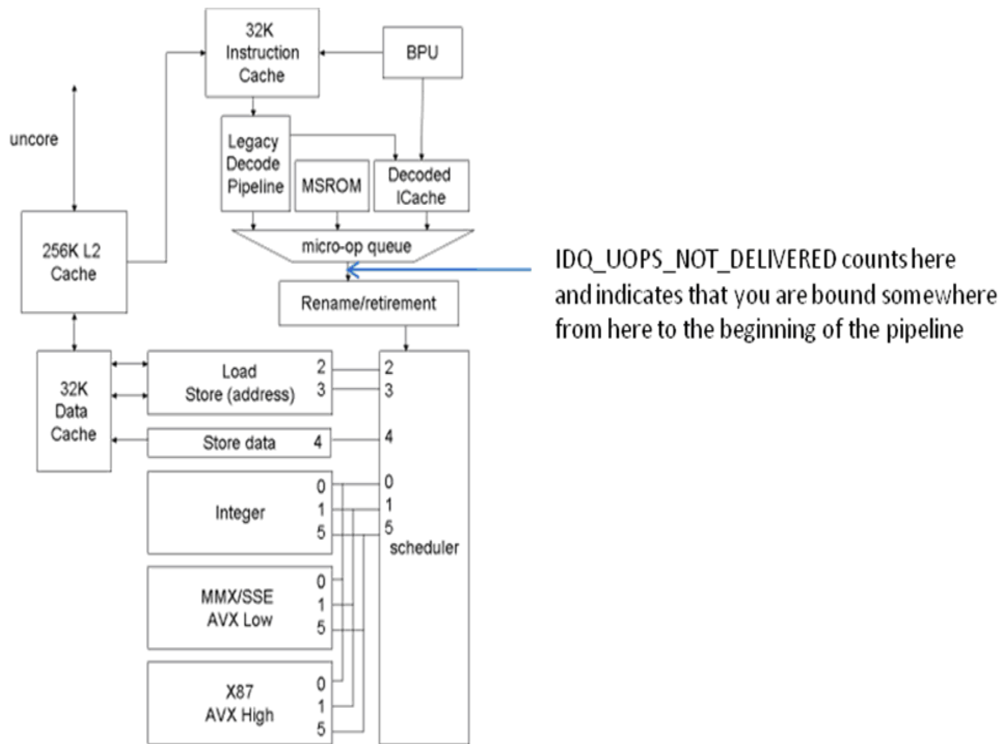
B.4.7 Front End Stalls

Stalls in the front end should not be investigated unless the analysis in Section B.4.2 showed at least 30% of a granularity being bound in the front end. This section explains the main issues that can cause delays in the front end of the pipeline. Events detected in the front end have unpredictable skid. Therefore do not try and associate the penalty at the IP level. Stay at the function, module, and process level for these events.

B.4.7.1 Understanding the Micro-op Delivery Rate

Usages of Counters

The event IDQ_UOPS_NOT_DELIVERED counts when the maximum of four micro-ops are not delivered to the rename stage, while it is requesting micro-ops. When the pipeline is backed up the rename stage does not request any further micro-ops from the front end. The diagram above shows how this event tracks micro-ops between the micro-op queue and the rename stage.



You can use the IDQ_UOPS_NOT_DELIVERED event to breakdown the distribution of cycles when 0, 1, 2, 3 micro-ops are delivered from the front end.

Percentage of cycles the front end is effective, or execution is back end bound:

%FE.DELIVERING =

100 * (CPU_CLK_UNHALTED.THREAD -
IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_3_UOP_DELIV.CORE) /
CPU_CLK_UNHALTED.THREAD;

Percentage of cycles the front end is delivering three micro-ops per cycle:

%FE.DELIVER.3UOPS =

100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_3_UOP_DELIV.CORE -
IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_2_UOP_DELIV.CORE) /
CPU_CLK_UNHALTED.THREAD;

Percentage of cycles the front end is delivering two micro-ops per cycle:

%FE.DELIVER.2UOPS =

100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_2_UOP_DELIV.CORE -
IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_1_UOP_DELIV.CORE) /
CPU_CLK_UNHALTED.THREAD;

Percentage of cycles the front end is delivering one micro-ops per cycle:

%FE.DELIVER.1UOPS =

100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_1_UOP_DELIV.CORE -
IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE) /
CPU_CLK_UNHALTED.THREAD;

Percentage of cycles the front end is delivering zero micro-ops per cycle:

%FE.DELIVER.0UOPS =

100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE) /
CPU_CLK_UNHALTED.THREAD;

Average Micro-ops Delivered per Cycle: This ratio assumes that the front end could potentially deliver four micro-ops per cycle when bound in the back end.

AVG.uops.per.cycle =

(4 * (%FE.DELIVERING) + 3 * (%FE.DELIVER.3UOPS) + 2 * (%FE.DELIVER.2UOPS) + (%FE.DELIVER.1UOPS)) / 100

Seeing the distribution of the micro-ops being delivered in a cycle is a hint at the front end bottlenecks that might be occurring. Issues such as LCPs and penalties from switching from the decoded ICache to the legacy decode pipeline tend to result in zero micro-ops being delivered for several cycles. Fetch bandwidth issues and decoder stalls result in less than four micro-ops delivered per cycle.

B.4.7.2 Understanding the Sources of the Micro-op Queue

The micro-op queue can get micro-ops from the following sources:

- Decoded ICache.
- Legacy decode pipeline.
- Microcode sequencer (MS).

A typical distribution is approximately 80% of the micro-ops coming from the Decoded ICache, 15% coming from legacy decode pipeline and 5% coming from the microcode sequencer. Excessive micro-ops coming from the legacy decode pipeline can be a warning sign that the Decoded ICache is not working effectively. A large portion of micro-ops coming from the microcode sequencer may be benign, such as complex instructions, or string operations, but can also be due to code assists handling undesired situations like Intel SSE to Intel AVX code transitions.

Description of Counters Required:

IDQ.DSB_UOPS - Micro-ops delivered to the micro-op queue from the Decoded ICache.

IDQ.MITE_UOPS - Micro-ops delivered to the micro-op queue from the legacy decode pipeline.

IDQ.MS_UOPS -Micro-ops delivered from the microcode sequencer.

Usage of Counters:

Percentage of micro-ops coming from Decoded ICache:

%UOPS.DSB =

IDQ.DSB_UOPS / **ALL_IDQ_UOPS**;

Percentage of micro-ops coming from legacy decoder pipeline:

%UOPS.MITE =

IDQ.MITE_UOPS / **ALL_IDQ_UOPS**;

Percentage of micro-ops coming from micro-sequencer:

%UOPS.MS =

IDQ.MS_UOPS / **ALL_IDQ_UOPS**;

ALL_IDQ_UOPS = (IDQ.DSB_UOPS + IDQ.MITE_UOPS + IDQ.MS_UOPS);

If your application is not bound in the front end then whether micro-ops are coming from the legacy decode pipeline or Decoded ICache is of lesser importance. Excessive micro-ops coming from the microcode sequencer are worth investigating further to see if assists might be a problem.

Cases to investigate are listed below:

- (%FE_BOUND > 30%) and (%UOPS.DSB < 70%)

  We use a threshold of 30% to define a “front end bound” case. This threshold may be applicable to many situations, but may also vary somewhat across different workloads.

  — Investigate why micro-ops are not coming from the Decoded ICache.

  — Investigate issues which can impact the legacy decode pipeline.
- (%FE_BOUND > 30%) and (%UOP_DSB > 70%)

— Investigate switches from Decoded ICache to legacy decode pipeline since it may be switching to run portions of code that are too small to be effective.

— Look at the amount of bad speculation, since branch mispredictions still impact FE performance.

— Determine the average number of micro-ops being delivered per 32-byte chunk hit. If there are many taken branches from one 32-byte chunk into another, it impacts the micro-ops being delivered per cycle.

— Micro-op delivery from the Decoded ICache may be an issue which is not covered.

- (%FE_BOUND < 20%) and (%UOPS_MS>25%)

  We use a threshold of 20% to define a "front end not bound" case. This threshold may be applicable to many situations, but may also vary somewhat across different workloads.

  The following steps can help determine why micro-ops came from the microcode, in order of most common to least common.

  — Long latency instructions - Any instruction over four micro-ops starts the microcode sequencer. Some instructions such as transcendentals can generate many micro-ops from the microcode.

  — String operations - string operations can produce a large amount of microcode. In some cases there are assists which can occur due to string operations such as REP MOVSB with trip count greater than 3, which costs 70+ cycles.

  — Assists - See Section B.4.5.2.

B.4.7.3 The Decoded ICache

The Decoded ICache has many advantages over the legacy decode pipeline. It eliminates many bottlenecks of the legacy decode pipeline such as instructions decoded into more than one micro-op and length changing prefix (LCP) stalls.

A switch to the legacy decode pipeline from the Decoded ICache only occurs when a lookup in the Decoded ICache fails and usually costs anywhere from zero to three cycles in the front end of the pipeline.

Required events:

The Decoded ICache events all have large skids and the exact instruction where they are tagged is usually not the source of the problem so only look for this issue at the process, module and function granularities.

DSB2MITE_SWITCHES.PENALTY_CYCLES - Counts the cycles attributed to the switch from the Decoded ICache to the legacy decode pipeline, excluding cycles when the micro-op queue cannot accept micro-ops because it is back end bound.

DSB2MITE_SWITCHES.COUNT - Counts the number of switches between the Decoded ICache and the legacy decode pipeline.

DSB_FILL.ALL_CANCEL - Counts when fills to the Decoded ICache are canceled.

DSB_FILL.EXCEED_DSB_LINES- Counts when a fill is canceled because the allocated lines for Decoded ICache has exceeded three for the 32-byte chunk.

Usage of Events:

Since these studies involve front end events, do not try to tag the event to a specific instruction.

Determining cost of switches from the Decoded ICache to the legacy decode pipeline.

```
%DSB2MITE.SWITCH.COST =
    100 * DSB2MITE_SWITCHES.PENALTY_CYCLES / CPU_CLK_UNHALTED.THREAD;
```

Determining the average cost per Decoded ICache switch to the legacy front end:

```
AVG.DSB2MITE.SWITCH.COST =
    DSB2MITE_SWITCHES.PENALTY_CYCLES / DSB2MITE_SWITCHES.COUNT;
```

Determining causes of misses in the decoded ICache

There are no partial hits in the Decoded ICache. If any micro-op that is part of that lookup on the 32-byte chunk is missing, a Decoded ICache miss occurs on all micro-ops for that transaction.

There are three primary reasons for missing micro-ops in the Decoded ICache:

- Portions of a 32-byte chunk of code were not able to fit within three ways of the Decoded ICache.
- A frequently run portion of your code section is too large for the Decoded ICache. This case is more common on server applications since client applications tend to have a smaller set of code which is “hot”.
- The Decoded ICache is getting flushed for example when an ITLB entry is evicted.

To determine if a portion of the 32-byte code is unable to fit into three lines within the Decoded ICache use the DSB_FILL.EXCEED_DSB_LINESevent at the process, module or function granularities

```
%DSB.EXCEED.WAY.LIMIT =
        100 * DSB_FILL.EXCEED_DSB_LINES/ DSB_FILL.ALL_CANCEL;
```

B.4.7.4 Issues in the Legacy Decode Pipeline

If a large percentage of the micro-ops going to the micro-op queue are being delivered from the legacy decode pipeline, you should check to see if there are bottlenecks impacting that stage. The most common bottlenecks in the legacy decode pipeline are:

- Fetch not providing enough instructions.

  This happens when hot code is poorly aligned. For example if the hot code being fetched to be run is on the 15th byte, then only one byte is fetched.
- Length changing prefix stalls in the instruction length decoder.

  Instructions that are decoded into two to four micro-ops may introduce a bubble in the decoder throughput. If the instruction queue, preceding the decoders, becomes full, this indicates that these instructions may cause a penalty.

```
%ILD.STALL.COST =
        100 * ILD_STALL.LCP * 3 / CPU_CLK_UNHALTED.THREAD;
```

B.4.7.5 Instruction Cache

Applications with large hot code sections tend to run into many issues with the instruction cache. This is more typical in server applications.

Required events:

ICACHE.MISSES - Counts the number of instruction byte fetches that miss the ICache

Usage of events:

To determine whether ICache misses are causing the issue, compare them to the instructions retired event count, using the same granularity (process, model, or function). Anything over 1% of instructions retired can be a significant issue.

```
ICACHE.PER.INST.RET =
        ICACHE.MISSES / INST_RETIRED.ANY;
```

If ICache misses are causing a significant problem, try to reduce the size of your hot code section, using the profile guided optimizations. Most compilers have options for text reordering which helps reduce the number of pages and, to a lesser extent, the number of pages your application is covering.

If the application makes significant use of macros, try to either convert them to functions, or use intelligent linking to eliminate repeated code.

B.5 USING PERFORMANCE EVENTS OF INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

There are performance events specific to the microarchitecture of Intel Core Solo and Intel Core Duo processors. See also: Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*).

B.5.1 Understanding the Results in a Performance Counter

Each performance event detects a well-defined microarchitectural condition occurring in the core while the core is active. A core is active when:

- It's running code (excluding the halt instruction).
- It's being snooped by the other core or a logical processor on the platform. This can also happen when the core is halted.

Some microarchitectural conditions are applicable to a sub-system shared by more than one core and some performance events provide an event mask (or unit mask) that allows qualification at the physical processor boundary or at bus agent boundary.

Some events allow qualifications that permit the counting of microarchitectural conditions associated with a particular core versus counts from all cores in a physical processor (see L2 and bus related events in Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*).

When a multi-threaded workload does not use all cores continuously, a performance counter counting a core-specific condition may progress to some extent on the halted core and stop progressing or a unit mask may be qualified to continue counting occurrences of the condition attributed to either processor core. Typically, one can adjust the highest two bits (bits 15:14 of the IA32_PERFEVTSELx MSR) in the unit mask field to distinguish such asymmetry (See Chapter 17, "Debug, Branch Profile, TSC, and Quality of Service," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*).

There are three cycle-counting events which will not progress on a halted core, even if the halted core is being snooped. These are: Unhalted core cycles, Unhalted reference cycles, and Unhalted bus cycles. All three events are detected for the unit selected by event 3CH.

Some events detect microarchitectural conditions but are limited in their ability to identify the originating core or physical processor. For example, bus_drdy_clocks may be programmed with a unit mask of 20H to include all agents on a bus. In this case, the performance counter in each core will report nearly identical values. Performance tools interpreting counts must take into account that it is only necessary to equate bus activity with the event count from one core (and not use not the sum from each core).

The above is also applicable when the core-specificity sub field (bits 15:14 of IA32_PERFEVTSELx MSR) within an event mask is programmed with 11B. The result of reported by performance counter on each core will be nearly identical.

B.5.2 Ratio Interpretation

Ratios of two events are useful for analyzing various characteristics of a workload. It may be possible to acquire such ratios at multiple granularities, for example: (1) per-application thread, (2) per logical processor, (3) per core, and (4) per physical processor.

The first ratio is most useful from a software development perspective, but requires multi-threaded applications to manage processor affinity explicitly for each application thread. The other options provide insights on hardware utilization.

In general, collect measurements (for all events in a ratio) in the same run. This should be done because:

- If measuring ratios for a multi-threaded workload, getting results for all events in the same run enables you to understand which event counter values belongs to each thread.
- Some events, such as writebacks, may have non-deterministic behavior for different runs. In such a case, only measurements collected in the same run yield meaningful ratio values.

B.5.3 Notes on Selected Events

This section provides event-specific notes for interpreting performance events listed in Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

- **L2_Reject_Cycles, event number 30H** — This event counts the cycles during which the L2 cache rejected new access requests.
- **L2_No_Request_Cycles, event number 32H** — This event counts cycles during which no requests from the L1 or prefetches to the L2 cache were issued.
- **Unhalted_Core_Cycles, event number 3C, unit mask 00H** — This event counts the smallest unit of time recognized by an active core.

  In many operating systems, the idle task is implemented using HLT instruction. In such operating systems, clock ticks for the idle task are not counted. A transition due to Enhanced Intel SpeedStep Technology may change the operating frequency of a core. Therefore, using this event to initiate time-based sampling can create artifacts.
- **Unhalted_Ref_Cycles, event number 3C, unit mask 01H** — This event guarantees a uniform interval for each cycle being counted. Specifically, counts increment at bus clock cycles while the core is active. The cycles can be converted to core clock domain by multiplying the bus ratio which sets the core clock frequency.
- **Serial_Execution_Cycles, event number 3C, unit mask 02H** — This event counts the bus cycles during which the core is actively executing code (non-halted) while the other core in the physical processor is halted.
- **L1_Pref_Req, event number 4FH, unit mask 00H** — This event counts the number of times the Data Cache Unit (DCU) requests to prefetch a data cache line from the L2 cache. Requests can be rejected when the L2 cache is busy. Rejected requests are re-submitted.
- **DCU_Snoop_to_Share, event number 78H, unit mask 01H** — This event counts the number of times the DCU is snooped for a cache line needed by the other core. The cache line is missing in the L1 instruction cache or data cache of the other core; or it is set for read-only, when the other core wants to write to it. These snoops are done through the DCU store port. Frequent DCU snoops may conflict with stores to the DCU, and this may increase store latency and impact performance.
- **Bus_Not_In_Use, event number 7DH, unit mask 00H** — This event counts the number of bus cycles for which the core does not have a transaction waiting for completion on the bus.
- **Bus_Snoops, event number 77H, unit mask 00H** — This event counts the number of CLEAN, HIT, or HITM responses to external snoops detected on the bus.

  In a single-processor system, CLEAN and HIT responses are not likely to happen. In a multiprocessor system this event indicates an L2 miss in one processor that did not find the missed data on other processors.

  In a single-processor system, an HITM response indicates that an L1 miss (instruction or data) found the missed cache line in the other core in a modified state. In a multiprocessor system, this event also indicates that an L1 miss (instruction or data) found the missed cache line in another core in a modified state.

B.6 DRILL-DOWN TECHNIQUES FOR PERFORMANCE ANALYSIS

Software performance intertwines code and microarchitectural characteristics of the processor. Performance monitoring events provide insights to these interactions. Each microarchitecture often provides a large set of performance events that target different sub-systems within the microarchitecture. Having a methodical approach to select key performance events will likely improve a programmer's understanding of the performance bottlenecks and improve the efficiency of code-tuning effort.

Recent generations of Intel 64 and IA-32 processors feature microarchitectures using an out-of-order execution engine. They are also accompanied by an in-order front end and retirement logic that enforces program order. Superscalar hardware, buffering and speculative execution often complicates the interpretation of performance events and software-visible performance bottlenecks.

This section discusses a methodology of using performance events to drill down on likely areas of performance bottleneck. By narrowed down to a small set of performance events, the programmer can take advantage of Intel VTune Performance Analyzer to correlate performance bottlenecks with source code locations and apply coding recommendations discussed in Chapter 3 through Chapter 8. Although the general principles of our method can be applied to different microarchitectures, this section will use performance events available in processors based on Intel Core microarchitecture for simplicity.

Performance tuning usually centers around reducing the time it takes to complete a well-defined workload. Performance events can be used to measure the elapsed time between the start and end of a workload. Thus, reducing elapsed time of completing a workload is equivalent to reducing measured processor cycles.

The drill-down methodology can be summarized as four phases of performance event measurements to help characterize interactions of the code with key pipe stages or sub-systems of the microarchitecture. The relation of the performance event drill-down methodology to the software tuning feedback loop is illustrated in Figure B-16.

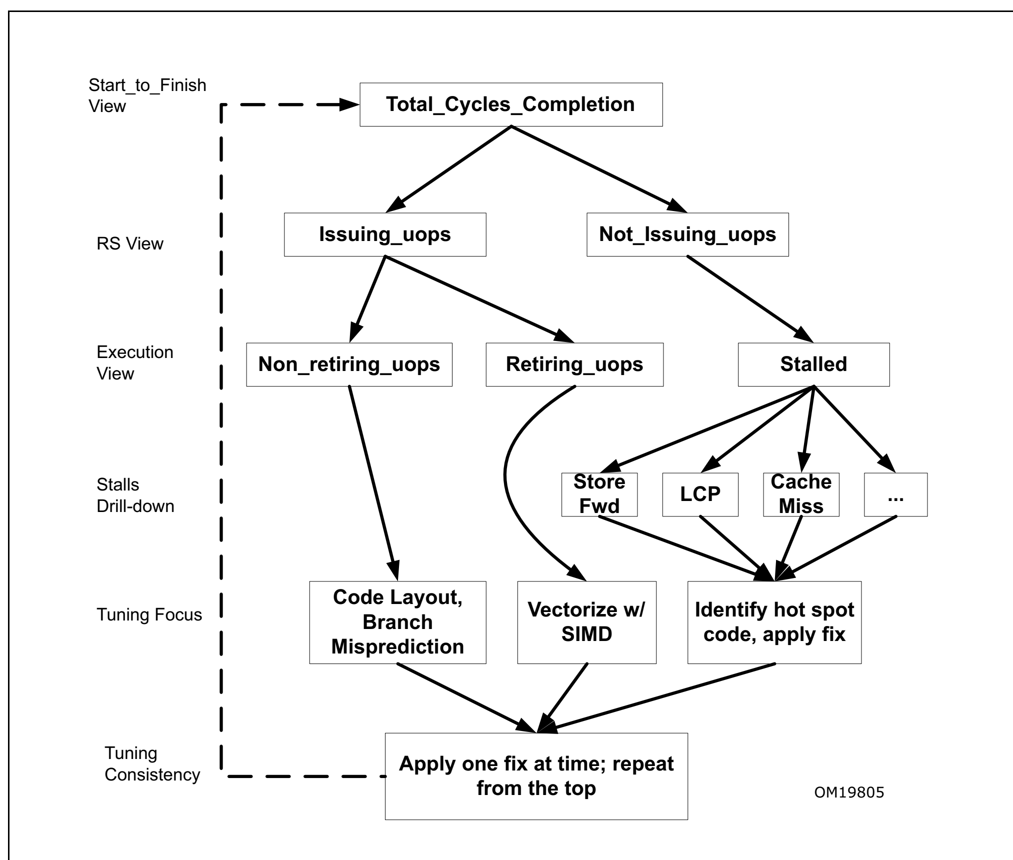


Figure B-16. Performance Events Drill-Down and Software Tuning Feedback Loop

Typically, the logic in performance monitoring hardware measures microarchitectural conditions that varies across different counting domains, ranging from cycles, micro-ops, address references, instances, etc. The drill-down methodology attempts to provide an intuitive, cycle-based view across different phases by making suitable approximations that are described below:

- **Total cycle measurement** — This is the start to finish view of total number of cycle to complete the workload of interest. In typical performance tuning situations, the metric Total_cycles can be measured by the event CPU_CLK_UNHALTED.CORE. See Chapter 19, "Performance Monitoring Events," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*).
- **Cycle composition at issue port** — The reservation station (RS) dispatches micro-ops for execution so that the program can make forward progress. Hence the metric Total_cycles can be

decomposed as consisting of two exclusive components: Cycles_not_issuing_uops representing cycles that the RS is not issuing micro-ops for execution, and Cycles_issuing_uops cycles that the RS is issuing micro-ops for execution. The latter component includes micro-ops in the architected code path or in the speculative code path.

- **Cycle composition of OOO execution** — The out-of-order engine provides multiple execution units that can execute micro-ops in parallel. If one execution unit stalls, it does not necessarily imply the program execution is stalled. Our methodology attempts to construct a cycle-composition view that approximates the progress of program execution. The three relevant metrics are: Cycles_stalled, Cycles_not_retiring_uops, and Cycles_retiring_uops.
- **Execution stall analysis** — From the cycle compositions of overall program execution, the programmer can narrow down the selection of performance events to further pin-point unproductive interaction between the workload and a micro-architectural sub-system.

When cycles lost to a stalled microarchitectural sub-system, or to unproductive speculative execution are identified, the programmer can use VTune Analyzer to correlate each significant performance impact to source code location. If the performance impact of stalls or misprediction is insignificant, VTune can also identify the source locations of hot functions, so the programmer can evaluate the benefits of vectorization on those hot functions.

B.6.1 Cycle Composition at Issue Port

Recent processor microarchitectures employ out-of-order engines that execute streams of micro-ops natively, while decoding program instructions into micro-ops in its front end. The metric Total_cycles alone, is opaque with respect to decomposing cycles that are productive or non-productive for program execution. To establish a consistent cycle-based decomposition, we construct two metrics that can be measured using performance events available in processors based on Intel Core microarchitecture. These are:

- **Cycles_not_issuing_uops** — This can be measured by the event RS_UOPS_DISPATCHED, setting the INV bit and specifying a counter mask (CMASK) value of 1 in the target performance event select (IA32_PERFEVTSELx) MSR (See Chapter 18 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*). In VTune Analyzer, the special values for CMASK and INV is already configured for the VTune event name RS_UOPS_DISPATCHED.CYCLES_NONE.
- **Cycles_issuing_uops** — This can be measured using the event RS_UOPS_DISPATCHED, clear the INV bit and specifying a counter mask (CMASK) value of 1 in the target performance event select MSR

Note the cycle decomposition view here is approximate in nature; it does not distinguish specificities, such as whether the RS is full or empty, transient situations of RS being empty but some in-flight uops is getting retired.

B.6.2 Cycle Composition of OOO Execution

In an OOO engine, speculative execution is an important part of making forward progress of the program. But speculative execution of micro-ops in the shadow of mispredicted code path represent unproductive work that consumes execution resources and execution bandwidth.

Cycles_not_issuing_uops, by definition, represents the cycles that the OOO engine is stalled (Cycles_stalled). As an approximation, this can be interpreted as the cycles that the program is not making forward progress.

The micro-ops that are issued for execution do not necessarily end in retirement. Those micro-ops that do not reach retirement do not help forward progress of program execution. Hence, a further approximation is made in the formalism of decomposition of Cycles_issuing_uops into:

- **Cycles_non_retiring_uops** — Although there isn't a direct event to measure the cycles associated with non-retiring micro-ops, we will derive this metric from available performance events, and several assumptions:

  - A constant issue rate of micro-ops flowing through the issue port. Thus, we define: uops_rate" = "Dispatch_uops/Cycles_issuing_uops, where Dispatch_uops can be measured with RS_UOPS_DISPATCHED, clearing the INV bit and the CMASK.
  - We approximate the number of non-productive, non-retiring micro-ops by [non_productive_uops = Dispatch_uops - executed_retired_uops], where executed_retired_uops represent productive micro-ops contributing towards forward progress that consumed execution bandwidth.
  - The executed_retired_uops can be approximated by the sum of two contributions: num_retired_uops (measured by the event UOPS_RETIRED.ANY) and num_fused_uops (measured by the event UOPS_RETIRED.FUSED).

  Thus, Cycles_non_retiring_uops = non_productive_uops / uops_rate.

- **Cycles_retiring_uops** — This can be derived from Cycles_retiring_uops = num_retired_uops / uops_rate.

The cycle-decomposition methodology here does not distinguish situations where productive uops and non-productive micro-ops may be dispatched in the same cycle into the OOO engine. This approximation may be reasonable because heuristically high contribution of non-retiring uops likely correlates to situations of congestions in the OOO engine and subsequently cause the program to stall.

Evaluations of these three components: Cycles_non_retiring_uops, Cycles_stalled, Cycles_retiring_uops, relative to the Total_cycles, can help steer tuning effort in the following directions:

- If the contribution from Cycles_non_retiring_uops is high, focusing on code layout and reducing branch mispredictions will be important.
- If both the contributions from Cycles_non_retiring_uops and Cycles_stalled are insignificant, the focus for performance tuning should be directed to vectorization or other techniques to improve retirement throughput of hot functions.
- If the contributions from Cycles_stalled is high, additional drill-down may be necessary to locate bottlenecks that lies deeper in the microarchitecture pipeline.

B.6.3 Drill-Down on Performance Stalls

In some situations, it may be useful to evaluate cycles lost to stalls associated with various stress points in the microarchitecture and sum up the contributions from each candidate stress points. This approach implies a very gross simplification and introduce complications that may be difficult to reconcile with the superscalar nature and buffering in an OOO engine.

Due to the variations of counting domains associated with different performance events, cycle-based estimation of performance impact at each stress point may carry different degree of errors due to over-estimation of exposures or under-estimations.

Over-estimation is likely to occur when overall performance impact for a given cause is estimated by multiplying the per-instance-cost to an event count that measures the number of occurrences of that microarchitectural condition. Consequently, the sum of multiple contributions of lost cycles due to different stress points may exceed the more accurate metric Cycles_stalled.

However an approach that sums up lost cycles associated with individual stress point may still be beneficial as an iterative indicator to measure the effectiveness of code tuning loop effort when tuning code to fix the performance impact of each stress point. The remaining of this sub-section will discuss a few common causes of performance bottlenecks that can be counted by performance events and fixed by following coding recommendations described in this manual.

The following items discuss several common stress points of the microarchitecture:

- **L2 Miss Impact** — An L2 load miss may expose the full latency of memory sub-system. The latency of accessing system memory varies with different chipset, generally on the order of more than a hundred cycles. Server chipset tend to exhibit longer latency than desktop chipsets. The number L2 cache miss references can be measured by MEM_LOAD_RETIRED.L2_LINE_MISS.

  An estimation of overall L2 miss impact by multiplying system memory latency with the number of L2 misses ignores the OOO engine's ability to handle multiple outstanding load misses. Multiplication of latency and number of L2 misses imply each L2 miss occur serially.

To improve the accuracy of estimating L2 miss impact, an alternative technique should also be considered, using the event BUS_REQUEST_OUTSTANDING with a CMASK value of 1. This alternative technique effectively measures the cycles that the OOO engine is waiting for data from the outstanding bus read requests. It can overcome the over-estimation of multiplying memory latency with the number of L2 misses.

- **L2 Hit Impact** — Memory accesses from L2 will incur the cost of L2 latency (See Table 2-27). The number cache line references of L2 hit can be measured by the difference between two events: MEM_LOAD_RETIRED.L1D_LINE_MISS - MEM_LOAD_RETIRED.L2_LINE_MISS.

  An estimation of overall L2 hit impact by multiplying the L2 hit latency with the number of L2 hit references ignores the OOO engine's ability to handle multiple outstanding load misses.
- **L1 DTLB Miss Impact** — The cost of a DTLB lookup miss is about 10 cycles. The event MEM_LOAD_RETIRED.DTLB_MISS measures the number of load micro-ops that experienced a DTLB miss.
- **LCP Impact** — The overall impact of LCP stalls can be directly measured by the event ILD_STALLS. The event ILD_STALLS measures the number of times the slow decoder was triggered, the cost of each instance is 6 cycles
- **Store forwarding stall Impact** — When a store forwarding situation does not meet address or size requirements imposed by hardware, a stall occurs. The delay varies for different store forwarding stall situations. Consequently, there are several performance events that provide fine-grain specificity to detect different store-forwarding stall conditions. These include:
  - A load blocked by preceding store to unknown address: This situation can be measure by the event Load_Blocks.Sta. The per-instance cost is about 5 cycles.
  - Load partially overlaps with proceeding store or 4-KByte aliased address between a load and a proceeding store: these two situations can be measured by the event Load_Blocks.Overlap_store.
  - A load spanning across cache line boundary: This can be measured by Load_Blocks.Until_Retire. The per-instance cost is about 20 cycles.

B.7 EVENT RATIOS FOR INTEL CORE MICROARCHITECTURE

Appendix B.7 provides examples of using performance events to quickly diagnose performance bottlenecks. This section provides additional information on using performance events to evaluate metrics that can help in wide range of performance analysis, workload characterization, and performance tuning. Note that many performance event names in the Intel Core microarchitecture carry the format of XXXX.YYY. This notation derives from the general convention that XXXX typically corresponds to a unique event select code in the performance event select register (IA32_PERFEVSELx), while YYY corresponds to a unique sub-event mask that uniquely defines a specific microarchitectural condition (See Chapter 18 and Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*).

B.7.1 Clocks Per Instructions Retired Ratio (CPI)

1. Clocks Per Instruction Retired Ratio (CPI): CPU_CLK_UNHALTED.CORE / INST_RETIRED.ANY.

The Intel Core microarchitecture is capable of reaching CPI as low as 0.25 in ideal situations. But most of the code has higher CPI The greater value of CPI for a given workload indicate it has more opportunity for code tuning to improve performance. The CPI is an overall metric, it does not provide specificity of what microarchitectural sub-system may be contributing to a high CPI value.

The following subsections defines a list of event ratios that are useful to characterize interactions with the front end, execution, and memory.

B.7.2 Front End Ratios

2. RS Full Ratio: RESOURCE_STALLS.RS_FULL / CPU_CLK_UNHALTED.CORE * 100
3. ROB Full Ratio: RESOURCE_STALLS.ROB_FULL / CPU_CLK_UNHALTED.CORE * 100
4. Load or Store Buffer Full Ratio: RESOURCE_STALLS.LD_ST / CPU_CLK_UNHALTED.CORE * 100

When there is a low value for the ROB Full Ratio, RS Full Ratio, and Load Store Buffer Full Ratio, and high CPI it is likely that the front end cannot provide instructions and micro-ops at a rate high enough to fill the buffers in the out-of-order engine, and therefore it is starved waiting for micro-ops to execute. In this case check further for other front end performance issues.

B.7.2.1 Code Locality

5. Instruction Fetch Stall: CYCLES_L1I_MEM_STALLED / CPU_CLK_UNHALTED.CORE * 100

The Instruction Fetch Stall ratio is the percentage of cycles during which the Instruction Fetch Unit (IFU) cannot provide cache lines for decoding due to cache and Instruction TLB (ITLB) misses. A high value for this ratio indicates potential opportunities to improve performance by reducing the working set size of code pages and instructions being executed, hence improving code locality.

6. ITLB Miss Rate: ITLB_MISS_RETIRED / INST_RETIRED.ANY

A high ITLB Miss Rate indicates that the executed code is spread over too many pages and cause many Instructions TLB misses. Retired ITLB misses cause the pipeline to naturally drain, while the miss stalls fetching of more instructions.

7. L1 Instruction Cache Miss Rate: L1I_MISSES / INST_RETIRED.ANY

A high value for L1 Instruction Cache Miss Rate indicates that the code working set is bigger than the L1 instruction cache. Reducing the code working set may improve performance.

8. L2 Instruction Cache Line Miss Rate: L2_IFETCH.SELF.I_STATE / INST_RETIRED.ANY

L2 Instruction Cache Line Miss Rate higher than zero indicates instruction cache line misses from the L2 cache may have a noticeable performance impact of program performance.

B.7.2.2 Branching and Front End

9. BACLEAR Performance Impact: 7 * BACLEARS / CPU_CLK_UNHALTED.CORE

A high value for BACLEAR Performance Impact ratio usually indicates that the code has many branches such that they cannot be consumed by the Branch Prediction Unit.

10. Taken Branch Bubble: (BR_TKN_BUBBLE_1+BR_TKN_BUBBLE_2) / CPU_CLK_UNHALTED.CORE

A high value for Taken Branch Bubble ratio indicates that the code contains many taken branches coming one after the other and cause bubbles in the front end. This may affect performance only if it is not covered by execution latencies and stalls later in the pipe.

B.7.2.3 Stack Pointer Tracker

11. ESP Synchronization: ESP.SYNCH / ESP.ADDITIONS

The ESP Synchronization ratio calculates the ratio of ESP explicit use (for example by load or store instruction) and implicit uses (for example by PUSH or POP instruction). The expected ratio value is 0.2 or lower. If the ratio is higher, consider rearranging your code to avoid ESP synchronization events.

B.7.2.4 Macro-fusion

12. Macro-Fusion: UOPS_RETIRED.MACRO_FUSION / INST_RETIRED.ANY

The Macro-Fusion ratio calculates how many of the retired instructions were fused to a single micro-op. You may find this ratio is high for a 32-bit binary executable but significantly lower for the equivalent 64-

bit binary, and the 64-bit binary performs slower than the 32-bit binary. A possible reason is the 32-bit binary benefited from macro-fusion significantly.

B.7.2.5 Length Changing Prefix (LCP) Stalls

13. LCP Delays Detected: ILD_STALL / CPU_CLK_UNHALTED.CORE

A high value of the LCP Delays Detected ratio indicates that many Length Changing Prefix (LCP) delays occur in the measured code.

B.7.2.6 Self Modifying Code Detection

14. Self Modifying Code Clear Performance Impact: MACHINE_NUKES.SMC * 150 / CPU_CLK_UNHALTED.CORE * 100

A program that writes into code sections and shortly afterwards executes the generated code may incur severe penalties. Self Modifying Code Performance Impact estimates the percentage of cycles that the program spends on self-modifying code penalties.

B.7.3 Branch Prediction Ratios

Appendix B.7.2.2 discusses branching that impacts the front end performance. This section describes event ratios that are commonly used to characterize branch mispredictions.

B.7.3.1 Branch Mispredictions

15. Branch Misprediction Performance Impact: RESOURCE_STALLS.BR_MISS_CLEAR / CPU_CLK_UNHALTED.CORE * 100

With the Branch Misprediction Performance Impact, you can tell the percentage of cycles that the processor spends in recovering from branch mispredictions.

16. Branch Misprediction per Micro-Op Retired: BR_INST_RETIRED.MISPRED/UOPS_RETIRED.ANY

The ratio Branch Misprediction per Micro-Op Retired indicates if the code suffers from many branch mispredictions. In this case, improving the predictability of branches can have a noticeable impact on the performance of your code.

In addition, the performance impact of each branch misprediction might be high. This happens if the code prior to the mispredicted branch has high CPI, such as cache misses, which cannot be parallelized with following code due to the branch misprediction. Reducing the CPI of this code will reduce the misprediction performance impact. See other ratios to identify these cases.

You can use the precise event BR_INST_RETIRED.MISPRED to detect the actual targets of the mispredicted branches. This may help you to identify the mispredicted branch.

B.7.3.2 Virtual Tables and Indirect Calls

17. Virtual Table Usage: BR_IND_CALL_EXEC / INST_RETIRED.ANY

A high value for the ratio Virtual Table Usage indicates that the code includes many indirect calls. The destination address of an indirect call is hard to predict.

18. Virtual Table Misuse: BR_CALL_MISSP_EXEC / BR_INST_RETIRED.MISPRED

A high value of Branch Misprediction Performance Impact ratio (Ratio 15) together with high Virtual Table Misuse ratio indicate that significant time is spent due to mispredicted indirect function calls.

In addition to explicit use of function pointers in C code, indirect calls are used for implementing inheritance, abstract classes, and virtual methods in C++.

B.7.3.3 Mispredicted Returns

19. Mispredicted Return Instruction Rate: BR_RET_MISSP_EXEC/BR_RET_EXEC

The processor has a special mechanism that tracks CALL-RETURN pairs. The processor assumes that every CALL instruction has a matching RETURN instruction. If a RETURN instruction restores a return address, which is not the one stored during the matching CALL, the code incurs a misprediction penalty.

B.7.4 Execution Ratios

This section covers event ratios that can provide insights to the interactions of micro-ops with RS, ROB, execution units, and so forth.

B.7.4.1 Resource Stalls

A high value for the RS Full Ratio (Ratio 2) indicates that the Reservation Station (RS) often gets full with micro-ops due to long dependency chains. The micro-ops that get into the RS cannot execute because they wait for their operands to be computed by previous micro-ops, or they wait for a free execution unit to be executed. This prevents exploiting the parallelism provided by the multiple execution units.

A high value for the ROB Full Ratio (Ratio 3) indicates that the reorder buffer (ROB) often gets full with micro-ops. This usually implies on long latency operations, such as L2 cache demand misses.

B.7.4.2 ROB Read Port Stalls

20. ROB Read Port Stall Rate: RAT_STALLS.ROB_READ_PORT / CPU_CLK_UNHALTED.CORE

The ratio ROB Read Port Stall Rate identifies ROB read port stalls. However it should be used only if the number of resource stalls, as indicated by Resource Stall Ratio, is low.

B.7.4.3 Partial Register Stalls

21. Partial Register Stalls Ratio: RAT_STALLS.PARTIAL_CYCLES / CPU_CLK_UNHALTED.CORE*100

Frequent accesses to registers that cause partial stalls increase access latency and decrease performance. Partial Register Stalls Ratio is the percentage of cycles when partial stalls occur.

B.7.4.4 Partial Flag Stalls

22. Partial Flag Stalls Ratio:RAT_STALLS.FLAGS / CPU_CLK_UNHALTED.CORE

Partial flag stalls have high penalty and they can be easily avoided. However, in some cases, Partial Flag Stalls Ratio might be high although there are no real flag stalls. There are a few instructions that partially modify the RFLAGS register and may cause partial flag stalls. The most popular are the shift instructions (SAR, SAL, SHR, and SHL) and the INC and DEC instructions.

B.7.4.5 Bypass Between Execution Domains

23. Delayed Bypass to FP Operation Rate: DELAYED_BYPASS.FP / CPU_CLK_UNHALTED.CORE

24. Delayed Bypass to SIMD Operation Rate: DELAYED_BYPASS.SIMD / CPU_CLK_UNHALTED.CORE

25. Delayed Bypass to Load Operation Rate: DELAYED_BYPASS.LOAD / CPU_CLK_UNHALTED.CORE

Domain bypass adds one cycle to instruction latency. To identify frequent domain bypasses in the code you can use the above ratios.

B.7.4.6 Floating-Point Performance Ratios

26. Floating-Point Instructions Ratio: X87_OPS_RETIRED.ANY / INST_RETIRED.ANY * 100

Significant floating-point activity indicates that specialized optimizations for floating-point algorithms may be applicable.

27. FP Assist Performance Impact: FP_ASSIST * 80 / CPU_CLK_UNHALTED.CORE * 100

Floating-Point assist is activated for non-regular FP values like denormals and NANs. FP assist is extremely slow compared to regular FP execution. Different assists incur different penalties. FP Assist Performance Impact estimates the overall impact.

28. Divider Busy: IDLE_DURING_DIV / CPU_CLK_UNHALTED.CORE * 100

A high value for the Divider Busy ratio indicates that the divider is busy and no other execution unit or load operation is in progress for many cycles. Using this ratio ignores L1 data cache misses and L2 cache misses that can be executed in parallel and hide the divider penalty.

29. Floating-Point Control Word Stall Ratio: RESOURCE_STALLS.FPCW / CPU_CLK_UNHALTED.CORE * 100

Frequent modifications to the Floating-Point Control Word (FPCW) might significantly decrease performance. The main reason for changing FPCW is for changing rounding mode when doing FP to integer conversions.

B.7.5 Memory Sub-System - Access Conflicts Ratios

A high value for Load or Store Buffer Full Ratio (Ratio 4) indicates that the load buffer or store buffer are frequently full, hence new micro-ops cannot enter the execution pipeline. This can reduce execution parallelism and decrease performance.

30. Load Rate: L1D_CACHE_LD.MESI / CPU_CLK_UNHALTED.CORE

One memory read operation can be served by a core each cycle. A high “Load Rate” indicates that execution may be bound by memory read operations.

31. Store Order Block: STORE_BLOCK.ORDER / CPU_CLK_UNHALTED.CORE * 100

Store Order Block ratio is the percentage of cycles that store operations, which miss the L2 cache, block committing data of later stores to the memory sub-system. This behavior can further cause the store buffer to fill up (see Ratio 4).

B.7.5.1 Loads Blocked by the L1 Data Cache

32. Loads Blocked by L1 Data Cache Rate: LOAD_BLOCK.L1D/CPU_CLK_UNHALTED.CORE

A high value for “Loads Blocked by L1 Data Cache Rate” indicates that load operations are blocked by the L1 data cache due to lack of resources, usually happening as a result of many simultaneous L1 data cache misses.

B.7.5.2 4K Aliasing and Store Forwarding Block Detection

33. Loads Blocked by Overlapping Store Rate: LOAD_BLOCK.OVERLAP_STORE/CPU_CLK_UNHALTED.CORE

4K aliasing and store forwarding block are two different scenarios in which loads are blocked by preceding stores due to different reasons. Both scenarios are detected by the same event: LOAD_BLOCK.OVERLAP_STORE. A high value for “Loads Blocked by Overlapping Store Rate” indicates that either 4K aliasing or store forwarding block may affect performance.

B.7.5.3 Load Block by Preceding Stores

34. Loads Blocked by Unknown Store Address Rate: LOAD_BLOCK.STA / CPU_CLK_UNHALTED.CORE

A high value for “Loads Blocked by Unknown Store Address Rate” indicates that loads are frequently blocked by preceding stores with unknown address and implies performance penalty.

35. Loads Blocked by Unknown Store Data Rate: LOAD_BLOCK.STD / CPU_CLK_UNHALTED.CORE

A high value for "Loads Blocked by Unknown Store Data Rate" indicates that loads are frequently blocked by preceding stores with unknown data and implies performance penalty.

B.7.5.4 Memory Disambiguation

The memory disambiguation feature of Intel Core microarchitecture eliminates most of the non-required load blocks by stores with unknown address. When this feature fails (possibly due to flaky load - store disambiguation cases) the event LOAD_BLOCK.STA will be counted and also MEMORY_DISAMBIGUATION.RESET.

B.7.5.5 Load Operation Address Translation

36. L0 DTLB Miss due to Loads - Performance Impact: DTLB_MISSES.L0_MISS_LD * 2 / CPU_CLK_UNHALTED.CORE

High number of DTLB0 misses indicates that the data set that the workload uses spans a number of pages that is bigger than the DTLB0. The high number of misses is expected to impact workload performance only if the CPI (Ratio 1) is low - around 0.8. Otherwise, it is likely that the DTLB0 miss cycles are hidden by other latencies.

B.7.6 Memory Sub-System - Cache Misses Ratios

B.7.6.1 Locating Cache Misses in the Code

Intel Core microarchitecture provides you with precise events for retired load instructions that miss the L1 data cache or the L2 cache. As precise events they provide the instruction pointer of the instruction following the one that caused the event. Therefore the instruction that comes immediately prior to the pointed instruction is the one that causes the cache miss. These events are most helpful to quickly identify on which loads to focus to fix a performance problem. The events are:

MEM_LOAD_RETIRE.L1D_MISS

MEM_LOAD_RETIRE.L1D_LINE_MISS

MEM_LOAD_RETIRE.L2_MISS

MEM_LOAD_RETIRE.L2_LINE_MISS

B.7.6.2 L1 Data Cache Misses

37. L1 Data Cache Miss Rate: L1D_REPL / INST_RETIRED.ANY

A high value for L1 Data Cache Miss Rate indicates that the code misses the L1 data cache too often and pays the penalty of accessing the L2 cache. See also Loads Blocked by L1 Data Cache Rate (Ratio 32).

You can count separately cache misses due to loads, stores, and locked operations using the events L1D_CACHE_LD.I_STATE, L1D_CACHE_ST.I_STATE, and L1D_CACHE_LOCK.I_STATE, accordingly.

B.7.6.3 L2 Cache Misses

38. L2 Cache Miss Rate: L2_LINES_IN.SELF.ANY / INST_RETIRED.ANY

A high L2 Cache Miss Rate indicates that the running workload has a data set larger than the L2 cache. Some of the data might be evicted without being used. Unless all the required data is brought ahead of time by the hardware prefetcher or software prefetching instructions, bringing data from memory has a significant impact on the performance.

39. L2 Cache Demand Miss Rate: L2_LINES_IN.SELF.DEMAND / INST_RETIRED.ANY

A high value for L2 Cache Demand Miss Rate indicates that the hardware prefetchers are not exploited to bring the data this workload needs. Data is brought from memory when needed to be used and the workload bears memory latency for each such access.

B.7.7 Memory Sub-system - Prefetching

B.7.7.1 L1 Data Prefetching

The event L1D_PREFETCH.REQUESTS is counted whenever the DCU attempts to prefetch cache lines from the L2 (or memory) to the DCU. If you expect the DCU prefetchers to work and to count this event, but instead you detect the event MEM_LOAD_RETIRE.L1D_MISS, it might be that the IP prefetcher suffers from load instruction address collision of several loads.

B.7.7.2 L2 Hardware Prefetching

With the event L2_LD.SELF.PREFETCH.MESI you can count the number of prefetch requests that were made to the L2 by the L2 hardware prefetchers. The actual number of cache lines prefetched to the L2 is counted by the event L2_LD.SELF.PREFETCH.I_STATE.

B.7.7.3 Software Prefetching

The events for software prefetching cover each level of prefetching separately.

40. Useful PrefetchT0 Ratio: SSE_PRE_MISS.L1 / SSE_PRE_EXEC.L1 * 100

41. Useful PrefetchT1 and PrefetchT2 Ratio: SSE_PRE_MISS.L2 / SSE_PRE_EXEC.L2 * 100

A low value for any of the prefetch usefulness ratios indicates that some of the SSE prefetch instructions prefetch data that is already in the caches.

42. Late PrefetchT0 Ratio: LOAD_HIT_PRE / SSE_PRE_EXEC.L1

43. Late PrefetchT1 and PrefetchT2 Ratio: LOAD_HIT_PRE / SSE_PRE_EXEC.L2

A high value for any of the late prefetch ratios indicates that software prefetch instructions are issued too late and the load operations that use the prefetched data are waiting for the cache line to arrive.

B.7.8 Memory Sub-system - TLB Miss Ratios

44. TLB miss penalty: PAGE_WALKS.CYCLES / CPU_CLK_UNHALTED.CORE * 100

A high value for the TLB miss penalty ratio indicates that many cycles are spent on TLB misses. Reducing the number of TLB misses may improve performance. This ratio does not include DTLB0 miss penalties (see Ratio 37).

The following ratios help to focus on the kind of memory accesses that cause TLB misses most frequently See “ITLB Miss Rate” (Ratio 6) for TLB misses due to instruction fetch.

45. DTLB Miss Rate: DTLB_MISSES.ANY / INST_RETIRED.ANY

A high value for DTLB Miss Rate indicates that the code accesses too many data pages within a short time, and causes many Data TLB misses.

46. DTLB Miss Rate due to Loads: DTLB_MISSES.MISS_LD / INST_RETIRED.ANY

A high value for DTLB Miss Rate due to Loads indicates that the code accesses loads data from too many pages within a short time, and causes many Data TLB misses. DTLB misses due to load operations may have a significant impact, since the DTLB miss increases the load operation latency. This ratio does not include DTLB0 miss penalties (see Ratio 37).

To precisely locate load instructions that caused DTLB misses you can use the precise event MEM_LOAD_RETIRE.DTLB_MISS.

47. DTLB Miss Rate due to Stores: DTLB_MISSES.MISS_ST / INST_RETIRED.ANY

A high value for DTLB Miss Rate due to Stores indicates that the code accesses too many data pages within a short time, and causes many Data TLB misses due to store operations. These misses can impact performance if they do not occur in parallel to other instructions. In addition, if there are many stores in a row, some of them missing the DTLB, it may cause stalls due to full store buffer.

B.7.9 Memory Sub-system - Core Interaction

B.7.9.1 Modified Data Sharing

48. Modified Data Sharing Ratio: EXT_SNOOP.ALL_AGENTS.HITM / INST_RETIRED.ANY

Frequent occurrences of modified data sharing may be due to two threads using and modifying data laid in one cache line. Modified data sharing causes L2 cache misses. When it happens unintentionally (aka false sharing) it usually causes demand misses that have high penalty. When false sharing is removed code performance can dramatically improve.

49. Local Modified Data Sharing Ratio: EXT_SNOOP.THIS_AGENT.HITM / INST_RETIRED.ANY

Modified Data Sharing Ratio indicates the amount of total modified data sharing observed in the system. For systems with several processors you can use Local Modified Data Sharing Ratio to indicates the amount of modified data sharing between two cores in the same processor. (In systems with one processor the two ratios are similar).

B.7.9.2 Fast Synchronization Penalty

50. Locked Operations Impact: (L1D_CACHE_LOCK_DURATION + 20 * L1D_CACHE_LOCK.MESI) / CPU_CLK_UNHALTED.CORE * 100

Fast synchronization is frequently implemented using locked memory accesses. A high value for Locked Operations Impact indicates that locked operations used in the workload have high penalty. The latency of a locked operation depends on the location of the data: L1 data cache, L2 cache, other core cache or memory.

B.7.9.3 Simultaneous Extensive Stores and Load Misses

51. Store Block by Snoop Ratio: (STORE_BLOCK.SNOOP / CPU_CLK_UNHALTED.CORE) * 100

A high value for “Store Block by Snoop Ratio” indicates that store operations are frequently blocked and performance is reduced. This happens when one core executes a dense stream of stores while the other core in the processor frequently snoops it for cache lines missing in its L1 data cache.

B.7.10 Memory Sub-system - Bus Characterization

B.7.10.1 Bus Utilization

52. Bus Utilization: BUS_TRANS_ANY.ALL_AGENTS * 2 / CPU_CLK_UNHALTED.BUS * 100

Bus Utilization is the percentage of bus cycles used for transferring bus transactions of any type. In single processor systems most of the bus transactions carry data. In multiprocessor systems some of the bus transactions are used to coordinate cache states to keep data coherency.

53. Data Bus Utilization: BUS_DRDY_CLOCKS.ALL_AGENTS / CPU_CLK_UNHALTED.BUS * 100

Data Bus Utilization is the percentage of bus cycles used for transferring data among all bus agents in the system, including processors and memory. High bus utilization indicates heavy traffic between the processor(s) and memory. Memory sub-system latency can impact the performance of the program. For compute-intensive applications with high bus utilization, look for opportunities to improve data and code

locality. For other types of applications (for example, copying large amounts of data from one memory area to another), try to maximize bus utilization.

54. Bus Not Ready Ratio: BUS_BNR_DRV.ALL_AGENTS * 2 / CPU_CLK_UNHALTED.BUS * 100

Bus Not Ready Ratio estimates the percentage of bus cycles during which new bus transactions cannot start. A high value for Bus Not Ready Ratio indicates that the bus is highly loaded. As a result of the Bus Not Ready (BNR) signal, new bus transactions might defer and their latency will have higher impact on program performance.

55. Burst Read in Bus Utilization: BUS_TRANS_BRD.SELF * 2 / CPU_CLK_UNHALTED.BUS * 100

A high value for Burst Read in Bus Utilization indicates that bus and memory latency of burst read operations may impact the performance of the program.

56. RFO in Bus Utilization: BUS_TRANS_RFO.SELF * 2 / CPU_CLK_UNHALTED.BUS * 100

A high value for RFO in Bus Utilization indicates that latency of Read For Ownership (RFO) transactions may impact the performance of the program. RFO transactions may have a higher impact on the program performance compared to other burst read operations (for example, as a result of loads that missed the L2). See also Ratio 31.

B.7.10.2 Modified Cache Lines Eviction

57. L2 Modified Lines Eviction Rate: L2_M_LINES_OUT.SELF.ANY / INST_RETIRED.ANY

When a new cache line is brought from memory, an existing cache line, possibly modified, is evicted from the L2 cache to make space for the new line. Frequent evictions of modified lines from the L2 cache increase the latency of the L2 cache misses and consume bus bandwidth.

58. Explicit WB in Bus Utilization: BUS_TRANS_WB.SELF * 2 / CPU_CLK_UNHALTED.BUS*100

Explicit Write-back in Bus Utilization considers modified cache line evictions not only from the L2 cache but also from the L1 data cache. It represents the percentage of bus cycles used for explicit write-backs from the processor to memory.

APPENDIX C INSTRUCTION LATENCY AND THROUGHPUT

This appendix contains tables showing the latency and throughput are associated with commonly used instructions[1]. The instruction timing data varies across processors family/models. It contains the following sections:

- **Appendix C.1, "Overview"** — Provides an overview of issues related to instruction selection and scheduling.
- **Appendix C.2, "Definitions"** — Presents definitions.
- **Appendix C.3, "Latency and Throughput"** — Lists instruction throughput, latency associated with commonly-used instruction.

C.1 OVERVIEW

This appendix provides information to assembly language programmers and compiler writers. The information aids in the selection of instruction sequences (to minimize chain latency) and in the arrangement of instructions (assists in hardware processing). The performance impact of applying the information has been shown to be on the order of several percent. This is for applications not dominated by other performance factors, such as:

- Cache miss latencies.
- Bus bandwidth.
- I/O bandwidth.

Instruction selection and scheduling matters when the programmer has already addressed the performance issues discussed in Chapter 2:

- Observe store forwarding restrictions.
- Avoid cache line and memory order buffer splits.
- Do not inhibit branch prediction.
- Minimize the use of `xchg` instructions on memory locations.

While several items on the above list involve selecting the right instruction, this appendix focuses on the following issues. These are listed in priority order, though which item contributes most to performance varies by application:

- Maximize the flow of µops into the execution core. Instructions which consist of more than four µops require additional steps from microcode ROM. Instructions with longer micro-op flows incur a delay in the front end and reduce the supply of micro-ops to the execution core.

  In Pentium 4 and Intel Xeon processors, transfers to microcode ROM often reduce how efficiently µops can be packed into the trace cache. Where possible, it is advisable to select instructions with four or fewer µops. For example, a 32-bit integer multiply with a memory operand fits in the trace cache without going to microcode, while a 16-bit integer multiply to memory does not.
- Avoid resource conflicts. Interleaving instructions so that they don't compete for the same port or execution unit can increase throughput. For example, alternate PADDQ and PMULUDQ (each has a throughput of one issue per two clock cycles). When interleaved, they can achieve an effective throughput of one instruction per cycle because they use the same port but different execution units.

---

1. Although instruction latency may be useful in some limited situations (e.g., a tight loop with a dependency chain that exposes instruction latency), software optimization on super-scalar, out-of-order microarchitecture, in general, will benefit much more on increasing the effective throughput of the larger-scale code path. Coding techniques that rely on instruction latency alone to influence the scheduling of instruction is likely to be sub-optimal as such coding technique is likely to interfere with the out-of-order machine or restrict the amount of instruction-level parallelism.

Selecting instructions with fast throughput also helps to preserve issue port bandwidth, hide latency and allows for higher software performance.

- Minimize the latency of dependency chains that are on the critical path. For example, an operation to shift left by two bits executes faster when encoded as two adds than when it is encoded as a shift. If latency is not an issue, the shift results in a denser byte encoding.

In addition to the general and specific rules, coding guidelines and the instruction data provided in this manual, you can take advantage of the software performance analysis and tuning toolset available at http://developer.intel.com/software/products/index.htm. The tools include the Intel VTune Performance Analyzer, with its performance-monitoring capabilities.

C.2 DEFINITIONS

The data is listed in several tables. The tables contain the following:

- **Instruction Name** — The assembly mnemonic of each instruction.
- **Latency** — The number of clock cycles that are required for the execution core to complete the execution of all of the μops that form an instruction.
- **Throughput** — The number of clock cycles required to wait before the issue ports are free to accept the same instruction again. For many instructions, the throughput of an instruction can be significantly less than its latency.
- The case of RDRAND instruction latency and throughput is an exception to the definitions above, because the hardware facility that executes the RDRAND instruction resides in the uncore and is shared by all processor cores and logical processors in a physical package. The software observable latency and throughput using the sequence of "rdrand followby jnc" in a single-thread scenario can be as low as ~100 cycles. In a third-generation Intel Core processor based on Intel microarchitecture code name Ivy Bridge, the total bandwidth to deliver random numbers via RDRAND by the uncore is about 500 MBytes/sec. Within the same processor core microarchitecture and different uncore implementations, RDRAND latency/throughput can vary across Intel Core and Intel Xeon processors.

C.3 LATENCY AND THROUGHPUT

This section presents the latency and throughput information for commonly-used instructions including: MMX technology, Streaming SIMD Extensions, subsequent generations of SIMD instruction extensions, and most of the frequently used general-purpose integer and x87 floating-point instructions.

Due to the complexity of dynamic execution and out-of-order nature of the execution core, the instruction latency data may not be sufficient to accurately predict realistic performance of actual code sequences based on adding instruction latency data.

- Instruction latency data is useful when tuning a dependency chain. However, dependency chains limit the out-of-order core's ability to execute micro-ops in parallel. Instruction throughput data are useful when tuning parallel code unencumbered by dependency chains.
- Numeric data in the tables is:
  - — Approximate and subject to change in future implementations of the microarchitecture.
  - — Not meant to be used as reference for instruction-level performance benchmarks. Comparison of instruction-level performance of microprocessors that are based on different microarchitectures is a complex subject and requires information that is beyond the scope of this manual.

Comparisons of latency and throughput data between different microarchitectures can be misleading.

Appendix C.3.1 provides latency and throughput data for the register-to-register instruction type. Appendix C.3.3 discusses how to adjust latency and throughput specifications for the register-to-memory and memory-to-register instructions.

In some cases, the latency or throughput figures given are just one half of a clock. This occurs only for the double-speed ALUs.

C.3.1 Latency and Throughput with Register Operands

Instruction latency and throughput data are presented in Table C-4 through Table C-18. Tables include AESNI, SSE4.2, SSE4.1, Supplemental Streaming SIMD Extension 3, Streaming SIMD Extension 3, Streaming SIMD Extension 2, Streaming SIMD Extension, MMX technology and most common Intel 64 and IA-32 instructions. Instruction latency and throughput for different processor microarchitectures are in separate columns.

Processor instruction timing data is implementation specific; it can vary between model encodings within the same family encoding (e.g. model = 3 vs model < 2). Separate sets of instruction latency and throughput are shown in the columns for CPUID signature 0xF2n and 0xF3n. The column represented by 0xF3n also applies to Intel processors with CPUID signature 0xF4n and 0xF6n. The notation 0xF2n represents the hex value of the lower 12 bits of the EAX register reported by CPUID instruction with input value of EAX = 1; ‘F’ indicates the family encoding value is 15, ‘2’ indicates the model encoding is 2, ‘n’ indicates it applies to any value in the stepping encoding.

Intel Core Solo and Intel Core Duo processors are represented by 06_0EH. Processors bases on 65 nm Intel Core microarchitecture are represented by 06_0FH. Processors based on Enhanced Intel Core microarchitecture are represented by 06_17H and 06_1DH. CPUID family/Model signatures of processors based on Intel microarchitecture code name Nehalem are represented by 06_1AH, 06_1EH, 06_1FH, and 06_2EH. Intel microarchitecture code name Westmere are represented by 06_25H, 06_2CH and 06_2FH. Intel microarchitecture code name Sandy Bridge are represented by 06_2AH, 06_2DH. Intel microarchitecture code name Ivy Bridge are represented by 06_3AH, 06_3EH. Intel microarchitecture code name Haswell are represented by 06_3CH, 06_45H and 06_46H.

Table C-1. CPUID Signature Values of Of Recent Intel Microarchitectures

DisplayFamily_DisplayModel	Recent Intel Microarchitectures
06_4EH, 06_5EH	Skylake microarchitecture
06_3DH, 06_47H, 06_56H	Broadwell microarchitecture
06_3CH, 06_45H, 06_46H, 06_3FH	Haswell microarchitecture
06_3AH, 06_3EH	Ivy Bridge microarchitecture
06_2AH, 06_2DH	Sandy Bridge microarchitecture
06_25H, 06_2CH, 06_2FH	Intel microarchitecture Westmere
06_1AH, 06_1EH, 06_1FH, 06_2EH	Intel microarchitecture Nehalem
06_17H, 06_1DH	Enhanced Intel Core microarchitecture
06_0FH	Intel Core microarchitecture

Instruction latency varies by microarchitectures. Table C-2 lists SIMD extensions introduction in recent microarchitectures. Each microarchitecture may be associated with more than one signature value given by the CPUID’s “display_family” and “display_model”. Not all instruction set extensions are enabled in all processors associated with a particular family/model designation. To determine whether a given instruction set extension is supported, software must use the appropriate CPUID feature flag as described in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

.

Table C-2. Instruction Extensions Introduction by Microarchitectures (CPUID Signature)

SIMD Instruction Extensions	DisplayFamily_DisplayModel							
	06_4EH, 06_5EH	06_3DH, 06_47H, 06_56H	06_3CH, 06_45H, 06_46H, 06_3FH	06_3AH, 06_3EH	06_2AH, 06_2DH	06_25H, 06_2CH, 06_2FH	06_1AH, 06_1EH, 06_1FH, 06_2EH	06_17H, 06_1DH
CLFLUSHOPT	Yes	No	No	No	No	No	No	No
ADX, RDSEED	Yes	Yes	No	No	No	No	No	No
AVX2, FMA, BMI1, BMI2	Yes	Yes	Yes	No	No	No	No	No
F16C, RDRAND, RWFSGSBASE	Yes	Yes	Yes	Yes	No	No	No	No
AVX	Yes	Yes	Yes	Yes	Yes	No	No	No
AESNI, PCLMULQDQ	Yes	Yes	Yes	Yes	Yes	Yes	No	No
SSE4.2, POPCNT	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
SSE4.1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SSSE3	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SSE3	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SSE2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SSE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
MMX	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table C-3. BMI1, BMI2 and General Purpose Instructions

Instruction	Latency [1]		Throughput	
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_4E, 06_5E	06_3D, 06_47, 06_56
ADCX	1	1	1	1
ADOX	1	1	1	1
RESEED	Similar to RDRAND	Similar to RDRAND	Similar to RDRAND	Similar to RDRAND

Table C-4. 256-bit AVX2 Instructions

Instruction	Latency [1]			Throughput		
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F
VEXTRACTI128 xmm1, ymm2, imm	1	1	1	1	1	1
VMPSADBW	4	6	6	2	2	2
VPACKUSDW/SSWB	1	1	1	1	1	1
VPADDB/D/W/Q	1	1	1	0.33	0.5	0.5
VPADDSB	1	1	1	0.5	0.5	0.5

Table C-4. 256-bit AVX2 Instructions (Contd.)

Instruction	Latency [1]			Throughput		
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F
VPADDUSB	1	1	1	0.5	0.5	0.5
VPALIGNR	1	1	1	1	1	1
VPAVGB	1	1	1	0.5	0.5	0.5
VPBLENDD	1	1	1	0.33	0.33	0.33
VPBLENDW	1	1	1	1	1	1
VPBLENDVB	1	2	2	1	2	2
VPBROADCASTB/D/SS/SD	3	3	3	1	1	1
VPCMPEQB/W/D	1	1	1	0.5	0.5	0.5
VPCMPEQQ	1	1	1	0.5	0.5	0.5
VPCMPGTQ	3	5	5	1	1	1
VPHADDW/D/SW	3	3	3	2	2	2
VINSERTI128 ymm1, ymm2, xmm, imm	3	3	3	1	1	1
VPMADDWD	5[b]	5	5	0.5	1	1
VPMADDUBSW	5[b]	5	5	0.5	1	1
VPMAXSD	1	1	1	0.5	0.5	0.5
VPMAXUD	1	1	1	0.5	0.5	0.5
VPMOVSX	3	3	3	1	1	1
VPMOVZX	3	3	3	1	1	1
VPMULDQ/UDQ	5[b]	5	5	0.5	1	1
VPMULHRSW	5[b]	5	5	0.5	1	1
VPMULHW/LW	5[b]	5	5	0.5	1	1
VPMULLD	10[b]	10	10	1	2	2
VPOR/VPXOR	1	1	1	0.33	0.33	0.33
VPSADBW	3	5	5	1	1	1
VPSHUFB	1	1	1	1	1	1
VPSHUFD	1	1	1	1	1	1
VPSHUFLW/HW	1	1	1	1	1	1
VPSIGNB/D/W/Q	1	1	1	0.5	0.5	0.5
VPERMD/PS	3	3	3	1	1	1
VPSLLVD/Q	2	2	2	0.5	2	2
VPSRAVD	2	2	2	0.5	2	2
VPSRAD/W ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLDQ ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLQ/D/W ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLQ/D/W ymm, ymm, ymm	4	4	4	1	1	1

Table C-4. 256-bit AVX2 Instructions (Contd.)

Instruction	Latency [1]			Throughput		
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F
VPUNPCKHBW/WD/DQ/QDQ	1	1	1	1	1	1
VPUNPCKLBW/WD/DQ/QDQ	1	1	1	1	1	1
ALL VFMA	4	5	5	0.5	0.5	0.5
VPMASKMOVD/Q mem, ymm[d], ymm				1	2	2
VPMASKMOVD/Q NUL, msk_0, ymm				>200[e]	2	2
VPMASKMOVD/Q ymm, ymm[d], mem	11	8	8	1	2	2
VPMASKMOVD/Q ymm, msk_0, [base+index][f]	>200	~200	~200	>200	~200	~200

b: includes 1-cycle bubble due to bypass.
c: includes two 1-cycle bubbles due to bypass
d: MASKMOV instruction timing measured with L1 reference and mask register selecting at least 1 or more elements.
e: MASKMOV store instruction with a mask value selecting 0 elements and illegal address (NUL or non-NUL) incurs delay due to assist.
f: MASKMOV Load instruction with a mask value selecting 0 elements and certain addressing forms incur delay due to assist.

Table C-5. Gather Timing Data from L1D*

Instruction	Latency [1]			Throughput		
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45/46/3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45/46/3F
VPGATHERDD/PS xmm, [vi128], xmm	~20	~17	~14	~4	~5	~7
VPGATHERQQ/PD xmm, [vi128], xmm	~18	~15	~12	~3	~4	~5
VPGATHERDD/PS ymm, [vi256], ymm	~22	~19	~20	~5	~6	~10
VPGATHERQQ/PD ymm, [vi256], ymm	~20	~16	~15	~4	~5	~7

* Gather Instructions fetch data elements via memory references. The timing data shown applies to memory references that reside within the L1 data cache and all mask elements selected

Table C-6. BMI1, BMI2 and General Purpose Instructions

Instruction	Latency [1]			Throughput		
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45/46/3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45/46/3F
ANDN	1	1	1	0.5	0.5	0.5
BEXTR	2	2	2	0.5	0.5	0.5
BLSI/BLSMSK/BLSR	1	1	1	0.5	0.5	0.5
BZHI	1	1	1	0.5	0.5	0.5
MULX r64, r64, r64	4	4	4	1	1	1

Table C-6. BMI1, BMI2 and General Purpose Instructions

Instruction	Latency [1]			Throughput		
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45 /46/3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45 /46/3F
PDEP/PEXT r64, r64, r64	3	3	3	1	1	1
RORX r64, r64, r64	1	1	1	0.5	0.5	0.5
SALX/SARX/SHLX r64, r64, r64	1	1	1	0.5	0.5	0.5
LZCNT/TZCNT	3	3	3	1	1	1

Table C-7. F16C,RDRAND Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/ 45/46/ 3F	06_3A/ 3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/ 45/46/ 3F	06_3A/ 3E
RDRAND* r64	Varies	Varies	Varies	<200	<300	~250	~250	<200
VCVTPH2PS ymm1, xmm2	7	6	6	7	1	1	1	1
VCVTPH2PS xmm1, xmm2	5	4	4	6	1	1	1	1
VCVTPS2PH ymm1, xmm2, imm	7	6	6	10	1	1	1	1
VCVTPS2PH xmm1, xmm2, imm	5	4	4	9	1	1	1	1

* See Section C.2

Table C-8. 256-bit AVX Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E
VADDPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VADDSUBPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VANDNPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VANDPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VBLENDPD/PS ymm1, ymm2, ymm3, imm	1	1	1	1	0.33	0.33	0.33	0.5
VBLENDVPD/PS ymm1, ymm2, ymm3, ymm	1	2	2	1	1	2	2	1
VCMPPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VCVTDQ2PD ymm1, ymm2	7	6	6	4	1	1	1	1
VCVTDQ2PS ymm1, ymm2	4	3	3	3	0.5	1	1	1
VCVT(T)PD2DQ ymm1, ymm2	7	6	6	4	1	1	1	1
VCVTPD2PS ymm1, ymm2	7	6	6	4	1	1	1	1
VCVT(T)PS2DQ ymm1, ymm2	4	3	3	3	1	1	1	1
VCVTPS2PD ymm1, xmm2	7	4	4	2	1	1	1	1

Table C-8. 256-bit AVX Instructions (Contd.)

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E
VDIVPD ymm1, ymm2, ymm3	14	16-23	25-35	27-35	8	16	27	28
VDIVPS ymm1, ymm2, ymm3	11	13-17	17-21	18-21	5	10	13	14
VDPPS ymm1, ymm2, ymm3	13	12	14	12	1.5	2	2	2
VEXTRACTF128 xmm1, ymm2, imm	3	3	3	3	1	1	1	1
VINSERTF128 ymm1, xmm2, imm	3	3	3	3	1	1	1	1
VMAXPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VMINPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VMOVAPD/PS ymm1, ymm2	1	1	1	1	0.25	0.5	0.5	1
VMOVDDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVDQA/U ymm1, ymm2	1	1	1	1	0.25	0.25	0.25	0.5
VMOVMSKPD/PS ymm1, ymm2	2	2	2	1	1	1	1	1
VMOVQ xmm1, xmm2	1	1	1	1	0.33	0.33	0.33	0.33
VMOVD/Q xmm1, r32/r64	2	1	1	1	1	1	1	1
VMOVD/Q r32/r64, xmm	2	1	1	1	1	1	1	1
VMOVNTDQ/PS/PD					1	1	1	1
VMOVSHDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVSLDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVUPD/PS ymm1, ymm2	1	1	1	1	0.25	0.5	0.5	1
VMULPD/PS ymm1, ymm2, ymm3	4	3	5	5	0.5	0.5	0.5	1
VORPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VPERM2F128 ymm1, ymm2, ymm3, imm	3	3	3	2	1	1	1	1
VPERMILPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VRCPPS ymm1, ymm2	4	7	7	7	1	2	2	2
VROUNDPD/PS ymm1, ymm2, imm	8	6	6	3	1	2	2	1
VRSQRTPS ymm1, ymm2	4	7	7	7	1	2	2	2
VSHUFPD/PS ymm1, ymm2, ymm3, imm	1	1	1	1	1	1	1	1
VSQRTPD ymm1, ymm2	<18	19-35	19-35	19-35	<12	16-27	16-27	28
VSQRTPS ymm1, ymm2	12	18-21	18-21	18-21	<6	13	13	14
VSUBPD/PS ymm1, ymm2, imm	4	3	3	3	0.5	1	1	1
VTESTPS ymm1, ymm2	3	2	2	2	1	1	1	1
VUNPCKHPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VUNPCKLPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VXORPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VZEROUPPER	0	0	0	0	1	1	1	1
VZEROALL					12	8	8	9

Table C-8. 256-bit AVX Instructions (Contd.)

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E
VEXTRACTPS reg, xmm2, imm	3	2	2	2	1	1	1	1
VINSERTPS xmm1, xmm2, reg, imm	1	1	1	1	1	1	1	1
VMASKMOVPD/PS mem[a], ymm, ymm					1	2	2	2
VMASKMOVPD/PS NUL, msk_0, ymm					>200[b]	2	2	2
VMASKMOVPD/PS ymm, ymm[a], mem	11	8	8	9	1	2	2	2
VMASKMOVPD/PS ymm, msk_0, [base+index][c]	>200	~200	~200	~200	>200	~200	~200	~200

Latency and Throughput data for CPUID signature 06_3AH are generally the same as those of 06_2AH, only those that differ from 06_2AH are shown in the 06_3AH column.
a: MASKMOV instruction timing measured with L1 reference and mask register selecting at least 1 or more elements.
b: MASKMOV store instruction with a mask value selecting 0 elements and illegal address (NUL or non-NUL) incurs delay due to assist.
c: MASKMOV Load instruction with a mask value selecting 0 elements and certain addressing forms incur delay due to assist.

Latency of VEX.128 encoded AVX instructions should refer to corresponding legacy 128-bit instructions.

Table C-9. AESNI and PCLMULQDQ Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E
AESDEC/AESDECLAST xmm1, xmm2	4	7	7	8	1	1	1	1
AESENC/AESENCLAST xmm1, xmm2	4	7	7	8	1	1	1	1
AESIMC xmm1, xmm2	8	14	14	14	2	2	2	2
AESKEYGENASSIST xmm1, xmm2, imm	12	10	10	10	12	8	8	8
PCLMULQDQ xmm1, xmm2, imm	7[b]	5	7	14	1	1	2	8

b: includes 1-cycle bubble due to bypass.

Table C-10. SSE4.2 Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D /47/56	06_3C /45/46 /3F	06_3A /3E/2A /2D	06_4E, 06_5E	06_3D /47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D
CRC32 r32, r32	3	3	3	3	1	1	1	1
PCMPESTRI xmm1, xmm2, imm	15	10	10	11	5	4	4	4
PCMPESTRM xmm1, xmm2, imm	10	10	10	11	6	5	5	4
PCMPISTRI xmm1, xmm2, imm	15	10	10	11	3	3	3	3
PCMPISTRM xmm1, xmm2, imm	15	11	11	11	3	3	3	3
PCMPGTQ xmm1, xmm2	3	5	5	5	0.33	1	1	1

Table C-10. SSE4.2 Instructions (Contd.)

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D /47/56	06_3C /45/46 /3F	06_3A /3E/2A /2D	06_4E, 06_5E	06_3D /47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D
POPCNT r32, r32	3	3	3	3	1	1	1	1
POPCNT r64, r64	3	3	3	3	1	1	1	1

Table C-11. SSE4.1 Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D /47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D	06_4E, 06_5E	06_3D/ 47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D
BLENDPD/S xmm1, xmm2, imm	1	1	1	1	0.33	0.33	0.33	0.5
BLENDVPD/S xmm1, xmm2	1	2	2	2	1	2	2	1
DPPD xmm1, xmm2	9	7	9	9	1	1	1	1
DPPS xmm1, xmm2	13	12	14	13	2	2	2	2
EXTRACTPS xmm1, xmm2, imm	3	2	2	2	1	1	1	1
INSERTPS xmm1, xmm2, imm	1	1	1	1	1	1	1	1
MPSADBW xmm1, xmm2, imm	4	6	6	6	2	2	2	1
PACKUSDW xmm1, xmm2	1	1	1	1	1	1	1	0.5
PBLENVB xmm1, xmm2	2	2	2	2	2	2	2	1
PBLENDW xmm1, xmm2, imm	1	1	1	1	1	1	1	0.5
PCMPEQQ xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PEXTRB/W/D reg, xmm1, imm	3	3	3	3	1	1	1	1
PHMINPOSUW xmm1,xmm2	4	5	5	5	1	1	1	1
PINSRB/W/D xmm1,reg, imm	2	2	2	2	1	1	1	1
PMAXSB/SD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMAXUW/UD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMINSB/SD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMINUW/UD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMOVSXBD/BW/BQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVSXWD/WQ/DQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVZXBD/BW/BQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVZXWD/WQ/DQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMULDQ xmm1, xmm2	5[b]	5	5	5	0.5	1	1	1
PMULLD xmm1, xmm2	10[c]	10	10	5	2	2	2	1
PTEST xmm1, xmm2	3	2	2	2	1	1	1	1

Table C-11. SSE4.1 Instructions (Contd.)

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D /47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D	06_4E, 06_5E	06_3D/ 47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D
ROUNDPD/PS xmm1, xmm2, imm	6	6	6	3	2	2	2	1
ROUNDSD/SS xmm1, xmm2, imm	6	6	6	3	2	2	2	1

b: includes 1-cycle bubble due to bypass
c: includes two 1-cycle bubbles due to bypass

Table C-12. Supplemental Streaming SIMD Extension 3 Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/ 47/56	06_3C/ 45/46/ 3F	06_3A/ 3E/2A/ 2D	06_4E, 06_5E	06_3D/ 47/56	06_3C/ 45/46/ 3F	06_3A/ 3E/2A/ 2D
PALIGNR xmm1, xmm2, imm	1	1	1	1	1	1	1	0.5
PHADDD xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHADDW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHADDSW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBD xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBSW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PMADDUBSW xmm1, xmm2	5[b]	5	5	5	0.5	1	1	1
PMULHRSW xmm1, xmm2	5[b]	5	5	5	0.5	1	1	1
PSHUFB xmm1, xmm2	1	1	1	1	1	1	1	0.5
PSIGNB/D/W xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PABSB/D/W xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5

b: includes 1-cycle bubble due to bypass

Table C-13. Streaming SIMD Extension 3 SIMD Floating-point Instructions

Instruction	Latency[1]				Throughput			
DisplayFamily_DisplayModel	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D
ADDSUBPD/ADDSUBPS	4	3	3	3	0.5	1	1	1
HADDPD xmm1, xmm2	6	5	5	5	2	2	2	2
HADDPS xmm1, xmm2	6	5	5	5	2	2	2	2
HSUBPD xmm1, xmm2	6	5	5	5	2	2	2	2
HSUBPS xmm1, xmm2	6	5	5	5	2	2	2	2
MOVDDUP xmm1, xmm2	1	1	1	1	1	1	1	1
MOVSHDUP xmm1, xmm2	1	1	1	1	1	1	1	1
MOVSLDUP xmm1, xmm2	1	1	1	1	1	1	1	1

Table C-13. Streaming SIMD Extension 3 SIMD Floating-point Instructions (Contd.)

Instruction	Latency[1]				Throughput			
DisplayFamily_DisplayModel	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D

Table C-14. Streaming SIMD Extension 2 128-bit Integer Instructions

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D
CVTPS2DQ xmm, xmm	3	3	3	3	1	1	1	1
CVTTPS2DQ xmm, xmm	3	3	3	3	1	1	1	1
MASKMOVDQU xmm, xmm					7	6	6	6
MOVD xmm, r64/r32	2	1	1	1	1	1	1	1
MOVD r64/r32, xmm	2	1	1	1	1	1	1	1
MOVDQA xmm, xmm	1	1	1	1	0.25	0.33	0.33	0.5
MOVDQU xmm, xmm	1	1	1	1	0.25	0.33	0.33	0.5
MOVQ xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PACKSSWB/PACKSSDW/ PACKUSWB xmm, xmm	1	1	1	1	1	1	1	0.5
PADDB/PADDW/PADDD xmm, xmm	1	1	1	1	0.33	0.5	0.5	0.5
PADDSB/PADDSW/ PADDUSB/PADDUSW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PADDQ/ PSUBQ[3] xmm, xmm	1	1	1	1	0.33	0.5	0.5	0.5
PAND xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PANDN xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PAVGB/PAVGW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PCMPEQB/PCMPEQD/ PCMPEQW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PCMPGTB/PCMPGTD/PCMP GTW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PEXTRW r32, xmm, imm8	3	3	3	3	1	1	1	1
PINSRW xmm, r32, imm8	2	2	2	2	2	2	2	1
PMADDWD xmm, xmm	5[b]	5	5	5	0.5	1	1	1
PMAX xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PMIN xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PMOVMSKB[3] r32, xmm	2	2	2	2	1	1	1	1
PMULHUW/PMULHW/ PMULLW xmm, xmm	5[b]	5	5	5	0.5	1	1	1
PMULUDQ xmm, xmm	5[b]	5	5	5	0.5	1	1	1
POR xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PSADBW xmm, xmm	3	5	5	5	1	1	1	1
PSHUFD xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSHUFHW xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSHUFLW xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSLLDQ xmm, imm8	1	1	1	1	1	1	1	0.5
PSLLW/PSLLD/PSLLQ xmm, imm8	1	1	1	1	1	1	1	1
PSLL/PSRL xmm, xmm	2	2	2	2	1	1	1	1
PSRAW/PSRAD xmm, imm8	1	1	1	1	1	1	1	1

Table C-14. Streaming SIMD Extension 2 128-bit Integer Instructions (Contd.)

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D
PSRAW/PSRAD xmm, xmm	2	2	2	2	1	1	1	1
PSRLDQ xmm, imm8	1	1	1	1	1	1	1	0.5
PSRLW/PSRLD/PSRLQ xmm, imm8	1	1	1	1	1	1	1	1
PSUBB/PSUBW/PSUBD xmm, xmm	1	1	1	1	0.33	0.5	0.5	0.5
PSUBSB/PSUBSW/PSUBUSB /PSUBUSW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PUNPCKHBW/PUNPCKHWD/ PUNPCKHDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKHQDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKLBW/PUNPCKLWD/ PUNPCKLDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKLQDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PXOR xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33

b: includes 1-cycle bubble due to bypass

Table C-15. Streaming SIMD Extension 2 Double-precision Floating-point Instructions

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)
ADDPD xmm, xmm	4	3	3	3	0.5	1	1	1
ADDSD xmm, xmm	4	3	3	3	0.5	1	1	1
ANDNPD xmm, xmm	1	1	1	1	0.33	1	1	1
ANDPD xmm, xmm	1	1	1	1	0.33	1	1	1
CMPPD xmm, xmm, imm8	4	3	3	3	0.5	1	1	1
CMPSD xmm, xmm, imm8	4	3	3	3	0.5	1	1	1
COMISD xmm, xmm	2	2	2	2	1	1	1	1
CVTDQ2PD xmm, xmm	5	4	4	4	1	1	1	1
CVTDQ2PS xmm, xmm	4	3	3	3	1	1	1	1
CVTPD2DQ xmm, xmm	5	4	4	4	1	1	1	1
CVTPD2PS xmm, xmm	5	4	4	4	1	1	1	1
CVT[T]PS2DQ xmm, xmm	4	3	3	3	1	1	1	1
CVTPS2PD xmm, xmm	5	2	2	2	1	1	1	1
CVT[T]SD2SI r64/r32, xmm	6	4	4	5	1	1	1	1
CVTSD2SS xmm, xmm	5	4	4	4	1	1	1	1
CVTSI2SD xmm, r64/r32	5	3	3	4	1	1	1	1
CVTSS2SD xmm, xmm	5	2	2	2	1	1	1	1
CVTTPD2DQ xmm, xmm	5	4	4	4	1	1	1	1

Table C-15. Streaming SIMD Extension 2 Double-precision Floating-point Instructions (Contd.)

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)
CVTTSD2SI r32, xmm	6	4	4	5	1	1	1	1
DIVPD xmm, xmm[1]	14	<14	14-20	16-22 (15-20)	4	8	13	22(14)
DIVSD xmm, xmm	14	<14	14-20	16-22 (15-20)	4	5	13	22(14)
MAXPD xmm, xmm	4	3	3	3	0.5	1	1	1
MAXSD xmm, xmm	4	3	3	3	0.5	1	1	1
MINPD xmm, xmm	4	3	3	3	0.5	1	1	1
MINSD xmm, xmm	4	3	3	3	0.5	1	1	1
MOVAPD xmm, xmm	1	1	1	1	0.33	0.5	0.5	1
MOVMSKPD r64/r32, xmm	2	2	2	2	1	1	1	1
MOVSD xmm, xmm	1	1	1	1	1	1	1	1
MOVUPD xmm, xmm	1	1	1	1	0.33	0.5	0.5	1
MULPD xmm, xmm	3	5	5	5	0.5	0.5	0.5	1
MULSD xmm, xmm	3	5	5	5	0.5	0.5	0.5	1
ORPD xmm, xmm	1	1	1	1	0.33	1	1	1
SHUFPD xmm, xmm, imm8	1	1	1	1	1	1	1	1
SQRTPD xmm, xmm[2]	18	20	20	22(21)	6	13	13	22(14)
SQRTSD xmm, xmm	18	20	20	22(21)	6	7	13	22(14)
SUBPD xmm, xmm	4	3	3	3	0.5	1	1	1
SUBSD xmm, xmm	4	3	3	3	0.5	1	1	1
UCOMISD xmm, xmm	2	2	2	2	1	1	1	1
UNPCKHPD xmm, xmm	1	1	1	1	1	1	1	1
UNPCKLPD xmm, xmm	1	1	1	1	1	1	1	1
XORPD[3] xmm, xmm	1	1	1	1	0.33	1	1	1

NOTES:

1. The latency and throughput of DIVPD/DIVSD can vary with input values. For certain values, hardware can complete quickly, throughput may be as low as ~ 6 cycles. Similarly, latency for certain input values may be as low as less than 10 cycles.
2. The latency throughput of SQRTPD/SQRTSD can vary with input value. For certain values, hardware can complete quickly, throughput may be as low as ~ 6 cycles. Similarly, latency for certain input values may be as low as less than10 cycles.

Table C-16. Streaming SIMD Extension Single-precision Floating-point Instructions

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)
ADDPS xmm, xmm	4	3	3	3	0.5	1	1	1
ADDSS xmm, xmm	4	3	3	3	0.5	1	1	1

Table C-16. Streaming SIMD Extension Single-precision Floating-point Instructions (Contd.)

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)
ANDNPS xmm, xmm	1	1	1	1	0.33	1	1	1
ANDPS xmm, xmm	1	1	1	1	0.33	1	1	1
CMPPS xmm, xmm	4	3	3	3	0.5	1	1	1
CMPSS xmm, xmm	4	3	3	3	0.5	1	1	1
COMISS xmm, xmm	2	2	2	2	1	1	1	1
CVTSI2SS xmm, r32	6	4	4	5	1	1	1	1
CVTSS2SI r32, xmm	6	4	4	5	1	1	1	1
CVT[T]SS2SI r64, xmm	6	4	4	5	1	1	1	1
CVTTSS2SI r32, xmm	6	4	4	5	1	1	1	1
DIVPS xmm, xmm[1]	11	<11	<13	10-14	3	4	6	14(6)
DIVSS xmm, xmm	11	<11	<13	10-14	3	2.5	6	14(6)
MAXPS xmm, xmm	4	3	3	3	0.5	1	1	1
MAXSS xmm, xmm	4	3	3	3	0.5	1	1	1
MINPS xmm, xmm	4	3	3	3	0.5	1	1	1
MINSS xmm, xmm	4	3	3	3	0.5	1	1	1
MOVAPS xmm, xmm	1	1	1	1	0.25	0.5	0.5	1
MOVHLPS xmm, xmm	1	1	1	1	1	1	1	1
MOVLHPS xmm, xmm	1	1	1	1	1	1	1	1
MOVMSKPS r64/r32, xmm	2	2	2	2	1	1	1	1
MOVSS xmm, xmm	1	1	1	1	1	1	1	1
MOVUPS xmm, xmm	1	1	1	1	0.25	0.5	0.5	1
MULPS xmm, xmm	4	3	5	5	0.5	0.5	0.5	1
MULSS xmm, xmm	4	3	5	5	0.5	0.5	0.5	1
ORPS xmm, xmm	1	1	1	1	0.33	1	1	1
RCPPS xmm, xmm	4	5	5	5	1	1	1	1
RCPSS xmm, xmm	4	5	5	5	1	1	1	1
RSQRTPS xmm, xmm	4	5	5	5	1	1	1	1
RSQRTSS xmm, xmm	4	5	5	5	1	1	1	1
SHUFPS xmm, xmm, imm8	1	1	1	1	1	1	1	1
SQRTPS xmm, xmm[2]	13	13	13	14	3	7	7	14(7)
SQRTSS xmm, xmm	13	13	13	14	3	4	7	14(7)
SUBPS xmm, xmm	4	3	3	3	0.5	1	1	1
SUBSS xmm, xmm	4	3	3	3	0.5	1	1	1
UCOMISS xmm, xmm	2	2	2	2	1	1	1	1
UNPCKHPS xmm, xmm	1	1	1	1	1	1	1	1
UNPCKLPS xmm, xmm	1	1	1	1	1	1	1	1
XORPS xmm, xmm	1	1	1	1	1	1	1	1
LFENCE[3]					6	5	5	4
MFENCE[3]					~40	~35	~35	~35

Table C-16. Streaming SIMD Extension Single-precision Floating-point Instructions (Contd.)

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)
SFENCE[3]					7	6	6	5
STMXCSR[3]					1	1	1	1
FXSAVE[3]					~90	~71	~75	~78

NOTES:

1. The latency and throughput of DIVPS/DIVSS can vary with input values. For certain values, hardware can complete quickly, throughput may be as low as ~ 6 cycles. Similarly, latency for certain input values may be as low as less than 10 cycles.
2. The latency and throughput of SQRTPS/SQRTSS can vary with input values. For certain values, hardware can complete quickly, throughput may be as low as ~ 6 cycles. Similarly, latency for certain input values may be as low as less than 10 cycles
3. The throughputs of FXSAVE/LFENCE/MFENCE/SFENCE/STMXCSR are measured with the destination in L1 Data Cache.

Table C-17. General Purpose Instructions

Instruction	Latency[1]				Throughput			
CPUID	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A, 06_3E	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A, 06_3E
ADC/SBB reg, reg	1	2	2	2	0.5	1	1	1
ADC/SBB reg, imm	1	2	2	2	0.5	1	1	1
ADD/SUB	1	1	1	1	0.25	0.25	0.25	0.33
AND/OR/XOR	1	1	1	1	0.25	0.25	0.25	0.33
BSF/BSR	3	3	3	3	1	1	1	1
BSWAP	2	2	2	2	0.5	0.5	0.5	1
BT	1	1	1	1	0.5	0.5	0.5	0.5
BTC/BTR/BTS	1	1	1	1	0.5	0.5	0.5	0.5
CBW/CWDE/CDQE	1	1	1	1	1	1	1	1
CDQ	1	1	1	1	1	1	1	1
CQO	1	1	1	1	0.5	0.5	0.5	0.5
CLC					0.25	0.33	0.33	0.33
CMC					0.25	0.33	0.33	0.33
STC					0.25	0.33	0.33	0.33
CLFLUSH[13]					~2 to 50	~3 to 50	~3 to 50	~5 to 50
CLFLUSHOPT[14]					~2to 10	NA	NA	NA
CMOVE/CMOVcc	1	1	2	2	0.5	0.5	0.5	0.5
CMOVBE/NBE/A/NA	2	2	3	3	1	1	1	1
CMP/TEST	1	1	1	1	0.25	0.25	0.25	0.33
CPUID (EAX = 0)					~100	~100	~100	~95
CPUID (EAX != 0)					>200	>200	>200	>200
CMPXCHG r64, r64	5	5	5	5	5	5	5	5

Table C-17. General Purpose Instructions (Contd.)

Instruction	Latency[1]				Throughput			
CPUID	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A, 06_3E	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A, 06_3E
CMPXCHG8B m64	15	8	8	8	15	8	8	8
CMPXCHG16B m128	19	10	10	10	19	10	10	10
Lock CMPXCHG8B m64	22	19	19	24	22	19	19	24
Lock CMPXCHG16B m128	32	28	28	29	32	28	28	29
DEC/INC	1	2	2	2	0.25	0.25	0.25	0.33
IMUL r64, r64	3	3	3	3	1	1	1	1
IMUL r64[11]	4, 5	3, 4	3, 4	3, 4	1	1	1	1
IMUL r32	5	4	4	4	1	1	1	1
IDIV r64 (RDX!= 0)[9]					~85-100	~85-100	~85-100	~85-100
IDIV r32[10]					~20-26	~20-26	~20-26	~19-25
LEA	1	1	1	1	0.5	0.5	0.5	0.5
LEA [base+index]disp	3	3	3	3	1	1	1	1
MOVSB/MOVSW	1	1	1	1	0..25	0..25	0..25	0.33
MOVZB/MOVZW	1	1	1	1	0.25	0.25	0.25	0.33
DIV r64 (RDX!= 0)[9]					~80-95	~80-95	~80-95	~80-95
DIV r32[10]					~20-26	~20-26	~20-26	~19-25
MUL r64[11]	4, 5	3, 4	3, 4	3, 4	1	1	1	1
NEG/NOT	1	2	2	2	0.25	0.25	0.25	0.33
PAUSE					~140	~10	~10	~10
RCL/RCR reg, 1	2	2	2	2	2	1.5	1.5	1.5
RCL/RCR	6	6	6	6	6	6	6	6
RDTSC					~13	~10	~10	~20
RDTSCP					~20	~30	~30	~30
ROL/ROR reg 1	1 (2 flg)	1 (2 flg)	1 (2 flg)	1 (2 flg)	1	1	1	1
ROL/ROR reg imm	1	1	1	1	0.5	0.5	0.5	0.5
ROL/ROR reg, cl	2	2	2	2	1.5	1.5	1.5	1.5
LAHF/SAHF	3	2	2	2				
SAL/SAR/SHL/SHR reg, imm	1	1	1	1	0.5	0.5	0.5	0.5
SAL/SAR/SHL/SHR reg, cl	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
SETBE	2	2	2	2	1	1	1	1
SETE	1	1	1	1	0.5	0.5	0.5	0.5
SHLD/RD reg, reg, cl	6	4	4	2 (4 flg)	1.5	1	1	1.5
SHLD/RD reg, reg, imm	3	3	3	1	0.5	0.5	0.5	0.5
XSAVE[12]					~98	~100	~100	~100
XSAVEOPT[12]					~86	~90	~90	~90
XADD	2	2	2	2	1	1	1	1
XCHG reg, reg	1	1	1	2	1	1	1	1
XCHG reg, mem	22	19	19	19	22	19	19	19

C.3.2 Table Footnotes

The following footnotes refer to all tables in this appendix.

1. Latency information for many instructions that are complex (> 4 µops) are estimates based on conservative (worst-case) estimates. Actual performance of these instructions by the out-of-order core execution unit can range from somewhat faster to significantly faster than the latency data shown in these tables.
2. The names of execution units apply to processor implementations of the Intel NetBurst microarchitecture with a CPUID signature of family 15, model encoding = 0, 1, 2. They include: ALU, FP_EXECUTE, FPMOVE, MEM_LOAD, MEM_STORE. Note the following:
   - The FP_EXECUTE unit is actually a cluster of execution units, roughly consisting of seven separate execution units.
   - The FP_ADD unit handles x87 and SIMD floating-point add and subtract operation.
   - The FP_MUL unit handles x87 and SIMD floating-point multiply operation.
   - The FP_DIV unit handles x87 and SIMD floating-point divide square-root operations.
   - The MMX_SHFT unit handles shift and rotate operations.
   - The MMX_ALU unit handles SIMD integer ALU operations.
   - The MMX_MISC unit handles reciprocal MMX computations and some integer operations.
   - The FP_MISC designates other execution units in port 1 that are separated from the six units listed above.
3. It may be possible to construct repetitive calls to some Intel 64 and IA-32 instructions in code sequences to achieve latency that is one or two clock cycles faster than the more realistic number listed in this table.
4. Latency and Throughput of transcendental instructions can vary substantially in a dynamic execution environment. Only an approximate value or a range of values are given for these instructions.
5. The FXCH instruction has 0 latency in code sequences. However, it is limited to an issue rate of one instruction per clock cycle.
6. The load constant instructions, FINCSTP, and FDECSTP have 0 latency in code sequences.
7. Selection of conditional jump instructions should be based on the recommendation of Section 3.4.1, “Branch Prediction Optimization,” to improve the predictability of branches. When branches are predicted successfully, the latency of jcc is effectively zero.
8. RCL/RCR with shift count of 1 are optimized. Using RCL/RCR with shift count other than 1 will be executed more slowly. This applies to the Pentium 4 and Intel Xeon processors.
9. The throughput of “DIV/IDIV r64” varies with the number of significant digits in the input RDX:RAX. The throughput is significantly higher if RDX input is 0, similar to those of “DIV/IDIV r32”. If RDX is not zero, the throughput is significantly lower, as shown in the range. The throughput decreases (increasing numerical value in cycles) with increasing number of significant bits in the input RDX:RAX (relative to the number of significant bits of the divisor) or the output quotient. The latency of “DIV/IDIV r64” also varies with the significant bits of input values. For a given set of input values, the latency is about the same as the throughput in cycles.
10. The throughput of “DIV/IDIV r32” varies with the number of significant digits in the input EDX:EAX and/or of the quotient of the division for a given size of significant bits in the divisor r32. The throughput decreases (increasing numerical value in cycles) with increasing number of significant bits in the input EDX:EAX or the output quotient. The latency of “DIV/IDIV r32” also varies with the significant bits of the input values. For a given set of input values, the latency is about the same as the throughput in cycles.
11. The latency of MUL r64 into 128-bit result has two sets of numbers, the read-to-use latency of the low 64-bit result (RAX) is smaller. The latency of the high 64-bit of the 128 bit result (RDX) is larger.
12. The throughputs of XSAVE and XSAVEOPT are measured with the destination in L1 Data Cache and includes the YMM states.

13. CLFLUSH throughput is representative from clean cache lines for a range of buffer sizes. CLFLUSH throughput can decrease significantly by factors including: (a) the number of back-to-back CLFLUSH being executed, (b) flushing modified cache lines incurs additional cost than cache lines in other coherent state. See Section 7.5.6.
14. CLFLUSHOPT throughput is representative from clean cache lines for a range of buffer sizes. CLFLUSHOPT throughput can decrease by factors including: (a) flushing modified cache lines incurs additional cost than cache lines in other coherent state, (b) the number of cache lines back-to-back. See Section 7.5.7.

C.3.3 Instructions with Memory Operands

The latency of an Instruction with memory operand can vary greatly due to a number of factors, including data locality in the memory/cache hierarchy and characteristics that are unique to each microarchitecture. Generally, software can approach tuning for locality and instruction selection independently. Thus Table C-4 through Table C-18 can be used for the purpose of instruction selection. Latency and throughput of data movement in the cache/memory hierarchy can be dealt with independent of instruction latency and throughput. Load-to-use Latency of the cache hierarchy can be found in Chapter 2.

C.3.3.1 Software Observable Latency of Memory References

When measuring latency of memory references of individual instructions, many factors can influence the observed latency exposure. Aside from access patterns, cache locality, effect of the hardware prefetchers, different microarchitectures may expose variability such register domains of the destination or memory addressing form with respect to the instruction encoding.

The table below gives a few selected sampling of the variability of L1D cache hit latency that software may observe using pointer-chasing constructs, due to memory reference encoding details, on recent Intel microarchitectures.

Table C-18. Pointer-Chasing Variability of Software Measurable Latency of L1 Data Cache Latency

Pointer Chase Construct	L1D latency Observation
MOV rax, [rax]	4
MOV rax, disp32[rax] , disp32 < 2048	4
MOV rax, [rcx+rax]	5
MOV rax, disp32[rcx+rax] , disp32 < 2048	5

INDEX

Numerics

64-bit mode
- arithmetic, 9-5
- coding guidelines, 9-1
- compiler settings, A-2
- CVTSI2SD instruction, 9-6
- CVTSI2SS instruction, 9-6
- default operand size, 9-1
- introduction, 2-56
- legacy instructions, 9-1
- multiplication notes, 9-2
- register usage, 9-1, 9-5
- REX prefix, 9-1
- sign-extension, 9-4
- software prefetch, 9-6

A

absolute difference of signed numbers, 5-16
absolute difference of unsigned numbers, 5-15
absolute value, 5-16
active power, 13-1
ADDSUBPD instruction, 6-11
ADDSUBPS instruction, 6-11, 6-13
algorithm to avoid changing the rounding mode, 3-72
alignment
- arrays, 3-52
- code, 3-8
- stack, 3-54
- structures, 3-52

Amdahl's law, 8-1
AoS format, 4-22
application performance tools, A-1
arrays
- aligning, 3-52

automatic vectorization, 4-16

B

battery life
- guidelines for extending, 13-6
- mobile optimization, 13-1
- OS APIs, 13-7
- quality trade-offs, 13-6

branch prediction
- choosing types, 3-9
- code examples, 3-5
- eliminating branches, 3-5
- optimizing, 3-4
- unrolling loops, 3-10

C

C4-state, 13-4
cache management
- blocking techniques, 7-19
- cache level, 7-3
- CLFLUSH instruction, 7-9
- coding guidelines, 7-1, 11-2
- compiler choices, 7-1
- compiler intrinsics, 7-1
- CPUID instruction, 3-3, 7-31
- function leaf, 3-3
- optimizing, 7-1
- simple memory copy, 7-27
- video decoder, 7-26
- video encoder, 7-26
- See also: optimizing cache utilization

CD/DVD, 13-7
changing the rounding mode, 3-71
classes (C/C++), 4-15
CLFLUSH instruction, 7-9
clipping to an arbitrary signed range, 5-19
clipping to an arbitrary unsigned range, 5-21
coding techniques, 4-12, 8-17
- 64-bit guidelines, 9-1
- absolute difference of signed numbers, 5-16
- absolute difference of unsigned numbers, 5-15
- absolute value, 5-16
- clipping to an arbitrary signed range, 5-19
- clipping to an arbitrary unsigned range, 5-21
- conserving power, 13-7
- data in segment, 3-56
- generating constants, 5-14
- interleaved pack with saturation, 5-6
- interleaved pack without saturation, 5-7
- latency and throughput, C-1
- methodologies, 4-13
- non-interleaved unpack, 5-8
- optimization options, A-2
- rules, 3-3
- signed unpack, 5-5
- simplified clip to arbitrary signed range, 5-20
- sleep transitions, 13-8
- suggestions, 3-3
- tuning hints, 3-3
- unsigned unpack, 5-5
- See also: floating-point code

coherent requests, 7-7
command-line options
- inline expansion of library functions, A-3
- vectorizer switch, A-3

comparing register values, 3-25, 3-27
compatibility mode, 9-1
compatibility model, 2-56
compiler intrinsics
- _mm_load, 7-1, 7-26
- _mm_prefetch, 7-1, 7-26
- _mm_stream, 7-1, 7-26

compilers
- branch prediction support, 3-11
- documentation, 1-3
- general recommendations, 3-2
- plug-ins, A-2
- supported alignment options, 4-19
- See also: Intel C++ Compiler & Intel Fortran Compiler

computation
- intensive code, 4-12

converting 64-bit to 128-bit SIMD integers, 5-34
converting code to MMX technology, 4-10
CPUID instruction
- AP-485, 1-3
- cache paramaters, 7-31
- function leaf, 7-31
- function leaf 4, 3-3
- Intel compilers, 3-3
- MMX support, 4-2
- SSE support, 4-2
- SSE2 support, 4-2
- SSE3 support, 4-3
- SSSE3 support, 4-3, 4-4
- strategy for use, 3-3

C-states, 13-1, 13-3
CVTSI2SD instruction, 9-6
CVTSI2SS instruction, 9-6
CVTTPS2PI instruction, 6-10

CVTTSS2SI instruction, 6-10

D

data
- access pattern of array, 3-53
- aligning arrays, 3-52
- aligning structures, 3-52
- alignment, 4-17
- arrangement, 6-2
- code segment and, 3-56
- deswizzling, 6-7
- swizzling, 6-5
- swizzling using intrinsics, 6-6

deeper sleep, 13-4
denormals-are-zero (DAZ), 6-10
deterministic cache parameters
- cache sharing, 7-31, 7-32
- multicore, 7-32
- overview, 7-31
- prefetch stride, 7-32

domain decomposition, 8-4
Dynamic execution, 2-44

E

eliminating branches, 3-6, 13-14
EMMS instruction, 5-2, 5-3
- guidelines for using, 5-2

Enhanced Intel SpeedStep Technology
- description of, 13-8
- multicore processors, 13-10
- usage scenario, 13-1

extract word instruction, 5-9

F

fencing operations, 7-5
- LFENCE instruction, 7-8
- MFENCE instruction, 7-8

FIST instruction, 3-71
FLDCW instruction, 3-71
floating-point code
- arithmetic precision options, A-3
- data arrangement, 6-2
- data deswizzling, 6-7
- data swizzling using intrinsics, 6-6
- guidelines for optimizing, 3-68
- horizontal ADD, 6-8
- improving parallelism, 3-73
- memory access stall information, 3-50
- operations, integer operands, 3-73
- optimizing, 3-68
- planning considerations, 6-1
- rules and suggestions, 6-1
- scalar code, 6-2
- transcendental functions, 3-73
- unrolling loops, 3-10
- vertical versus horizontal computation, 6-3
- See also: coding techniques

flush-to-zero (FTZ), 6-10
front end
- branching ratios, B-58
- characterizing mispredictions, B-59
- loop unrolling, 8-22
- optimization, 3-4

functional decomposition, 8-4
FXCH instruction, 3-73, 6-2

G

generating constants, 5-14
GetActivePwrScheme, 13-7
GetSystemPowerStatus, 13-7

H

HADDPD instruction, 6-11
HADDPS instruction, 6-11, 6-14
hardware multithreading
- support for, 3-3

hardware prefetch
- cache blocking techniques, 7-23
- description of, 7-2
- latency reduction, 7-12
- memory optimization, 7-11
- operation, 7-12

horizontal computations, 6-8
hotspots
- definition of, 4-12
- identifying, 4-12
- VTune analyzer, 4-12

HSUBPD instruction, 6-11
HSUBPS instruction, 6-11, 6-14
Hyper-Threading Technology
- avoid excessive software prefetches, 8-18
- bus optimization, 8-9
- cache blocking technique, 8-19
- conserve bus command bandwidth, 8-17
- eliminate 64-K-aliased data accesses, 8-22
- excessive loop unrolling, 8-22
- front-end optimization, 8-22
- full write transactions, 8-19
- functional decomposition, 8-4
- improve effective latency of cache misses, 8-18
- memory optimization, 8-19
- minimize data sharing between physical processors, 8-20
- multitasking environment, 8-2
- optimization, 8-1
- optimization guidelines, 8-8
- optimization with spin-locks, 8-13
- overview, 2-53
- parallel programming models, 8-4
- pipeline, 2-55
- placement of shared synchronization variable, 8-15
- prevent false-sharing of data, 8-14
- processor resources, 2-54
- shared execution resources, 8-25
- shared-memory optimization, 8-20
- synchronization for longer periods, 8-13
- synchronization for short periods, 8-11
- system bus optimization, 8-17
- thread sync practices, 8-8
- thread synchronization, 8-10
- tools for creating multithreaded applications, 8-7

I

IA-32e mode, 2-56
IA32_PERFEVSELx MSR, B-57
increasing bandwidth
- memory fills, 5-30
- video fills, 5-30

Indefinite
- description of, 2-61, 2-62

indirect branch, 3-9
inline assembly, 5-3
inline expansion library functions option, A-3
inlined-asm, 4-14
insert word instruction, 5-10
instruction latency/throughput
- overview, C-1

instruction scheduling, 3-56
Intel 64 and IA-32 processors, 2-1
Intel 64 architecture

- and IA-32 processors, 2-56
- features of, 2-56
- IA-32e mode, 2-56

Intel Advanced Digital Media Boost, 2-31
Intel Advanced Memory Access, 2-38
Intel Advanced Smart Cache, 2-31, 2-42
Intel Core Duo processors
- 128-bit integers, 5-34
- packed FP performance, 6-14
- performance events, B-52
- prefetch mechanism, 7-3
- SIMD support, 4-1
- special programming models, 8-4
- static prediction, 3-6

Intel Core microarchitecture, 2-1, 2-30
- advanced smart cache, 2-42
- branch prediction unit, 2-33
- event ratios, B-57
- execution core, 2-36
  - execution units, 2-36
  - issue ports, 2-36
- front end, 2-32
- instruction decode, 2-35
- instruction fetch unit, 2-33
- instruction queue, 2-34
  - advanced memory access, 2-38
- micro-fusion, 2-35
- pipeline overview, 2-13, 2-31
- special programming models, 8-4
- stack pointer tracker, 2-35
- static prediction, 3-8

Intel Core Solo processors
- 128-bit SIMD integers, 5-34
- performance events, B-52
- prefetch mechanism, 7-3
- SIMD support, 4-1
- static prediction, 3-6

Intel C++ Compiler, 3-1
- 64-bit mode settings, A-2
- branch prediction support, 3-11
- description, A-1
- IA-32 settings, A-2
- multithreading support, A-3
- OpenMP, A-3
- optimization settings, A-2
- related Information, 1-3

Intel Debugger
- description, A-1

Intel developer link, 1-3
Intel Enhanced Deeper Sleep
- C-state numbers, 13-3
- enabling, 13-9
- multiple-cores, 13-11

Intel Fortran Compiler
- description, A-1
- multithreading support, A-3
- OpenMP, A-3
- optimization settings, A-2
- related information, 1-3

Intel Integrated Performance Primitives
- for Linux, A-5
- for Windows, A-5

Intel Math Kernel Library for Linux, A-5
Intel Math Kernel Library for Windows, A-5
Intel Mobile Platform SDK, 13-7
Intel NetBurst microarchitecture
- introduction, 2-6, 2-12, 2-44
- prefetch characteristics, 7-2
- trace cache, 3-8

Intel Performance Libraries
- benefits, A-5
- optimizations, A-5

Intel performance libraries
- description, A-1

Intel Performance Tools, 3-1
Intel Smart Memory Access, 2-31
Intel software network link, 1-3
Intel Thread Checker
- example output, A-5, A-6, A-7, A-8

Intel Threading Tools, A-7
Intel VTune Performance Analyzer
- code coach, 4-12
- coverage, 3-2
- related information, 1-3

Intel Wide Dynamic Execution, 2-30, 2-31, 2-44
interleaved pack with saturation, 5-6
interleaved pack without saturation, 5-7
interprocedural optimization, A-3
introduction
- chapter summaries, 1-1
- optimization features, 2-1
- processors covered, 1-1
- references, 1-3

IPO. See interprocedural optimization

L

large load stalls, 3-50
latency, 7-3, 7-13
legacy mode, 9-1
LFENCE Instruction, 7-8
links to web data, 1-3
load instructions and prefetch, 7-4
loading-storing to-from same DRAM page, 5-30
loop
- blocking, 4-24
- unrolling, 7-17, A-3

M

MASKMOVDQU instruction, 7-5
memory bank conflicts, 7-2
memory optimizations
- loading-storing to-from same DRAM page, 5-30
- overview, 5-27
- partial memory accesses, 5-28, 5-31
- performance, 4-20
- reference instructions, 3-25
- using aligned stores, 5-31
- using prefetch, 7-11

MFENCE instruction, 7-8
misaligned data access, 4-17
misalignment in the FIR filter, 4-18
mobile computing
- ACPI standard, 13-1, 13-3
- active power, 13-1
- battery life, 13-1, 13-6, 13-7
- C4-state, 13-4
- CD/DVD, WLAN, WiFi, 13-7
- C-states, 13-1, 13-3
- deep sleep transitions, 13-8
- deeper sleep, 13-4, 13-9
- Intel Mobile Platform SDK, 13-7
- OS APIs, 13-7
- OS changes processor frequency, 13-1
- OS synchronization APIs, 13-7
- overview, 13-1, 14-1, 15-1
- performance options, 13-6
- platform optimizations, 13-7
- P-states, 13-1
- Speedstep technology, 13-8
- spin-loops, 13-7
- state transitions, 13-1
- static power, 13-1

WM_POWERBROADCAST message, 13-8
MOVAPD instruction, 6-2
MOVAPS instruction, 6-2
MOVDDUP instruction, 6-11
move byte mask to integer, 5-12
MOVHLPS instruction, 6-8
MOVLHPS instruction, 6-8
MOVNTDQ instruction, 7-5
MOVNTI instruction, 7-5
MOVNTPD instruction, 7-5
MOVNTPS instruction, 7-5
MOVNTQ instruction, 7-5
MOVQ Instruction, 5-30
MOVSHDUP instruction, 6-11, 6-13
MOVSLDUP instruction, 6-11, 6-13
MOVUPD instruction, 6-2
MOVUPS instruction, 6-2
multicore processors
architecture, 2-1
C-state considerations, 13-11
energy considerations, 13-10
SpeedStep technology, 13-10
thread migration, 13-10
multiprocessor systems
dual-core processors, 8-1
HT Technology, 8-1
optimization techniques, 8-1
See also: multithreading & Hyper-Threading Technology
multithreading
Amdahl's law, 8-1
application tools, 8-7
bus optimization, 8-9
compiler support, A-3
dual-core technology, 3-3
environment description, 8-1
guidelines, 8-8
hardware support, 3-3
HT technology, 3-3
Intel Core microarchitecture, 8-4
parallel & sequential tasks, 8-1
programming models, 8-3
shared execution resources, 8-25
specialized models, 8-4
thread sync practices, 8-8
See Hyper-Threading Technology

N

Newton-Raphson iteration, 6-1
non-coherent requests, 7-7
non-interleaved unpack, 5-8
non-temporal stores, 7-6, 7-25
NOP, 3-27

O

OpenMP compiler directives, 8-7, A-3
optimization
branch prediction, 3-4
branch type selection, 3-9
eliminating branches, 3-5
features, 2-1
general techniques, 3-1
spin-wait and idle loops, 3-6
static prediction, 3-6
unrolling loops, 3-10
optimizing cache utilization
cache management, 7-26
examples, 7-8
non-temporal store instructions, 7-5, 7-8
prefetch and load, 7-4
prefetch instructions, 7-4
prefetching, 7-4
SFENCE instruction, 7-8
streaming, non-temporal stores, 7-5
See also: cache management
OS APIs, 13-7

P

pack instructions, 5-6
packed average byte or word), 5-22
packed multiply high unsigned, 5-21
packed shuffle word, 5-12
packed signed integer word maximum, 5-21
packed sum of absolute differences, 5-22
parallelism, 4-12, 8-4
partial memory accesses, 5-28
PAUSE instruction, 3-6, 8-8
PAVGB instruction, 5-22
PAVGW instruction, 5-22
PeekMessage(), 13-7
Pentium 4 processors
static prediction, 3-6
Pentium M processors
prefetch mechanisms, 7-3
static prediction, 3-6
performance models
Amdahl's law, 8-1
multithreading, 8-1
parallelism, 8-1
usage, 8-1
performance monitoring events
analysis techniques, B-53
Bus_Not_In_Use, B-53
Bus_Snoops, B-53
DCU_Snoop_to_Share, B-53
drill-down techniques, B-53
event ratios, B-57
Intel Core Duo processors, B-52
Intel Core Solo processors, B-52
Intel Netburst architecture, B-1
Intel Xeon processors, B-1
L1_Pref_Req, B-53
L2_No_Request_Cycles, B-53
L2_Reject_Cycles, B-53
Pentium 4 processor, B-1
performance counter, B-52
ratio interpretation, B-52
See also: clock ticks
Serial_Execution_Cycles, B-53
Unhalted_Core_Cycles, B-53
Unhalted_Ref_Cycles, B-53
performance tools, 3-1
PEXTRW instruction, 5-9
PGO. See profile-guided optimization
PINSRW instruction, 5-10
PMINSW instruction, 5-21
PMINUB instruction, 5-21
PMOVMSKB instruction, 5-12
PMULHUW instruction, 5-21
predictable memory access patterns, 7-4
prefetch
64-bit mode, 9-6
coding guidelines, 7-1
compiler intrinsics, 7-1
concatenation, 7-16
deterministic cache parameters, 7-31
hardware mechanism, 7-2
characteristics, 7-12
latency, 7-12
how instructions designed, 7-4
innermost loops, 7-4
instruction considerations

cache block techniques, 7-19
checklist, 7-15
concatenation, 7-16
hint mechanism, 7-3
minimizing number, 7-17
scheduling distance, 7-15
single-pass execution, 7-2, 7-24
spread with computations, 7-18
strip-mining, 7-21
summary of, 7-3
instruction variants, 7-4
latency hiding/reduction, 7-13
load Instructions, 7-4
memory access patterns, 7-4
memory optimization with, 7-11
minimizing number of, 7-17
scheduling distance, 7-2, 7-15
software data, 7-3
spreading, 7-19
when introduced, 7-1
PREFETCHNTA instruction, 7-4, 7-21
usage guideline, 7-2
PREFETCHT0 instruction, 7-4, 7-21
usage guideline, 7-2
PREFETCHT1 instruction, 7-4
PREFETCHT2 instruction, 7-4
producer-consumer model, 8-4
profile-guided optimization, A-3
PSADBW instruction, 5-22
PSHUF instruction, 5-12
P-states, 13-1

Q

-Qparallel, 8-8

R

ratios, B-57
branching and front end, B-58
references, 1-3
releases of, 2-58

S

sampling
event-based, A-6
scheduling distance (PSD), 7-15
Self-modifying code, 3-56
SFENCE Instruction, 7-8
SHUFPS instruction, 6-3
signed unpack, 5-5
SIMD
auto-vectorization, 4-16
cache instructions, 7-1
classes, 4-15
coding techniques, 4-12
data alignment for 128-bits, 4-19
data alignment for MMX, 4-18
data and stack alignment, 4-17
example computation, 2-57
history, 2-57
identifying hotspots, 4-12
instruction selection, 4-26
loop blocking, 4-24
memory utilization, 4-20
microarchitecture differences, 4-27
MMX technology support, 4-2
padding to align data, 4-17
parallelism, 4-12
SSE support, 4-2
SSE2 support, 4-2
SSE3 support, 4-3
SSSE3 support, 4-3, 4-4
stack alignment for 128-bits, 4-18
strip-mining, 4-23
using arrays, 4-17
vectorization, 4-12
VTune capabilities, 4-12
SIMD floating-point instructions
data arrangement, 6-2
data deswizzling, 6-7
data swizzling, 6-5
different microarchitectures, 6-11
general rules, 6-1
horizontal ADD, 6-8
Intel Core Duo processors, 6-14
Intel Core Solo processors, 6-14
planning considerations, 6-1
reciprocal instructions, 6-1
scalar code, 6-2
SSE3 complex math, 6-12
SSE3 FP programming, 6-11
using
ADDSUBPS, 6-13
CVTTPS2PI, 6-10
CVTTSS2SI, 6-10
FXCH, 6-2
HADDPS, 6-14
HSUBPS, 6-14
MOVAPD, 6-2
MOVAPS, 6-2
MOVHLPS, 6-8
MOVLHPS, 6-8
MOVSHDUP, 6-13
MOVSLDUP, 6-13
MOVUPD, 6-2
MOVUPS, 6-2
SHUFPS, 6-3
vertical vs horizontal computation, 6-3
with x87 FP instructions, 6-2
SIMD technology, 2-58
SIMD-integer instructions
64-bits to 128-bits, 5-34
data alignment, 5-3
data movement techniques, 5-5
extract word, 5-9
integer intensive, 5-1
memory optimizations, 5-27
move byte mask to integer, 5-12
optimization by architecture, 5-34
packed average byte or word), 5-22
packed multiply high unsigned, 5-21
packed shuffle word, 5-12
packed signed integer word maximum, 5-21
packed sum of absolute differences, 5-22
rules, 5-1
signed unpack, 5-5
unsigned unpack, 5-5
using
EMMS, 5-2
MOVDQ, 5-30
MOVQ2DQ, 5-14
PABSW, 5-16
PACKSSDW, 5-6
PADDQ, 5-23
PALIGNR, 5-4
PAVGB, 5-22
PAVGW, 5-22
PEXTRW, 5-9
PINSRW, 5-10
PMADDWD, 5-22
PMAXSW, 5-21
PMAXUB, 5-21

PMINSW, 5-21
PMINUB, 5-21
PMOVMSKB, 5-12
PMULHUW, 5-21
PMULHW, 5-21
PMULUDQ, 5-21
PSADBW, 5-22
PSHUF, 5-12
PSHUFB, 5-17, 5-18
PSHUFLW, 5-13
PSLLDQ, 5-23
PSRLDQ, 5-23
PSUBQ, 5-23
PUNPCHQDQ, 5-14
PUNPCKLQDQ, 5-14
simplified 3D geometry pipeline, 7-13
simplified clipping to an arbitrary signed range, 5-20
single vs multi-pass execution, 7-24
sleep transitions, 13-8
SoA format, 4-22
software write-combining, 7-25
spin-loops, 13-7
optimization, 3-6
PAUSE instruction, 3-6
related information, 1-3
SSE, 2-59
SSE2, 2-59
SSE3, 2-59
SSSE3, 2-60, 2-61
stack
alignment 128-bit SIMD, 4-18
alignment stack, 3-54
dynamic alignment, 3-54
state transitions, 13-1
static power, 13-1
static prediction, 3-6
streaming stores, 7-5
coherent requests, 7-7
improving performance, 7-5
non-coherent requests, 7-7
strip-mining, 4-23, 4-24, 7-21, 7-22
prefetch considerations, 7-23
structures
aligning, 3-52
system bus optimization, 8-17

T

time-consuming innermost loops, 7-4
TLB. See transaction lookaside buffer
transaction lookaside buffer, 7-27
transcendental functions, 3-73

U

unpack instructions, 5-8
unrolling loops
benefits of, 3-10
code examples, 3-11
unsigned unpack, 5-5
using MMX code for copy, shuffling, 6-8

V

vector class library, 4-16
vectorized code
auto generation, A-4
automatic vectorization, 4-16
high-level examples, A-4
parallelism, 4-12
SIMD architecture, 4-12
switch options, A-2
vertical vs horizontal computation, 6-3

W

WaitForSingleObject(), 13-7
WaitMessage(), 13-7
weakly ordered stores, 7-5
WiFi, 13-7
WLAN, 13-7
write-combining
buffer, 7-26
memory, 7-26
semantics, 7-6

X

XCHG EAX,EAX, support for, 3-28
XFEATURE_ENALBED_MASK, 4-5
XRSTOR, 4-5
XSAVE, 4-5, 4-8, 4-9