Cache Organization and Memory Management of the Intel Nehalem Computer Architecture

Trent Rolf
University of Utah Computer Engineering
CS 6810 Final Project
December 2009

***Abstract*—Intel is now shipping microprocessors using their new architecture codenamed "Nehalem" as a successor to the Core architecture. This design uses multiple cores like its predecessor, but claims to improve the utilization and communication between the individual cores. This is primarily accomplished through better memory management and cache organization. Some benchmarking and research has been performed on the Nehalem architecture to analyze the cache and memory improvements. In this paper I take a closer look at these studies to determine if the performance gains are significant.**

I. Introduction

The predecessor to Nehalem, Intel's Core architecture, made use of multiple cores on a single die to improve performance over traditional single-core architectures. But as more cores and processors were added to a high-performance system, some serious weaknesses and bandwidth bottlenecks began to appear.

After the initial generation of dual-core Core processors, Intel began a Core 2 series processor which was not much more than using two or more pairs of dual-core dies. The cores communicated via system memory which caused large delays due to limited bandwidth on the processor bus [5]. Adding more cores increased the burden on the processor and memory buses, which diminished the performance gains that could be possible with more cores.

The new Nehalem architecture sought to improve core-to-core communication by establishing a point-to-point topology in which microprocessor cores can communicate directly with one another and have more direct access to system memory.

II. Overview of Nehalem

A. Architectural Approach

The approach to the Nehalem architecture is more modular than the Core architecture which makes it much more flexible and customizable to the application. The architecture really only consists of a few basic building blocks. The main blocks are a microprocessor core (with its own L2 cache), a shared L3 cache, a Quick Path Interconnect (QPI) bus controller, an integrated memory controller (IMC), and graphics core.

With this flexible architecture, the blocks can be configured to meet what the market demands. For example, the Bloomfield model, which is intended for a performance desktop application, has four cores, an L3 cache, one memory controller, and one QPI bus controller. Server microprocessors like the Beckton model can have eight cores, and four QPI bus controllers [5]. The architecture allows the cores to communicate very effectively in either case. The specifics of the memory organization are described in detail later.

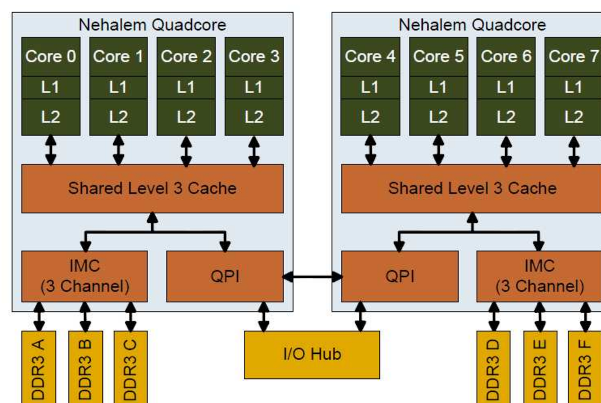

Fig. 1. Eight-core Nehalem Processor [1]

Figure 1 is an example of an eight-core Nehalem processor with two QPI bus controllers. This is the configuration of the processor used in [1].

B. Branch Prediction

Another significant improvement in the Nehalem microarchitecture involves branch prediction. For the Core architecture, Intel designed what they call a "Loop Stream Detector," which detects loops in code execution and saves the instructions in a special buffer so they do not need to be continually fetched from cache. This increased branch prediction success for loops in the code and improved performance. Intel engineers took the concept even further with the Nehalem architecture by placing the Loop Stream Detector *after* the decode stage eliminating the instruction decode from a loop iteration and saving CPU cycles.

C. Out-of-order Execution

Out-of-order execution also greatly increases the performance of the Nehalem architecture. This feature allows the processor to fill pipeline stalls with useful instructions so the pipeline efficiency is maximized. Out-of-order execution was present in the Core architecture, but in the Nehalem

architecture the reorder buffer has been greatly increased to allow more instructions to be ready for immediate execution.

D. Instruction Set

Intel also added seven new instructions to the instruction set. These are single-instruction, multiple-data (SIMD) instructions that take advantage of data-level parallelism for today's data-intensive applications (like multimedia). Intel refers to the new instructions as Applications Targeted Accelerators (ATA) due to their specialized nature. For example, a few instructions are used explicitly for efficient text processing such as XML parsing. Another instruction is used just for calculating check-sums.

E. Power Management

For past architectures Intel has used a single power management circuit to adjust voltage and clock frequencies even on a die with multiple cores. With many cores, this strategy becomes wasteful because the load across cores is rarely uniform. Looking forward to a more scalable power management strategy, Intel engineers decided to put yet another processing unit on the die called the Power Control Unit (PCU).

The PCU firmware is much more flexible and capable than the dedicated hardware circuit on previous architectures. Figure 2 shows how the PSU interacts with the cores. It uses sensors to read temperature, voltage, and current across all cores in the system and adjusts the clock frequency and supply voltage accordingly. This enables the cores to get exactly what they need, including putting a core to sleep if it is not being used at all.

While these and other features contribute to the performance and efficiency of a Nehalem processor, the remainder of this paper will focus on the cache organization, memory architecture, and communication between cores.

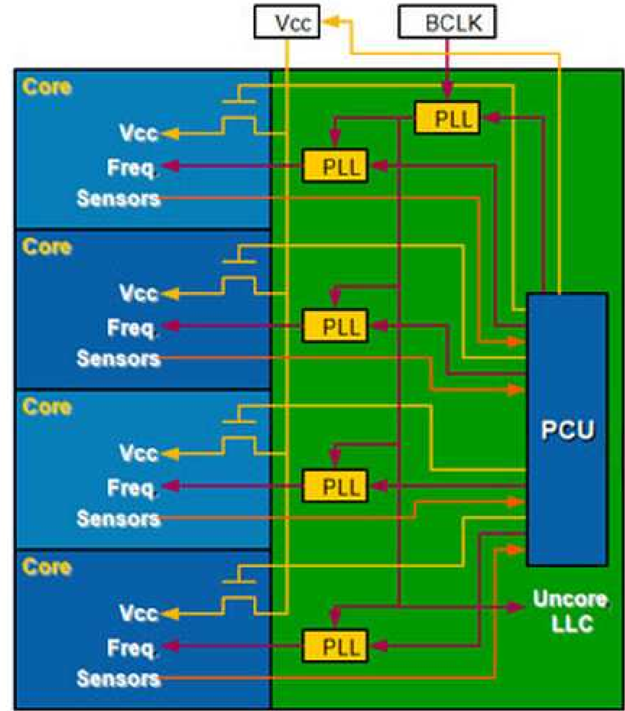

Fig. 2. Power Control Unit (PSU) in a Multi-core Nehalem Architecture [5]

III. Cache and Memory Specifics

A. Transition Lookaside Buffer

The transition lookaside buffer (TLB) plays a critical role in the cache performance. It is a high-speed buffer that maps virtual addresses to physical addresses in the cache or memory. When a page of memory is mapped in the TLB, it is accessed quickly in the cache. When the TLB is too small, misses occur more frequently. The TLB in the Nehalem architecture is much larger than previous architectures which allows for many more memory page references to remain in the TLB.

In addition, Intel made the TLB dual-level by adding an L2 TLB. The second-level TLB is larger than the first level and can store up to 512 entries [5]. The gains from the TLB changes are significant, but the most dramatic improvements come from the changes to the overall cache-memory layout.

B. Cache and Cache Coherency

In the Core architecture, each pair of cores shared an L2 cache. This allowed the two cores to communicate efficiently with each other, but as more cores were added it proved difficult to implement efficient communication with more pairs of cores. For the Nehalem architecture each core has its own L2 cache of 256KB. Although this is smaller than the L2 cache of the Core architecture, it is lower latency allowing for faster L2 cache performance.

Nehalem does still have shared cache, though, implemented as L3 cache. This cache is shared among all cores and is relatively large. For example, a quad-core Nehalem processor will have an 8MB L3 cache. This cache is inclusive, meaning that it duplicates all data stored in each individual L1 and L2 cache. This duplication greatly adds to the inter-core communication efficiency because any given core does not have to locate data in another processor's cache. If the requested data is not found in any level of the core's cache, it knows the data is also not present in any other core's cache.

To insure coherency across all caches, the L3 cache has additional flags that keep track of which core the data came from. If the data is modified in L3 cache, then the L3 cache knows if the data came from a different core than last time, and that the data in the first core needs its L1/L2 values updated with the new data. This greatly reduces the amount of traditional "snooping" coherency traffic between cores.

This new cache organization is known as the MESIF (Modified, Exclusive, Shared, Invalid, Forward) protocol, which is a modification of the popular MESI protocol. Each cache line is in one of the five states:

- **Modified** - The cache line is only present in the current cache and does not match main memory (dirty). This line must be written back to main memory before any other reads of that address take place.

Everest 4.60	Nehalem 3.2GHz			Penryn 3.2GHz	
	3 channel DDR3-1067 7-7-7-20	2 channel DDR3-1067 7-7-7-20	1 channel DDR3-1067 7-7-7-20	2 channel DDR3-1067 7-7-7-20	2 channel DDR3-1600 7-7-7-20
Read, MB/s	15057	14030	8199	8125	10101
Write, MB/s	14727	12038	8200	8464	8495
Copy, MB/s	15522	13314	8560	7207	8175
Latency, ns	39.2	34.9	33.5	64.2	51.7

Fig. 3. Comparison of Memory Channel Performance - Nehalem vs. Core 2 (Penryn model) [5]

- **Exclusive** - The cache line is only present in the current cache and matches main memory (clean).
- **Shared** - The cache line is clean similar to the exclusive state, but the data has been read and may exist in another cache. This other cache should be updated somehow if the line changes.
- **Invalid** - The cache line is invalid.
- **Forward** - This cache line is designated as the responder to update all caches who are sharing this line.

With the extra "Forward" state, the excessive responding among shared cache lines is eliminated.

C. Memory Controller

The location of the memory controller was a significant change from the Core processors. Previously the memory controller was located off-chip on the motherboard's northbridge, but Nehalem integrates the memory controller to the processor die with the hope to reduce the latency of memory accesses. In keeping with the modular design approach, Intel engineers introduced flexibility into the size of the memory controller and the number of channels.

The first Nehalem processors were the quad-core models which had a triple-channel memory controller. To show the effectiveness of this on-chip design, the authors of [5] performed memory subsystem tests in which they compared the new architecture to the Core 2 (Penryn model) architecture. They varied the number of channels for each architecture and found that even a single-channel Nehalem processor was faster than the dual-channel Core 2 system with an external memory controller. Figure 3 shows the exact results of the test.

Another benefit to an on-chip memory controller is that it is totally independent of the motherboard hardware. This provides the processor more predictable memory performance that will run just as fast on any hardware platform.

D. QuickPath Interconnect Bus

With the memory controller now located on the processor die, the load on the Front-side Bus (FSB) for a single-processor system has been greatly reduced. But for multi-processor systems (like servers) there is a need for faster and more direct chip-to-chip communication, and the FSB does not have the bandwidth to fill that need. So Intel developed the QuickPath Interconnect (QPI) bus as a means of connecting multiple processors to each other in addition to the chip sets.

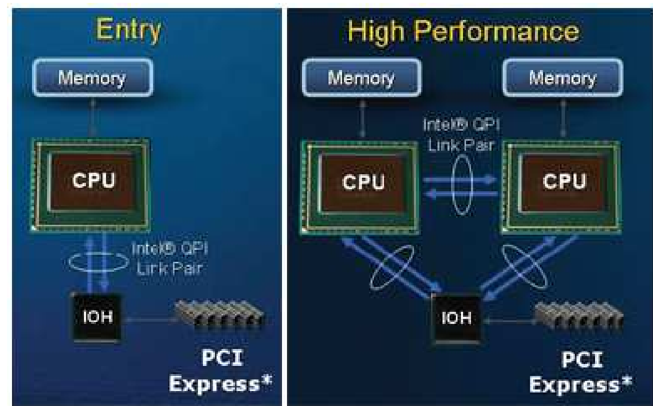

Fig. 4. Examples of QuickPath Interconnect in a Single and Multi-processor Nehalem System [5]

Figure 4 shows how this might work. On an entry-level Nehalem system with one processor the QPI bus becomes an improved FSB allowing for higher bandwidth communication between the processor and high-speed hardware like PCI Express. As more processors are added to the sytem, the QPI bus also provides an efficient point-to-point communication path between processors by facilitating high-speed non-uniform memory accesses (NUMA).

Now that we have given an overview of the most important improvements to the Nehalem architecture, let's take a closer look at some studies that have been performed on actual Nehalem processors that validate the performance improvement claims.

IV. A Study of Memory Performance and Cache Coherency

The paper *Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System* [1] is a study that focuses on the improvements to cache and memory. The authors have devised new benchmarks that can measure latency and bandwidth for accesses to main memory and the other processors' cache. Previous studies and benchmarks only measure memory bandwidth in general (for example, the STREAM benchmark [1]) without providing any cache-specific metrics. The authors consider their study to be the first to provide metrics that measure the effectiveness of the cache coherency protocol.

To perform these tests, the authors used BenchIT open-source performance measurement software (www.benchit.org). It compares algorithms and implementations of algorithms on a given architecture.

The details of the system under test are given in Figure 5, and a diagram of the system is shown in Figure 1. To remove unnecessary variables from the test, the authors disabled some hardware features such as dynamic overclocking, simultaneous multi-threading (SMT), and hardware prefetching.

A. Assembler Benchmarks

The first benchmarks that were performed on the system were a series of hand-optimized assembly routines compiled

Processors	2x Intel Xeon X5570
Core arrangement	Processor 0: core 0-3 Processor 1: core 4-7
Core frequency	2.933 GHz
Uncore frequency	2.666 GHz
QPI bandwidth	25.6 GB/s per link
L1 cache size	32 KiB/32 KiB
L2 cache size	256 KiB
L3 cache size	8 MiB
Cache line size	64 Bytes
Page size	4 KiB/2 MiB (small/huge pages)
L1 data TLB	48/32 entries for small/huge pages
L2 TLB	512 entries for small pages
Memory type	6x 2 GiB DDR3-1333, registered, ECC 3 channels per processor, 10.66 GB/s each
Operating System	Debian 5.0, Kernel 2.6.28.1
Compiler	gcc 4.3.2 and icc 11.0

Fig. 5. Test System Configuration for [1]

in-line with a test program written in C. These benchmarks are designed to run on 64-bit x86 processors, and by design execute code that would not be generated by a compiler. The authors are using a high-resolution time stamp counter that adds very little overhead to the measurements. The overhead is somewhat noticable in the higher-speed tests like the L1 test.

To make a meaningful evaluation of the memory and cache, the authors use the following techniques throughout the tests:

- Each thread of the benchmark program is "pinned" to a certain core.
- All memory pages of a given thread are physically located on the corresponding memory module to that core to help identify the effects of the NUMA architecture.
- Before the benchmarks begin, the caches can be placed in certain coherency states–modified, exclusive, or shared–by performing initial reads and writes.
- With the help of a special cache flush routine, the authors can invalidate an entire level of cache to isolate the performance of just one cache.
- To reduce the effects of the TLB on memory latency tests the authors use "huge pages" that prevent TLB misses for data sets of up to 64MB of memory.

In this assembler test, three specific benchmarks are taken for each iteration. First, the authors take a *latency benchmark* to determine the latency for accesses to main memory and all three cache levels. Next they measure a *single-core bandwidth benchmark* to determine how the cache coherency scheme affects memory performance on a single core. Finally they use a *multi-core bandwidth benchmark* that runs multiple threads simultaneously to demonstrate the effectiveness of the shared L3 cache and the integrated memory controller. The results of these benchmarks will be described in the following sections.

B. Modified STREAM Benchmarks

To verify the results obtained using the assembler benchmarks the authors use a modified version of the well-known set of benchmarks known as the STREAM benchmarks. In addition, the benchmarks are used to perform more complicated memory access patterns that are not easily implemented in the assembler benchmarks.

The unmodified STREAM benchmarks implement read/write tests using four different memory access patterns. These tests perform calculations on a one-dimensional array of double precision floating point numbers.

The authors made the following modifications to the original STREAM benchmarks to better compliment the assembler benchmarks:

- Every thread is bound to a specific core and allocates its own memory.
- The time stamp counter is used for time measurement which excludes overhead caused by the spawning of new threads.
- The authors added `#pragma vector aligned` preprocessor commands to the code to optimize memory accesses and `#pragma vector nontemporal` to perform explicit writes of data to main memory without affecting cache.

C. Latency Benchmarks

The latency benchmark uses pointer chasing to measure the latency for cache/memory accesses. It is perfomed using the following general strategy where thread 0 is running on core 0 and is accessing memory associated with core N.

First, thread 0 "warms up" the TLB by accessing all the required pages of memory. This ensures that the TLB entries needed for the test are present in core 0.

Next, thread N places the data in the caches of core N into one of the three cache coherency states described earlier (modified, exclusive, or shared).

The latency measurement then takes place on core 0 as the test runs a constant number of access and uses the time stamp counter to collect data. Each cache line is accessed once and only once in a pseudo-random order.

The following figures show the results of testing different starting cache coherency states among the different levels of the memory hierarchy. The measurements for the lines in the exclusive cache coherency state are shown in Figure 6. Figure 7 shows the lines in the modified state. Figure 8 shows a summary of the graphical results including the results from the shared state test.

The latency of the local accesses is the same regardless of the cache coherency state since a cache is always coherent with itself. The authors measure a latency of 4 clock cycles for L1 cache, 10 clock cycles for L2 cache, and 38 cycles for L3 cache.

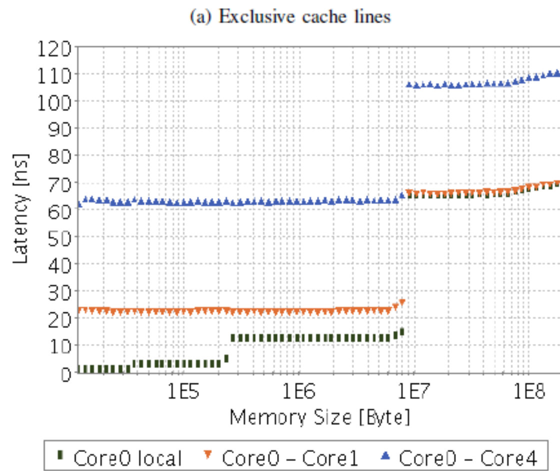

Fig. 6. Read latencies of core 0 accessing cache lines of core 0 (local), core 1 (on die) or core 4 (via QPI) with cache lines initialized to the exclusive coherency state[1]

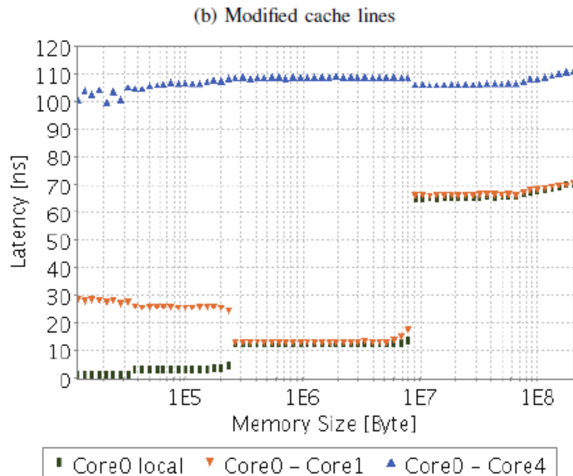

Fig. 7. Read latencies of core 0 accessing cache lines of core 0 (local), core 1 (on die) or core 4 (via QPI) with cache lines initialized to the modified coherency state[1]

The latency to a core on the same die shows a strong correlation to the cache coherency state. Shared cache lines can be accessed within 13 ns. This is because the inclusive L3 cache contains a valid copy of the data and can immediately respond to the request. Exclusive lines, however, may have been modified in the other core's higher level cache which forces the L3 cache to check the data in that core. This check costs 9.2 ns which increases the latency for reads to exclusive lines to 22.2 ns.

Data accesses to cores on a different processor suffer an additional penalty due to the transfer over the QPI bus. Any access off-chip to an exclusive cache line requires at least one snoop to the other core's cache/memory which takes 63 ns. The latency for shared lines is only slightly better at 58 ns due to the shared L3 cache. Latency for modified cache lines can be 100ns or more because the caches are forced to write back the value to main memory due to the cache coherency protocol.

Main memory was measured to have a latency of 65 ns for local accesses. When the memory of another core is accessed via the QPI bus it takes an additional 41 ns bringing the total latency to 106 ns.

D. Bandwidth Benchmarks

The tests for the bandwidth benchmarks were very similar to the latency tests. The caches and memory were initialized in the same way. But instead of using a timer the authors developed a measurement routine that can access any location in the memory hierarchy, including other cores' caches. Using this tool the authors were able to determine local, inter-core, and inter-processor bandwidth.

The test continuously accesses data using load and store assembly calls eliminating any arithmetic instructions. This way the bandwidth can be measured as accurately as possible.

It is important to note that the architecture does not allow one core to write directly to another core's cache. This has to be done in a two-step process. The core must read the value first to obtain ownership then write to its local cache. This must be considered when analyzing the results.

Each core in the architecture has a 128-bit write port and a 128-bit read port to the L1 cache. This means that at a clock rate 2.933 GHz (the core clock for the specific chip they used for the test) the theoretical limit for L1 cache bandwidth is 46.9 Gbps in each direction. The L2 and L3 caches each have a 256-bit port for reading or writing, but the L3 cache must share its port with three other cores on the chip.

The benchmarks also measure main memory bandwidth. Each integrated memory controller has a theoretical bandwidth peak of 32 Gbps.

Figure 9 shows the measured bandwidth (in Gbps) for different memory sizes where the cache has been initialized to the exclusive state. Figure 10 shows a similar test where the cache has been initialized to the modified state. The read bandwidth measurements are compared in Figure 11, and write bandwidth measurements are compared in Figure 12.

As with latency results, we should expect to see bandwidth the same independent of the cache coherency protocol. This is indeed the case with the L1 bandwidth measuring near its peak performance at 45.6 Gbps regardless of the coherency states.

As was expected, the write bandwidth is lower than the read bandwidth because a write is made up of a read and and write. This would seem like it would double the amount of traffic and cut bandwidth in half, but the measurements suggest that the penalty is not that large. The authors suggest that this is due to the large 256-bit interfaces on the L2 cache.

The bandwidth to *other* cores does have strong ties to the cache coherency state. Reading or writing data that hits in the L3 cache achieves a bandwidth of 26.2 Gbps for reading and 19.9 Gbps for writing. If the core must get the data from

Source	Exclusive cache lines			Modified cache lines			Shared cache lines			RAM
	L1	L2	L3	L1	L2	L3	L1	L2	L3	
Local	1.3 (4)	3.4 (10)	13.0 (38)	1.3 (4)	3.4 (10)	13.0 (38)	1.3 (4)	3.4 (10)	13.0 (38)	65.1
Core1 (on die)	22.2 (65)			28.3 (83)	25.5 (75)		13.0 (38)			
Core4 (QPI)	63.4 (186)			102 - 109			58.0 (170)			106.0

Fig. 8. Read latencies of core 0 accessing cache lines of core 0 (local), core 1 (on die) or core 4 (via QPI) [1]

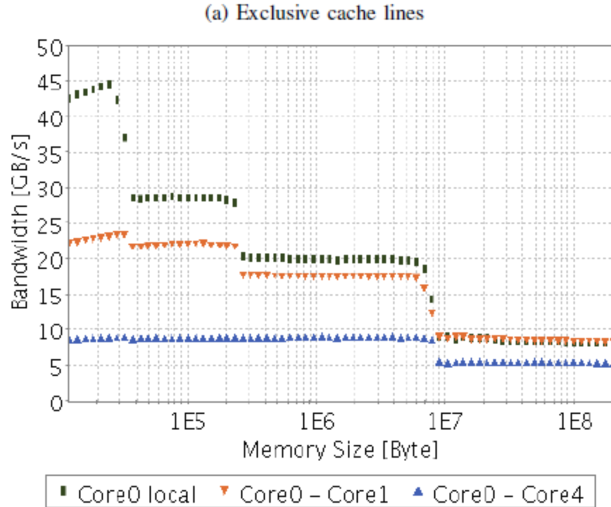

Fig. 9. Read bandwidth of core 0 accessing cache lines of core 0 (local), core 1 (on die) or core 4 (via QPI) with cache lines initialized to the exclusive coherency state[1]

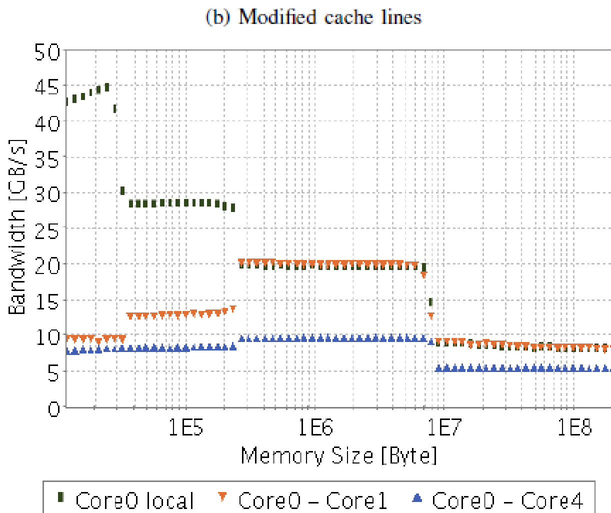

Fig. 10. Read bandwidth of core 0 accessing cache lines of core 0 (local), core 1 (on die) or core 4 (via QPI) with cache lines initialized to the modified coherency state[1]

	Exclusive			Modified			RAM
	L1	L2	L3	L1	L2	L3	
Local	45.6	31.1	26.2	45.6	31.1	26.2	10.1
Core1	19.3	19.7		9.4	13.2		
Core4	9.0	9.2		5.6			6.3

Fig. 11. Core 0 Read Bandwidth in Gbps [1]

	Exclusive			Modified			RAM
	L1	L2	L3	L1	L2	L3	
Local	45.6	28.8	19.9	45.6	28.8	19.9	8.4
Core1	23.4	22.2	17.6	9.4	13.0		
Core4	9.0			8.3		9.6	5.5

Fig. 12. Core 0 Write Bandwidth in Gbps [1]

another processor's cache, then the bandwidth decreases to 13.2 Gbps in the L2 cache and 9.4 Gbps in the L1 cache.

These results show us that the memory subsystem in the Nehalem architecture is able to maintain a high level of efficiency in terms of latency and bandwidth due to the well-designed memory hierarchy and cache coherency strategy. It performs particularly well for on-chip data transfers between cores. The shared L3 cache seems to eliminate much of the snooping traffic and allow the architecture to be more scalable.

V. A Comparison with Other Architectures

To gain some perspective on the efficiency of the Nehalem architecture, we will take a look at the study *A Performance Evaluation of the Nehalem Quad-core Processor for Scientific Computing* [4]. This paper compares Nehalem with its predecessors and competing architectures for a scientific computing application. Specifically, the three nodes used in the test are:

1) Two quad-core Intel Core i7 (Nehalem) processors (8 cores total), 45nm fabrication technology
2) Four quad-core AMD Opteron 8350 (Barcelona) processors (16 cores total), 65nm fabrication technology
3) Four quad-core Intel Xeon X7350 (Tigerton) processors (16 cores total), 65nm fabrication technology

	Processor					Memory		Node	
	Speed GHz	Peak Gflops	L1 KB	L2 MB	L3 MB	Type	Speed MHz	Memory controllers	Peak Gflops
Nehalem	2.8	44.8	64	0.25	8	DDR3	1333	2	89.6
Barcelona	2.0	32	64	0.25	2	DDR2	667	4	128.0
Tigerton	2.93	46.9	64	4	-	FBDIMM	667	1	187.6

Fig. 13. Characteristics of the quad-core processors, memory, and node organization [4]

To compare these architectures, the authors of [4] used a suite of scientific applications taken from existing U.S. Department of Energy workloads. They are most interested in the capability of each architecture to efficiently implement parallelism at the core level, the processor level, and the node level. These processors will be the building blocks of large-scale parallel scientific computers.

A. Description of Other Architectures

The Nehalem architecture has been described in detail in previous sections. To make the comparisons more meaningful the following is a brief overview of the Barcelona and Tigerton processors.

1) Barcelona: The Barcelona is a first generation quad-core processor developed by Advanced Micro Devices (AMD) as part of the Opteron series. The Barecelona architecture combines four Opteron cores onto a single die with 65nm technology. Each core has its own L1 cache (64 KB) and L2 cache (512 KB). The four cores on a single processor share a 2MB L3 cache. The Opteron architecture has special instructions that enable each core to execute 4 double-precision floating-point operations per clock cycle. The clock speed of each core is 2.0 GHz which gives each processor a theoretical peak performance of 32 billion floating-point operations per second.

For this test, the Barcelona node has four quad-core processors each with 4 GB of DDR2 memory for a total of 16 GB of memory for the node. The processors are connected in a 2x2 HyperThread mesh which classifies this as a NUMA architecture.

2) Tigerton: The Intel Tigerton processor is a first generation quad-core processor which contains two dual-core dies that are packaged into a single module. Each core contains a private L1 cache (64 KB), and two cores on each die share a 4 MB L2 cache.

The Tigerton node in this setup contains four processors and 16 GB of memory using fully-buffered DIMMs. Unlike the Barcelona or Nehalem that have a NUMA configuration, Tigerton uses a single memory controller hub in what is called a symmetric multiprocessor (SMP) configuration. This memory controller hub contains a Dedicated High Speed Interconnect (DHSI) to provide point-to-point communication between processors and memory channels.

Some key features of each architecture are compared in Figure 13.

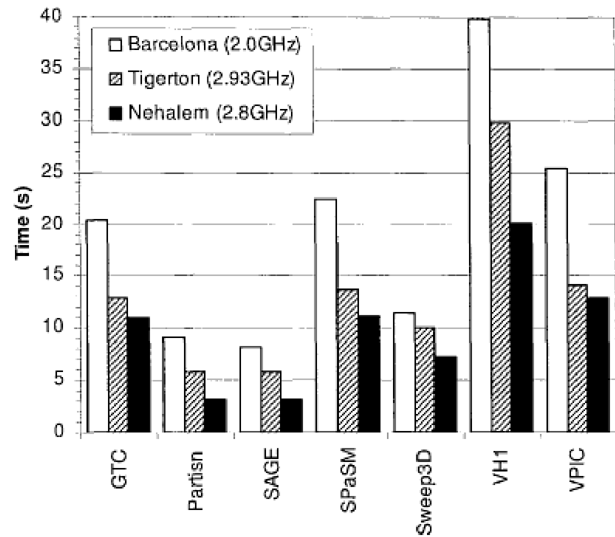

Fig. 14. Application iteration time, single-core [4]

B. Application Testing

To provide a comparison of the three architectures for scientific computations, the authors used applications currently in use by the U.S. Department of Energy. The testing consisted of:

- Comparing single core performance of each architecture.
- Comparing scaling performance by using all processors in the node.
- Determining the combinations of cores or number of cores per processor that gives the best performance.

1) Single-core Measurements: Figures 14 and 15 illustrate the performance of each architecture using just one core. The Y-axis in Figure 14 denotes the time it takes for one iteration of the main computational loop, and the X-axis shows the results of each test application. In Figure 15, a value of 1.0 indicates the same performance between the two processors, a value of 2.0 indicates a 2x improvement, etc.

As you can see in the figures the Nehalem core is 1.1-1.8 times faster than a Tigerton core and 1.6-2.9 times faster than a Barcelona core. Note that Nehalem has a slower clock than Tigerton but achieves much better performance for all the applications in the test. This suggests that Nehalem can "do less with more" with the improvements such as the memory and cache organization.

2) Node Measurements: The authors took the best results from running the applications on each node (exercising all cores on the node) and compared their execution time side-by side in Figure 16. The relative performance of the Nehalem architecture is shown once again in Figure 17. As you can see, the Nehalem node achieves performance 1.4-3.6 times faster than the Tigerton node and 1.5-3.3 times faster than the Barcelona node.

The scaling behavior (going from a single core to many cores) of the Tigerton seems to be particularly bad. This

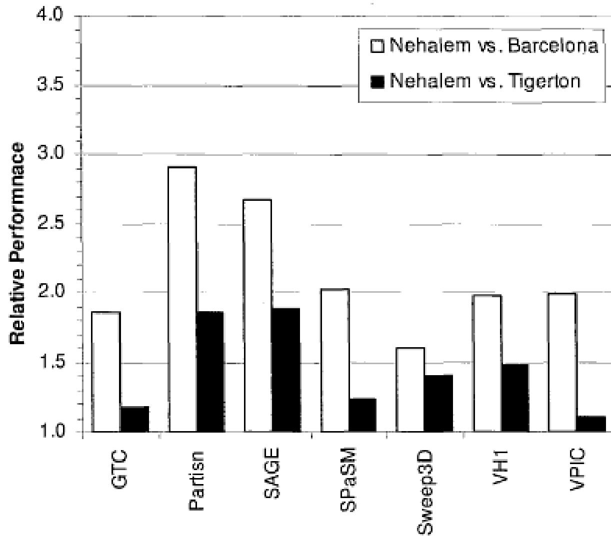

Fig. 15. Performance advantage of Nehalem, single-core [4]

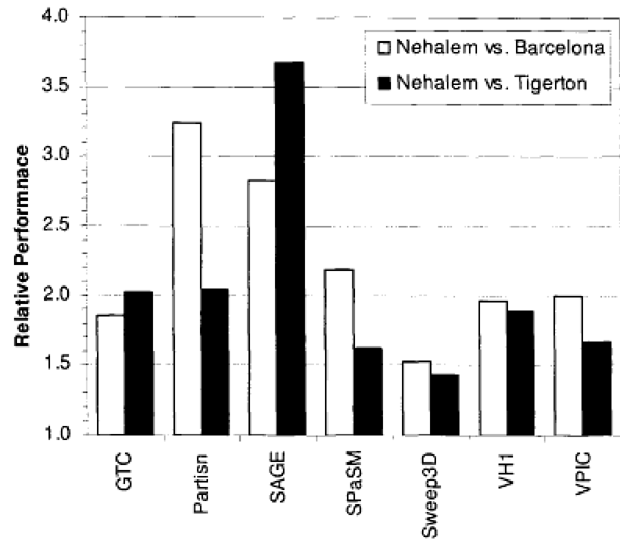

Fig. 17. Performance advantage of Nehalem, multi-core [4]

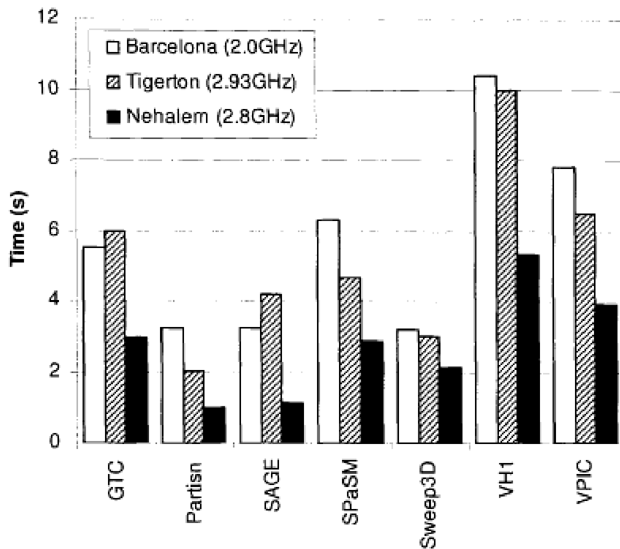

Fig. 16. Application iteration time, multi-core [4]

is likely because the memory subsystem of the Tigerton architecture was not designed for scalabilty. It was mostly designed with just the two cores in mind.

These tests show that the improvements made to the memory architecture in the Nehalem processor make a huge impact on performance when it comes to performing data-intensive parallel computations for scientific applications.

VI. Conclusion

In this paper I have taken a close look at Intel's Nehalem architecture. I have given an overview of the major improvements to the architecture over Intel's previous multi-core architectures with a special focus on the memory organization and cache coherency scheme. I have looked into several studies that have shown by benchmarking measurements the effectiveness of these improvements. The inclusive, shared L3 cache has reduced much of the complexity and overhead associated with keeping caches coherent between cores. The integrated memory controller reduces memory access latencies. I have shown that the Nehalem architecture scales well; it allows for as many cores as are needed for a particular application. Previous and competing architectures do not scale nearly as well; more cores can be added, but the limitations of the memory organization introduce bottlenecks to data movement. Nehalem has identified and removed these bottlenecks. This positions Nehalem well to be a flexible solution to future parallel processing needs.

References

[1] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System", in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, September 2009

[2] JJ. Treibig, G. Hager, and G. Wellein, "Multi-core architectures: Complexities of performance prediction and the impact of cache topology", Regionales Rechenzentrum Erlangen, Friedrich-Alexander Universiat Erlangen-urnberg, Erlangen, Germany, October 2009

[3] BB. Qian, L. Yan, "The Research of the Inclusive Cache used in Multi-Core Processor", Key Laboratory of Advanced Display & System Applications, Ministry of Education, Shanghai University, July 2008

[4] KK.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, J.C. Sancho, "A Performance Evaluation of the Nehalem Quad-core Processor for Scientific Computing", *Parallel Processing Letters*, Vol. 18, No. 4 (2008), World Scientific Publishing Company

[5] II. Gavrichenkov, "First Look at Nehalem Microarchitecture", November 2008, http://www.xbitlabs.com/articles/cpu/display/nehalem-microarchitecture.html