THE NEXT-GENERATION INTEL® CORE™ MICROARCHITECTURE

Contributors

Martin Dixon
Intel Corporation

Per Hammarlund
Intel Corporation

Stephan Jourdan
Intel Corporation

Ronak Singhal
Intel Corporation

Abstract

The next-generation Intel® microarchitecture was designed to allow products (under the *Intel® microarchitecture code name Nehalem and Intel® microarchitecture code name Westmere*) to be scaled from low-power laptops to high-performance servers. The core was created with power efficiency in mind and offers performance improvements for both lightly-threaded and highly-threaded workloads, while also adding key segment-specific features. We describe the innovative techniques used to achieve these goals in the core, including the development of traditional microarchitecture enhancements, new instructions, the addition of Intel Hyper-Threading Technology, innovative power-management schemes, and other performance and power improvements throughout the core.

Index Words

Nehalem
Westmere
Core
Power consumption
Loop Stream Detector
New Instructions
Physical Addressing

Introduction

The Intel® microarchitecture that appears in Nehalem and Westmere processors is the follow-on to the successful Intel® Core™ and Intel® Core 2 products and forms the basis of the Intel® Core™ i3, Intel® Core™ i5 and the Intel® Core™ i7 series of chips and the Intel Xeon® 5500/5600/7500 CPUs [1]. Each of these products is built by combining a processor core that contains the instruction processing logic and an "Uncore" that contains the logic that glues together multiple cores and interfaces with the system beyond the processor. The core is common to all products, while Uncores are designed for specific market segments. An example Uncore is described in another article in this issue of the *Intel Technology Journal* [2]. In this article, we detail the key features of the core that forms the basis of all the Nehalem and Westmere family of products. Given that the processor core had to scale from low-power laptops to high-end servers, we designed it with the overall goal of power efficiency for a wide range of operations, while achieving improved performance over the prior generation of processors.

"We added features only if they added performance in a power-efficient manner. If features were beneficial for performance, but were not power efficient, we did not pursue them for this design."

Guiding Principles

Given the range of products to be covered, the focus on power efficiency required a new mindset in choosing which features to include. Even though our goal was still to provide high performance, we added features only if they added performance in a power-efficient manner. If features were beneficial for performance, but were not power efficient, we did not pursue them for this design. For this design, the starting point was the Intel Core 2 microarchitecture (code-named Penryn).

The rule of thumb for judging the power efficiency of any particular performance feature was to compare performance without the feature versus performance with the feature within the same power envelope. Total power consumed by a component includes dynamic power plus leakage power. Dynamic power is equal to the capacitance multiplied by the frequency and the square of the voltage. Features that increase capacitance require the frequency and voltage to be dropped to remain at the same power level. Making the assumption that frequency and voltage are linearly related and that leakage is directly proportional to dynamic power, then power is cubically related to frequency. Therefore, our basic rule of thumb is that a feature that adds 1% performance needs to do so by adding less than 3% power. For our analysis, we make the assumption that the interesting power-scaling range starts at the maximum frequency achievable at the minimum supported voltage. To drop below that frequency, the voltage is held constant, so there is only a linear relationship in that range. The power efficiency that needs to be achieved in that linear scaling range is much more strict and therefore a much less power-efficient operating point, one that we seek to avoid. Consequently, if a performance feature is worse than cubic in its power-performance relationship, then we are actually losing performance in a fixed power envelope. Overall, for this microarchitecture generation, the majority of the features added were much better than this 3:1 ratio, with most features approaching 1:1.

"Our basic rule of thumb is that a feature that adds 1% performance needs to do so by adding less than 3% power."

Beyond these power-efficient performance features, we also needed to add market segment-specific features. These were features that were critical for one of our target segments (mobile, desktop, server) but not in other segments. For instance, the amount of memory that needs to be addressed is manifested in the number of physical address bits that are supported in the core. The server segment has a much higher demand for memory capacity than either desktop or mobile products. It was important to find a balance between the needs of each market segment in order to maintain our power efficiency while still hitting our segment targets.

Figure 1 shows the power-performance characteristics of the core changes with the addition of segment-specific features, design and microarchitecture improvements, and performance work. Starting with a baseline curve (Baseline), we add the segment-specific features (Segment-Specific Features). With these features, the curve moves up because the power increases without a performance change. The design and microarchitecture enhancements (improvements) do three things to the curve:

- The enhancements move the curve down due to power reduction work.
- They extend the operating point towards the left, by enabling operation at lower voltages.
- They extend the operating point to the right by enabling higher frequency operation.

"We end up with a core that yields higher performance at lower power envelopes and that covers a wider range of power envelopes."

The power-efficient performance features increase the instructions per clock (IPC) and move the curve up and to the right. Overall, we end up with a core that yields higher performance at lower power envelopes and that covers a wider range of power envelopes.

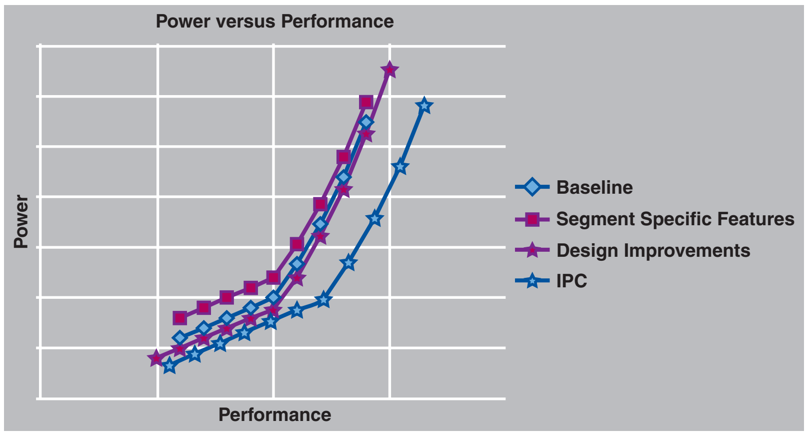

Figure 1: Power-performance for different changes to a processor design

Summing up our approach for this microarchitecture, the guiding principles we used in designing the core microarchitecture were these:

- Simply aggregating more and more cores is not sufficient to provide a well-balanced performance increase. We must also deliver a per-core performance increase since not all software has been, or even can be, upgraded to use all available threads. Moreover, even highly parallel software still has serial sections that benefit from a faster core. The need for increased per-core performance has been echoed by many of our key customers.
- Customers should not have to choose between high performance when all cores are active or high performance when only some cores are being used. Prior to this generation of processors, customers had to make a tradeoff between number of cores and the maximum frequency that cores could run at. We sought to eliminate the need for this tradeoff.
- Power envelopes should remain flat or move down generation after generation. This enables smaller form factors in laptops, while it also addresses critical power constraints that face servers today. Therefore, while we strived for higher performance, we could not do it by increasing power.

Microarchitecture Overview

The Core 2 microarchitecture had a theoretical maximum throughput of four instructions per cycle. We chose to maintain that maximum theoretical throughput, so our focus was on how to achieve a greater utilization of the possible peak performance.

The core consists of several sub-blocks, known as clusters, that are responsible for key pieces of overall functionality. In each cluster, a set of new features was added to improve the overall efficiency of the core. As seen in Figure 2, there are three basic clusters that make up the core. The front end is responsible for fetching instruction bytes and decoding those bytes into micro-operations (µops) that the rest of the core can consume. The Out of Order (OOO) and execution engine are responsible for allocating the necessary resources to µops, scheduling the µops for execution, executing them, and then reclaiming the resources. The memory cluster is responsible for handling load and store operations.

"The core consists of several sub-blocks, known as clusters, that are responsible for key pieces of overall functionality."

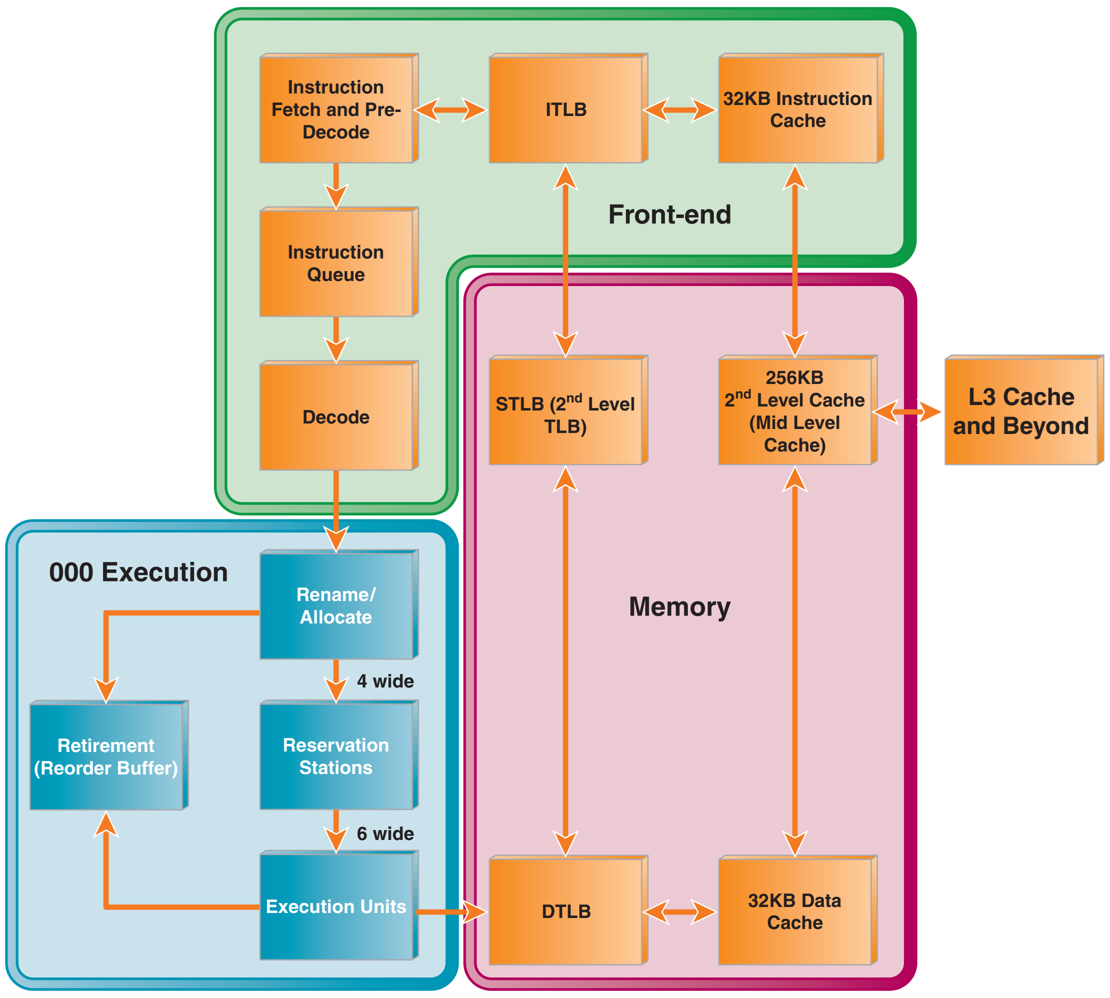

Figure 2: Block diagram of the core

Front End

One goal for the front-end cluster is to provide a steady stream of µops to the other clusters. Otherwise, the rest of the machine starves waiting for operations to execute. The focus for performance improvements is providing higher effective throughput through the front end while keeping latencies low.

"Many fundamental portions of the front end remain unchanged from previous generations of processors."

"The instruction cache size remains at 32 kilobytes and is organized in 64-byte lines."

"With each generation, branch prediction is typically near the top of that list. The rationale for this is simple: more efficient branch prediction gives better efficiency with no other changes to the machine."

Many fundamental portions of the front end remain unchanged from previous generations of processors. Specifically, the instruction cache size remains at 32 kilobytes and is organized in 64-byte lines. Similarly, there are four decoders that are used to translate raw instruction bytes into µops for the rest of the machine to consume.

One area in the front end that has traditionally been improved in each generation of processors is the accuracy of branch prediction. At the beginning of each project, we look at what areas of the processor have the greatest leverage for improving overall performance. And with each generation, branch prediction is typically near the top of that list. The rationale for this is simple: more efficient branch prediction gives better efficiency *with no other changes to the machine.*

Even though the predictors today have a very high accuracy rate, improving the prediction accuracy still has a significant impact on performance, because of the opportunity cost of a mispredicted branch. When a branch is mispredicted, the impact of flushing the pipeline increases with each generation of processor as we improve the throughput and increase the speculation depth of the core.

Accurate branch prediction is also critical for power efficiency. Each mispredicted branch represents a case where instructions were fetched, decoded, renamed, allocated, and possibly executed, and then thrown away. Because they are thrown away, the work provides almost no benefit, yet it costs power. More accurate branch prediction reduces speculative operations and can result in higher power efficiency.

In this microarchitecture generation, we make several notable improvements to our branch predictors. First, we add a second-level (L2) branch predictor. The purpose of this predictor is to aid in improving the prediction accuracy for applications that have large code footprints that do not fit well in the existing predictors. This addition is in line with providing better performance for server workloads, like databases, that typically have large code footprints.

"We also added a Renamed Return Stack Buffer (RSB). This idea was first implemented in the Core 2 Penryn Processor family."

"In prior microarchitecture generations, mispredicting branches could corrupt the RSB."

We also added a Renamed Return Stack Buffer (RSB). This idea was first implemented in the Core 2 Penryn Processor family [2]. CALL instructions are branches that are typically used to enter into functions, and RET (Return) instructions are branches used to exit functions and return back to where the function CALL occurred. An RSB is a microarchitecture feature that implements a simple stack. The content of the stack is maintained such that when CALLs occur, the address of the CALL is pushed onto the stack. When a RET occurs, it pops an address off the stack and uses the popped address as its prediction of where the RET is branching to. However, in prior microarchitecture generations, mispredicting branches could corrupt the RSB. For instance, a CALL instruction would push an address onto the stack, but then would itself be flushed from the machine due to a mispredicted branch. A subsequent RET would then pop the address from the mispredicted CALL off the stack and would also then mispredict. With the renamed RSB, we are able to avoid these corruption scenarios.

Beyond branch prediction, another area of improvement in this generation is increasing the number of *macrofusion* cases. Macrofusion is a capability introduced in the Core 2 microarchitecture where a TEST or CMP instruction followed by certain conditional branch instructions could be combined into a single µop. This is obviously good for power by reducing the number of operations that flow down the pipeline, and it is good for performance by eliminating the latency between the TEST/CMP and the branch.

"Macrofusion is a capability introduced in the Core 2 microarchitecture where a TEST or CMP instruction followed by certain conditional branch instructions could be combined into a single µop."

In this generation, we extend macrofusion in two ways. First, Core 2 would only macrofuse operations in 32-bit mode. In this generation, macrofusion can be applied in both 32-bit and 64-bit mode. Second, the number of conditional branch cases that support macrofusion was increased. Newly supported macrofusion cases in this generation are a compare (CMP) instruction followed by these instructions:

- JL (Jump if less than)
- JGE (Jump if greater than or equal)
- JLE (Jump if less than or equal)
- JG (Jump if greater)

Another area in the front end where we sought to improve both power efficiency and overall performance was in the Loop Stream Detector (LSD). The motivation behind the LSD is fairly straightforward. Short loops that execute for many iterations are very common in software. In a short loop, the front end is fetching and decoding the same instructions over and over, which is not power efficient. The LSD was created to capture these short loops in a buffer, reissue operations from that buffer, and then reduce power by disabling pieces of logic that are not needed—since their work is captured in the state of the buffer. Figure 3a shows the LSD as it existed in the Core 2 microarchitecture. The branch prediction and instruction fetch sub-blocks are powered down when running out of the LSD.

"In a short loop, the front end is fetching and decoding the same instructions over and over, which is not power efficient. The LSD was created to capture these short loops in a buffer, reissue operations from that buffer, and then reduce power by disabling pieces of logic that are not needed."

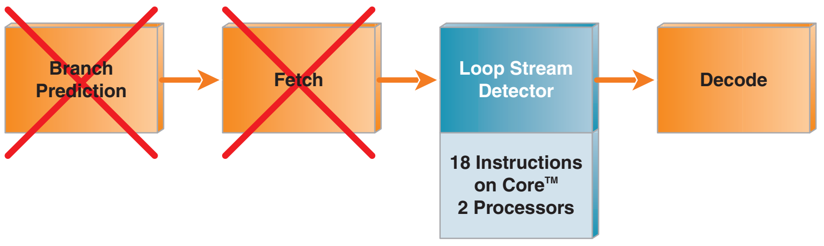

Figure 3a: Loop Stream Detector in the Core 2 microarchitecture

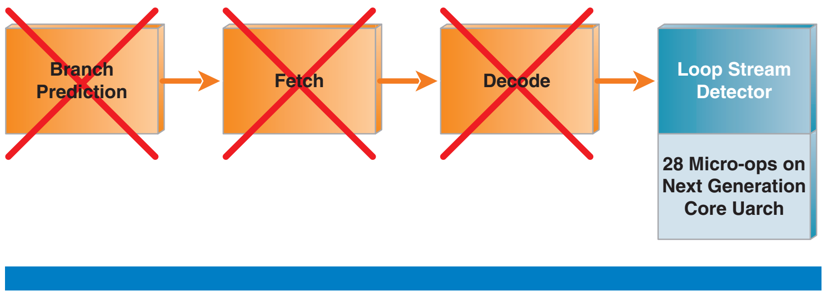

Figure 3b: Loop Stream Detector in this microarchitecture generation

In this microarchitecture generation, we made two key improvements to the LSD. First, we move the LSD to a later point in the pipeline, as seen in Figure 3b. By moving it after the decoders, we can now turn off the decoders when running from the LSD, allowing even more power to be saved. This change also provides for higher performance by eliminating the decoders as a possible performance bottleneck for certain code sequences. Second, by moving the location of the LSD, we can now take advantage of a larger buffer. In the prior architecture generation, the LSD could cover loops of up to 18 instructions. Now, in this generation, loops of up to 28 μops in length can be executed out of the LSD. Our internal studies show that, on average, the number of μops per instructions is nearly 1, so the 28 μop deep buffer is virtually equivalent to a 28-instruction deep buffer.

"In the prior architecture generation, the LSD could cover loops of up to 18 instructions. Now, in this generation, loops of up to 28."

Out of Order and Execution Engine

The OOO and Execution Engine cluster are responsible for scheduling and executing operations. In this generation, we looked to improve overall performance by exploiting greater levels of parallelism.

To exploit greater parallelism during execution, we need the processor core to scan across a greater range of operations to identify cases that are independent and ready to execute. In this generation of processors, we increased the size of the OOO window that the hardware scans by 33% from 96 operations in the Core 2 microarchitecture to 128 operations. This window is implemented as the Reorder Buffer (ROB), which tracks all operations in flight. Because the pipeline depth in this processor generation is effectively the same as in the prior generation, the entire benefit of this increase is used for performance instead of for simply covering a deeper pipeline.

"We increased the size of the OOO window that the hardware scans by 33% from 96 operations in the Core 2 microarchitecture to 128 operations."

Additionally, when increasing the size of the ROB, we need to increase the size of other corresponding buffers to keep the processor well balanced. If we do not make those increases, all we would be doing is shifting the location of the performance bottleneck and not actually getting any value out of the larger

ROB. Consequently, in this generation, we also increased the load buffer by 50% and the store buffer sizes by more than 50%. The increase in the size of these buffers was proportionally greater than the increase in the ROB size, based on data we collected that showed that performance bottlenecks in the previous technology generation were too often caused by the limited number of load and store buffers rather than by the ROB. Table 1 summarizes the sizes of these structures in this generation of products as compared to those in the prior product generation.

"We also increased the load buffer by 50% and the store buffer sizes by more than 50%."

Structure	**Core 2**	**Next-Generation (Nehalem/Westmere)**	**Comment**
Reservation Station	32	36	Dispatches operations to execution units
Load Buffers	32	48	Tracks all load operations allocated
Store Buffers	20	32	Tracks all store operations allocated

Table 1: Size of Key Structures

Operations are scheduled from our unified reservation station (RS). "Unified" means that all operations, regardless of type, are scheduled from this single RS. We increased the size of the RS from 32 to 36 entries in this processor generation as another means of expanding the amount of parallelism that can be exploited.

"Operations are scheduled from our unified reservation station (RS). "Unified" means that all operations, regardless of type, are scheduled from this single."

The RS is capable of scheduling an operation every cycle on each of its six execution ports:

"The RS is capable of scheduling an operation every cycle on each of its six execution ports."

- Ports 0, 1, 5: Integer operations plus Floating Point/SSE operations.
- Port 2: Load operations.
- Port 3: Store Address operations; store operations are broken into two pieces: an address operation and a data operation.
- Port 4: Store data operation.

This number of execution ports is the same as in the prior generation microarchitecture. The RS and its connection to the execution units are shown in Figure 4.

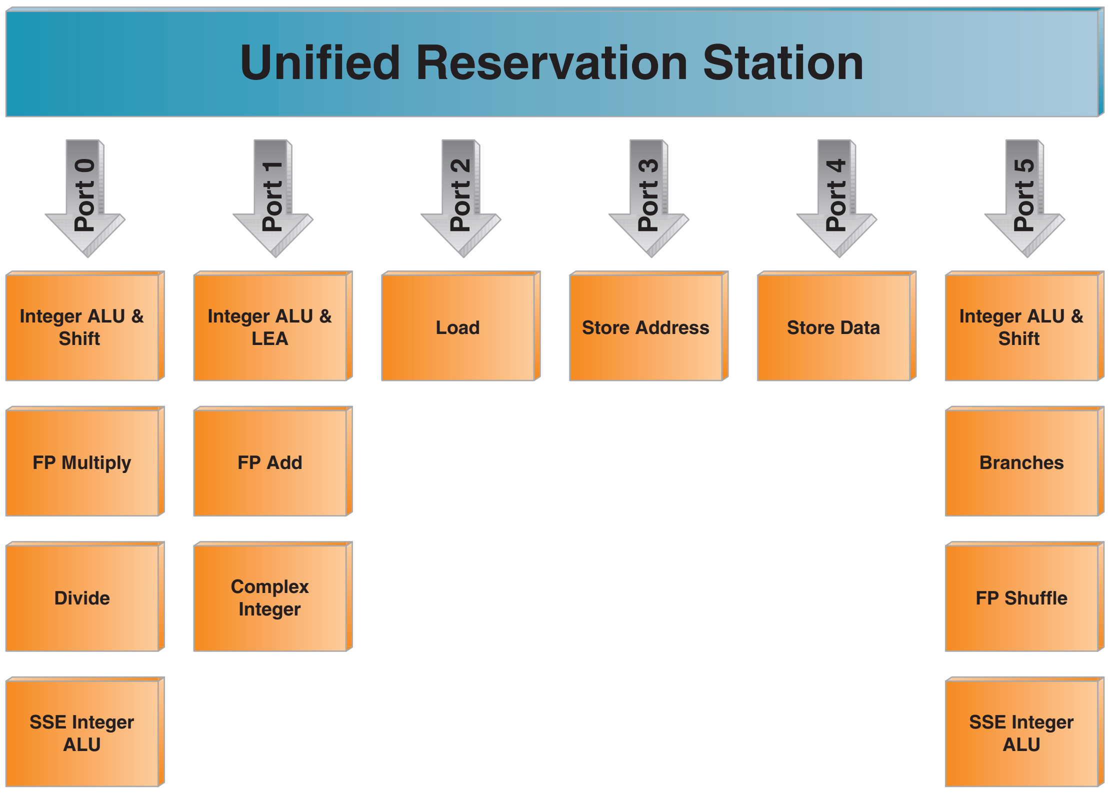

Figure 4: Reservation station and execution units

"In the prior generation microarchitecture, the new stream of instructions that is fetched on the correct path after a mispredicted branch was not allowed to allocate into the ROB, until the mispredicted branch in question was retired."

Another major performance improvement made in this microarchitecture generation was to reduce the cost of mispredicted branches. Mispredicted branches still have to flush instructions from the wrong path from the pipeline and cause a new stream of instructions to be fetched, so the minimum latency impact from a mispredicted branch is not affected. However, in the prior generation microarchitecture, the new stream of instructions that is fetched on the correct path after a mispredicted branch was not allowed to allocate into the ROB, until the mispredicted branch in question was retired. In this generation of microarchitecture, we remove that restriction and allow the new stream to allocate and execute without regards to whether the mispredicted branch in question has retired. By removing this limitation, significant latency can be saved in certain cases. For example, if a load operation misses all on-die caches and has to read memory, it may take hundreds of cycles to complete. Following that load may be a branch instruction that is not dependent on the load. If that branch mispredicts, in the prior generation microarchitecture, the new stream of instructions could not execute until the branch retired, which meant that the long latency load also had to complete, effectively creating a dependence between the branch and the load. In this new technology generation, no such dependence is created. This allows the new stream of instructions to start executing in parallel with the completion of the load that missed the caches.

Memory Subsystem

The memory cluster is responsible for handling load and store operations. The most noticeable change in the memory subsystem is in the cache hierarchy. In the prior generation of microarchitecture, the first level of cache was private to a core while the second level of cache was shared between pairs of cores. In this generation, we moved to having two levels of cache that are dedicated to the core and a third level that is shared between cores and is located in the Uncore.

"In this generation, we moved to having two levels of cache that are dedicated to the core and a third level that is shared between cores."

The first-level data cache remains 32 kilobytes in size and continues to use a writeback policy. Also, we left the cache line size and associativity the same—64 bytes and 8-ways, respectively. The load-to-use latency of the first-level data cache increased from three cycles to four cycles in this generation. We added a new second level, or mid-level, cache (MLC) that is 256 kilobytes in size, also with 64-byte lines and 8-way set associative. It is also a writeback cache. The MLC was designed to achieve high performance through a low, 10-cycle, load-to-use latency. The MLC is a unified cache, holding both instructions and data. Cache misses from both the first-level instruction cache and from the first-level data cache look up the MLC on cache misses.

"We added a new second level, or mid-level, cache (MLC) that is 256 kilobytes in size."

"Cache misses from both the first-level instruction cache and from the first-level data cache look up the MLC on cache misses."

The rationale for adding a second level of cache dedicated to each core was twofold:

- We decided to move to having a last-level cache shared between all cores. Having such a shared cache allows the entire cache to be used by any subset of the cores, in line with our goal of not penalizing applications that cannot take advantage of all cores. Therefore, we needed the MLC to buffer the last-level shared cache from a high rate of requests coming from all of the cores. This bandwidth buffering also enables greater scalability in core counts, so that as we build products with larger core counts, we do not necessarily need to make any changes to the CPU core.
- Provide a middle latency alternative between the very fast first-level cache and the third-level cache that would be much slower. By adding this cache, we optimize the overall effective latency (weighted latency average) across a wide range of workloads.

Beyond the caching hierarchy, we also updated the Translation Lookaside Buffer (TLB) hierarchy inside the core with the addition of a second-level TLB (STLB). The STLB is 512 entries in size and can hold both instruction-page and data-page translations. The workloads that benefit most from this structure have large data and code working sets, for example, sets often seen in high-performance computing and database workloads. By adding the STLB, numerous page walks can be eliminated, resulting in a performance gain—as page walks can be costly operations.

"We also updated the Translation Lookaside Buffer (TLB) hierarchy inside the core with the addition of a second-level TLB (STLB)."

"By adding the STLB, numerous page walks can be eliminated, resulting in a performance gain."

"This microarchitecture (Westmere family) also adds support for 1GB pages."

In addition to the STLB, the 32nm version of this microarchitecture (Westmere family) also adds support for 1GB pages. Prior to this technology generation, the page sizes supported were 4KB, 2MB, and 4MB. With the appropriate operating system support, larger page sizes offer the opportunity for higher performance by again reducing the number of page walks that are needed to read a given page. Table 2 details each level of the TLB hierarchy.

	# of Entries
first Level Instruction TLBs	
Small Page (4k)	128
Large Page (2M/4M)	7 per thread
first Level Data TLBs	
Small Page (4k)	64
Large Page (2M/4M)	32
New 2nd Level Unified TLB	
Small Page Only	512

Table 2: TLB Hierarchy Description

Another set of memory cluster optimizations made in this microarchitecture generation revolved around unaligned memory accesses to the first-level data cache. Specifically, two optimizations were made to improve performance on these types of operations.

The first optimization relates to 16-byte SSE vector load and store operations. The SSE architecture defines two forms of 16-byte memory accesses, one that can be used when the memory location being accessed is aligned on a 16-byte boundary (for example, the MOVDQA instruction), and a second form that allows any arbitrary byte alignment on these operations (for example, the MOVDQU instruction). The latter case is important because compilers often have to be conservative when generating code; they cannot always guarantee that a memory access will be aligned on a 16-byte boundary. Prior to this, the aligned form of these memory accesses had lower latency and higher throughput than the unaligned forms. In this new microarchitecture generation, we optimized the 16-byte unaligned memory access instruction to have the same latency and throughput as the aligned version, for cases that are aligned. By doing this, compilers are free to use the unaligned form in all cases and not have to worry about checking for alignment considerations.

"We optimized the 16-byte unaligned memory access instruction to have the same latency and throughput as the aligned version, for cases that are aligned."

"We took significant steps to reduce the latency of these cache-line split accesses through low-level, microarchitectural techniques."

The second optimization in the first-level data cache is for memory accesses that span a 64-byte cache line. As vectorization becomes more pervasive, we are seeing the need to improve the performance on these operations, as the compiler, again, cannot guarantee alignment of operations in many cases. With this generation of processors, we took significant steps to reduce the latency of these cache-line split accesses through low-level, microarchitectural techniques.

The final area of optimization in the memory subsystem was synchronization latency. As more threads are added to systems, and as software developers recognize that there are significant performance gains to be had by writing parallel code, we want to ensure that such code is not severely limited by the need to synchronize threads. To this end, we have been working to reduce the latency of cacheable LOCK and XHCG operations, as these are the primitives used primarily for synchronization. In this processor generation, we reduce both the latency and side-effect costs of these operations. The significant reduction in latency was achieved through careful re-pipelining. We also worked to minimize the pipeline stalls that these synchronization operations caused. Prior to this technology, all memory operations younger than the LOCK/XCHG had to wait for the LOCK/XCHG to complete. However, there was no architectural reason that we had to be this conservative, so this time around, we allow younger load operations to proceed even while an older LOCK/XCHG is still executing, as long as the other instructions do not overlap with the LOCK/XCHG in the memory they are accessing. This improvement does not show up in the latency of the LOCK/XCHG instructions, but it does show up in overall performance by allowing subsequent instructions to complete faster than in previous processors.

"We have been working to reduce the latency of cacheable LOCK and XHCG operations, as these are the primitives used primarily for synchronization."

"Prior to this technology, all memory operations younger than the LOCK/XCHG had to wait for the LOCK/XCHG to complete."

Intel® Hyper-Threading Technology

Even with all of the techniques previously described, very few software applications are able to sustain a throughput near the core's theoretical capability of four instructions per cycle. Therefore, there was still an opportunity to further increase the utilization of the design. To take advantage of these resources, Intel Hyper-Threading Technology, which was first implemented in the Intel Pentium® 4 processor family, was re-introduced to improve the throughput of the core for multi-threaded software environments in an extremely area- and power-efficient manner.

"Very few software applications are able to sustain a throughput near the core's theoretical capability of four instructions per cycle."

"Intel Hyper-Threading Technology, which was first implemented in the Intel Pentium® 4 processor family, was re-introduced."

The basic idea of Intel Hyper-Threading Technology is to allow two logical processors to execute simultaneously within the core. Each logical processor has its own software thread of execution state. Because a single software thread rarely fully exploits the peak capability of the core on a sustained basis, the central idea was that by introducing a second software thread, we could increase the overall throughput of the CPU core. This design yields an extremely efficient performance feature, since a minimal amount of hardware is added to provide this performance improvement.

"Because a single software thread rarely fully exploits the peak capability of the core on a sustained basis, the central idea was that by introducing a second software thread, we could increase the overall throughput of the CPU core."

Several key philosophies went into the design of Intel Hyper-Threading Technology in this processor generation:

- When Hyper-Threading Technology is enabled, but only a single software thread is scheduled to a core, the performance of that thread should be basically identical to when Intel Hyper-Threading Technology is disabled. This behavior was achieved by making sure that resources that are shared or partitioned between logical processors can be completely used when only one thread is active.

- There are points in the processor pipeline where the logic has to either operate on one thread or the other. When faced with these arbitration points, design the selection algorithm to achieve fairness between the threads. Therefore, if both threads are active and have work to do, the arbitration points will use a "ping-pong" scheme to switch between the two threads on a cycle-by-cycle basis.
- When faced with an arbitration point in the pipeline, if only one thread has work to do, allow that thread to get full bandwidth. It is often the case that at an arbitration point in the pipeline, only one thread has work to do. If that case occurs, we designed the core to give full bandwidth to that thread until the other thread has work to do. In other words, we do not unnecessarily constrict throughput by ping-ponging between the threads unless both threads are active and have work to do.

"We do not unnecessarily constrict throughput by ping-ponging between the threads unless both threads are active and have work to do."

Given these principles, decisions still had to be made on how various structures are handled in the face of Intel Hyper-Threading Technology. Four different policies were available for managing structures and are summarized in Table 3:

- **Replicated:** For structures that were replicated, each thread would have its own copy of the structure. If only a single thread were active in the core, the structures for the other thread would be unused. The most obvious example of this type of scheme is in the case of the architectural state. Each thread must maintain its own architectural state so the structures that hold that state, such as the retired RF, must be replicated. Cases that are replicated represent a real area and power cost for Hyper Threading Technology, but the number of cases where structures are replicated is limited.
- **Partitioned:** For structures that are partitioned, when two threads are active, each one is able to access only half of the structure. When only a single thread is active, we make the entire structure available to that thread. Prime examples where we partitioned structures are the various control buffers: reorder, store, and load. By partitioning structures, we guarantee each thread a set of resources to achieve reasonable performance. Moreover, partitioning of structures typically comes at only a small cost for managing the structures, but without any increase in the area or the power of the structures themselves.
- **Competitively shared:** For structures that are competitively shared, we allow the two threads to use as much of the structure as they need. The best example of this scheme is the caches on the processor. In the caches, we do not limit a thread to a percentage of the cache. Therefore, if two threads are active, we allow one thread to occupy a majority of the cache, if that is what its dynamic program behavior demands.
- **Unaware:** Finally, there are parts of the core that are completely unaware that Hyper-Threading Technology exists. The execution units are the best example of this, where the computed result is not affected by which thread is doing the computation.

"By partitioning structures, we guarantee each thread a set of resources to achieve reasonable performance."

Policy	Description	Examples
Replicated	Duplicate logic per thread	Register State Renamed RSB Large Page ITLB
Partitioned	Statically allocated between threads	Load Buffer Store Buffer Reorder Buffer Small Page ITLB
Competitively Shared	Depends on thread's dynamic behavior	Reservation Station Caches Data TLB 2nd level TLB
Unaware	No impact	Execution units

Table 3: Comparison of Intel® Hyper-Threading Technology Policies

The key arbitration points in the pipeline that we needed to consider when implementing Hyper-Threading Technology are these:

- **Instruction Fetch:** We support only one read of the Instruction Cache per cycle; therefore, we need to arbitrate between the two threads to decide which one gets to read the cache in any given cycle.
- **Instruction Decode:** In any given cycle, the bytes from only one thread can be decoded into µops.
- **Allocation:** We can allocate resources (ROB entries, store/load buffer entries, RS slots) only to a single thread each cycle.
- **Retirement:** Similarly, when we retire µops and reclaim their resources, we can only work on a single thread in a given clock cycle.

Note that the RS is not a point of arbitration between the threads. The RS schedules µops to the execution units without regard to which thread they belong to. Instead, its decisions are based upon which instructions are ready to execute and then choosing the "oldest," as measured by when they allocated into the RS.

The overall performance benefit of Hyper-Threading Technology varies depending on the nature of the software application in question. Some applications do not benefit from this technology. For example, applications or benchmarks, like Streams, that are memory bandwidth-bound when not using Hyper-Threading Technology will likely not see a performance benefit when Hyper-Threading is enabled, because no additional memory bandwidth

"The overall performance benefit of Hyper-Threading Technology varies depending on the nature of the software application in question."

is made available by using this technology. Another category of applications that will not benefit are those that already operate near the peak throughput of four instructions per clock or near the peak of another execution's resource, since that means that no idle resources can be exploited by the other thread. Linpack* is a primary example of a benchmark that falls into this category [3].

However, since there are very few workloads that operate near the CPU's peak IPC capabilities, there is ample opportunity for software to benefit from Intel Hyper-Threading Technology. Various reports on real-world workloads have shown that the performance boost can vary significantly based on the workload. Facebook has shown 15% higher throughput on one of their production workloads due to the use of Hyper-Threading Technology [4]. Citrix Systems has shown that Hyper-Threading Technology provides a 57% increase in the number of users that could be consolidated onto a server [5]. Additionally, measurements on the SpecMPI2007* benchmark suite, a high-performance computing proxy, have shown gains ranging from −2% to 35% depending on the specific benchmark with an overall 9% impact on the final benchmark score [6].

"Citrix Systems has shown that Hyper-Threading Technology provides a 57% increase in the number of users that could be consolidated onto a server."

New Instructions and Operations

Another means of improving performance in a power-efficient manner is through the addition of new instructions that software can exploit. New instructions improve overall power efficiency by completing in a single instruction a task that previously was handled by multiple instructions. We can achieve higher performance at lower power through the addition of these new instructions. Of course, exploiting the benefit of the new instructions requires software to be rewritten or re-compiled. Pre-existing software will not see a benefit from these instructions.

"New instructions improve overall power efficiency by completing in a single instruction a task that previously was handled by multiple instructions."

New instructions were introduced in both the 45nm (Nehalem) and 32nm (Westmere) versions of the core. The instructions were chosen because they offered significantly higher performance on critical and common operations in computing today.

In the 45nm version of the Nehalem microarchitecture, seven new instructions were added for performance, as well as new virtualization functionality and new timestamp counter (TSC) functionality. Full specification details on these instructions can be found in *Intel® 64 and IA-32 Architectures Software Developer's Manual Volumes 2A and 2B* [7]. The seven new instructions were branded as SSE4.2 and comprise the following:

"Seven new instructions were branded as SSE4.2."

- PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM: These four instructions accelerate manipulation operations on text strings. Such operations can be helpful, for instance, when parsing a stream of XML or doing regular expression comparisons. These instructions offer powerful capabilities such as identifying substrings within a string, finding a specific character in a string, or finding a range of characters, such as finding all numbers, in a string. The instructions operate on the 128-bit SSE register set, allowing for processing of 16 bytes at a time per operation. An example usage has shown roughly a 25% average throughput improvement on parsing XML by use of these new instructions [8].

- POPCNT: This instruction returns the number of 1's that are set in an integer register. This is a common operation in recognition and search algorithms.
- CRC32: This instruction accelerates the computation of a Cyclic Redundancy Check (CRC) by using the iSCSI polynomial, which is used in several networking protocols [9]. Additionally, this type of operation can also be used to provide a fast hash function.
- PCMPGTQ: This instruction compares two SSE integer vector registers and checks for a "greater than" condition.

"This instruction accelerates the computation of a Cyclic Redundancy Check (CRC) by using the iSCSI polynomial, which is used in several networking protocols [9]. Additionally, this type of operation can also be used to provide a fast hash function."

Two other architecture-visible features were added to the Nehalem microarchitecture. Specifically, support was added for a constant time stamp counter, which aids timing across deep sleep states. Additionally, enhanced page table (EPT) support was added that allows virtualization vendors to avoid shadow page tables, removing a source of performance overhead when running in virtualized environments. Coupled with EPT, two instructions (INVVPID, INVEPT), were added to the architecture to support invalidations of EPT translation caches.

In the Westmere architecture, seven new instructions were added that focus on providing instructions to accelerate the performance of cryptographic operations. These operations are prevalent today in computing in areas such as Web commerce and in setting up secure transactions between a host and a client. The seven new instructions are as follows:

"In the Westmere architecture, seven new instructions were added that focus on providing instructions to accelerate the performance of cryptographic operations."

- AESENC, AESENCLAST, AESDEC, AESDESCLAST, AESKEYGENASSIST, AESIMC: This collection of six instructions can be used together to accelerate encryption and decryption by using the Advanced Encryption Standard (AES) [10]. With these instructions, performance gains three to ten times better than previous-generation technology performance have been achieved on commercial software [11].
- PCLMULQDQ: This instruction performs a carryless multiply operation. A Galois (binary) field multiply consists of the carryless multiply followed by a reduction. Galois fields are common in cryptographic operations, such as AES as well as cyclic redundancy checks, elliptic curve cryptography, and error correcting codes. AES Galois Counter Mode is probably the most widely known application of PCLMULQDQ [12].

"Galois fields are common in cryptographic operations, such as AES as well as cyclic redundancy checks, elliptic curve cryptography, and error correcting codes."

- Also in the Westmere processor architecture, the page table support, when running in virtualized environments, was further enhanced to support real-mode guests. This design substantially reduces the time required to boot virtual machines through BIOS as well as reduces the time to execute device option ROMs.

These new instructions focus on accelerating specific usage models for critical and common tasks. Going forward, we will continue to look for opportunities for such targeted acceleration opportunities, as well as more general-purpose instruction-set additions, such as the upcoming Intel Advanced Vector Extensions (AVX). Full details on AVX are available in the *Intel Advanced Vector Extensions Programming Reference* [13].

Segment-Specific Features

In this section, we detail a few examples of new segment-specific features, and we discuss some of the tradeoffs that were made in their design.

Physical and Virtual Addressing

The client segment platforms, mobile and desktop, typically support a maximum of 4GB-8GB of main memory; large sever systems need to support very large memory capacities, on the order of 100 GBytes or more. This wide range of requirements provides a very stark set of tradeoffs. Additional address bits needed for large memory systems have little value in the client segments, but they do consume extra power. In addition to increasing power consumption and adding area, additional address bits can cause stress on the timing of critical paths in the machine, such as address generation for cache look-ups. To strike a balance between how many address bits to add, we settled on a modest two physical address bits over the Core 2 microarchitecture, but we did not increase the virtual address bits from the number in previous-generation technology. Our rationale was that physical address bits are critical to enable certain large systems, and that the number of virtual address bits was already adequate to enable operating systems for the server systems that needed to be built.

"We settled on a modest two physical address bits over the Core 2 microarchitecture, but we did not increase the virtual address bits from the number in previous-generation technology."

Reliability Features

To satisfy large server systems, we enhanced the reliability features found in the prior-generation architecture. Reliability features come in two major categories: enhancements to the Machine Check Architecture (MCA) and enhancements to the design and microarchitecture. To achieve the targets for "soft error rates" we set to enable scaling to the socket and processor counts needed. For both of these feature additions, it is important to keep in mind that the cores were designed to meet the requirements of high-end, multi-processor servers with 8-10 cores per socket and up to 8 sockets. Clearly the requirements set by these high-end server systems far exceed the ones that are set by the client segments.

"Because the Westmere core would be used mostly in the same platforms as the Nehalem core, we opted against adding additional addressing bits in the Westmere version."

For machine-check enhancements, we added more reporting capabilities for various failure modes to increase coverage and insights into state for diagnostic software. The cost of MCA features is not high in power, area, or design effort; however, due to its complexity it is a fairly substantial cost against the project's validation budget. A reasonable balance of added features included looking at the tradeoffs between value and validation cost.

We also added Soft Error Rate (SER) features to meet scalability requirements. Deciding what SER features to add was arrived at by carefully modeling possible failure behavior and also looking at a careful modeling of possible and critical failure behaviors to determine the protection of specific micro-architectural assets. The SER features we added included items such as parity on various structures. To further add high-end server reliability, we also added a mode where the first-level data cache operates at half the size with additional data reliability protection enabled. This cache mode is a customer-enabled option for the expandable server (EX) product line, where customers are willing to trade some performance for additional reliability.

Power Features

Low average power is important to battery life for traditional productivity workloads and for battery life for specific usages, like video, DVD, or streaming media playback usages. For this generation of processors, we modeled the impact of both of these usage cases when making feature decisions. At a high level, there are two aspects to achieving a good average power behavior: 1) being able to run at a low power for a minimum performance point, and 2) being able to transition in and out of low power states quickly, making sure that the power consumed in these low power states is as low as possible.

Achieving a low-power operating point is valued by the server segment, where it enables more cores to be aggregated in a socket. To achieve this operating point, we typically work on lowering the power consumption in general and specifically lowering the minimum voltage at which the core can operate (MinVCC). Having a low MinVCC provides a very efficient way to reduce the power due to the cubic scaling we discussed earlier. Due to the cubic benefit from lowering the operating voltage, even small changes have very beneficial effects. There is typically an area tradeoff involved in the device sizing needed to achieve a low MinVCC, which is a factor to take into account for the overall product cost.

Efficient low-power states, also known as C-states, and quick entry and exit into and out of those states are features that are fairly uniquely valued by the client mobile segment. At a high level, the lower the power is for a C state, the more power we save. Moreover, the faster we can enter and exit these states, the more often we can utilize them without harming overall system performance characteristics. We added new low-power states, made the power in those states lower, and optimized entry and exit latencies.

"For machine-check enhancements, we added more reporting capabilities for various failure modes to increase coverage and insights into state for diagnostic software."

"Cache mode is a customer-enabled option for the expandable server (EX) product line, where customers are willing to trade some performance for additional reliability."

"Efficient low-power states, also known as C-states, and quick entry and exit into and out of those states are features that are fairly uniquely valued by the client mobile segment."

"We added new low-power states, made the power in those states lower, and optimized entry and exit latencies."

Integration and Scalability

We also added features to make it easier to integrate cores into a large scalable processor and to better utilize the power for such multi-core systems.

To achieve better performance scaling in systems with high core counts, we added the previously mentioned MLC to reduce the bandwidth requirement of the core-Uncore interface. The benefit of the MLC is two-fold. It adds performance by, on average, improving the perceived latency of memory accesses through its low latency and by not overtaxing the bandwidth at the shared last-level cache in the Uncore.

Additionally, the Nehalem generation of processors supports buffers that allow the core to independently run at a different frequency than the Uncore they are attached to. This is a key power-efficiency feature where we can adjust the frequency of the core and the Uncore individually to operate at the best power/performance efficiency point. For example, we can set the Uncore frequency to match the desired performance of external interfaces, such as memory speed, while we let the cores independently run at the frequency that is demanded by the task, as asked for by the operating system.

"We can set the Uncore frequency to match the desired performance of external interfaces, such as memory speed, while we let the cores independently run at the frequency that is demanded by the task."

Building on the ability to dynamically run the cores at a different frequency, and isolating a core from the Uncore, this generation of microarchitecture also supports power gates, where the power to the cores can be completely turned off when the core is not being used. This feature is important for mobile average power as it prevents draining of the battery when the core is doing nothing. Power gates are also key enablers for Intel Turbo Boost Technology which dynamically allows active cores to run at higher frequencies. Fundamentally, the turbo boost feature allows products built with this generation of cores to dynamically use the full-power envelope without artificially limiting the performance of the cores. With this technology, the frequency of active cores can scale up if there is power headroom. Power gates help this by eliminating the power overhead of idle cores. However, even when all cores are active, Intel® Turbo Boost Technology can kick in and increase the frequency of all cores if the workload being run is not power intensive. This technology helps satisfy the philosophy of not penalizing customers whose software does not take advantage of all cores.

"The power to the cores can be completely turned off when the core is not being used."

"The frequency of active cores can scale up if there is power headroom."

Conclusion

The Nehalem and Westmere architecture is designed to work efficiently for a wide range of operating points, from mobile to high-end server. High-value, segment-specific features were added, while at the same time taking care that the power overhead could be absorbed by all segments. Intel achieves the needed increase in efficiency of both power and performance through appropriate buffer size increases and more intelligent algorithms. These changes were made without fundamentally affecting the pipeline latencies of the baseline design. The changes deliver exciting features and a superior performance increase inside equal or lesser power envelopes.

References

[1] Harikrishna Baliga, Niranjan Cooray, Edward Gamsaragan, Peter Smith, Ki Yoon, James Abel, Antonio Valles. "The Original 45-mn Intel® Core™ 2 Processor Performance," *Intel Technology Journal*. Available at ***http://download.intel.com/technology/itj/2008/v12i3/paper4.pdf***

[2] David L Hill, Derek Bachand, Selim Bilgin, Robert Greiner, Per Hammarlund, Thomas Huff, Steve Kulick, and Robert Safranek. "The Uncore: A Modular Approach to Feeding the High-Performance Cores, *Intel Technology Journal*, Volume 15, Issue 01, 2011.

[3] Pawel Gepner, Michal F. Kowalik, David L. Fraser, Kazimierz Wackowski. "Early Performance Evaluation of New Six-Core Intel® Xeon® 5600 Family Processors for HPC," ispdc, pp. 117–124, 2010 *Ninth International Symposium on Parallel and Distributed Computing*, 2010.

[4] *Real-World Application Benchmarking. Available at* ***http://www.facebook.com/note.php?note_id=203367363919***

[5] *Nehalem and XenServer Raise the Bar for XenApp Performance.* Available at ***http://community.citrix.com/pages/viewpage.action?pageId=73564465***

[6] Bugge. H. "An evaluation of Intel's core i7 architecture using a comparative approach," Computer Science—Research and Development, Vol. 23(3–4), 203–209, 2009.

[7] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volumes 2A and 2B.*

[8] Chris Newburn. "High Performance XML Processing with Intel SSE4.2 Hardware and Software Algorithms*," XML in Practice*, 2008.

[9] Vinodh Gopal, et al. "Fast CRC Computation for iSCSI Polynomial Using CRC32 Instruction." Available at ***http://download.intel.com/design/intarch/papers/323405.pdf***

[10] Gueron, S. "Advanced encryption standard (AES) instructions set," *Technical report,* Intel Corporation, 2008. Available at ***http://softwarecommunity.intel.com/isn/downloads/intelavx/AES-Instructions-Set WP.pdf***

[11] Refer to ***http://www.tomshardware.com/reviews/clarkdale-aes-ni-encryption,2538-7.html***

[12] Gueron, S., et al. "Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode," *White Paper*, Intel Corporation, 2010.

[13] *Intel Advanced Vector Extensions Programming Reference. Available at* ***http://software.intel.com/file/19151***

Authors' Biographies

Martin Dixon is a Principal Engineer in the Intel Architecture Group, working closely with software partners to develop and enhance the overall instruction set. He received his B.S. degree in Electrical and Computer Engineering from Carnegie Mellon University.

Per Hammarlund is a Senior Principal Engineer in the Intel Architecture Group working on microarchitecture development, Intel Hyper-Threading Technology, power efficiency, and performance modeling. Per started at Intel in 1997 working on the Willamette processors (Pentium 4). He received his PhD degree in Computer Science from the Royal Institute of Technology (KTH) in 1996.

Stephan Jourdan is a Senior Principal Engineer in the Intel Architecture Group.

Ronak Singhal is a Senior Principal Engineer in the Intel Architecture Group working on microarchitecture development, Instruction Set Architecture development, and performance analysis. He joined Intel in 1997 and has worked on multiple generations of processors, starting with the initial Intel Pentium 4 processor. He received his B.S. and M.S. degrees in Electrical and Computer Engineering from Carnegie Mellon University.