The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms

Michael E. Thomadakis, Ph.D.
Supercomputing Facility
miket AT tamu DOT edu
Texas A&M University

March, 17, 2011

Abstract

Nehalem is an implementation of the CISC Intel64 instruction specification based on 45nm and high-k + metal gate transistor technology. Nehalem micro-architectures and system platforms employ a number of state-of-the-art technologies which enable high computation rates for scientific and other demanding workloads. Nehalem based processors incorporate multiple cores, on-chip DDR3 memory controller, a shared Level 3 cache and high-speed Quick-Path Interconnect ports for connectivity with other chips and the I/O sub-system. Each core has superscalar, out-of-order and speculative execution pipelines and supports 2-way simultaneous multi-threading. Each core offers multiple functional units which can sustain high instruction level parallelism rates with the assistance of program development tools, compilers or special coding techniques. A prominent feature of Intel64 is the processing of SIMD instructions at a nominal rate of 4 double or 8 single precision floating-point instructions per clock cycle. Nehalem platforms are cc-NUMA shared-memory processor systems.

Complex processors and platforms, such as those based on Nehalem, present several challenges to application developers, as well as, system level engineers. Developers are faced with the task of writing efficient code on increasingly complex platforms. System engineers need to understand the system level bottlenecks in order to configure and tune the system to yield good performance for the application mix of interest. This report discusses technical details of the Nehalem $\mu-$architecture and platforms with an emphasis on inner workings and the cost of instruction execution. The discussion presented here can assist developers and engineers in their respective fields. The first to produce efficient scalar and parallel code on the Nehalem platform and the latter ones to configure and tune a system to perform well under complex application workloads.

Keywords: *Intel64, Nehalem, Core and Uncore, Superscalar Processors, Micro-architecture, Cache Memory Hierarchy, cc-NUMA, Global Queue, Integrated-Memory Controllers, Local and Remote Memory Performance.*

Contents

List of Figures

1 Introduction

Intel "Nehalem" is the nickname for the "Intel Micro-architecture", where the latter is a specific implementation of the "Intel64" Instruction Set Architecture (ISA) specification [1, 2, 3]. For this report, "Nehalem" refers to the particular implementation where a processor chip contains four cores, the fabrication process is 45nm with high-k + metal gate transistor technology. We further focus on the Nehalem-EP platform which has two processor sockets per node and where the interconnection between sockets themselves and between processors and I/O is through Intel's Quick-Path Interconnect. Nehalem is the foundation of Intel Core i7 and Xeon processor 5500 series. Even though Intel64 is a classic Complex-Instruction Set Computer ("CISC") instruction set type, its Intel Micro-architecture implementation shares many mechanisms in common with modern Reduced-Instruction Set Computer ("RISC") implementations.

Each Nehalem chip is a multi-core chip multiprocessor, where each core is capable of sustaining high degrees of instruction-level parallelism. Nehalem based platforms are cache-coherent non-uniform memory access shared memory multi-processors. However, to sustain the high instruction completion rates, the developer must have an accurate appreciation of the cost of using the various system resources.

1.1 Motivation for this Study

Every application, scalar or parallel, experiences a "critical-path" in the system resources it accesses. For a developer a basic objective is to deploy algorithms and coding techniques which minimize the total application execution time. A developer has to identify and improve the code which utilizes critical path resources. One "well" known consequence is that right after of alleviating "the" bottleneck, the next bottleneck emerges. Developers should strike a balance between the effort to remove a bottleneck and the expected benefits. They need to understand the degree of Instruction-Level Parallelism (ILP) or Task-Level Parallelism (TLP) available in their code. They have to strike the right balance at what should be done in parallel between these two extreme levels. A consequence of the ccNUMA memory architecture of Nehalem is that there is very different cost in accessing data items residing in the local or the remote physical memories. Developers have to make wise data and computation thread placement decisions for their code in order to avoid accessing data remotely.

System engineers objectives include increasing system resource utilization, tuning a system to perform well under mixed application workloads and reducing overall cost of system operation. We can increase system utilization by allowing more workload into the system to minimize the times resources remain idle. However, it is well known that the more utilized the service centers (*i.e.*, the resources) get the longer the queuing effects become with the consequence of individual tasks experiencing higher latencies through the system. Increasing system utilization therefore, requires one to know which tasks to allow to proceed in parallel so that individual task performance is not seriously compromised.

Developers and engineers tasks become increasingly challenging as platforms become more complex, while information is not available to them in a coherent fashion about the inner workings or about system performance cost.

In this study we attempt to provide in a single place, a coherent discussion of the inner workings and performance of the Nehalem processor and platform. The sources are scattered around and are in different forms. We proceed by studying the system operation and cost in layers, namely, from the individual cores all the way out to the shared memory platform level. Note that this study is the result of continuous and challenging effort and should be considered to be in the state of continuous expansion. A subsequent publication will focus on the architecture and performance of Nehalem based clusters, interconnected by high speed low latency interconnects.

Nehalem builds upon and expands the new features introduced by the previous micro-architecture, namely the 45nm "Enhanced Intel Core Micro-architecture" or "Core-2" for short [4, 5, 6, 7]. A short discussion below focuses on the innovations introduced by the "Penryn" the predecessor to Nehalem micro-architecture and finally the additional enhancements Nehalem has implemented.

1.2 Overview of Features in the Intel Core Micro-Architecture

The "Core-2" micro-architecture introduced [1, 7] a number of interesting features, including the following

1. "Wide Dynamic Execution" which enabled each processor core to fetch, dispatch, execute and retire up to *four* instructions per clock cycle. This architecture had
   - 45nm fabrication technology,
   - **14**-stage core pipeline,
   - **4** decoders to decode up to 5 instructions per cycle,
   - **3** clusters of arithmetic logical units,

- *macro-fusion* and *micro-fusion* to improve front-end throughput,
- *peak* dispatching rate of up to **6** micro-ops per cycle,
- *peak* retirement rate of up to **4** micro-ops per cycle,
- advanced branch prediction algorithms,
- stack pointer tracker to improve efficiency of procedure entries and exits.

2. “Advanced Smart Cache” which improved bandwidth from the second level cache to the core, and improved support for single- and multi-threaded applications computation
   - 2nd level cache up to 4 MB with 16-way associativity,
   - 256 bit internal data path from L2 to L1 data caches.
3. “Smart Memory Access” which pre-fetches data from memory responding to data access patterns, reducing cache-miss exposure of out-of-order execution
   - hardware pre-fetchers to reduce effective latency of 2nd level cache misses,
   - hardware pre-fetchers to reduce effective latency of 1st level data cache misses,
   - “memory disambiguation” to improve efficiency of speculative instruction execution.
4. “Advanced Digital Media Boost” for improved execution efficiency of most 128-bit SIMD instruction with single-cycle throughput and floating-point operations
   - single-cycle inter-completion latency (“throughput”) of most 128-bit SIMD instructions,
   - up to **eight** single-precision floating-point operation per cycle,
   - 3 issue ports available to dispatching SIMD instructions for execution.

1.3 Summary of New Features in the Intel Micro-Architecture

Intel Micro-architecture (Nehalem) provides a number of distinct feature enhancements over those of the Enhanced Intel Core Micro-architecture, discussed above, including:

1. “Enhanced” processor core:
   - improved branch prediction and recovery cost from mis-prediction,
   - enhancements in loop streaming to improve front-end performance and reduce power consumption,
   - deeper buffering in out-of-order engine to sustain higher levels of instruction level parallelism,
   - enhanced execution units with accelerated processing of CRC, string/text and data shuffling.
2. Hyper-threading technology (SMT):
   - support for two hardware threads (logical processors) per core,
   - a **4**-wide execution engine, larger L3, and large memory bandwidth.
3. “Smarter” Memory Access:
   - integrated (on-chip) memory controller supporting low-latency access to local system memory and overall scalable memory bandwidth (previously the memory controller was hosted on a separate chip and it was common to all dual or quad socket systems),
   - new cache hierarchy organization with shared, inclusive L3 to reduce snoop traffic,
   - two level TLBs and increased TLB sizes,
   - faster unaligned memory access.
4. Dedicated Power management:

- integrated micro-controller with embedded firmware which manages power consumption,
- embedded real-time sensors for temperature, current, and power,
- integrated power gate to turn off/on per-core power consumption;
- Versatility to reduce power consumption of memory and QPI link subsystems.

In the sections which follow we study and analyze in-depth all the technologies and innovations which make Nehalem platforms state-of-the-art systems for high-performance computing. This study brings together in one place information which is not available in a one place. The objective is to study the system at various detail levels with an emphasis on the cost of execution of code which can be purely scalar, shared or distributed-memory parallel and hybrid combinations.

2 The "Intel®64" Architecture

An *Instruction Set Architecture* (ISA) is the formal definition of the logical or "architected" view of a computer [8, 9]. This is the view that machine language code has of the underlying hardware. An ISA specifies precisely all machine instructions, the objects which can be directly manipulated by the processor and their binary representation. Data objects can be, for instance, n-bit integers, floating point numbers of a certain precision, single-byte characters or complex structures consisting of combinations of simpler objects. Machine instructions determine which elementary operations the processor can carry out directly on the various data operands. Instruction sets have traditionally been classified as Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) specifications.

The "Intel64" ISA, historically derives from the 64-bit extensions AMD applied on Intel's popular 32-bit "IA-32" ISA for its "K8" processor family. Later on AMD used the name "AMD64" while Intel the names "EM64T" and "IE-32e". Finally, Intel settled on "Intel64" as their official 64-bit ISA deriving from the IA-32. The Intel64 architecture supports IA-32 ISA and extends it to fully support natively 64-bit OS and 64-bit applications [1]. The physical address space in the Intel64 platform can reach up to 48 bits which implies that 256 Tera-binary-Bytes (TiB) can by directly addressed by the hardware. The logical address size of Intel64 is 64-bit to allow support for a 64-bit flat linear address space. However, Intel64 hardware addressing currently uses only the last 48-bits. Furthermore, the address size of the physical memory itself in a Nehalem processor can be up to **40** bits. Intel is reserving the address filed sizes for future expansions. Intel64 is one of the most prominent CISC instruction sets. Fig. 1 (see pp. 5) presents the logical (or "architected") view of the Intel64 ISA [1]. The architected view of an ISA is the collection of objects which are visible at the machine language code level and can be directly manipulated by machine instructions. In the 64-bit mode of Intel64 architecture, software may access

- a 64-bit linear ("flat") logical address space,
- uniform byte-register addressing,
- 16 64-bit-wide General Purpose Registers (GPRs) and instruction pointers
- 16 128-bit "XMM" registers for streaming SIMD extension instructions (SSE, SSE2, SSE3 and SSSE3, SSE4), in addition to 8 64-bit MMX registers or the 8 80-bit x87 registers, supporting floating-point or integer operations,
- fast interrupt-prioritization mechanism, and
- a new instruction-pointer relative-addressing mode.

64-bit applications can use 64-bit register operands and 64-bit address pointers through a set of modifier prefixes in the code. Intel compilers can produce code which takes full advantage of all the features in Intel64 ISA. Application optimization requires a fair level of understanding of the hardware resources and the cost in using them.

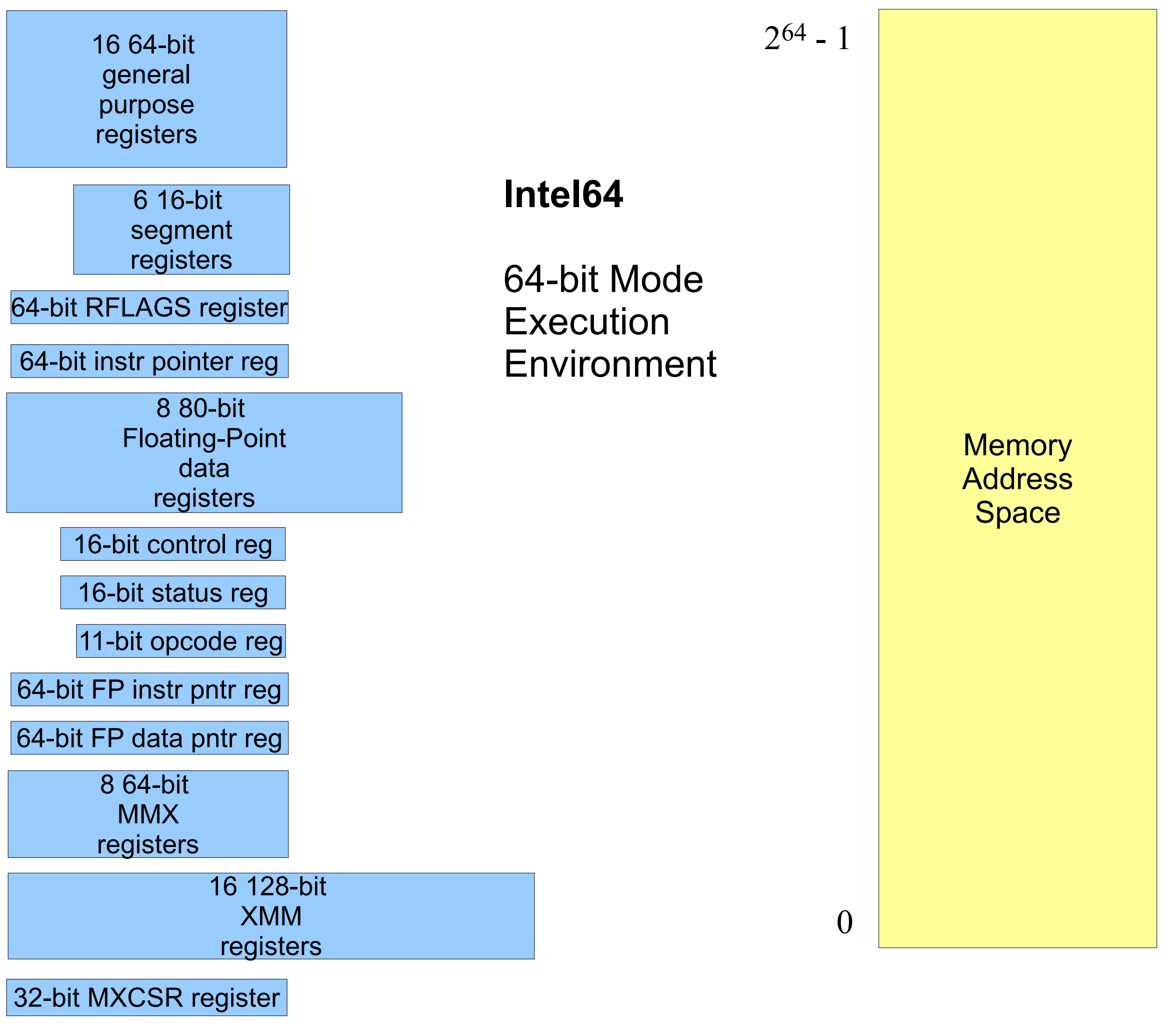


Figure 1: “Intel64” 64-bit execution environment for Nehalem processor

3 The Nehalem Processor

3.1 Instruction and Data Flow in Modern Processors

Nehalem implements a number of techniques to process efficiently the stream of Intel64 ISA CISC "*macro-instructions*" in the user code. A core internally consists of a large number of functional units (FUs) each capable of carrying out an elementary "micro-operation" (micro-op). An example of a FU is an ALU (arithmetic and logic unit) which can carry out an operation against input operands. Micro-ops would specify the operation type and its operands. Micro-ops are RISC-like type of instructions and they require similar effort and resources to process.

Micro-operations having no dependencies on the results of each other could proceed *in parallel* if separate FUs are available. The CISC type of Intel64 macro-instructions are translated by the early stages of the core into one or more micro-ops. The micro-operations eventually reach the execution FUs where they are dispatched to FUs and "retire", that is, have their results saved back to visible ("architected") state (*i.e.*, data registers or memory). When all micro-ops of a macro-instruction retire, the macro-instruction itself retires. It is clear that the basic objective of the processor is to **maximize** the macro-instruction retirement rate.

The fundamental approach Nehalem (and other modern processors) take to maximize instruction completion rates is to allow the micro-ops of as many instructions as feasible, proceed in parallel with micro-op occupying independent FUs at each clock cycle. We can summarize the Intel64 instruction flow through the core as follows.

1. The early stages of the processor **fetch**-in several macro-instructions at a time (say in a cache block) and
2. **decode** them (break them down) into sequences of micro-ops.
3. The micro-ops are buffered at various places where they can be picked up and scheduled to use the FUs in parallel if data dependencies are not violated. In Nehalem, micro-ops are **issued** to stations were they reserve their position for subsequent,
4. **dispatching** as soon as their input operands become available.
5. Finally, completed micro-ops **retire** and post their results to permanent storage.

The entire process proceeds in stages, in a "pipelined" fashion. Pipelining is used to break down a lengthy task into sub-tasks where intermediate results flow downstream the pipeline stages. In microprocessors, subtasks handled within each stage take one clock cycle. The amount of hardware logic which goes into each stage has been carefully selected so that there is approximately an equal amount of work which takes place in every stage. Since adding a pipeline stage includes some additional fixed overhead for buffering intermediate results, pipeline designs carefully balance the total number of stages and the duration per stage.

Complex FUs are usually themselves pipelined. A floating-point ALU may require several clock cycles to produce the results of complex FP operations, such as, FP division or square root. The advantage of pipelining here is that with proper intermediate result buffering, we could supply a new set of input operands to the pipelined FU in each clock cycle and then correspondingly expect a new result to be produced at each clock cycle at the output of the FU.

A pipeline **bubble** takes place when the input operands of a downstream stage are not available. Bubbles flow downstream at each clock cycle. When the entire pipeline has no input to work with it can **stall**, that is, it can suspend operation completely. Bubbles and stalls are detrimental to the efficiency of pipelined execution if they take place with a "high" frequency. Common reasons for a bubble is when say data has to be retrieved from slower memory or from a FU which takes multiple cycles to produce them. Compilers and processor designers invest heavily in minimizing the occurrence and the impact of stalls. A common way to alleviate the frequency of stalls is to allow micro-ops proceed **out of chronological order** and use any available FUs. **Dynamic instruction scheduling** logic in the processor determines which micro-ops can proceed in parallel while the program execution remains semantically correct. Dynamic scheduling utilizes the "Instruction Level Parallelism" (ILP) which is possible within the instruction stream of a program. Another mechanism to avoid pipeline stalling is called *speculative* execution.

A processor may speculatively start fetching and executing instructions from a code path before the outcome of a conditional branch is determined. Branch prediction is commonly used to "predict" the outcome and the target of a branch instruction. However, when the path is determined not to be the correct one, the processor has to cancel all intermediate results and start fetching instructions from the right path. Another mechanism relies on data pre-fetching when it is determined that the code is retrieving data with a certain pattern. There are many other mechanisms which are however beyond the scope of this report to describe.

Nehalem, as other modern processors, invests heavily into pre-fetching as many instructions, from a predicted path and translating them into micro-ops, as possible. A dynamic scheduler then attempts to maximize the number of concurrent micro-ops which can be in progress ("in-flight") at a time, thus increasing the completion instruction rates. Another interesting feature of Intel64 is the direct support for SIMD instructions which increase the effective ALU throughput for FP or integer operations.

3.2 Overview of the Nehalem Processor Chip

A Nehalem processor chip is a "Chip-Multi Processor" (CMP), consisting of several functional parts within a single silicon die. Fig. 2 illustrates a Nehalem CMP chip and its major parts.

Referring to Fig. 2, a Nehalem chip consists of the following components

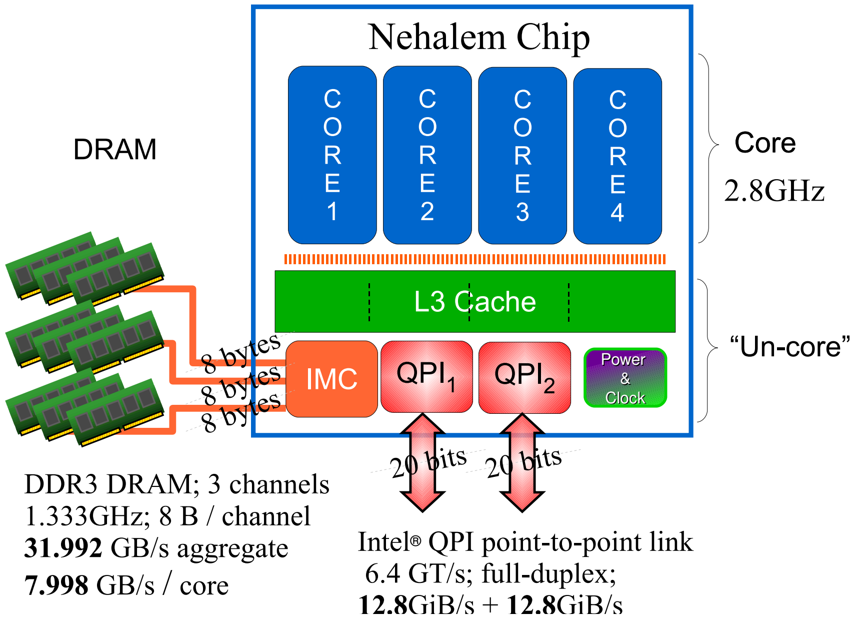
- four identical compute cores,
- UIU: Un-Core Interface Unit (switch connecting the 4 cores to the 4 L3 cache segments, the IMC and QPI ports),
- L3: level-3 cache controller and data block memory,
- IMC: 1 integrated memory controller with 3 DDR3 memory channels,
- QPI: 2 Quick-Path Interconnect ports, and
- auxiliary circuitry for cache-coherence, power control, system management and performance monitoring logic.

A Nehalem chip is divided into two broad domains, namely, the "*core*" and the "*un-core*". Components in the core domain operate with the same clock frequency as that of the actual computation core. In EOS's case this is 2.8GHz. The un-core domain operates under a different clock frequency. This modular organization reflects one of Nehalem's objectives of being able to consistently implement chips with different levels of computation abilities and power consumption profiles. For instance, a Nehalem chip may have from two to eight cores, one or more high-speed QPI interconnects, different sizes for L3 caches, as well as, memory sub-systems with different DRAM bandwidths. Similar partitioning of CMP chip into different clock domains can be found in other processors, such as, in IBM's Power5, 6 and 7, in AMDs multi-core chips and serves very similar purposes.

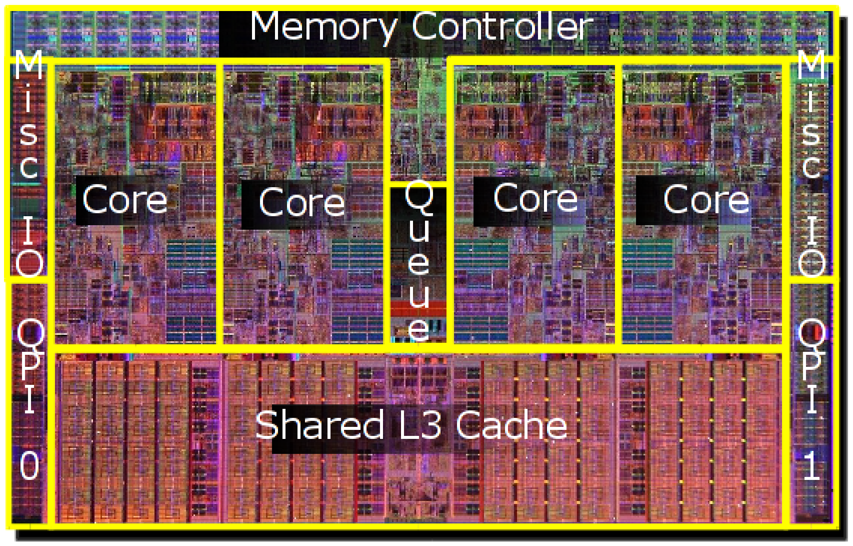
Outside the Nehalem chip, but at close physical proximity, we find the DRAM which is accessible by means of **three 8**-byte DDR3 channels, each capable to operate at up to **1.333** GigaTransfers/sec. The aggregate nominal main memory bandwidth is **31.992** GB/s per chip, or on the average **7.998** GB/s per core. This is a significant improvement over all previous Intel micro-architectures. The maximum operating frequency of the DDR3 buses is determined by the number of DIMMs in the slots.

In essence the "un-core" domain contains the memory controller and cache coherence logic which in earlier Intel architectures used to be implemented by the separate "North-bridge" chip.

The high performance of the Nehalem architecture relies, among other things, on the fact that the DRAM controller, the L3 and the QPI ports are all housed within the same silicon die as the four cores. This saves a *significant amount of off-chip communications* and makes possible a tightly coupled, low-latency, high bandwidth CMP system. This particular processor to memory implementation is a significant departure from all previous ones by Intel. Prior to Nehalem, the memory controller was housed on a separate "Northbridge" chip and it was shared by all



A. Nehalem Chip and DDR3 Memory Module. The processor chip contains four cores, a shared L3 cache and DRAM controllers, and Quick-path Interconnect ports.



B. Nehalem micro-photograph.

Figure 2: A Nehalem Processor/Memory module and Nehalem micro-photograph

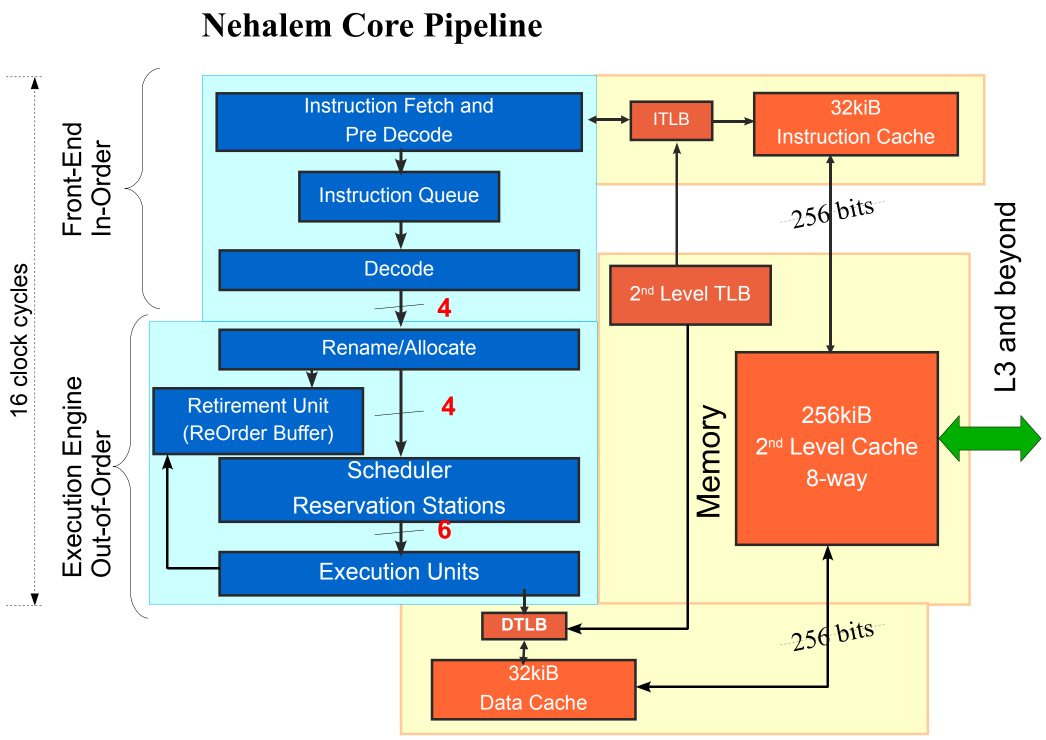


Figure 3: High-level diagram of a Nehalem core pipeline.

processor chips. The Northbridge has been one of the often cited bottlenecks in previous Intel architectures. Nehalem has substantially increased the main memory bandwidth and shortened the latency to access main memory. However, now that a separate DRAM is associated with every IMC and chip, platforms with more than one chips are Non-Uniform Memory Access ("NUMA"). NUMA organizations have distinct performance advantages and disadvantages and with proper care multi-threaded computation can make efficient use of the available memory bandwidth. In general data and thread placement becomes an important part of the application design and tuning process.

3.3 Nehalem Core Pipeline

3.3.1 Instruction and Data Flow in Nehalem Cores

Nehalem cores are modern micro-processors with in-order instruction issue, super-scalar, out-of-order execution data-paths, which are coupled with a multilevel storage hierarchy. Nehalem cores have extensive support for branch prediction, speculative instruction execution, data pre-fetching and multiple pipelined FUs. An interesting feature is the direct support for integer and floating point SIMD instructions by the hardware.

Nehalem's pipeline is designed to maximize the macro-instruction flow through the multiple FUs. It continues the four-wide micro-architecture pipeline pioneered by the 65nm "Intel Core Micro-architecture" ("Merom") and the 45nm "Enhanced Core Micro-architecture" ("Penryn"). Fig. 3 illustrates a functional level overview of a Nehalem instruction pipeline. The total length of the pipeline, measured by branch mis-prediction delay, is **16** cycles, which is two cycles longer than that of its predecessor. Referring to Fig. 3, the core consists of

- an in-order **Front-End Pipeline** (FEP) which retrieves Intel64 instructions from memory, uses four decoders to decode them into micro-ops and buffers them for the downstream stages;

- an out-of-order super-scalar **Execution Engine** (EE) that can dynamically schedule and dispatch up to six micro-ops per cycle to the execution units, as soon as source operands and resources are ready,
- an in-order **Retirement Unit** (RU) which ensures the results of execution of micro-ops are processed and the "architected" state is updated according to the original program order, and
- multi-level cache hierarchy and address translation resources.

We describe in the next two Sub-sections in detail the front-end and back-end pf the core.

3.3.2 Nehalem Core: Front-End Pipeline

Fig. 4 illustrates in more detail key components of Nehalem's Front-End Pipeline (FEP). The FEP is responsible for retrieving blocks of macro-instructions from memory and translating them into micro-ops and buffering them for handling at the execution back-end. FEP handles the code instructions "in-order". It can decode up to **4** macro-instructions in a single cycle. It is designed to support up to two hardware SMT threads by decoding the instruction streams of the two threads in *alternate* cycles. When SMT is not enabled, the FEP handles the instruction stream of only one thread. Front-End pipeline of a Nehalem core

The **Instruction Fetch Unit** (IFU) consists of the Instruction Translation Look-aside Buffer (ITLB, discussed in section 6), an instruction pre-fetcher, the L1 instruction cache and the pre-decode logic of the Instruction Queue (IQ). The IFU always fetches **16** bytes (128 bits) of aligned instruction bytes on each clock cycle from the Level 1 instruction cache into the Instruction Length Decoder. There is a 128-bit wide direct path from L1 to the IFU. The IFU always brings in 16 byte blocks.

The IFU uses the ITLB to locate the 16-byte block in the L1 instruction cache and instruction pre-fetch buffers. Instructions are referenced by virtual address and translated to physical address with the help of a 128 entry ITLB. A hit in the instruction cache causes 16 bytes to be delivered to the instruction pre-decoder. Programs average slightly less than 4 bytes per instruction, and since most instructions can be decoded by all decoders, an entire fetch can often be consumed by the decoders in one cycle. Instruction fetches are always 16-byte aligned. A non-16 byte aligned target reduces the number of instruction bytes by the amount of offset into the 16 byte fetch quantity. A taken branch reduces the number of instruction bytes delivered to the decoders since the bytes after the taken branch are not decoded.

The **Branch-Prediction Unit** (BPU) allows the processor to begin fetching and processing instructions before the outcome of a branch instruction is determined. For microprocessors with lengthy pipelines successful branch prediction allows the processor to fetch and execute speculatively instructions over the "predicted" path without "stalling" the pipeline. When a prediction is not successful, Nehalem simply cancels all work already done by the micro-ops already in the pipeline on behalf of instructions along the wrong path. This may get costly in terms of resources and execution cycles already spent. Modern processors invest heavily in silicon estate and algorithms for the BPU in order to minimize the frequency and impact of wrong branch predictions.

On Nehalem the BPU makes predictions for the following types of branch instructions

- direct calls and jumps: targets are read as a target array, without regarding the taken or not-taken prediction,
- indirect calls and jumps: these may either be predicted as having a fixed behavior, or as having targets that vary according to recent program behavior,
- conditional branches: BPU predicts the branch target and whether the branch will be taken or not.

Nehalem improves branch handling in several ways. The **Branch Target Buffer** (BTB) has been increased in size to improve the accuracy of branch predictions. Furthermore, hardware enhancements improve the handling of branch mis-prediction by expediting resource reclamation so that the front-end would not be waiting to decode instructions in an "architected" code path (the path in which instructions will reach retirement) while resources were

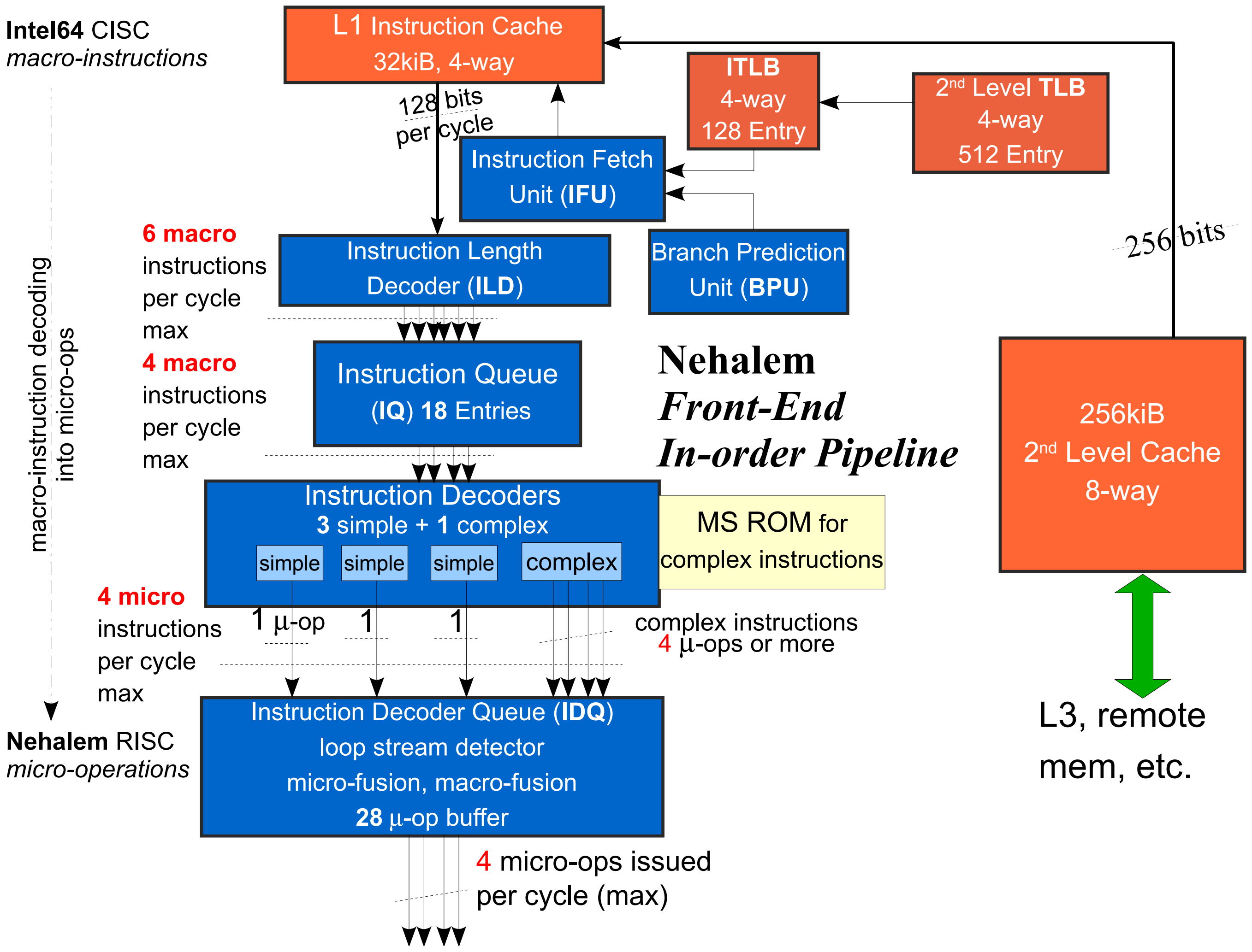

Intel64 CISC
macro-instructions
macro-instruction decoding into micro-ops
Nehalem RISC
micro-operations
L1 Instruction Cache
32kiB, 4-way
128 bits per cycle
ITLB
4-way
128 Entry
2nd Level TLB
4-way
512 Entry
Instruction Fetch Unit (IFU)
Branch Prediction Unit (BPU)
256 bits
6 macro instructions per cycle max
Instruction Length Decoder (ILD)
4 macro instructions per cycle max
Instruction Queue (IQ) 18 Entries
Nehalem
Front-End
In-order Pipeline
256kiB
2nd Level Cache
8-way
Instruction Decoders
3 simple + 1 complex
simple
simple
simple
complex
MS ROM for complex instructions
4 micro instructions per cycle max
1 µ-op
1
1
complex instructions
4 µ-ops or more
L3, remote mem, etc.
Instruction Decoder Queue (IDQ)
loop stream detector
micro-fusion, macro-fusion
28 µ-op buffer
4 micro-ops issued per cycle (max)

Figure 4: High-level diagram of the In-Order Front-End Nehalem Pipeline (FEP).

allocated to executing mis-predicted code path. Instead, new micro-ops stream can start forward progress as soon as the front end decodes the instructions in the architected code path. The BPU includes the following mechanisms

- **Return Stack Buffer** (RSB) A 16-entry RSB enables the BPU to accurately predict RET instructions. Renaming is supported with return stack buffer to reduce mis-predictions of return instructions in the code.
- **Front-End Queuing** of BPU lookups. The BPU makes branch predictions for 32 bytes at a time, twice the width of the IFU. Even though this enables taken branches to be predicted with no penalty, software should regard taken branches as consuming more resources than do not-taken branches.

Instruction Length Decoder (ILD or "Pre-Decoder") accepts 16 bytes from the L1 instruction cache or pre-fetch buffers and it prepares the Intel64 instructions found there for instruction decoding downstream. Specifically the ILD

- determines the length of the instructions,
- decodes all prefix modifiers associated with instructions and
- notes properties of the instructions for the decoders, as for example, the fact that an instruction is a branch.

The ILD can write up to 6 instructions per cycle, maximum, into the downstream Instruction Queue (IQ). A 16-byte buffer containing more than 6 instructions will take 2 clock cycles. Intel64 allows modifier prefixes which dynamically modify the instruction length. These length changing prefixes (LCPs) prolong the ILD process to up to 6 cycles instead of 1.

The **Instruction Queue** (IQ) buffers the ILD-processed instructions and can deliver up to five instructions in one cycle to the downstream instruction decoder. The IQ can buffer up to 18 instructions.

The **Instruction Decoding Unit** (IDU) translates the pre-processed Intel64 macro-instructions into a stream of micro-operations. It can handle several instructions in parallel for expediency.

The IDU has a total of **four** decoding units. Three units can decode one simple instruction each, per cycle. The other decoder unit can decode one instruction every cycle, either a simple instruction or complex instruction, that is one which translates into several micro-ops. Instructions made up of more than four micro-ops are delivered from the micro-sequencer ROM (MSROM). All decoders support the common cases of single micro-op flows, including, micro-fusion, stack pointer tracking and macro-fusion. Thus, the three simple decoders are not limited to decoding single micro-op instructions. Up to four micro-ops can be delivered each cycle to the downstream instruction decoder queue (IDQ).

The IDU also parses the micro-op stream and applies a number of transformations to facilitate a more efficient handling of groups of micro-ops downstream. It supports the following.

Loop Stream Detection (LSD). For small iterative segments of code whose micro-ops fit within the 28-slot Instruction Decoder Queue (IDQ), the system only needs to decode the instruction stream once. The LSD detects these loops (backward branches) which could be streamed directly from the IDQ. When such a loop is detected, the micro-ops are locked down and the loop is allowed to stream from the IDQ until a mis-prediction ends it. When the loop plays back from the IDQ, it provides higher bandwidth at reduced power, (since much of the rest of the front end pipeline is shut off. In the previous micro-architecture the loop detector was working with the instructions within the IQ upstream. The LSD provides a number of benefits, including,

- no loss of bandwidth due to taken-branches,
- no loss of bandwidth due to misaligned instructions,
- no LCP penalties, as the pre-decode stage are used once for
- the instruction stream within the loop,

- reduced front-end power consumption, because the instruction cache, BPU and pre-decode unit can go to idle mode. However, note that loop unrolling and other code optimizations may make the loop too big to fit into the LSD. For high performance code, loop unrolling is generally considered superior for performance even when it overflows the loop cache capability.

Stack Pointer Tracking (SPT) implements the Stack Pointer Register (RSP) update logic of instructions which manipulate the program stack (PUSH, POP, CALL, LEAVE and RET) within the IDU. These macro-instructions were implemented by several micro-ops in previous architectures. The benefits with SPT include

- using a single micro-op for these instructions improves decoder bandwidth,
- execution resources are conserved since RSP updates do not compete for them,
- parallelism in the execution engine is improved since the implicit serial dependencies have already been taken care of,
- power efficiency improves since RSP updates are carried out by a small hardware unit.

Micro-Fusion The instruction decoder supports micro-fusion to improve pipeline front-end throughput and increase the effective size of queues in the scheduler and re-order buffer (ROB). Micro-fusion fuses multiple micro-ops from the same instruction into a single complex micro-op. The complex micro-op is dispatched in the out-of-order execution core. This reduces power consumption as the complex micro-op represents more work in a smaller format (in terms of bit density), and reduces overall "bit-toggling" in the machine for a given amount of work. It virtually increases the amount of storage in the out-of-order execution engine. Many instructions provide register and memory flavors. The flavor involving a memory operand will decodes into a longer flow of micro-ops than the register version. Micro-fusion enables software to use memory to register operations to express the actual program behavior without worrying about a loss of decoder bandwidth.

Macro-Fusion The IDU supports macro-fusion which translates adjacent macro-instructions into a single micro-op if possible. Macro-fusion allows logical compare or test instructions to be combined with adjacent conditional jump instructions into one micro-operation.

3.3.3 Nehalem Core: Out-of-Order Execution Engine

The execution engine (EE) in a Nehalem core selects micro-ops from the upstream IDQ and dynamically schedules them for dispatching and execution by the execution units downstream. The EE is a dynamically scheduled "out-of-order", super-scalar pipeline which allows micro-ops to use available execution units in parallel when correctness and code semantics are not violated. The EE scheduler can dispatch up to **6** micro-ops in one clock cycle through the six dispatch ports to the execution units. There are several FUs, arranged in three clusters, for integer, FP and SIMD operations. Finally, **four** micro-ops can retire in one cycle, which is the same as in Nehalem's predecessor cores. Results can be written-back at the maximum rate of one register per per port per cycle. Fig. 5 presents a high-level diagram of the Execution Engine along with its various functional units.

3.3.4 Execution pipelines of a Nehalem core

The execution engine includes the following major components:

Register Rename and Allocation Unit (RRAU) – Allocates EE resources to micro-ops in the IDQ and moves them to the EE.

Reorder Buffer (ROB) – Tracks all micro-ops in-flight,

Unified Reservation Station (URS) – Queues up to **36** micro-ops until all source operands are ready, schedules and dispatches ready micro-ops to the available execution units.

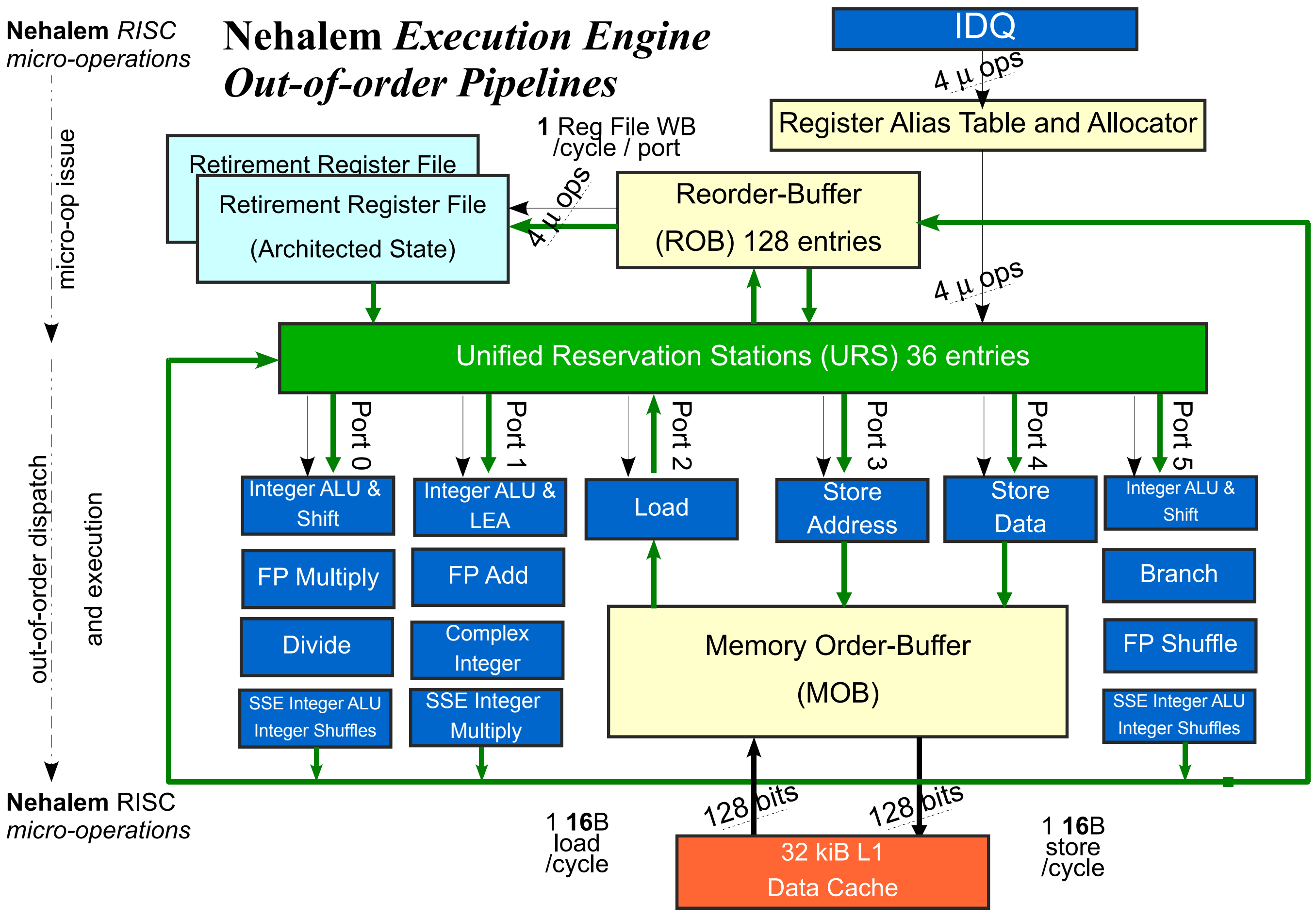

Nehalem RISC micro-operations
Nehalem Execution Engine Out-of-order Pipelines
IDQ
4 µ ops
Register Alias Table and Allocator
1 Reg File WB /cycle / port
Retirement Register File
Retirement Register File (Architected State)
4 µ ops
Reorder-Buffer (ROB) 128 entries
4 µ ops
micro-op issue
Unified Reservation Stations (URS) 36 entries
Port 0
Port 1
Port 2
Port 3
Port 4
Port 5
Integer ALU & Shift
Integer ALU & LEA
Load
Store Address
Store Data
Integer ALU & Shift
FP Multiply
FP Add
Branch
Divide
Complex Integer
Memory Order-Buffer (MOB)
FP Shuffle
SSE Integer ALU Integer Shuffles
SSE Integer Multiply
SSE Integer ALU Integer Shuffles
out-of-order dispatch and execution
Nehalem RISC micro-operations
1 16B load /cycle
128 bits
128 bits
32 kiB L1 Data Cache
1 16B store /cycle

Figure 5: High-level diagram of a the out-of-order execution engine in the Nehalem core. All units are fully pipelined and can operate independently.

Memory Order Buffer (MOB) – Supports speculative and out of order loads and stores and ensures that writes to memory take place in the right order and with the right data.

Execution Units and Operand Forwarding Network The execution units are fully pipelined and can produce a result for most micro-ops with latency 1 cycle.

The IDQ unit (see Fig. 4) delivers a stream of micro-ops to the allocation/renaming stage of the EE pipeline. The execution engine of Nehalem supports up to **128** micro-ops in flight. The input data associated with a micro-op are generally either read from the ROB or from the retired register file. When a "dependency chain" across micro-ops causes the machine to wait for a "slow" resource (such as a data read from L2 data cache), the EE allows other micro-ops to proceed. The primary objective of the execution engine is to increase the flow of micro-ops, maximizing the overall rate of instructions reaching completion per cycle (IPC), without compromising program correctness.

Resource Allocation and Register Renaming for micro-ops The initial stages of the out of order core advance the micro-ops from the front end to the ROB and RS. This process is called micro-op *issue*. The RRAU in the out of order core carries out the following steps.

1. It allocates resources to micro-ops, such as,
   - an entry in the re-order buffer (ROB),
   - an entry in the reservation station (RS),
   - and a load/store buffer if a memory access is required.
2. It binds the micro-op to an appropriate "dispatch" port.
3. It "renames" source and destination operands of micro-ops in-flight, enabling out of order execution. Operands are registers or memory in general. Architected (program visible) registers are renamed onto a larger set of "micro-architectural" (or "non-architectural") registers. Modern processors contain a large pool of non-architectural registers, that is, registers which are not accessible from the code. These registers are used to capture results which are produced by independent computations but which happen to refer to the same architected register as destination. Register renaming eliminates these false dependencies which are known as "write-after-write" and "write-after-read" hazards. A "hazard" is any condition which could force a pipeline to stall to avoid erroneous results.
4. It provides data to the micro-op when the data is either an immediate value (a constant) or a register value that has already been calculated.

Unified Reservation Station (URS) queues micro-ops until all source operands are ready, then it schedules and dispatches ready micro-ops to the available execution units. The RS has **36** entries, that is, at any moment there is a window of up to 36 micro-ops waiting in the EE to receive input. A single scheduler in the Unified-Reservation Station (URS) dynamically selects micro-ops for dispatching to the execution units, for all operation types, integer, FP, SIMD, branch, etc. In each cycle, the URS can dispatch up to **six** micro-ops, which are ready to execute. A micro-op is ready to execute as soon as its input operands become available. The URS dispatches micro-ops through the **6 issue ports** to the execution units clusters. Fig. 5 shows the 6 issue ports in the execution engine. Each cluster may contain a collection of integer, FP and SIMD execution units.

The result produced by an execution unit computing a micro-op are eventually written back permanent storage. Each clock cycle, up to **4** results may be either written back to the RS or to the ROB. New results can be forwarded immediately through a **bypass network** to a micro-op in-flight that requires it as input. Results in the RS can be used as early as in the next clock cycle.

The EE schedules and executes next common micro-operations, as follows.

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

Nehalem Issue Ports and Execution Units The URS scheduler can dispatch up to six micro-ops per cycle through the six issue ports to the execution engine which can execute up to 6 operations per clock cycle, namely

- 3 memory operations (1 integer and FP load, 1 store address and 1 store data) and
- 3 arithmetic/logic operations.

The ultimate goal is to keep the execution units utilized most of the time. Nehalem contains the following components which are used to buffer micro-ops or intermediate results until the retirement stage

- 36 reservation stations
- 48 load buffers to track all allocate load operations,
- 32 store buffers to track all allocate store operations, and
- 10 fill buffers.

The execution core contains the three execution clusters, namely, SIMD integer, regular integer and SIMD floating-point/x87 units. Each blue block in Fig. 5 is a cluster of execution units (EU) in the execution engine. All EUs are fully pipelined which means they can deliver one result on each clock cycle. Latencies through the EU pipelines vary with complexity of the micro-op from 1 to 5 cycles Specifically, the EUs associated with each port are the following:

Port 0 supports

- Integer ALU and Shift Units
- Integer SIMD ALU and SIMD shuffle
- Single precision FP MUL, double precision FP MUL, FP MUL (x87), FP/SIMD/SSE2 Move and Logic and FP Shuffle, DIV/SQRT

Port 1 supports

- Integer ALU, integer LEA and integer MUL
- Integer SIMD MUL, integer SIMD shift, PSAD and string compare, and
- FP ADD

Port 2 Integer loads

Port 3 Store address

Port 4 Store data

Port 5 Supports

- Integer ALU and Shift Units, jump
- Integer SIMD ALU and SIMD shuffle
- FP/SIMD/SSE2 Move and Logic

The execution core also contains connections to and from the memory cluster (see Fig. 5).

Forwarding and By-pass Operand Network Nehalem can support write back throughput of one register file write per cycle per port. The bypass network consists of three domains of integer, FP and SIMD. Forwarding the result within the same bypass domain from a producer micro-op to a consumer micro-op is done efficiently in hardware without delay. Forwarding the result across different bypass domains may be subject to additional bypass delays. The bypass delays may be visible to software in addition to the latency and throughput characteristics of individual execution units.

The **Re-Order Buffer** (ROB) is a key structure in the execution engine for ensuring the successful out-of-order progress-to-completion of the micro-ops. The ROB holds micro-ops in various stages of completion, it buffers completed micro-ops, updates the architectural state in macro-instruction program order, and manages ordering of the various machine exceptions. On Nehalem the ROB has *128* entries to track micro-ops in flight.

Retirement and **write-back** of state to architected registers is only done for instructions and micro-ops that are on the correct instruction execution path. Instructions and micro-ops of incorrectly predicted paths are flushed as soon as mis-prediction is detected and the correct paths are then processed.

Retirement of the correct execution path instructions can proceed when two conditions are satisfied:

1. all micro-ops associated with the macro-instruction to be retired have completed, allowing the retirement of the entire instruction. In the case of instructions that generate very large numbers of micro-ops, enough to fill the retirement window, micro-ops may retire.
2. Older instructions and their micro-ops of correctly predicted paths have retired.

These requirements ensure that the processor updates the visible state consistently with the in-order execution of the macro-instructions of the code.

The advantages of this design is that older instructions which have to block waiting, for example, for the arrival of data from memory, cannot block younger, but *independent*, instructions and micro-ops, whose inputs are available. The micro-ops of these younger instructions can be dispatched to the execution units and warehoused in the ROB until completion.

3.3.5 Nehalem Core: Load and Store Operations

The memory cluster in the Nehalem core supports:

- peak issue rate of one **128**-bit (**16** bytes) load and one 128-bit store operation per clock cycle
- deep buffers for data load and store operations:
  - 48 load buffers,
  - 32 store buffers and

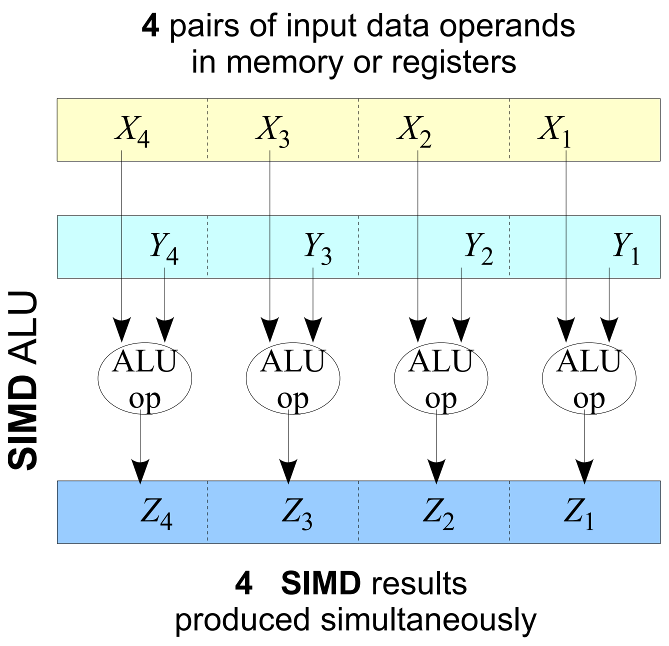

4 pairs of input data operands in memory or registers
X_4
X_3
X_2
X_1
Y_4
Y_3
Y_2
Y_1
SIMD ALU
ALU op
ALU op
ALU op
ALU op
Z_4
Z_3
Z_2
Z_1
4 SIMD results produced simultaneously

Figure 6: SIMD instructions apply the same FP or integer operation to collections of input data pairs simultaneously.

  – 10 fill buffers;

- fast unaligned memory access and robust handling of memory alignment hazards;
- improved store-forwarding for aligned and non-aligned scenarios, and
- store-to-load data forwarding for most address alignments.

Note that the h/w for memory access and its capabilities as seen by the core are described in detail in a later subsection.

3.4 Nehalem Core: Intel Streaming SIMD Extensions Instruction Set

Single-Instruction Multiple-Data (SIMD) is a processing technique were the same operation is applied simultaneously to different sets of input operands. Vector operations, such as, vector additions, subtractions, etc. are examples of computation where SIMD processing can be applied directly. SIMD requires the presence of multiple Arithmetic and Logic Units (ALUs) and multiple source and destination operands for these operations. The multiple ALUs can produce multiple results simultaneously using input operands. Fig. 6 illustrates an example SIMD computation against four operands. SIMD operating principle

Nehalem supports SIMD processing to integer or floating-point ALU intensive code with the Streaming SIMD Extensions (SSE) instruction set. This technology has evolved with time and now it represents a rather significant capability in Nehalem's micro-architectures. Fig. 7 illustrates the SIMD computation mode in Nehalem. On the left part of Fig. 7, two double-precision floating-point operations are applied to 2 DP input operands. On the right part of Fig. 7, four single-precision floating-point operations are applied to 4 SP input operands.

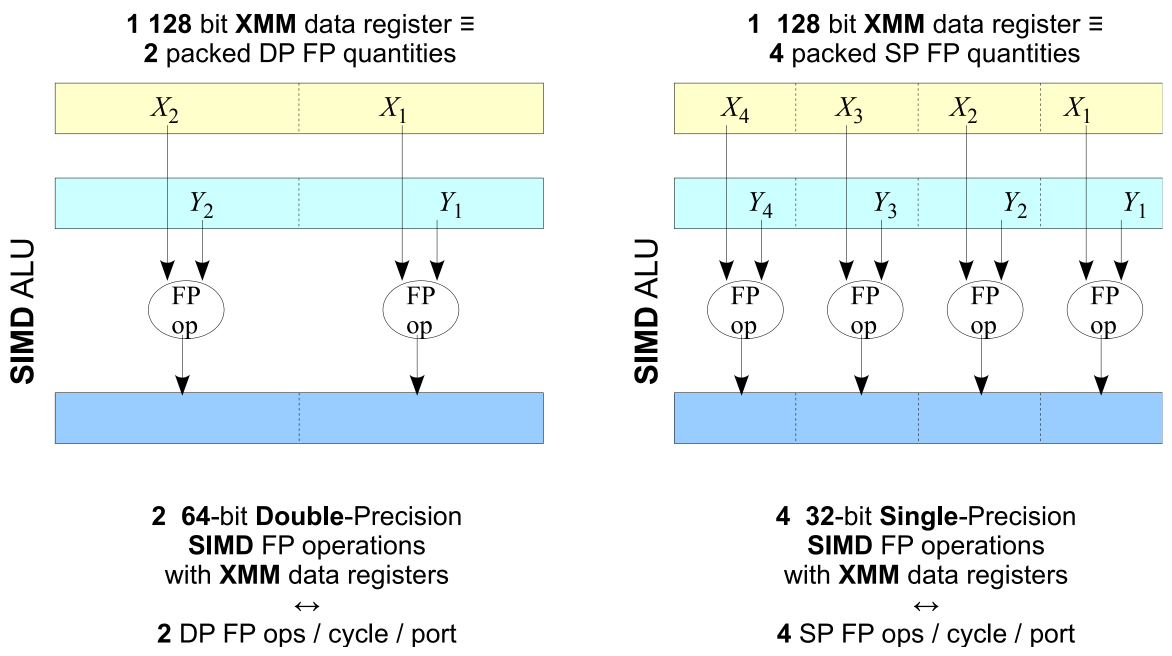

1 128 bit XMM data register ≡
2 packed DP FP quantities
X_2
X_1
Y_2
Y_1
SIMD ALU
FP op
2 64-bit Double-Precision SIMD FP operations with XMM data registers
↔
2 DP FP ops / cycle / port
1 128 bit XMM data register ≡
4 packed SP FP quantities
X_4
X_3
X_2
X_1
Y_4
Y_3
Y_2
Y_1
4 32-bit Single-Precision SIMD FP operations with XMM data registers
↔
4 SP FP ops / cycle / port

Figure 7: Floating-Point SIMD Operations in Nehalem.

3.4.1 Floating-Point SIMD operations in Nehalem core

Nehalem's execution engine (see Fig. 5) contains the ALU circuitry necessary to carry out **two** double-precision, or **four** single-precision "simple" FP operations, such as addition or subtraction, in each one of the two FP units accessible through ports 0 and 1. Note that Nehalem execution engine can retire up to **4** operations per clock cycle, including the SIMD FP ones.

Ideal Floating-Point Throughput For the Xeon 5560 which operates at 2.8GHz, we can say that in the steady state and under ideal conditions each core can retire **4** double-precision or **8** single-precision floating-point operations each cycle. Therefore, the nominal, ideal throughput of a Nehalem core, a quad core and a 2-socket system are, respectively,

$$\begin{aligned} \mathbf{11.2}\text{ Giga FLOPs/sec/core} &= 2.8\text{GHz} \times 4\text{FLOPs/Hz} \\ \mathbf{44.8}\text{ Giga FLOPs/sec/socket} &= 11.2\text{Giga FLOPs/sec/core} \times 4\text{cores} \\ \mathbf{89.6}\text{ Giga FLOPs/sec/node} &= 44.8\text{Giga FLOPs/sec/socket} \times 2\text{sockets}, \end{aligned} \tag{1}$$

in terms of double-precision FP operations.

3.4.2 Floating-Point Registers in Nehalem core

SIMD instructions use sets of separate core registers called MMX and XMM registers (shown in Fig. 8). The MMX registers are 64-bit in size and are aliased to the operand stack for the legacy x87 instructions. XMM registers are **128**-bit in size and each can store either **4** SP or **2** DP floating-point operands. The load and store units can retrieve and save 128-bit operands from cache or from the main memory.

One interesting feature of Nehalem's memory subsystem is that certain memory areas can be treated as *non-temporal*, that is, they can be used as *buffers* for vector data streaming in and out of the core, without requiring

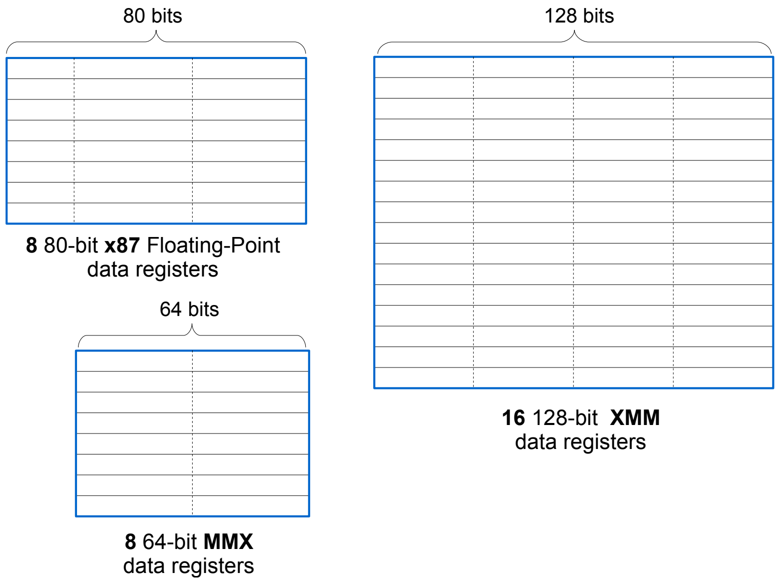

80 bits
8 80-bit x87 Floating-Point data registers
64 bits
8 64-bit MMX data registers
128 bits
16 128-bit XMM data registers

Figure 8: Floating-Point Registers in a Nehalem core.

their temporary storage in a cache. This is an efficient way to retrieve a stream of sub-vector operands from memory to XMM registers, carry out SIMD computation and then stream the results out directly to memory.

Overview of the SSE Instruction Set Intel introduced and extended the support for SIMD operations in stages over time as new generations of micro-architectures and SSE instructions were released. Below we summarize the main characteristics of the SSE instructions in the order of their appearance.

MMX(TM) Technology Support for SIMD computations was introduced to the architecture with the "MMX technology". MMX allows SIMD computation on packed byte, word, and double-word integers. The integers are contained in a set of eight 64-bit MMX registers (shown in Fig. 8).

Streaming SIMD Extensions (SSE) SSE instructions can be used for 3D geometry, 3D rendering, speech recognition, and video encoding and decoding. SSE introduced 128-bit XMM registers, 128-bit data type with four packed single-precision floating-point operands, data pre-fetch instructions, non-temporal store instructions and other cache-ability and memory ordering instructions, extra 64-bit SIMD integer support.

Streaming SIMD Extensions 2 (SSE2) SSE2 instructions are useful for 3D graphics, video decoding/encoding, and encryption. SSE2 add 128-bit data type with two packed double-precision floating-point operands, 128-bit data types for SIMD integer operation on 16-byte, 8-word, 4-double-word, or 2-quadword integers, support for SIMD arithmetic on 64-bit integer operands, instructions for converting between new and existing data types, extended support for data shuffling and extended support for cache-ability and memory ordering operations.

Streaming SIMD Extensions 3 (SSE3) SSE3 instructions are useful for scientific, video and multi-threaded applications. SSE3 add SIMD floating-point instructions for asymmetric and horizontal computation, a special-purpose 128-bit load instruction to avoid cache line splits, an x87 FPU instruction to convert to integer independent of the floating-point control word (FCW) and instructions to support thread synchronization.

Supplemental Streaming SIMD Extensions 3 (SSSE3) SSSE3 introduces 32 new instructions to accelerate eight types of computations on packed integers.

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

SSE4.2 Intel during 2008 introduced a new set of instructions collectively called as SSE4.2. SSE4 has been defined for Intel's 45nm products including Nehalem. A set of 7 new instructions for SSE4.2 were introduced in Nehalem architecture in 2008. The first version of SSE4.1 was present in the Penryn processor. SSE4.2 instructions are further divided into 2 distinct sub-groups, called "STTNI" and "ATA".

- STring and Text New Instructions (STTNI) operate on strings of bytes or words of 16bit size. There are four new STTNI instructions which accelerate string and text processing. For example, code can parse XML strings faster and can carry out faster search and pattern matching. Implementation supports parallel data matching and comparison operations.
- Application Targeted Accelerators (ATA) are instructions which can provide direct benefit to specific application targets. There are two ATA instructions, namely "POPCNT" and "CRC32".
  - POPCNT is an ATA for fast pattern recognition while processing large data sets. It improves performance for DNA/Genome Mining and handwriting/voice recognition algorithms. It can also speed up Hamming distance or population count computation.
  - CRC32 is an ATA which accelerates in hardware CRC calculation. This targets Network Attached Storage (NAS) using iSCSI. It improves power efficiency and reduces time for software I-SCSI, RDMA, and SCTP protocols by replacing complex instruction sequences with a single instruction.

Intel Advanced Vector Extensions AVX are several vector SIMD instruction extensions of the Intel64 architecture that will be introduced to processors based on **32**nm process technology. AVX will expand current SIMD technology as follows.

- AVX introduces **256**-bit vector processing capability and includes two components which will be introduced on processors built on 32nm fabrication process and beyond:
  - the first generation Intel AVX will provide 256-bit SIMD register support, 256- bit vector floating-point instructions, enhancements to 128-bit SIMD instructions, support for **three** and **four** operand syntax.
  - FMA is a future extension of Intel AVX, which provides **fused** floating-point multiply-add instructions supporting 256-bit and 128-bit SIMD vectors.
- General-purpose encryption and AES: 128-bit SIMD extensions targeted to accelerate high-speed block encryption and cryptographic processing using the Advanced Encryption Standard.

AVX will be introduced with the new Intel 32nm micro-architecture called "Sandy-Bridge".

Compiler Optimizations for SIMD Support in Executables User applications can leverage the SIMD capabilities of Nehalem through the Intel Compilers and various performance libraries which have been tuned up to take advantage of this feature. On EOS, use the following compiler options and flags.

- `-xHost` (or the `-xSSE4.2`) compiler options to instruct the compiler to use the entire set of SSE instructions in the generated binary
- `-vec` This option enables "vectorization" (better term would be "SIMDizations") and transformations enabled for vectorization. This effectively asks the compiler to attempt to use the SIMD SSE instructions available in Nehalem. Use the `-vec-report`*N* option to see which lines could use SIMD and which could not and why.
- `-O2` or`-O3`

Libraries Optimized for SIMD Support Intel provides user Libraries tuned up for SIMD computation. These include, Intel's **Math-Kernel Library** (MKL), Intel's standard math library (**libimf**) and the **Integrated-Performance Primitive** library (IPP). Please review the "`~/README`" file on your EOS home directory with information on the available software and instructions how to access it. This document contains, among other things, a useful discussion on compiler flags used for optimization of user code, including SIMD.

3.5 Floating-Point Processing and Exception Handling

Nehalem processors implement a floating-point system compliant with the ANSI/IEEE Standard 754-1985, "*IEEE Standard for Binary Floating-Point Arithmetic*". IEEE 754 defines required arithmetic operations (addition, subtraction, sqrt, *etc.*), the binary representation of floating and fixed point quantities and conditions which render machine arithmetic valid or invalid. Before this standard, different vendors used to have their own incompatible FP arithmetic implementations making portability of FP computation virtually impossible. When the result of an arithmetic operation cannot be considered valid or when precision is lost, the h/w handles a Floating-Point Exception (FPE).

The following floating-point exceptions are detected by the processor:

1. IEEE standard exception: invalid operation exception for invalid arithmetic operands and unsupported formats (#IA)
   - Signaling NaN
   - $\infty - \infty$
   - $\infty \div \infty$
   - $0 \div 0$
   - $\infty \times 0$
   - Invalid Compare
   - Invalid Square Root
   - Invalid Integer Conversion
2. Zero Divide Exception (#Z)
3. Numeric Overflow Exception (#O)
4. Underflow Exception (#U)
5. Inexact Exception (#P)

The standard defines the exact conditions raising floating point exceptions and provides well-prescribed procedures to handle them. A user application has a set of choices in how to treat and/or respond, if necessary, to these exceptions. However, detailed treatment of FPEs is far beyond the scope of this write up.

Please review the following presentation on IEEE 754 Floating-Point Standard and Floating Point Exception handling `http://sc.tamu.edu/systems/hydra/SC-FP.pdf` which apply to Nehalem. Note that this presentation is *under revision* but it is provides useful material for FP arithmetic.

3.6 Intel Simultaneous Multi-Threading

A Nehalem core supports "**Simultaneous Multi-Threading**" (SMT), or as Intel calls it "*Hyper-Threading*". SMT is a pipeline design and implementation scheme which permits *more than one* hardware threads to execute simultaneously within each core and share its resources. For Nehalem, two threads can be simultaneously executing within each core. Fig. 5 shows the different execution units within a Nehalem core which the two SMT threads can share.

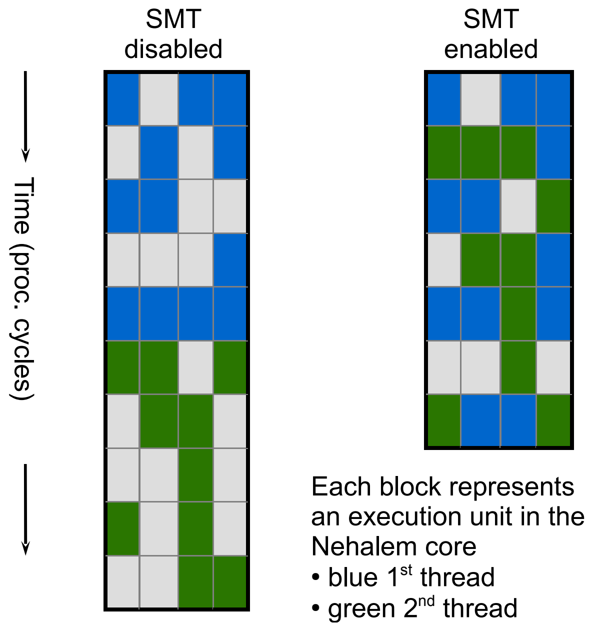


Figure 9: Simultaneous Multi-Treading (SMT) concept on Nehalem cores.

3.6.1 Basic SMT Principles

The objective of SMT is to allow the 2nd hardware thread to utilize functional units in a core which the 1st hardware thread leaves *idle*. In Fig. 9, the right-hand side part demonstrates the case where two threads execute simultaneously within a core with SMT enabled. The horizontal dimension shows the occupancy of the functional units of a core and the vertical one shows consecutive clock cycles. As you can see, both SMT threads may "simultaneously" (*i.e.*, at the same clock period) utilize these units, making progress.

The alternative to SMT would be to let a thread run until it has to stall (*e.g.*, waiting for a lengthy FP operation to finish or a cache memory miss to be handled), at which point in time the OS dispatcher would have to carry out a costly context-switching operation with processor state swapping. This is illustrated in an idealized fashion (*i.e.*, without accounting for the resource waste due to context-switching overhead) on the right-hand side part of the figure. SMT can potentially exploit "task-level" concurrency at a very fine level and produces cost saving by avoiding context-switching.

In short, the potential advantages of SMT are several, including among others, the increased utilization of functional units that would have remained idle, the overall increased throughput in instructions completed per clock cycle and the overhead savings from the lower number of thread switching operations. It implicitly can save power consumed by the idle units.

3.6.2 SMT in Nehalem cores

When SMT is ON, each Nehalem core appears to the Operating System as two logical processors. An SMT enabled dx360-M2 node appears as 16 logical processors to Linux.

On Nehalem, SMT takes advantage of the 4-wide execution engine. The units are kept busy with the two threads. SMT hides the latency experienced by a single thread. One prominent advantage is that with SMT it is more likely that an active unit will be producing some result on behalf of a thread as opposed to consuming power while it is waiting for work. Overall, SMT is much more efficient in terms of power than adding another core. One Nehalem, SMT is supported by the high memory bandwidth and the larger cache sizes.

3.6.3 Resources on Nehalem Cores Shared Among SMT Threads

The Nehalem core supports SMT by replicating, partitioning or sharing existing functional units in the core. Specifically the following strategies are used:

Replication The unit is replicated for each thread.

- register state
- renamed RSB
- large page ITLB

Partitioning The unit is statically allocated between the two threads

- load buffer
- store buffer
- reorder buffer
- small page ITLB

Competitive Sharing The unit is dynamically allocated between the two threads

- reservation station
- cache memories
- data TLB
- 2nd level TLB

SMT Insensitive All execution units are SMT transparent

3.7 CISC and RISC Processors

From the discussion above, it is clear that on the Nehalem processor, the CISC nature of the Intel64 ISA exits the scene after the instruction decoding phase by the IDU. By that time all CISC macro-instructions have been converted into RISC like micro-ops which are then scheduled dynamically for parallel processing at the execution engine. The specific execution engine of the Nehalem we described above could have been be part of any RISC or CISC processor. In deed one cannot tell by examining it if it is part of a CISC or a RISC processor. Please see a companion article on Power5+ `http://sc.tamu.edu/systems/hydra/hardware.php` processors and systems to make comparisons and draw some preliminary conclusions.

Efficient execution of applications is the ultimate objective and this requires an efficient flow of ISA macro-instructions through the processor. This implies accurate branch prediction and efficient fetching of instructions, their efficient decoding into micro-ops and a *maximal flow* of micro-ops from issue to retirement through the execution engine.

This points to one of the successes of the RISC approach where sub-tasks are simple and can be executed in parallel in multiple FUs by dynamic dispatching. Conversely, Nehalem has invested heavily in silicon real estate and clock cycles into preprocessing the CISC macro-instructions so that can be smoothly converted into sequences of micro-ops. The varying length of the CISC instructions requires the additional overhead in the ILD. A RISC ISA would had avoided this overhead and instructions would had moved directly from fetch to decoding stage.

At the same time, it is obvious that Intel has done a great job in processing very efficiently a heavy-weight CISC ISA, using all the RISC techniques. Thus the debate of RISC vs. CISC remains a valid and open question.

- Given modern back-end engines, which ISA style is more efficient to capture at a higher-level the semantics of applications?
- Is it more efficient to use a RISC back-end engine with a CISC or a RISC ISA and front-ends?
- It would be very interesting to see how well the Nehalem back-end execution engine would perform when fitted in a RISC processor front-end, handling a classical RISC ISA. For instance, how would a classical RISC, such as a Power5+ would perform if the Nehalem execution engine were to replace its own?
- Conversely, how would the Nehalem perform if it were fitted with the back-end execution engine of a classical RISC, such as that of an IBM Power5+ processor ?
- From the core designer point of view, can I select different execution engines for the same ISA ?

The old CISC vs. RISC debate is resurfacing as a question of how more aptly and concisely RISC or a CISC ISA can express the semantics of applications, so that when the code is translated into micro-ops powerful back-end execution engines can produce results at a lower cost, *i.e.*, in shorter amount of time and/or using less power?

4 Memory Organization, Hierarchy and Enhancements in Nehalem Processors

4.1 Cache-Memory and the Locality Phenomenon

The demand for increasingly larger data and instruction sections in applications requires that the size of the main memory hosting them be also sufficiently large. Experience with modern processors suggests that 2 to 4 GiB are needed per compute core to provide a comfortable size for a main memory. However, cost and power consumption for this large amounts of memory, necessitates the use of the so called, Dynamic Random Access Memory (DRAM) technology. DRAM allows the manufacturing of large amounts of memory using simpler memory elements (*i.e.*, by a transistor and a capacitor which needs to be dynamically refreshed every a few 10s of Milli-seconds). However, the bandwidth rates at which modern processors require to access memory in order to operate efficiently, far exceed the memory bandwidth that can be supported with current DRAM technologies.

Another type of memory, called “Static RAM” (SRAM) implements memory elements with more complex organization (5-6 transistors). SRAM is much faster than the DRAM and it does not require periodic refreshing of the bit contents. However, with more electronic components per bit, memory density per chip decreases dramatically while the power consumption grows. We cannot currently provide 2-4 GiB of RAM per core using just SRAM with a viable cost.

Computer architects design modern processors with multiple levels of faster, smaller and more expensive cache memories. Cache memories, mostly implemented with SRAM logic, maintain copies of recently and frequently used instruction and data blocks of the “main” (DRAM) memory. When an object is accessed for the first time, the hardware retrieves a block of memory containing it from the DRAM and stores it in the cache. Subsequent object accesses go directly to the faster cache and avoid the lengthy access to DRAM.

This is a viable approach due to the phenomenon of “**temporal**” and “**spatial locality**” in the memory access patterns exhibited by executable code. Simply speaking, temporal locality means that objects (data or instructions) accessed recently, have a higher probability to get accessed in the near future than other memory objects. Spatial locality means that objects physically adjacent in memory to those accessed recently have a higher probability of getting accessed “soon”. Temporal locality stems from the fact that within in a short span of time instructions in iterations (loops) are executed repeatedly likely accessing common data. Spatial locality is the result of code accessing dense array locations in linear order or simply accessing the next in sequence instruction. Hardware and compiler designers invest heavily in mechanisms which can leverage the locality phenomenon. Compilers strive to

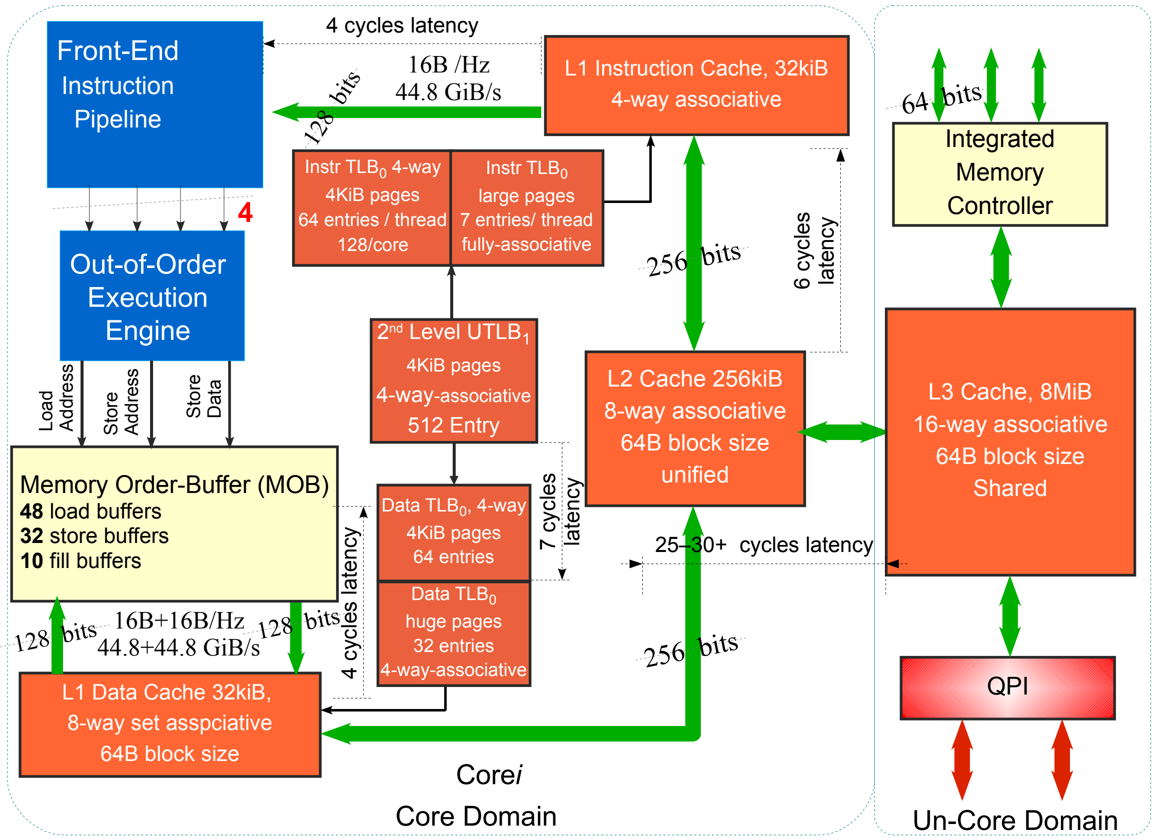

Front-End Instruction Pipeline
4 cycles latency
16B /Hz
44.8 GiB/s
128 bits
L1 Instruction Cache, 32kiB 4-way associative
Instr TLB0 4-way 4KiB pages 64 entries / thread 128/core
Instr TLB0 large pages 7 entries/ thread fully-associative
4
Out-of-Order Execution Engine
256 bits
6 cycles latency
2nd Level UTLB1 4KiB pages 4-way-associative 512 Entry
L2 Cache 256kiB 8-way associative 64B block size unified
Load Address
Store Address
Store Data
Memory Order-Buffer (MOB) 48 load buffers 32 store buffers 10 fill buffers
Data TLB0, 4-way 4KiB pages 64 entries
7 cycles latency
25-30+ cycles latency
Data TLB0 huge pages 32 entries 4-way-associative
4 cycles latency
128 bits
16B+16B/Hz
44.8+44.8 GiB/s
128 bits
256 bits
L1 Data Cache 32kiB, 8-way set asspciative 64B block size
Corei
Core Domain
64 bits
Integrated Memory Controller
L3 Cache, 8MiB 16-way associative 64B block size Shared
QPI
Un-Core Domain

Figure 10: Overview of Cache Memory Hierarchy and Data Flow Paths to and from Nehalem's Core.

co-locate items which are likely to be accessed together within short time spans. Hardware logic detects sequential memory access and attempts to pre-fetch subsequent blocks ahead of time. The cache memories eventually have to evict least used contents to make room for incoming new ones.

4.2 Cache-Memory Organization in Nehalem

Nehalem [1, 2] divides the physical memory into blocks 64 byte in size. These blocks, referred to as "cache blocks" or "cache lines", are the units of data the memory system transfers among the major subsystems.

The architecture supports a hierarchy of up to three levels of cache memory and DRAM memory. Fig. 10 shows the different caches in a Nehalem chip, their connectivity with the common L3, QPI and IMC, along with the TLBs translation structures.

Referring to Fig. 10, a Nehalem core contains an instruction cache, a first-level data cache and a second-level unified cache. Each physical processor chip may contain several processor cores and a shared collection of subsystems that are referred to as "uncore". Specifically, in Intel Xeon 5560 processors, there are four cores and the uncore provides a unified third-level cache shared by all cores in the chip, Intel QuickPath Interconnect ports and auxiliary logic such as, a performance monitoring unit, control configuration registers and power management units, among others.

The processor always reads a cache line from system memory beginning on a 64-byte boundary (which has an address with its 6 least-significant bits zero). A cache line can be filled from memory with 8-byte transfer burst transactions. The caches do not support partially-filled cache lines, so caching even a single double-word requires caching an entire line.

L1 Cache At Level 1 (L1), separate instruction and data caches are part of the Nehalem core (called a "Harvard" style). The instruction and the data cache are each 32 KiB in size. The L1 data-cache has a single access data port, and a block size of 64 bytes. In SMT mode, the caches are shared by the two hardware threads running in the core.

The instruction and the data caches have **4**-way and **8**-way set associative organization, respectively. The access latency to retrieve data already in L1 data-cache is **4** clocks and the "throughput" period is 1 clock.

L2 Cache Each core also contains a private, 256KiB, 8-way set associative, unified level 2 (L2) cache (for both instructions and data). L2's block size is 64 bytes and access time for data already in the cache is **10** clocks. The write policy is **write-back** and the cache is **non-inclusive**.

L3 Cache The Level 3 (L3) cache is a unified, 16-way set associative, 8 MiB cache shared by all four cores on the Nehalem chip. The latency of L3 access may vary as a function of the frequency ratio between the processor and the uncore sub-system. Access latency is around **35–40+** clock cycles, depending on the clock ration of the core and the uncore domains.

The L3 is **inclusive** (unlike L1 and L2), meaning that a cache line that exists in either L1 data or instruction, or the L2 unified caches, also exists in L3. The L3 is designed to use the inclusive nature to minimize "snoop" traffic between processor cores and processor sockets. A 4-bit valid vector indicates if a particular L3 block is already cached in the L2 or L1 cache of a particular core in the socket. If the associated bit is not set, it is certain that this core is not caching this block. A cache block in use by a core in a socket, is cached by its L3 cache which can respond to snoop requests by other chips, without disturbing (snooping into) L2 or L1 caches on the same chip. The write policy is write-back.

4.3 Nehalem Memory Access Enhancements

The data path from L1 data cache to the memory cluster is 16 bytes in each direction. Nehalem maintains load and store buffers between the L1 data cache and the core itself.

4.3.1 Store Buffers

Intel64 processors temporarily store data for each write (store operation) to memory in a **store buffer** (SB). SBs are associated with the execution unit in Nehalem cores. They are located between the core and the L1 data-cache. SBs improve processor performance by allowing the processor to continue executing instructions without having to wait until a write to memory and/or to a cache is complete. It also allows writes to be delayed for more efficient use of memory-access bus cycles.

In general, the existence of store buffers is transparent to software, even in multi-processor systems like the Nehalem-EP. The processor ensures that write operations are always carried out in program order. It also insures that the contents of the store buffer are always drained to memory when necessary.

- when an exception or interrupt is generated;
- when a serializing instruction is executed;
- when an I/O instruction is executed;
- when a LOCK operation is performed;
- when a BINIT operation is performed;
- when using an SFENCE or MFENCE instruction to order stores.

4.3.2 Load and Store Enhancements

The memory cluster of Nehalem supports a number of mechanisms which speed up memory operations, including

- out of order execution of memory operations,
- peak issue rate of one **128**-bit load and one **128**-bit store operation per cycle from L1 cache,
- "deeper" buffers for load and store operations: **48** load buffers, **32** store buffers and **10** fill buffers,
- data pre-fetching to L1 caches,
- data pre-fetch logic for pre-fetching to the L2 cache
- fast unaligned memory access and robust handling of memory alignment hazards,
- memory disambiguation,
- store forwarding for most address alignments and
- pipelined read-for-ownership operation (RFO).

Data Load and Stores Nehalem can execute up to one 128-bit load and up to one 128-bit store per cycle, each to different memory locations. The micro-architecture enables execution of memory operations out-of-order with respect to other instructions and with respect to other memory operations.

Loads can

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

Loads cannot

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

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

Stores to memory are executed in two phases:

Execution Phase Prepares the store buffers with address and data for store forwarding (see below). Consumes dispatch ports 3 and 4.

Completion Phase The store is retired to programmer-visible memory. This may compete for cache banks with executing loads. Store retirement is maintained as a background task by the Memory Order Buffer, moving the data from the store buffers to the L1 cache.

Data Pre-fetching to L1 Caches Nehalem supports hardware logic (DPL1) for two data pre-fetchers in the L1 cache. Namely

Data Cache Unit Prefetcher DCU (also known as the "streaming prefetcher"), is triggered by an ascending access to recently loaded data. The logic assumes that this access is part of a streaming algorithm and automatically fetches the next line.

Instruction Pointer-based Strided Prefetcher IPSP keeps track of individual load instructions. When load instructions have a regular stride, a prefetch is sent to the next address which is the sum of the current address and the stride. This can prefetch forward or backward and can detect strides of up to half of a 4KB-page, or 2 KBytes.

Data prefetching works on loads only when loads is from write-back memory type, the request is within the page boundary of 4 KiB, no fence or lock is in progress in the pipeline, the number of outstanding load misses in progress are below a threshold, the memory is not very busy and there is no continuous stream of stores waiting to get processed.

L1 prefetching usually improves the performance of the memory subsystem, but in rare occasions it may degrade it. The key to success is to issue the pre-fetch to data that the code *will use in the near future* when *the path from memory to L1 cache is not congested*, thus effectively spreading out the memory operations over time. Under these circumstances pre-fetching improves performance by anticipating the retrieval of data in large sequential structures in the program. However, it may cause some performance degradation due to bandwidth issues if access patterns are sparse instead of having spatial locality.

On certain occasions, if the algorithm's working set is tuned to occupy most of the cache and *unneeded* pre-fetches evict lines required by the program, hardware prefetcher may cause severe performance degradation due to cache capacity of L1.

In contrast to hardware pre-fetchers, software prefetch instructions relies on the programmer or the compiler to anticipate data cache miss traffic. Software prefetch act as hints to bring a cache line of data into the desired levels of the cache hierarchy.

Data Pre-fetching to L2 Caches DPL2 pre-fetch logic brings data to the L2 cache based on past request patterns of the L1 to the L2 data cache. DPL2 maintains two independent arrays to store addresses from the L1 cache, one for upstreams (12 entries) and one for down streams (4 entries). Each entry tracks accesses to one 4K byte page. DPL2 pre-fetches the next data block in a stream. It can also detect more complicated data accesses when intermediate data blocks are skipped. DPL2 adjusts its pre-fetching effort based on the utilization of the memory to cache paths. Separate state is maintained for each core.

Memory Disambiguation A load instruction micro-op may depend on a preceding store. Many micro-architectures block loads until all preceding store address are known. The memory disambiguator predicts which loads will not depend on any previous stores. When the disambiguator predicts that a load does not have such a dependency, the load takes its data from the L1 data cache. Eventually, the prediction is verified. If an actual conflict is detected, the load and all succeeding instructions are re-executed.

Store Forwarding When a load data follows a store which reloads the data the store just wrote to memory, the micro-architecture can forward the data directly from the store to the load in many cases. This is called "tore-to-load" forwarding, and it saves several cycles by allowing a data requester receive data already available on the processor instead of waiting for a cache to respond. However several conditions must be met for store to load forwarding to proceed without delays:

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

In previous micro-architectures specific address alignments and data sizes between the store and load operations would determine whether a store-to-load forwarding might proceed directly or get delayed going through the cache/memory sub-system. Intel micro-architecture (Nehalem) allows store-to-load forwarding to proceed regardless of store address alignment.

Efficient Access to Unaligned Data The cache and memory subsystems handle a significant amount of instructions and data with different address alignment scenarios. Different address alignments have varying performance impact on memory and cache operations based on the implementation of these subsystems. On Nehalem the data path to the L1 caches are 16 bytes wide. The L1 data cache can deliver 16 bytes of data in every cycle, regardless how their addresses are aligned. However, if a 16-byte load spans across a cache line boundary, the data transfer will suffer a mild delay in the order of 4 to 5 clock cycles. Prior micro-architectures imposed much heavier delays.

5 Nehalem-EP Main Memory Organization

5.1 Integrated Memory Controller

The integrated memory controller (IMC) for Nehalem supports three 8-byte channels of DDR3 memory operating at up to **1.333** GigaTransfer/sec (GT/s). Fig. 11 shows the IMC in a Nehalem chip. Total theoretical bandwidth between DRAM and the IMC in the un-core domain of the chip is **31.992** GB/s. The memory controller supports both registered and un-registered DDR3 DRAM. Each channel of memory can operate independently and the controller services requests out-of-order to minimize latency. Each core supports up to **10** data cache misses and **16** total outstanding misses. This places a *strict upper bound on the memory bandwidth per core.*

5.2 Cache-Coherence Protocol for Multi-Processors

The conveniences of the cache memories come with some extra cost when the system has multiple processors. Copies of data which have been retrieved and modified by a processor in its local cache are inconsistent with the original copy in main memory. When another processor accesses the same data item it should receive the latest up-to-date copy and not an older stale version of it. This problem of **Memory Consistency** is addressed with **Cache Coherence** (CC) mechanisms. CC ensures that the value of an item retrieved by any processor in the system is the most up-to-date one. CC may add considerable overhead in accessing memory in multi-processors. CC logic is in the critical path of accessing memory and can easily become the main bottleneck, exacerbating the processor and memory speed gap. Recent processors provide increasingly tuned and adaptive CC protocols which try to stay to any extend feasible out of the way in accessing memory. Ideally, accesses to disjoint data by separate processors should proceed without any additional overhead. Conflicting access to the same data item (reads and writes) by different processors should extend the latency as minimally as possible, maintain fairness and avoid indefinite postponement. Cache coherence mechanism have been studied extensively in the literature and still are hot topics as there is always an increasing demand for larger multi-processors and more efficient concurrent access to shared memory.

5.2.1 Cache-Coherence Protocol (MESI+F)

Practical reasons require that CC protocols maintain memory consistency in terms of 64-byte memory blocks and not in individual bytes or words. Memory blocks are the units of physical memory transfer. Each block has a unique identification, and belongs to a unique Nehalem socket ("home location") and is managed by the local IMC. Based on the way and time they propagate modifications of the blocks, CC protocols are divided into 2 categories, namely write-update and write-invalidate. Based on the way they locate multiple copies of the same block, are divided into "snoopy" and the directory based protocols. For the discussion which follows please refer to Fig. 11.

Nehalem processors use the **MESIF** (Modified, Exclusive, Shared, Invalid and Forwarding) cache protocol to maintain cache coherence with caches on the same chip and on other chips via the QPI. MESIF belongs to the write-invalidate, snoopy (with a small directory part) category, and it is a variation of the well known MESI protocol. The designations used in the its acronym are the possible states that cache memory blocks can be in as they are transfered among cores, caches, I/O and DRAM. When a core reads or modifies memory objects causes the their corresponding

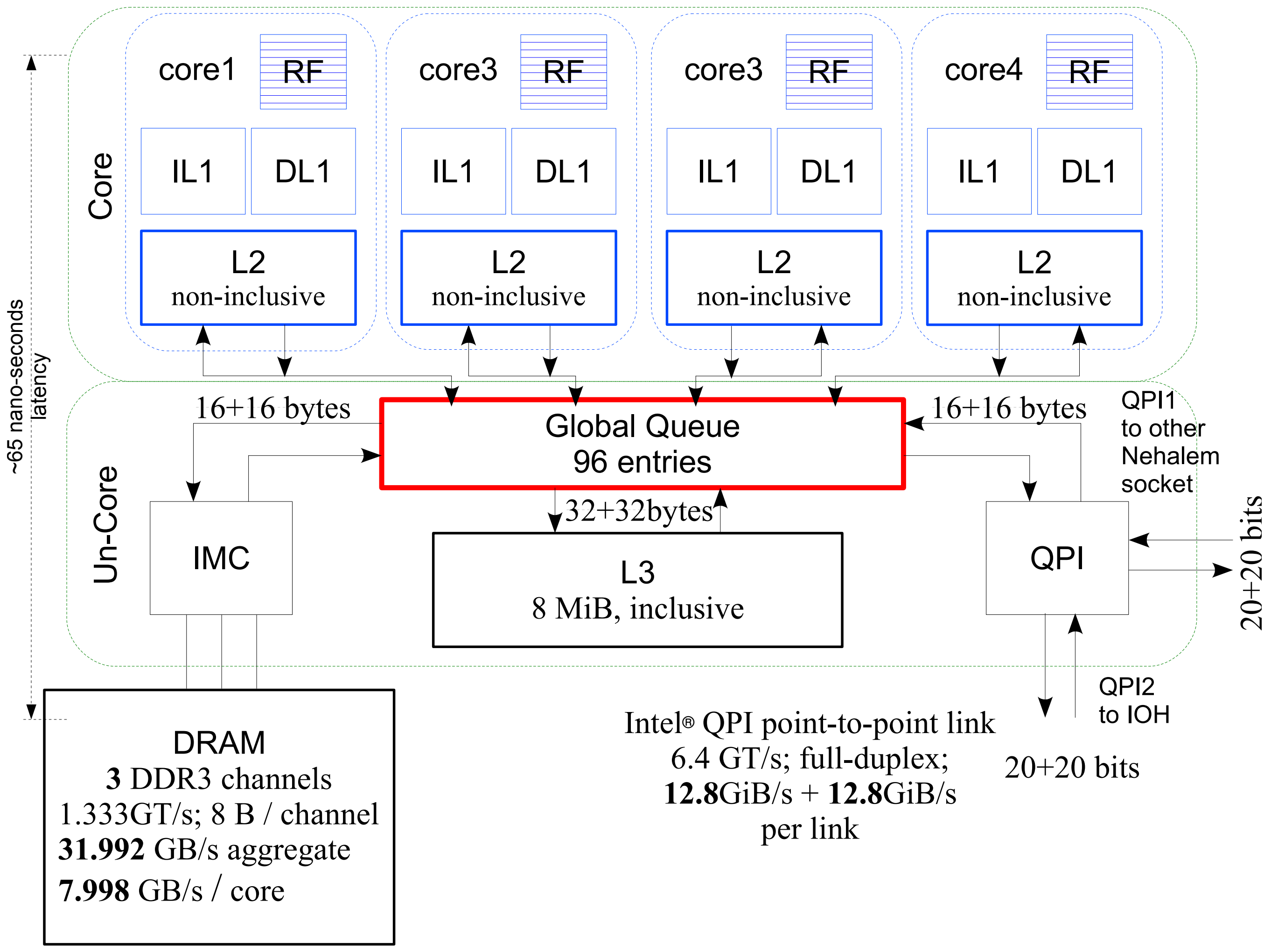


Figure 11: Nehalem On-Chip Memory Hierarchy and Data Traffic through the Chip and Global Queue.

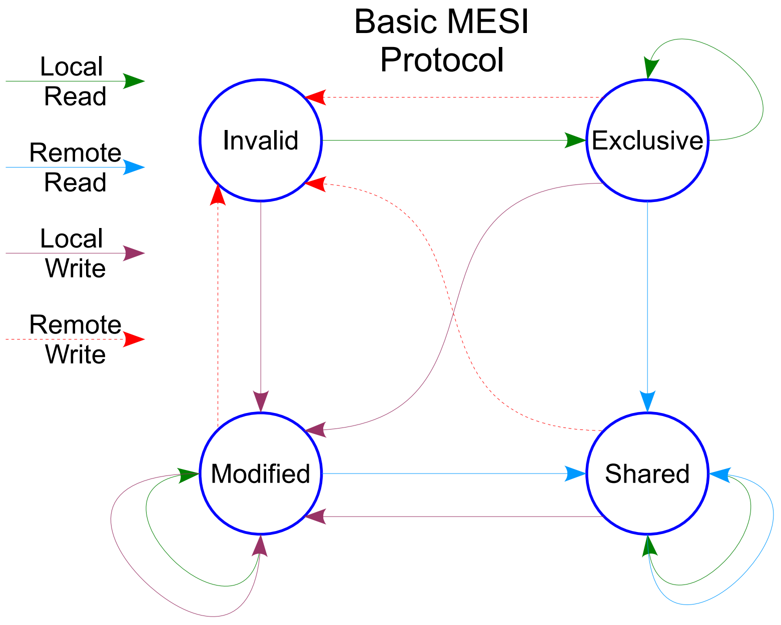


Figure 12: The State Transitions of Cache Blocks in the Basic MESI Cache-Coherence Protocol.

block to transition from one of these states to another. The current state of a block and the requested operation against it prescribes the h/w to follow a different sequence of tasks which provably maintain memory consistency.

5.2.2 Basic MESI Protocol

Initially all blocks in a cache do not store actual data and they are in the "**Invalid**" state. When a core reads a data object, it always checks first the L1 memory to see if the block is already there. The first time a block is accessed results in a cache **read miss** and it is in the Invalid state. If the block is not in L1 but is found in the L2 cache (L1 miss and L2 hit), it is transferred to the L1 data cache and the data access instruction proceeds. If the block is neither in the L2, then it must be retrieved from the "un-core". In general, a read-miss causes the core to retrieve the entire 64-byte cache block containing the object into the appropriate cache (L1, L2, L3, or all). This operation is called a **cache-line fill**. A block read for the first time by any core transitions to the "**Exclusive**" state.

The next time a core needs to access the same or nearby memory locations, if the block is still in the cache the data object is retrieved directly from the cache instead of going back to memory. This is called a **cache hit**.

When a core A has already retrieved a block in the Exclusive state and core B requires to read the same block, the cache coherence h/w stores a copy of this block in the cache of core B and changes its state to "**Shared**".

When a core wants to write an operand to memory, it first checks if the corresponding block is already in the cache. If a valid cache line does exist, the processor can write the operand into the cache instead of writing it out to system memory. This operation is called a **write hit**.

A write which refers to a memory location not currently in the cache causes a **write-miss**. In this case the core performs a cache line-fill, **write allocation** and proceeds to modify the value of the operand in the cache line without writing directly to memory. The Intel64 architecture does not write-allocate on a write miss if when the write operation is "non-temporal". When data is "streamed" in and out of the processor, as the case may be with SIMD/vector operands, there is no need to store this data in the cache as it is not expected to be needed in the near future. Write allocate operations are costly especially when the block to enter the cache would have to evict a modified block already resident in a conflicting cache location. The Intel compilers may select the non-temporal write operations when it is evident that the written data are streamed out of the processor.

When a core attempts to modify data in a block in the Shared state, the h/w issues a "Request-for-Ownership" (RfO) transaction which **invalidates** all copies in other caches and transitions its own (unique now) copy to Modified state. The owning core can read and write to this block without having to notify the other cores. If any of the cores previously sharing this block attempts to read this block, it will receive a cache-miss since the block is Invalid in that core's cache. Note that when a core attempts to modify data in a Exclusive state block, NO "Request-for-Ownership" transaction is necessary since it is certain that no other processor is caching copies of this block.

For Nehalem which is a multi-processor platform, the processors have the ability to "**snoop**" (eavesdrop) the address bus for other processor's accesses to system memory and to their internal caches. They use this snooping ability to keep their internal caches consistent both with system memory and with the caches in other interconnected processors.

If through snooping one processor detects that another processor intends to write to a memory location that it currently has cached in Shared state, the snooping processor will invalidate its cache block forcing it to perform a cache line fill the next time it accesses the same memory location.

If a core detects that another core is trying to access a memory location that it has modified in its cache, but has not yet written back to system memory, the owning core signals the requesting core (by means of the "HITM#" signal) that the cache block is held in Modified state and will perform an implicit write-back of the modified data. The implicit write-back is transferred directly to the requesting core and snooped by the memory controller to assure that system memory has been updated. Here, the processor with the valid data can transfer the block directly to the other core without actually writing it to system memory; however, it is the responsibility of the memory controller to snoop this operation and update memory.

Each memory block can be stored in a unique set of cache locations, based on a subset of their memory block identification. A cache memory with associativity K can store each memory block to up to K alternative locations. If all K cache slots are occupied by memory blocks, the $K + 1$ request will not have room to store this latest memory block. This requires that one of the existing K blocks has to be written out to memory (or the inclusive L3 cache) if this block is in Modified state. Cache memories commonly use a Least Recently Used (LRU) **cache replacement** strategy where they evict the block which has not been accessed recently.

As we mentioned, before written out to memory, data operands are first saved in a store buffer. They are then written from the store buffer to memory when the system path to memory is available.

Note that when all 10 of the line-fill buffers in a core become occupied, outstanding data access operations queue up in the load and store buffers and cannot proceed. When this happens the core's front end suspends issuing new micro-ops to the RS and OOO engine to maintain pipeline consistency.

5.2.3 The Un-Core Domain and Multi-Socket Cache Coherence

In the Nehalem processor the "un-core" domain essentially is a shared last level L3 cache ("LLC"), a memory access chip-set ("Northbridge"), and a QPI socket interconnection interface [10]. Fig. 11 illustrates a quad-core Nehalem processor chip and the connections between the cores, the cache memories, the local DRAM and the inter-socket QPI links. Technical information published about the un-core does not go into sufficient depth to permit an understanding of the underlying h/w capabilities and limitations. Note that in Section 7 we attempt to quantify the performance of the Nehalem-EP based actual low level cache and memory access experiments.

The un-core supports cache line access requests (such as, L2 cache misses, "un-cacheable" loads and stores) from the on-chip cores and transfers data among the L3 cache, the local IMC, the remote socket and the I/O Hub.

Memory consistency in a multi-core, multi-socket system like the Nehalem-PE, is maintained across sockets. With the introduction of the Intel Quick-Path Interconnect protocol the, 4 MESI states are supplemented with a fifth, "Forward" (F) state. A specific cache memory is chosen to store a block in the Forward state and it is allowed to forward it to other requesters. Blocks in the Shared state cannot be directly forwarded. Forwarding of a block from a single cached location only is done to avoid cache coherence protocol complexity and unnecessary data transfers. This would occur if multiple caches were allowed to forward the same block to a requester. Forwarding is done beneficially when a block belongs to a remote IMC, it is already cached by the remote L3 and it can be forwarded by th L3 faster than accessing the remote DRAM.

Cache line requests from the on-chip four cores, from a remote chip or the I/O hub are handled by the **Global Queue** (GQ) (see Fig. 11) which resides in the Uncore. The GQ buffers, schedules and manages the flow of data traffic through the uncore. The operations of the GQ is most-critical for the efficient exchange of data within and among Nehalem processor chips. The GQ contains **3** request queues for the different request types:

Write Queue (WQ): is a 16-entry queue for store (write) memory access operations from the local cores.

Load Queue (LQ): is a 32-entry queue for load (read) memory requests by the local cores.

QPI Queue (QQ): is a 12-entry queue for off-chip requests delivered by the QPI links.

According to [10], each pair of cores share one port to the Global Queue. Thus for the two core pairs on a chip there is a total of 64 request buffers for read operations to the local DRAM. We suspect that this buffer limit places an artificial upper bound in the available read bandwidth to the local DRAM. Unfortunately, no other information is available to elucidate the limitations of the GQ h/w. A “cross-bar” switch assists GQ to exchange data among the connected parts.

When the GQ receives a cache line request from one of the cores, it first checks the on-chip Last Level Cache (L3) to see if the line is already cached there. As the L3 is inclusive, the answer can be quickly determined. If the line is in the L3 and was owned by the requesting core it can be returned to the core from the L3 cache directly. If the line is being used by multiple cores, the GQ snoops the other cores to see if there is a modified copy. If so the L3 cache is updated and the line is sent to the requesting core. In the event of an L3 cache miss, the GQ sends out requests for the line. Since the cache line could be cached in the other Nehalem chip, a request through the QPI to the remote L3 cache is made. As each Nehalem processor chip has its own local integrated memory controller, the GQ must identify the “**home**” location of the requested cache line from the physical address. If the address identifies home as being on the local chip, then the GQ makes a simultaneous request to the local IMC. If home belongs to the remote chip, the request sent by the QPI will also be used to access the remote IMC.

This process can be viewed in the terms of the QPI protocol as follows. Each socket has a “Caching Agent” (CA) which might be thought of as the GQ plus the L3 cache and a “Home agent” (HA) which is the IMC. An L3 cache miss results in simultaneous queries for the line from all the CAs and the HA (wherever home is). In a Nehalem-EP system there are **3** caching agents, namely the 2 sockets and an I/O hub. If none of the CAs has the cache line, the home agent ultimately delivers it to the caching agent that requested it. Clearly, the IMC has queues for handling local and remote, read and write requests.

5.2.4 Local vs. Remote Memory Access

In Nehalem, the integrated memory controller substantially improved memory latency and bandwidth, compared to predecessor micro-architectures. For the two socket implementations of Nehalem EP (see Fig. 13), the remote latency is higher, since the memory request and response must go through a QPI link. This shared memory organization is called “cache-coherent Non-Uniform Memory Access” (cc-NUMA) and it is very common in modern SMP platforms. The latency to access the local memory is, approximately, **65** nano-seconds. The latency to access the remote memory is, approximately, **105** nano-seconds. That is, remote accesses are 1.6 to 1.7 times the latency of local memory access.

The available bandwidth through the QPI link is **12.8** GB/s which is approximately **%40** of the theoretical bandwidth of the three local DDR3 channels.

Access to Local Memory DRAM Fig. 14 demonstrates access to a memory block whose home location is in the directly (locally) attached DRAM. The sequence of steps are the following:

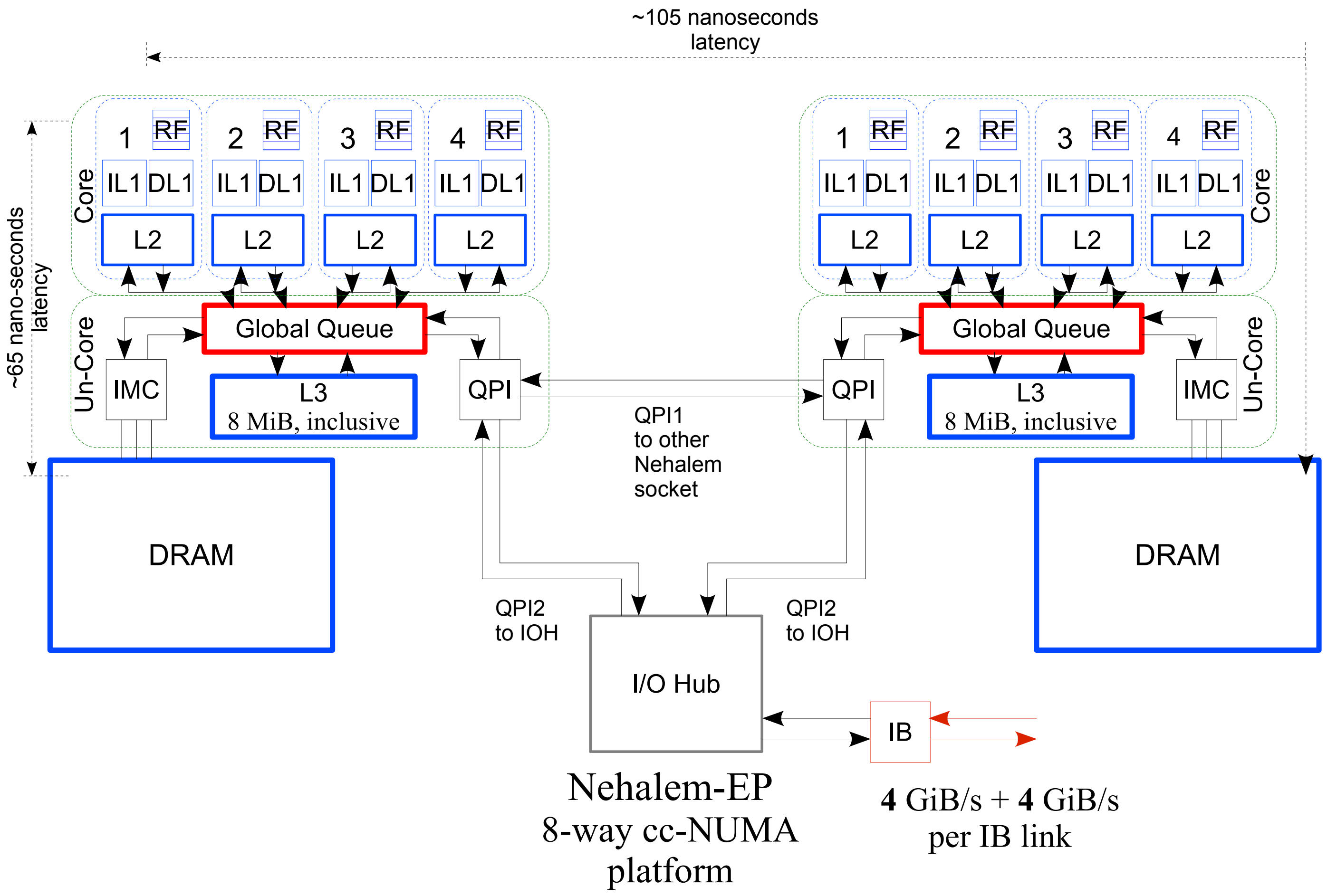


Figure 13: Nehalem-EP 8-way cc-NUMA SMP, Memory Hierarchies and Local vs. Remote Memory Organization.

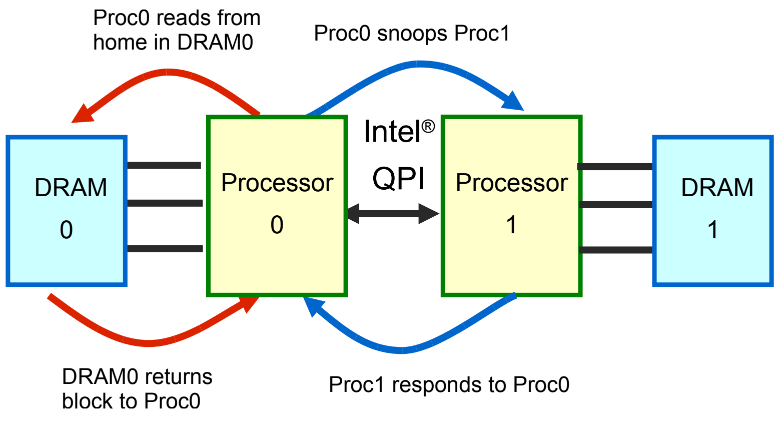


Figure 14: Nehalem Local Memory Access Event Sequence.

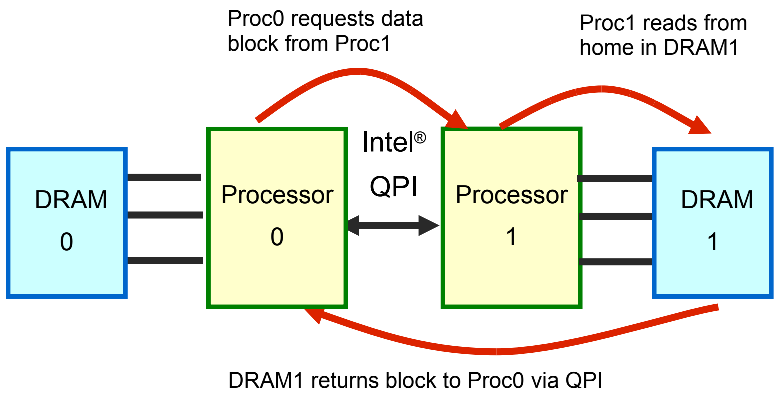

Proc0 requests data block from Proc1
Proc1 reads from home in DRAM1
DRAM 0
Processor 0
Intel®
QPI
Processor 1
DRAM 1
DRAM1 returns block to Proc0 via QPI

Figure 15: Nehalem Remote Memory Access Event Sequence.

1. Proc0 requests a cache line which is not in its L1, l2 nor in shared L3 cache
   - Proc0 requests data from its DRAM,
   - Proc0 snoops Proc1 to check if data is present there.
2. Response
   - local DRAM returns data;
   - Proc1 returns snoop response;
   - Proc0 installs block in its L3, L2 and L1 cache and retrieves target memory word.

The local memory access latency is the maximum of the above two steps.

Access to Remote Memory DRAM Fig. 15 illustrates access to a memory block whose home location is the remote DRAM memory (directly attached to the other Nehalem chip). The steps to access the remote memory block are the following:

1. Proc0 requests a cache line which is in not in Proc0's L1, L2 or L3 cache;
2. Request sent over QPI to Proc1
3. Proc1's probes for cache line
   - Proc1's IMC makes requests to its own DRAM;
   - Proc1 snoops internal caches;
4. Response
   - Data block returns to Proc0 via the QPI;
   - Proc0 installs cache block in L3, L2 and L1;
   - Proc0 sets the state of cache block to Shared or Exclusive.

The remote memory access latency is the sum of steps 1, 2, 3 and 4 and clearly is a function of QPI latencies.

The cache coherency protocol messages, among the multiple sockets, are exchanged over the Intel QPI. The inclusive L3 cache mode allows this protocol to operate rather fast, with the latency to the L3 cache of the adjacent socket being *even less than the latency to the local memory.*

One of the main virtues of the integrated memory controller is the separation of the cache coherency traffic and the memory access traffic. This enables a tangible increase in memory access bandwidth, compared to previous architectures, but it results in a non-uniform memory access (NUMA). The latency to the memory DIMMs attached to a remote socket is *considerably longer than to the local DIMMs.* A second advantage is that the memory control logic can run at processor frequencies and thereby reduce the latency.

6 Virtual Memory in Nehalem Processors

6.1 Virtual Memory

In modern processors, an executable is logically divided into "sections", or "segments" which have distinct function. For instance, an executable consists, among others, of a "code", a "stack" and a "heap" segment. For reasons of effective space management and efficient utilization, each segment is divided into logical units, called **pages**. A page in different varies from a few KiBytes in size, (*e.g.*, 4 KiB) to several MiBytes or even GiBytes.

Each page can actually be stored anywhere in the physical memory, in any of the available main memory slots known as **page frames**. We can consider the main memory as an array of pages frames. As an example, a physical memory with 4GiBytes capacity has available exactly 1 Mi page frames for pages having 4KiB size. Applications can define vast amounts of memory, but they usually refer or access a very small subset of it. When a program references for the first time a memory location (for instance a new subroutine call or a reference to a data item in an array) the system selects a free page frame and "pages in" the corresponding page. Application pages already in page slots which have not been recently used become candidates for eviction to make room for new pages.

The mechanism which dynamically manages the page frames and keeps track of the mapping between pages and page frames is called the **Virtual Memory** (VM) management system.

While applications execute refer to memory using "Effective Addresses" (EA) which are virtual memory addresses. "Physical Addresses" (PA) are actual addresses the memory hardware uses to identify specific memory locations. The VM system dynamically translates EAs into PAs, in process called "Virtual Address Translation". VM systems keeps track of the various program segments and corresponding pages with in memory in data structures called VM segment and page tables. These structures end up taking plenty of memory space. Multiple levels of page tables are used to cut down on the actual space used. This multi-level indirection requires the *traversal of multiple tables for each address* an application uses and it is in the critical path of the tasks the processor has to carry out in order to retire each macro-instruction. For this reason, special hardware called "Translation Look-aside Buffers" (TLBs) is used to speed up this process.

6.2 Nehalem Address Translation Process

Address Sizes Intel64 architecture defines translation from a "flat", linear **64**-bit "Effective Address" (EA) into a "Physical Address" (PA) with a width to up to **52** bits [2, 11]. Note that even though this mode produces 64-bit linear addresses, the processor ensures that bits 63:47 of such an address are identical ("sign extension" of bit 47). This implies that the Virtual Address (VA) the paging system uses has an effective width of 48 bits. Although 52 bits corresponds to 4 PiBytes, since linear addresses are limited to 48 bits, at most 256 TiBytes of linear-address space may be accessed at any given time by a single process. Note finally that on Nehalem, the physical address size is actually **40** bits.

VM Page Sizes Supported Nehalem support virtual memory page sizes of **4** KiB, **2** MiB and **1** GiB "huge" pages. Please refer to Fig. 11 which shows in detail the hardware components involved in the EA to PA translation process

and the various levels of cache memories.
TLBs The processor architecture specifies **two-levels** of translation look-aside buffers, TLB_0 and TLB_1 to speed-up the EA to PA translation process. The TLB is a cache of *recent EA to PA translations.* Intel64 allows TLBs and other h/w structures to cache address mappings information which is stored in main memory. When a process executes it is associated with an integer **Process-Context Identifier** (PCId). In this way TLBs and other structures may cache multiple mappings associated with different processes simultaneously. The first level TLB_0 consists of separate TLBs for data $DTLB_0$ and instructions $ITLB_0$. $DTLB_0$ handles address translation for data accesses, it provides 64 entries to support 4KiB pages and 32 entries for large pages. The $ITLB_0$ provides 64 entries (per thread) for 4KiB pages and 7 entries (per thread) for large pages. The second level unified $UTLB_1$ handles both code and data accesses for 4KiB pages. It support 4KiB page translation operation that missed $ITLB_0$ or $DTLB_0$.
Here is a list of entries in each TLB

$ITLB_0$ for 4-KiB pages: 64 entries / SMT thread, 4-way associative;

$ITLB_0$ for 2-MiB "huge" pages: 7 entries / SMT thread, fully associative;

$DTLB_0$ for 4-KiBe pages: 64 entries, 4-way associative;

$DTLB_0$ for 2-MiB pages : 32 entries, 4-way associative;

$UTLB_1$ for 4-KiB pages: 512 entries for both data and instruction look-ups.

An $DTLB_0$ miss and $UTLB_1$ hit causes a penalty of **7** cycles. Software only pays this penalty if the $DTLB_0$ is used in some dispatch cases. The delays associated with a miss to the $UTLB_1$ and Page-Miss Handler are largely non-blocking.

7 Nehalem Main Memory Performance

This section presents actual performance measurements of the cache and memory systems in a Nehalem-EP platforms. The experiments investigate the performance of basic, low-level memory operations ad allow us to quantify the actual performance limits. We compare latencies and bandwidth of different access patterns with those allowed by the hardware under ideal conditions. Even though not comprehensive, these results offer a good insight into actual performance limits of the memory subsystems in the Nehalem-EP platform. A subsequent publication will present more thorough memory system performance evaluation of Nehalem and likely of the Westmere which is the follow-on platform for Intel64 systems.

7.1 Ideal Performance Limits

Ideal or "Theoretical" data transfer figures are obtained by simply multiplying the transfer rates times the width of the various system channels. Please review Sections 4 and 5 for a discussion of the memory architecture on Nehalem-EP platforms and the ideal performance figures. Here we focus on the Xeon 5560 processor which has a core domain clock of 2.8GHz. Each core in a socket should be able to access the DRAM attached to the local IMC at the maximum bandwidth of the DDR3 paths connecting the IMC to the memory DIMMs. The three DDR3 channels to local DRAM support a bandwidth of $31.992\text{GB}/s = 3 \times 8 \times 1.333\text{G transfers}/s$. The available bandwidth to access memory blocks on the other socket should be bounded by the QPI inter-core links which can achieve

When remote memory locations are access data is transfered over the QPI link connecting the Nehalem chips. The available bandwidth through the QPI link is **12.8** GB/s (approximately **%40** of the theoretical bandwidth to the local DRAM) is the absolute upper bound to access remote DRAM.

We divide the experimental results below into bandwidth and latency sections, for single reader, single writer, combined reader and writer, multiple reader and writer threads. By varying the amount of total memory accessed we

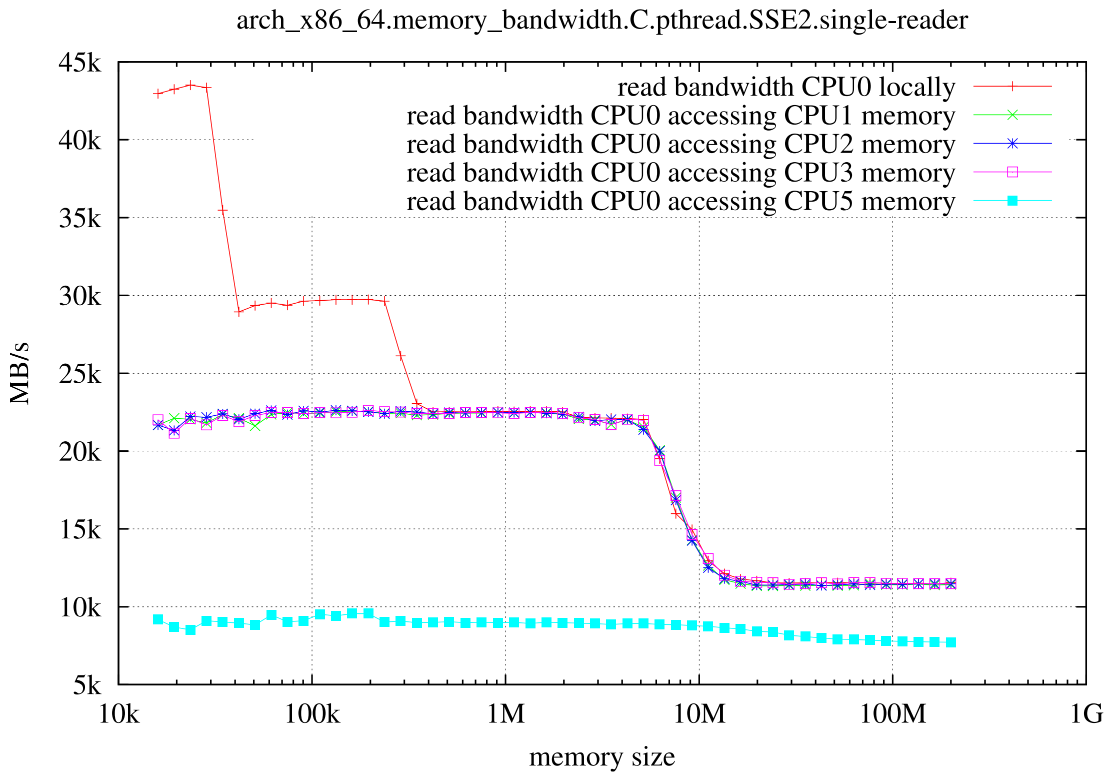


Figure 16: Bandwidth of a Single Reader Thread

are able to demonstrate the bandwidth and latencies to access memory at all 3 levels of cache and of the main memory. Finally, we present the different cost to access memory blocks owned by the local *vs.*remote memory controller. This last results demonstrates the intensity the NUMA effect has on the bandwidth and latency of applications accessing Nehalem system memory. In all experiments CPU0, . . ., CPU3, and CPU4, . . ., CPU7, belong to chips 0 and 1, respectively. All experiments were carried out on our Nehalem-EP cluster called EOS [12] and they utilized only 4KiB size pages.

For all experiments we utilize the "BenchIT" [13, 14] open source package[1] which was built on our target Nehalem-EP target system. We used a slightly modified version of a collection of benchmarks called "`X86membench`" [15] which are also available at the BenchIT site.

7.2 Memory Bandwidth Results

7.2.1 Single Reader Thread

In this experiment, a single thread reads memory segments of size varying from 10KiBs to 200MBs. All memory blocks are in the Exlusive state. The top curve in Fig. 16 (labeled "read bandwidth CPU0 locally") plots the observed bandwidth when a thread running on CPU0, accesses data which have as home location DRAM managed by the IMC on the same socket as CPU0. No other thread is reading any of these data blocks. This data traverse only the memory path which connects the CPU0 core to the common L3, and then the private L2 and L1 data cache

[1]Visit the official web site of the BenchIt project at `http://www.benchit.org`.

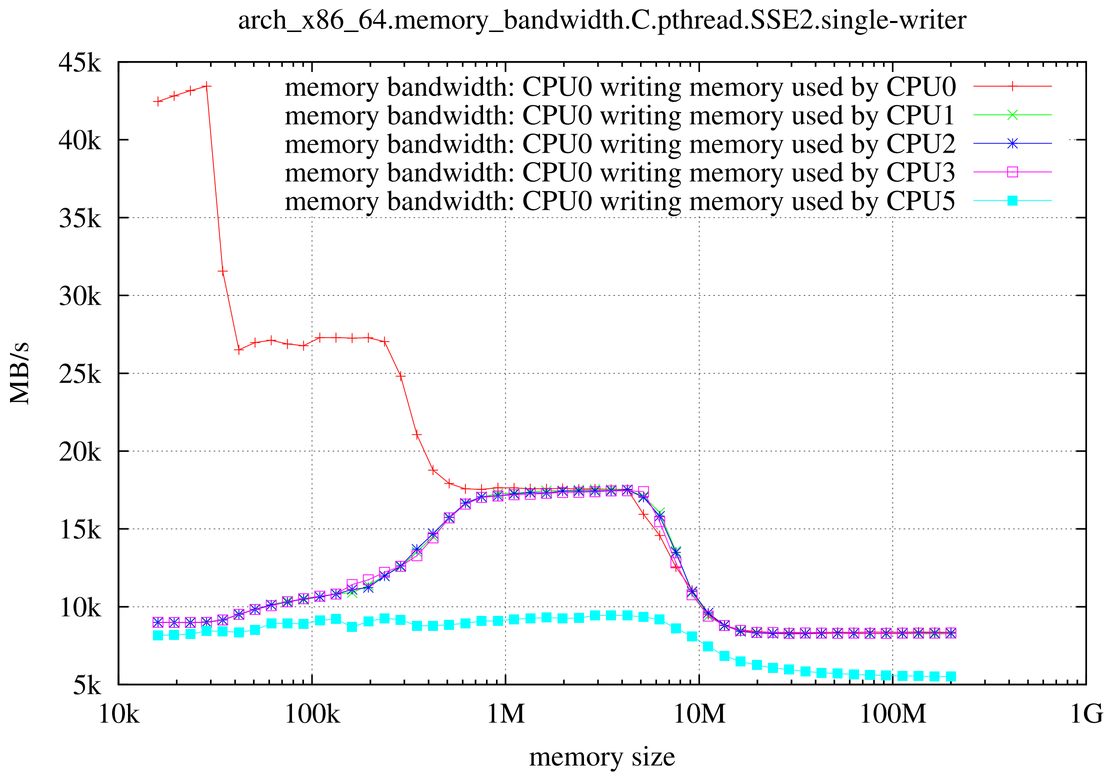


Figure 17: Bandwidth of Single Writer Thread

memories of CPU0. We can see that the observed bandwidth in accessing data areas which fit entirely within the L1 data cache (32 KiB) is very close to the ideal L1 bandwidth of 44.8GB/sec = 2.8GHz $\times$ 16bytes.

Next, the memory ready bandwidth of segments of size up to the L2 cache (**256** KiBs) stays a little below **30** GB/sec. As soon as the data cannot entirely fit in the L2, data is retrieved from the inclusive **8** MiB L3 cache. While data sizes stay below 8MiB, performance stays at around **22.5** GB/sec.

Note that the next three from the top curves namely those labeled "CPU0 accessing CPU1", and so on, up to "CPU0 accessing CPU3", represent threads running respectively on CPU and accessing data already read in by CPU1, CPU2 and CPU3 which are on the same quad-core socket as CPU0. Read performance is essentially that of L3 as data are served off the inclusive L3 cache to the rest of the cores of the same socket.

Soon after data sizes exceed the **8** MiBs of L3 capacity, all data is fetched from DRAM. Since our system has two sockets, each owning a different set of DRAM chips, we have two distinct performance cases. The first case is when all data is brought in from the DRAM which is owned by the same socket as the cores on which the threads execute and this achieves a bandwidth of approximately **11.5** GiB/sec.

The second case is when the thread running on CPU0 accesses data which are homed on the other Nehalem socket. The achieved bandwidth is around **8.8** GiB/sec when the data size is less than **8** MiBs and around **7.8** GiB/sec otherwise.

7.2.2 Single Writer Thread

In this experiment, a single thread writes to memory segments of size varying from 10KBs to 200MBs. When a core has to write to a data block, the MESI(F) protocol requires a "Read-for-Ownership" operation which snoops and

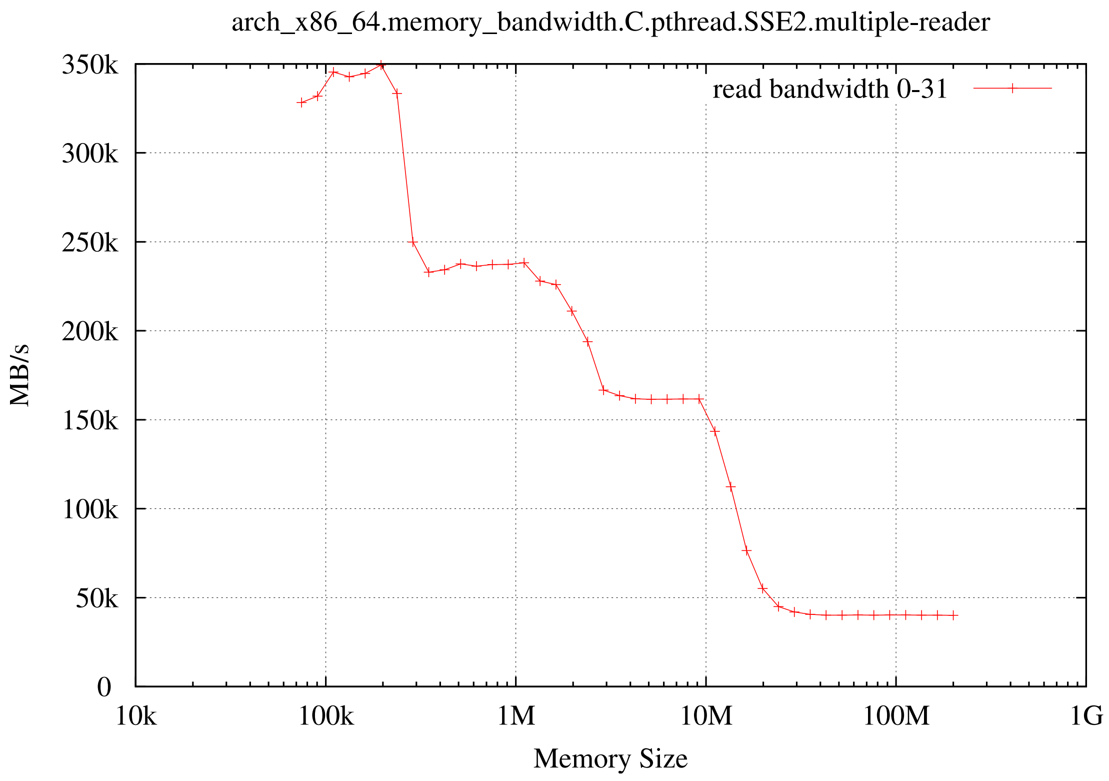


Figure 18: Aggregate Bandwidth of Multiple Reader Threads

invalidates this block already stored on other caches. All data blocks are in the Modified state when the writes are performed against them.

The top curve in Fig. 17 (labeled "CPU0 writing memory used by CPU0") demonstrates the case a thread exclusively writes to data only owned by itself. The next three curves plot the write bandwidth of a thread running on CPU0 writing to data blocks which are already cached by the other three cores on the same chip and are in the Modified state. Updating data in the L1 of another core takes longer than writing to the L2 of the same core.

As soon as a core starts writing data blocks which are larger than L3, data spill over to the local DRAM at a rate of **8.2** GiB/sec, approximately.

Finally, the bottom curve represents the bandwidth of writing to memory which is owned by the IMC on the other socket. It can be seen that as soon as the L3 capacity of the socket is exceeded, the write bandwidth to remote DRAM drops to **5.5** GiB/sec, approximately.

7.2.3 Multiple Reader Threads

In this experiment, 8 threads are split evenly across all cores and read simultaneously from disjoint locations memory segments of sizes up to 200MBs. Cached memory blocks are in the Exclusive state. As we can see, while the L1 data cache can satisfy the requests, the aggregate read bandwidth is very close to the ideal one which is equal to 358GB/sec $= 2.8\text{GHz} \times 8 \times 16$bytes.

We can then see the clear pattern of the bandwidths while memory reads can be satisfied from L1, L2 and L3 caches. Finally, when all requests go directly to DRAM, read bandwidth settles to approximately **40** GB/sec, or correspondingly **20** GB/sec per socket.

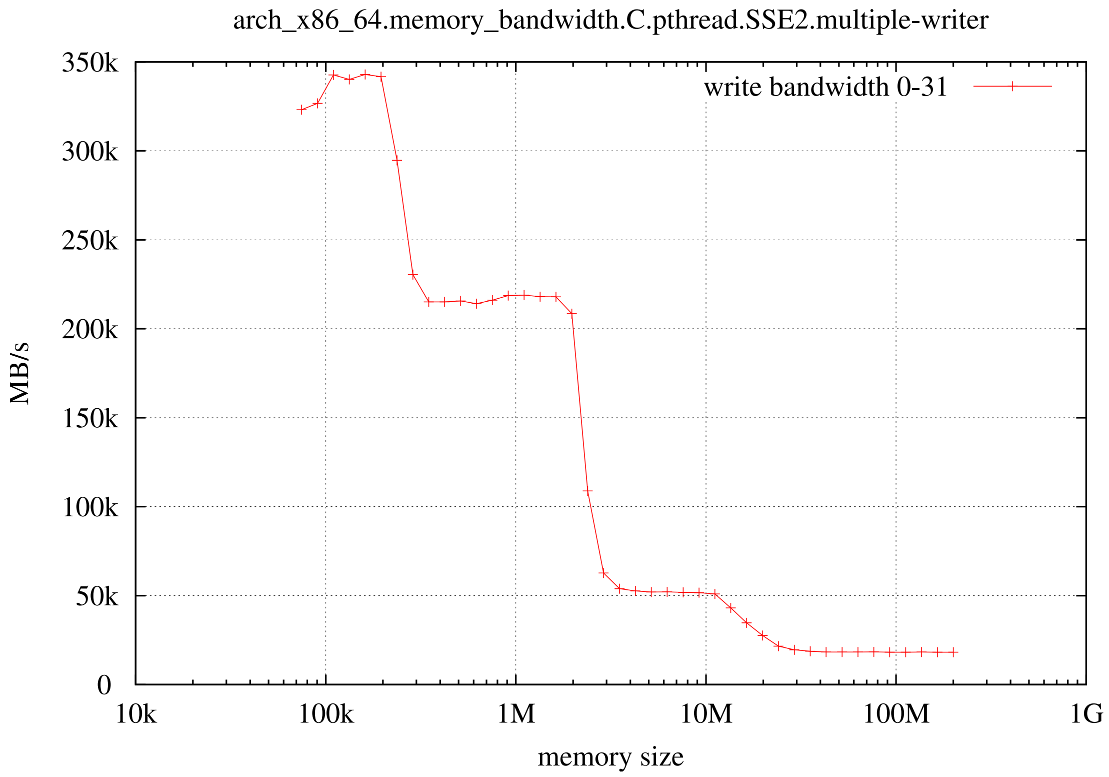


Figure 19: Aggregate Bandwidth of Multiple Writer Threads

7.2.4 Multiple Writer Threads

In this experiment, 8 threads are split evenly across all cores and write simultaneously to disjoint locations memory segments of sizes up to 200MBs. As we can see, while the L1 data cache can satisfy the requests, the aggregate write bandwidth is also very close to the ideal one which is equal to 358GB/sec = 2.8GHz × 8 × 16bytes.

We can then again see as with the multiple reader case above, the clear pattern of the attainable bandwidths while memory writes can stay within L2 and L3 caches. Finally, when all requests go directly to DRAM, aggregate write bandwidth settles to approximately **20** GB/sec.

7.2.5 Single Pair of Combined Reader and Writer Threads

In this experiment, a single pair of threads reads and writes memory segments of size varying from 10KBs to 200MBs. The reader thread retrieves data belonging to the remote DRAM, whereas the writer thread writes into data belonging to the local DRAM. The top curve in Fig. 20 (labeled "bandwidth CPU0–CPU0") plots the observed bandwidth when the pair of threads is running on CPU0 and they access data not accessed by other cores. When the L3 capacity is overran the limitation becomes the QPI link one way bandwidth.

7.2.6 Multiple Pairs of Combined Reader and Writer Threads

This experiment is a combined 8-way experiment of the "single reader and writer pair" of threads above.

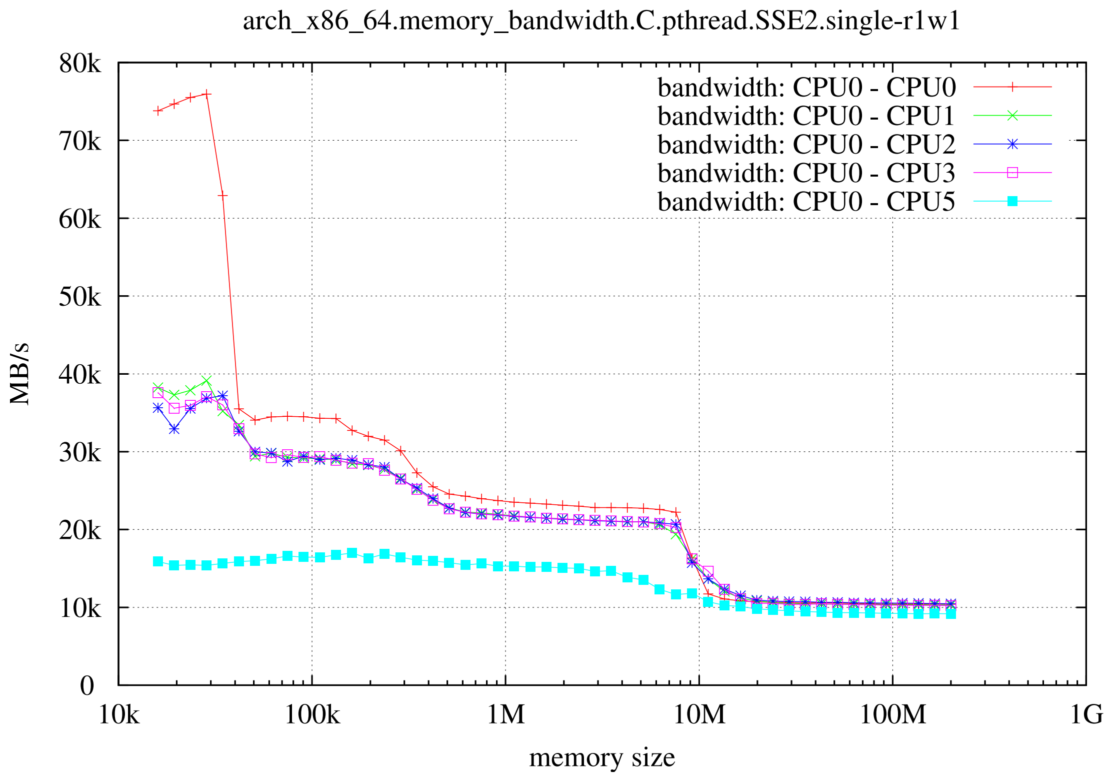


Figure 20: Bandwidth of a Single Pair of Reader and Writer Threads

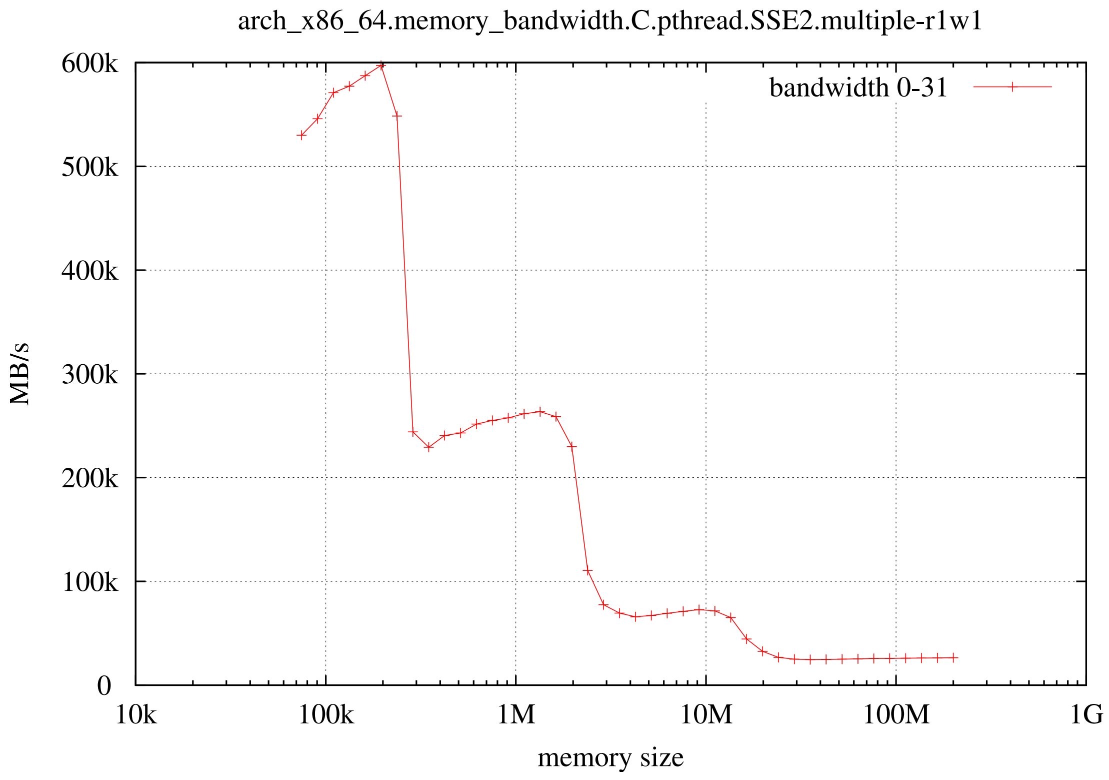


Figure 21: Aggregate Bandwidth of 8 Pairs of Reader and Writer Threads

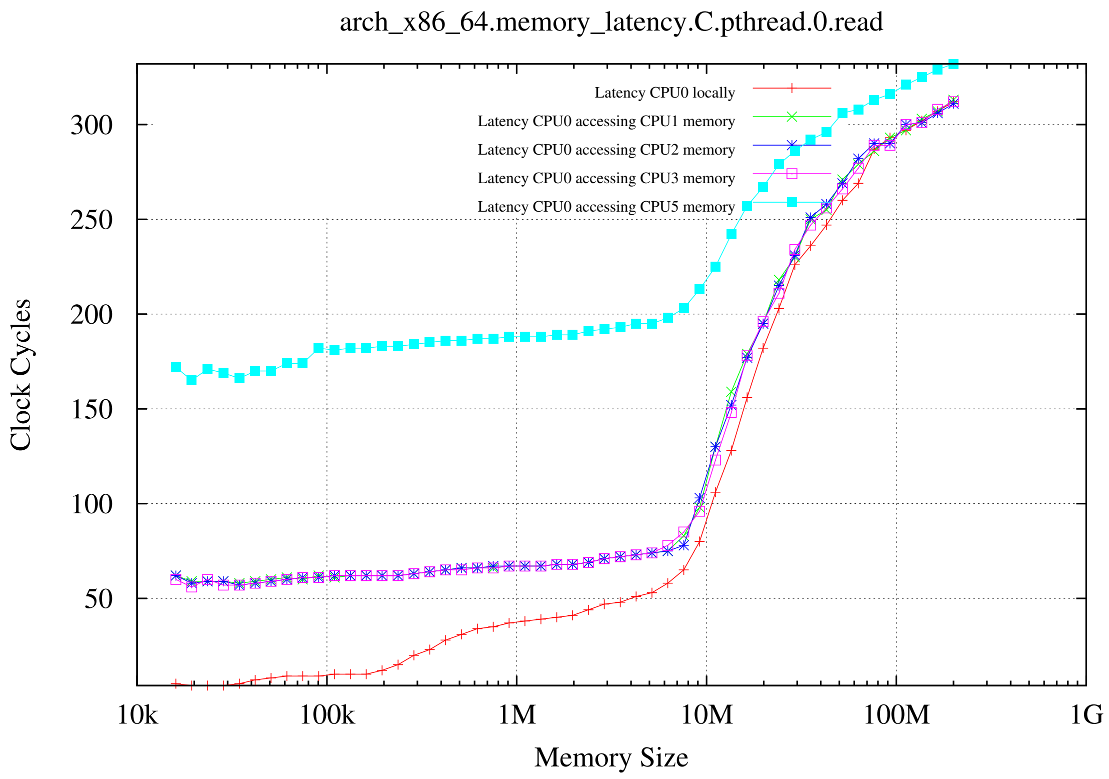


Figure 22: Latency to Read a Data Block in Processor Cycles

7.3 Memory Latency

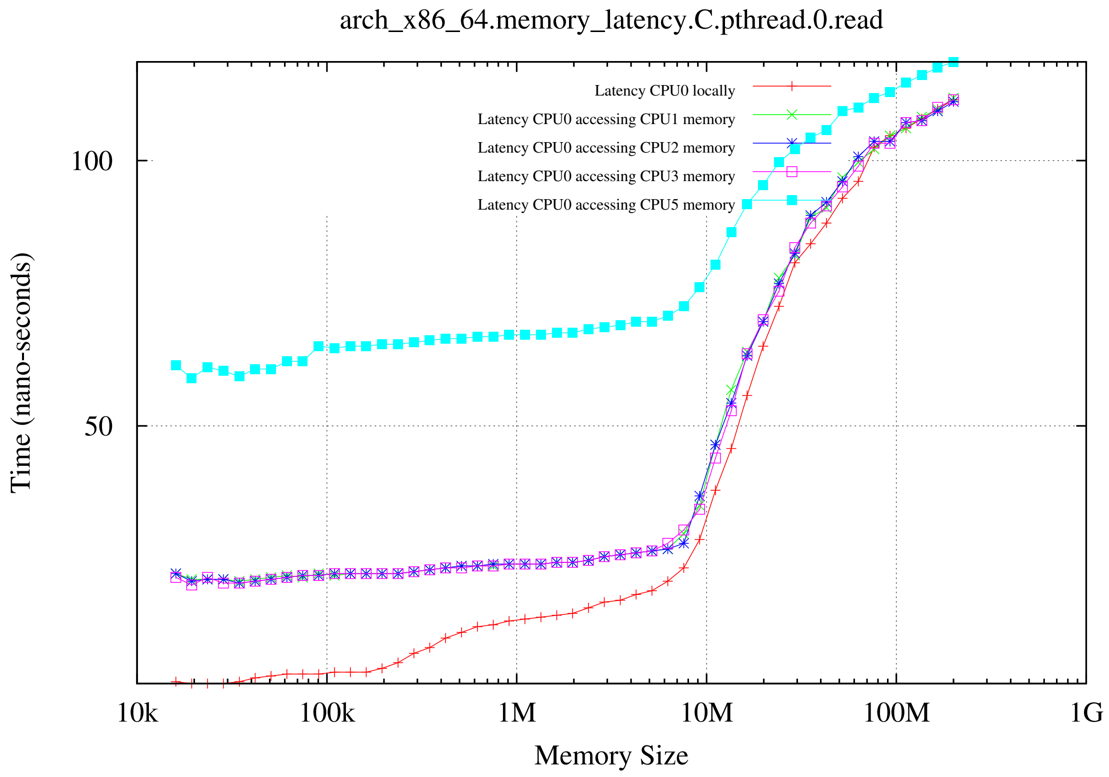


Figure 23: Latency to Read a Data Block in Seconds

The memory latency experiments measure the absolute amount of time in clock cycles and seconds it takes a core to retrieve a new cache block directly from the L1, L2, L3 caches and finally from the DRAM. The target memory blocks are initialized to be in the Exclusive state before access. We have only used 4KiB size pages for these experiments.

Referring to Fig. 22 and Fig. 23, the bottom "red" curve plots the access time it takes CPU 0 access data residing on the local DRAM block. The cost to retrieve memory clearly reflects the location of the specific block, namely the cache level (L1, L2, L3) or the local DRAM.

The three curves labeled as "CPU0 accessing CPUM memory", for $M = 1, 2, 3$, reflect the time it takes to retrieve a memory block which is already stored in Exclusive state in another core within the sane Nehalem chip. We can readily notice that the cost is identical regardless of which particular core owns this block.

Finally, the top curve demonstrates the case where the block to be retrieved resides on the DRAM of the other Nehalem chip. The latency increases considerably as the UnCore has to send cache coherence and data transfer messages over the QPI link. The distance of the bottom (CPU0 to local DRAM) and top (CPU0 to remote DRAM) curves trace the "NUMA effect". A thread accessing data belonging to the remote DRAM incurs much higher memory access cost.

One may readily notice that as cores attempt to retrieve at once data which far exceeding the capacity of the cache memories, the remote and local DRAM access costs start getting close to each other. We believe that this is due to the fact that the data TLB level 0 and 1 for 4KiB size pages are becoming overwhelmed by the excessive number of VM addresses requested in sequence. Visit Section 6 for a discussion on the memory address translation process in Intel64 architectures. There are only **64** and **512** entries, respectively, on the level 0 and 1 DTLBs. 64 DTLB entries cover up to **256** KiBs of memory and the 512 DTLB level 1 cover **2** MiBs. As a core accesses memory sizes which exceed these limits, the address translation hardware becomes the bottleneck as the page tables in DRAM have to be accessed to locate complete page address information.

Wes strongly recommend that code which accesses lengthy blocks of data use large pages (known as “Huge Pages” in the Linux world) as there is significant difference in the latencies for memory access.

8 Intel Turbo Boost Technology

Nehalem supports a number of power saving features which switch off the power to idle cores or other parts of the system[2]. An interesting related feature is called “Turbo Boost Technology” (TBT) which not only dynamically turns off unused processor cores but it can also increase the clock speed of the cores in use as long as the power and head profiles of the chip is maintained. TBT will increase the frequency in steps of 133 MHz (to a maximum of three steps or 400 MHz) as long as the processors’ predetermined thermal and electrical requirements are still met. For example, with three cores active, a 2.26 GHz processor can run the cores at 2.4 GHz. With only one or two cores active, the same processor can run those cores at 2.53 GHz. Similarly, a 2.93 GHz processor can run at 3.06 GHz or even 3.33 GHz. When the cores are needed again, they are dynamically turned back on and the processor frequency is adjusted accordingly. This feature can be enabled or disabled in the UEFI BIOS of each node.

References

[1] *Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 1:Basic Architecture*, Intel Corporation, Jun. 2010.

[2] *Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide, Part 1*, Intel Corporation, Jun. 2010.

[3] *Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3B: System Programming Guide, Part 2*, Intel Corporation, Jun. 2010.

[4] S. Gunther and R. Singhal, “Next generation intel microarchitecture (nehalem) family: Architectural insights and power management,” in *Intel Developer Forum*, San Francisco, Mar. 2008.

[5] P. P. Gelsinger, “Intel architecture press briefing,” in *Intel Developer Forum*, San Francisco, Mar. 2008.

[6] R. Singhal, “Inside intel next generation nehalem microarchitecture,” in *Intel Developer Forum*, San Francisco, Mar. 2008.

[7] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, Nov. 2009.

[8] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th ed. Morgan-Kaufmann Publishers Inc., 2009, iSBN: 978-0-12-374493-7.
Publisher’s URL: http://www.elsevierdirect.com/product.jsp?isbn=9780123744937.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan-Kaufmann Publishers Inc., 2007, iSBN: 978-0-12-370490-0.
Publisher’s URL: http://www.elsevierdirect.com/product.jsp?isbn=9780123704900.

[10] D. Levinthal, “Performance analysis guide for intel® core$^{(TM)}$ i7 processor and intel xeon$^{(TM)}$ 5500 processors,” Intel Corporation, Tech. Rep. Version 1.0, 2009.

[11] Intel Technical Staff, “TLBs, paging-structure caches, and their invalidation,” Intel Corporation, Application Note, 2008.

[12] M. E. Thomadakis, “A High-Performance Nehalem iDataPlex Cluster and DDN S2A9990 Storage for Texas A&M University,” Texas A&M University, College Station, TX, Technical Report, 2011. [Online]. Available: http://sc.tamu.edu/systems/eos/

[2]This discussion is beyond hte scope of the present article and only short description is given.

[13] G. Juckeland, S.Börner, M. Kluge, S. Kölling, W. Nagel, S. Pflüger, H. Röding, S. Seidl, T. William, and R. Wloch, "BenchIT – performance measurement and comparison for scientific applications," in *Parallel Computing - Software Technology, Algorithms, Architectures and Applications*, ser. Advances in Parallel Computing, F. P. G.R. Joubert, W.E. Nagel and W. Walter, Eds. North-Holland, 2004, vol. 13, pp. 501–508. [Online]. Available: http://www.sciencedirect.com/science/article/B8G4S-4PGPT1D-28/2/ceadfb7c3312956a6b713e0536929408

[14] G. Juckeland, M. Kluge, W. E. Nagel, and S. Pflüger, "Performance analysis with benchit: Portable, flexible, easy to use," in *Proceedings of the International Conference on Quantitative Evaluation of Systems*, 2004, pp. 320–321.

[15] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore smp systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 413–422. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669165