Performance Evaluation of Hyper Threading Technology Architecture Using Microsoft Operating System Platform

[1] Okonta O.E., [2] Ajani D, [3] Owolabi A.A, [4] Imiere E.E., [5] Uzomah L.
[1,2,3,5] Department Of Computer Science Federal College Of Education (T), Asaba,
[4] Department Of Physics Federal College Of Education (T), Asaba
Okeyokonta@Yahoo.Com, Ajanidele@Gmail.Com, Abudulhakimowolabi@Gmail.Com,
Emmaedekinimiere@Gmail.Com, , Lawrenceuzomah@Rocketmail.Com

Abstract

This paper describes the Hyper-Threading Technology architecture, and discusses the micro architecture details of Intel's structure. Hyper-Threading Technology is an important addition to Intel's enterprise product line and has been integrated into a wide variety of products. Intel provides Hyper-Threading (HT) in processors based on its Pentium and more recent processor die. HT enables two threads to execute on each core in order to hide latencies related to data access. These two threads can execute simultaneously, filling unused stages in the functional unit pipelines. To aid better understanding of HT related issues, we look at Performance Monitoring Unit (PMU) data (instructions retired; un-halted core cycles; L2 and L3 cache hits and misses; vector and scalar floating-point operations, etc.). We then use the PM data to make deduction on a new metric of efficiency in order to quantify processor resource utilization and make comparisons of that utilization between single-threading (ST) and HT modes. We also study performance gain using unhalted core cycles, code efficiency of using vector units of the processor, and the impact of HT mode on various shared resources like L2 and L3 cache. Results using four full-scale, production-quality scientific applications from computational fluid dynamics indicate that HT generally improves processor resource utilization efficiency, but does not necessarily translate into overall application performance gain.

Key words: *Hyper-Threading, core circles, code efficiency, processor resource utitilization, applications performance.*

Introduction

Intel's Hyper-Threading Technology brings the concept of simultaneous multi-threading to the Intel Architecture. Hyper-Threading Technology makes a single physical processor appear as two logical processors; the physical execution resources are shared and the architecture state is duplicated for the two logical processors. From a software or architecture perspective, this means operating systems and user programs can schedule processes or threads to logical processors as they would on multiple physical processors. From a micro-architecture perspective, this means that instructions from both logical processors will persist and execute simultaneously on shared execution resources.

Hyper-threading is Intel's trademarked term for its simultaneous multithreading implementation in their Pentium 4, Atom, Core i7, and certain Xeon CPUs and more recent processors. Hyper-threading (officially termed Hyper-Threading Technology or HTT) is an Intel-proprietary technology used to improve parallelization of computations (doing multiple tasks at once) performed on PC microprocessors [10],[11]. A processor with hyper-threading enabled is treated by the operating system as two processors instead of one. This means that only one processor is physically present but the operating system

sees two virtual processors, and shares the workload between them. Hyper-threading requires both operating system and CPU support for efficient usage; conventional multiprocessor support is not enough, and may actually decrease performance if the Operating System is not sufficiently aware of the distinction between a physical core and a HTT-enabled core. For example, Intel does not recommend that hyper-threading be enabled under Windows 2000, even though the operating system supports multiple CPUs (but is not HTT-compliant).

So in simple term, Hyper-threading is using one physical processor but logically dividing it into two so that it gives the user the benefit of two processors with only using the resources equivalent to almost one. This is achieved by sharing, partitioning and duplicating the various resources almost into two processors. Used by the latest Pentium processors, which are Hyper Thread enabled, in layman's terms, it allows you to use more than two applications at the same time without slowing down processing speed. Hyper-threading was intended to solve an issue with a waste of potential resources in the CPU. It does not change the physical resources (the CPU cores) but more resources can be potentially tapped into by allowing two threads to be processed by the same execution resource simultaneously -with each physical core being an execution resource. When hyper-threading is enabled it doubles the number of logical processors presented by the BIOS. For example, a 2 socket, 8 core system with 16 cores total now presents 32 cores to the virtual hardware. Therefore the CPU scheduler can make efficient use of hyper-threading and generally it should be enabled. The number of logical processors now doubles, providing a performance benefit in the range of 10-15% in most virtual sphere environments (depending on workload/applications, processors speed and memory capacity) [9].

Parallel Computing

Multithreading computers have hardware support to efficiently execute multiple threads. These are distinguished from multiprocessing systems (such as multi-core systems) in that the threads have to share the resources of single core: the computing units, the CPU caches and the translation lookaside buffer (TLB). While multiprocessing systems include multiple complete processing units, multithreading aims to increase utilization of a single core by leveraging thread-level as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and in CPUs with multiple multithreading cores.[2] Multi-threading is when various processes are time sliced such that it gives the user the impression that all the programs are being run at the same time. This is what happens on your computer regularly

Multithreading refers to the general task of running more than one thread of execution within an operating system. Multithreading is more generically called "multiprocessing", which can include multiple system processes (a simple example on Windows would be, e.g., running Internet Explorer and Microsoft Word at the same time), or it can consist of one process that has multiple threads within it. Multithreading is a software concept.[1],[6],[7] Practically any Turing-complete CPU can perform multithreading, even if the computer only has one CPU core and that core does not support hyper-threading. In order to support multiprocessing, the CPU will interleave execution of different threads of execution, by executing one, then another, then another, where the operating system will divide up the time available into "slices" and give a roughly equal amount of time to each thread (the time doesn't have to be equal, but that's typically how it's done unless a process requests a higher priority).

Super-threading

Super-threading allows threads from different processes to be executed at the same time unlike Multi-threading where every process has a time slot during which, thread from only one process will be executed. But every time, if for example, there are four instructions issued to the processor. They will all be from the same process. Hyper-threading takes it a step further. It allows threads from different processes to be issued at the same

time, in turn, utilizing the waste cycles of the processor. Super-threading is a multithreading approach that weaves together the execution of different threads on a single processor without truly executing them at the same time. This qualifies it as time-sliced or temporal multithreading rather than simultaneous multithreading. It is motivated by the observation that the processor is occasionally left idle while executing an instruction from one thread. Super-threading seeks to make use of unused processor cycles by applying them to the execution of an instruction from another thread

Understanding Hyper-Threading Technology

Current trends in microprocessor design have made high resource utilization a key requirement for achieving good performance. This means that while deeper pipelines have led to 3 GHz processors, each new generation of micro-architecture technology comes with increased memory latency and a decrease in relative memory speed. This results in the processor spending a significant amount of time waiting for the memory system to fetch data. This "memory wall" problem continues to remain a major shortcoming and as a result, sustained performance of most real-world applications is less than 10% of peak. Over the years, a number of multithreading techniques have been employed to hide this memory latency. One approach is simultaneous multi-threading (SMT), which exposes more parallelism to the processor by fetching and retiring instructions from multiple instruction streams, thereby increasing processor utilization. Simultaneous multi-threading requires only some extra hardware instead of replicating the entire core.[6] Price and performance benefits make it a common design choice as, laid out in ***Intel's Nehalem*** micro-architecture, where it is called Hyper-Threading (HT). As is the case with other forms of on-chip parallelism, such as multiple cores and instruction-level parallelism,[8],[9] Simultaneous multi-threading uses resource sharing to make the parallel implementation economical. With Simultaneous multi-threading, this sharing has the potential for improving utilization of resources such as that of the floating-point unit through the hiding of latency in the memory hierarchy. When one thread is waiting for a load instruction to complete, the core can execute instructions from another thread without stalling.

Review of and related work

Intel introduced Simultaneous multi-threading (SMT), which it called Hyper-Threading (HT), into its product line in 2002 with new models of their Pentium 4 processors curtailing the use Co-Maths processor **[10],[11],[4].** The advantage of HT is its ability to better utilize processor resources and to hide memory latency. There have been a few efforts studying the effectiveness of Hyper-Threading on application performance**.** Boisseau et al. conducted a performance evaluation of Hyper-Threading on a Dell 2650 dual processor-server based on Pentium 4 using matrix multiplication and a 256-particle molecular dynamics benchmark written in OpenMP **[13].** Haung et al. characterized the performance of Java applications using Pentium 4 processors with Hyper-Threading **[25].** Blackburn et al. studied the performance of garbage collection in Hyper-Threading mode by using some of the Pentium 4 performance counters [21]. A key finding of these investigations was that the Pentium 4's implementation of Hyper-Threading was not very advantageous, as the processor had very limited memory bandwidth and issued only two instructions per cycle.

After that, Hyper-Threading was extended to processors that use ***Intel's Nehalem*** micro-architecture [9]. In these processors, memory bandwidth was enhanced significantly by overcoming the front-side bus memory bandwidth problem and by increasing instruction issuance from two to four per cycle. Saini et al. conducted a performance evaluation of Hyper-Threading on small numbers of Nehalem nodes using NPB [23]. Results showed that for one node, Hyper-Threading provided a slight advantage only for LU**.** BT, SP, MG, and LU achieved the greatest benefit from Hyper-Threading at 4 nodes: factors of 1.54, 1.43, 1.14, and 1.14, respectively, while FT did not achieve any

benefit independent of the number of nodes. Later on Saini et al. extended their work on Hyper-Threading to measure the relative efficiency *E* of the processor in terms of cycle per instruction using the formula

E = 100*(2*CPIST / CPIHT) – 100 (1)

where CPIST and CPIHT are cycle per instruction in ST and HT modes respectively [23].

In the study focus was on the ***Westmere-EP Xeon processor***, which is based on the Nehalem micro-architecture. The contributions of the paper are as follows: • We present *efficiency*, a new performance metric in terms of instruction per cycle to quantify the utilization of the processor, by collecting PMU data in both ST and HT modes using a range of core counts. It analyse the PMU data to identify the factors that influence the performance of the codes, in particular focusing on the impact of shared resources, such as execution units and memory hierarchy, when executing in Hyper-Threading mode.[12]

Hyper-threading in Nehalem Micro-architecture

Hyper-Threading (HT) allows instructions from multiple threads to run on the same core. When one thread stalls, a second thread is allowed to proceed. To support Hyper-Threading, the Nehalem micro-architecture has several advantages over the Pentium 4. Firstly, the newer design has much more memory bandwidth and larger caches, giving it the ability to get data to the core faster. Secondly, Nehalem is a much wider architecture than Pentium 4. It supports two threads per core, presenting the abstraction of two independent logical cores. The physical core contains a mixture of resources, some of which are shared between threads [11]: • *replicated resources* for each thread, such as register state, return stack buffer (RSB), and the instruction queue; • *partitioned resources* tagged by the thread number, such as load buffer, store buffer, and reorder buffer; • *shared resources*, such as L1, L2, and L3 cache; and • *shared resources unaware of the presence of threads*, such as execution units.

The RSB (return stack buffer) is an improved branch target prediction mechanism. Each thread has a dedicated RSB to avoid any cross-contamination. Such replicated resources should not have an impact on Hyper-Threading performance. Partitioned resources are statically allocated between the threads and reduce the resources available to each thread. However there is no competition for these resources. On the other hand, the two threads do compete for shared resources and the performance depends on the dynamic behaviour of the threads. Some of the shared resources are unaware of Hyper-Threading. For example, the scheduling of instructions to execution units is independent of threads, but there are limits on the number of instructions from each thread that can be queued. **Figure 1** is a schematic description of Hyper-Threading for the Nehalem micro-architecture. In the diagram, the rows depict each of the Westmere-EP processor's six execution units— two floating-point units (FP0 and FP1), one load unit (LD0), one store unit (ST0), one load address unit (LA0), and one branch unit (BR0). It is a sixteen-stage pipeline. Each box represents a single micro-operation running on an execution unit.

Figure 1(a) shows the ST mode (no HT) in a core where the core is executing only one thread (Thread 0 shown in green) and white space denotes unfilled stages in the pipeline. The peak execution bandwidth of the Nehalem micro-architecture is four micro-operations per cycle. Often ST does not utilize the execution units optimally and operates at less than peak bandwidth, as indicated by the large number of idle (white) execution units.

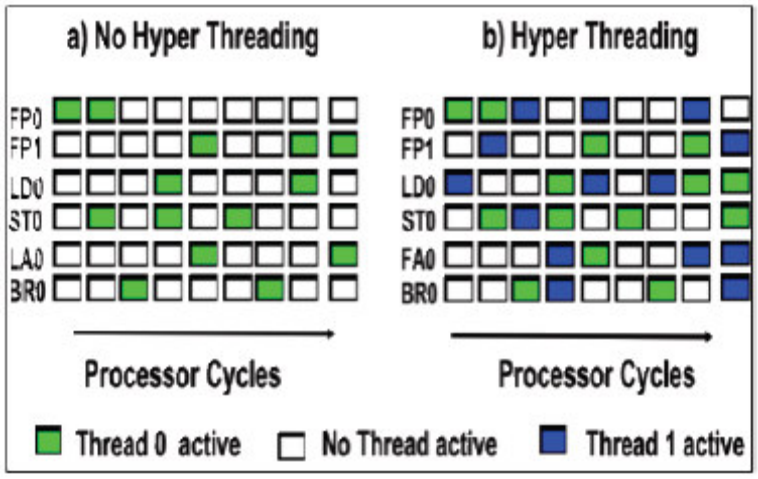

Figure 1. Hyper threading on the sixteen-stage pipeline Nehalem architecture with six execution units.

Figure 1(b) shows the HT feature in one of the processor cores. This core in HT mode executes the micro-operations, from both threads (Thread 0 and Thread 1 shown in green and blue, respectively). This arrangement can operate closer to peak bandwidth, as indicated by the smaller number of idle (white) execution units. In HT mode, the processor can utilize execution units more efficiently.

Computing Platform

The study on Computing Platform was conducted using NASA's Pleiades supercomputer, and SGI Altix ICE 8400EX system located at NASA Ames Research Centre. Pleiades comprises of 10,752 nodes interconnected with an InfiniBand (IB) network in a hypercube topology. The nodes are based on three different Intel Xeon processors: Harpertown, Nehalem-EP, and Westmere-EP. In the study, they used the Westmere-EP based nodes [12]. This subset of Pleiades is interconnected via 4X Quad Data Rate (QDR) IB switches. As shown in Figure 2, the Westmere-EP based nodes have two Xeon X5670 processors, each with six cores. Each processor is clocked at 2.93 GHz, with a peak performance of 70.32 Gflop/s. The total peak performance of the node is therefore 140.64Gflop/s. Each Westmere-EP processor has two parts: "core" and "un-core". The core part consists of six cores with per-core L1 and L2 caches. The un-core part has a shared L3 cache, an integrated memory controller, and Quick Path Interconnect (QPI). Each core has 64 KB of L1 cache (32 KB data and 32 KB instruction) and 256 KB of L2 cache. All six cores share 12 MB of L3 cache. The on-chip memory controller supports three DDR3 channels running at 1333 MHz, with a peak memory bandwidth per socket of 32 GB/s (and twice that per node). Each processor has two QPI links: one connects the two processors of a node to form a non-uniform-memory access (NUMA) architecture, while the other connects to the I/O hub. Each QPI link runs at 6.4 GT/s ("T" for transactions), at which rate 2 bytes can be transferred in each direction, for an aggregate of 25.6 GB/s. HT was enabled on each processor for our experiments. Pleiades utilizes SUSE Linux Enterprise Server (SLES) based on the 2.6.32 Linux kernel and SGI overlays as its operating system and has a Lustre file system for I/O.[12]

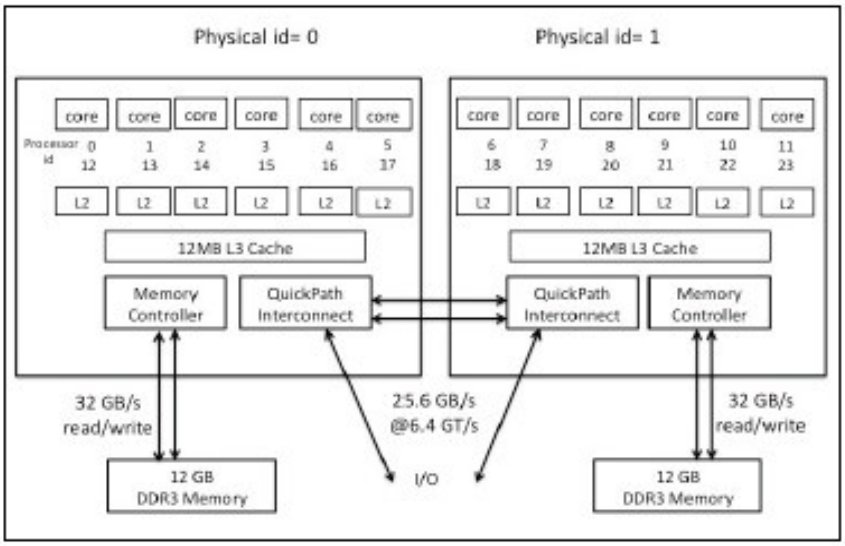

Figure 2. Configuration of an Intel Westmere-EP node.

Experimental Setup and Performance Analysis

Software optimization based on performance analysis of large existing applications, in most cases, reduces to optimizing the code generation by the compiler and optimizing the memory access. This paper will focus on this approach. Optimizing the code generation by the compiler requires inspection of the assembler of the time consuming parts of the application and verifying that the compiler generated a reasonable code stream. Optimizing the memory access is a complex issue involving the bandwidth and latency capabilities of the platform, hardware and software prefetching efficiencies and the virtual address layout of the heavily accessed variables. The memory access is where the non-uniform memory access nature of the Intel® Core™ i7 processor based platforms becomes an issue.

Performance analysis illuminates how the existing invocation of an algorithm executes. It allows a software developer to improve the performance of that invocation. It does not offer much insight about how to change an algorithm, as that really requires a better understanding of the problem being solved rather than the performance of the existing solution. That being said, the performance gains that can be achieved on a large existing code base can regularly exceed a factor of 2, (particularly in HPC) which is certainly worth the comparatively small effort required.

Basic Intel® Core™ i7 Processor and Intel® Xeon™ 5500

Processor Architecture and Performance Analysis

Performance analysis on a micro architecture is the experimental investigation of the micro architecture's response to a given instruction and data stream. As such, a reasonable understanding of the micro architecture is required to understand what is actually being measured with the performance events that are available. Here we cover the basics of the Intel® Core™ i7 processor and Intel® Xeon™ 5500 processor architecture. It is not meant to be complete but merely the briefest of introductions. For more details please consult the Software Developers Programming Optimization Guide.[22]

The Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors are multi core, Intel® Hyper-Threading Technology (HT) enabled designs. Each socket has one to eight cores, which share a last level cache (L3 CACHE), a local integrated memory controller and an Intel® QuickPath interconnect. Thus a 2 socket platform with quad core sockets might be drawn as:

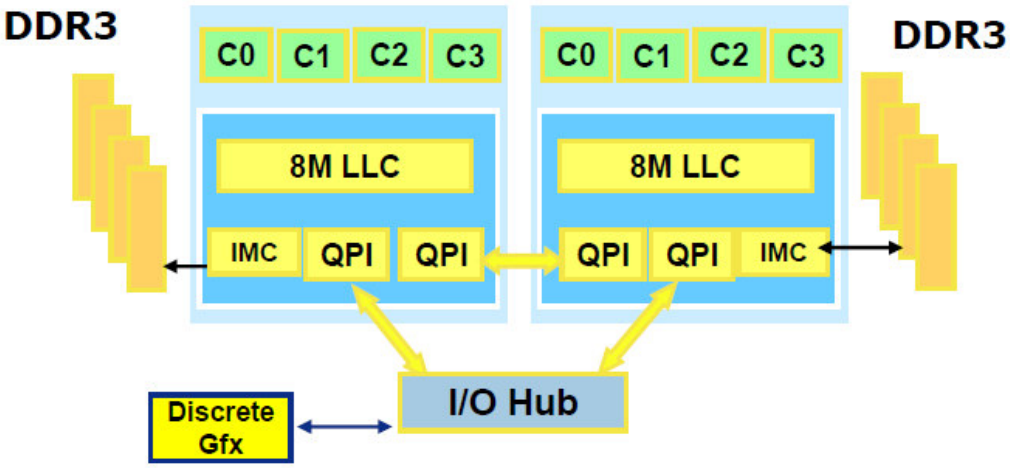

Figure 3 Memory Sub System

What follows here is a brief description of the experimental setup for collecting and analysing the data based on the hardware performance. Note that each core is quite similar to that of the Intel® Core™2 processor. The pipelines are rather similar except that the Intel® Core™ i7 core and pipeline supports Intel® Hyper- Threading Technology (HT), allowing the hardware to interleave instructions of two threads during execution to maximize utilization of the core's resources. The Intel® Hyper-Threading Technology (HT) can be enabled or disabled through a bios setting. Each core has a 32KB data and instruction cache, a 256 KB unified mid-level cache and 2 level DTLB system of 64 and 512 entries. There is a single, 32 entry large page DTLB. The cores in a socket share an inclusive last level cache.[11] The inclusive aspect of this cache is an important issue and in the usual DP configuration the shared, inclusive last level cache is 8MB and 16 wav associative. The cache coherency protocol messages, between the multiple sockets, are exchanged over the Intel® QuickPath Interconnects. The inclusive L3 CACHE allow this protocol to be extremely 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 an enormous increase in memory access bandwidth and results in a non-uniform memory access. 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. The development of a reasonably hierarchical structure and usage of the performance events will require a fairly detailed knowledge of exactly how the components of Intel® Core™ i7 processor execute an application's stream of instructions and delivers the required data. What follows is a minimal introduction to these components.[3],[5]

Core Out of Order Pipeline

The basic analysis methodology starts with an accounting of the cycle usage for execution. The out of order execution can be considered from the perspective of a simple block diagram as shown below:

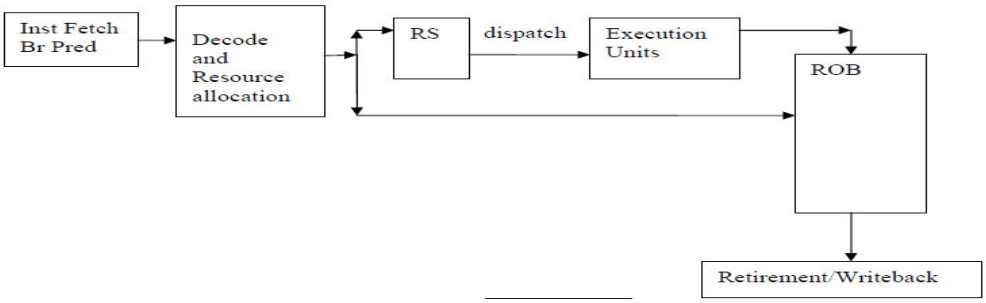

Figure 4 Block Diagram of Out of Order Execution

After instructions are decoded into the executable micro operations they are assigned their required resources. They can only be **issued** to the downstream stages when there are sufficient free resources. This would include (among other requirements):
1) space in the Reservation Station (RS), where the micro operations wait until their inputs are available
2) space in the Reorder Buffer, where the micro operations wait until they can be retired
3) sufficient load and store buffers in the case of memory related micro operations (loads and stores)

Retirement and write back of state to visible registers is only done for instructions and micro operations that are on the correct execution path. Instructions and micro operations of incorrectly predicted paths are flushed upon identification of the error in prediction and the correct paths are then processed. Retirement of the correct execution path instructions can proceed when two conditions are satisfied
1) The micro operations associated with the instruction to be retired have completed, allowing the retirement of the entire instruction, or in the case of instructions that generate very large number of micro operations, enough to fill the retirement window
2) Older instructions and their micro operations of correctly predicted paths have retired The mechanics of following these requirements ensures that the visible state is always consistent with in-order execution of the instructions. The core of this design is that if the oldest instruction is blocked, for instance waiting for the arrival of data from memory, younger independent instructions and micro operations, whose inputs are available, can be **dispatched** to the **execution** units and warehoused in the ROB upon completion. They will then retire when all the older work has completed.

The terms "**issued**", "**dispatched**", "**executed**" and "**retired**" have very precise meanings as to where in this sequence they occur and are used in the event names to help document what is being measured.

In the Intel® Core™ i7 Processor, the reservation station has 36 entries which are shared between the Hyper-threads when that mode is enabled in the bios, with some entries reserved for each thread to avoid locking. If not, all 36 could be available to the single running thread, making restarting a blocked thread inefficient. There are 128 positions in the reorder buffer, which are again divided if Hyper-threads are enabled or entirely available to the single thread if Hyper-threads is not enabled. As on Core™2 processors, the RS dispatches the micro operations to one of 6 dispatch ports where they are consumed by the execution units. This implies that on any cycle between 0 and 6 micro operations can be dispatched for execution.

The hardware branch prediction requests the bytes of instructions for the predicted code paths from the 32KB L1 instruction cache at a maximum bandwidth of 16 bytes/cycle. Instructions fetches are always 16 byte aligned, so if a hot code path starts on the 15th byte, the FE will only receive 1 byte on that cycle. This can aggravate instruction bandwidth issues. The instructions are referenced by virtual address and translated to physical address with the help of a 128 entry instruction translation lookaside buffer (ITLB). The x86 instructions are decoded into the processors micro operations by the pipeline front end. Four instructions can be decoded and issued per cycle. If the branch prediction hardware wrongly predicts the execution path, the micro operations from the incorrect path which are in the instruction pipeline are simply removed where they are, without stalling execution. This reduces the cost of branch wrong predictions. Thus the "cost" associated with such wrong predictions is only the wasted work associated with any of the incorrect path micro operations that actually got dispatched and executed and any cycles that are idle while the correct path instructions are located, decoded and inserted into the execution pipeline.

Core Memory Subsystem

In applications working with large data footprints, memory access operations can dominate the application's performance.

Consequently a great deal of effort goes into the design and instrumentation of the data delivery subsystem. Data is organized as a contiguous string of bytes and is transferred around the memory subsystem in cache lines of 64 bytes.

Generally, load operations copy contiguous subsets of the cache lines to registers, while store operations copy the contents of registers back into the local copies of the cache lines. SSE streaming stores are an exception as they create local copies of cache lines which are then used to overwrite the versions in memory, thus are slightly different. The local copies of the lines that are accessed in this way are kept in the 32KB L1 data cache. The access latency to this cache is 4 cycles. While the program references data through virtual addresses, the hardware identifies the cache lines by the physical addresses. The translation between these two mappings is maintained by the operating system in the form of translation tables. These tables list the translations of the standard 4KB aligned address ranges called pages. They also handle any large pages that the application might have allocated. When a translation is used it is kept in the data translation lookaside buffers (DTLBs) for future reuse, as all load and store operations require such a translation to access the data caches. Programs reference virtual addresses but access the cache lines in the caches through the physical addresses.

As mentioned earlier, there is a multi-level TLB system in each core for the 4KB pages. The level 1 caches have TLBs of 64 and 128 entries respectively for the data and instruction caches. There is a shared 512 entry second level TLB. There is a 32 entry DTLB for the large 2/4MB pages should the application allocate and access any large pages. There are 7 large page ITLB entries per HT. When a translation entry cannot be found in the DTLBs the hardware page walker (HPW) works with the OS translation data structures to retrieve the needed translation and updates the DTLBs. The hardware page walker begins its search in the cache for the table entry and then can continue searching in memory if the page containing the entry required is not found. Cache line coherency in a multi core multi socket system must be maintained to ensure that the correct values for the data variables can be retrieved. This has traditionally been done through the use of a 4 value state for each copy of each cache line. The four state (MESI) cache line protocol allows for a coherent use of data in a multi-core, multi-socket platform. A line that is only read can be shared and the cache line access protocol supports this by allowing multiple copies of the cache line to coexist in the multiple cores. Under these conditions, the multiple copies of the cache line would be in what is called a Shared state (S). A cache line can be put in an Exclusive state (E) in response to a "read for ownership" (RFO) in order to store a value. All instructions containing a lock prefix will result in a (RFO) since they always result in a write to the cache line. The F0 lock prefix will be present in the opcode or is implied by the exchange and complex exchange instructions when a memory access is one of the operands. The exclusive state ensures exclusive access of the line. Once one of the copies is modified the cache line's state is changed to Modified (M).

Then change of state is propagated to the other cores, whose copies are changed to the Invalid state (I). With the introduction of the Intel® QuickPath Interconnect protocol the 4 MESI states are supplemented with a fifth, Forward (F) state, for lines forwarded from on socket to another. When a cache line, required by a data access instruction, cannot be found in the L1 data cache it must be retrieved from a higher level and longer latency component of the memory access subsystem. Such a cache miss results in an invalid state being set for the cache line. This mechanism can be used to count cache misses. The L1D miss creates an entry in the 16 element super queue and allocates a line fill buffer. If the line is found in the 256KB mid-level cache (MLC, also referred to as L2), it is transferred to the L1 data cache and the data access instruction can be serviced. The load latency from the L2 CACHE is 10 cycles, resulting in a performance penalty of around 6 cycles, the difference of the effective L2 CACHE and L1D latencies. If the line is not found in the L2 CACHE, then it must be retrieved from the un-core. When all the line fill buffers are in use, the data access operations in the load and

store buffers cannot be processed. They are thus queued up in the load and store buffers. When all the load or store buffers are occupied, the front end is inhibited from issuing micro operations to the RS and OOO engine. This is the same mechanism as used in Core™2 processors to maintain pipeline consistency.

The Intel® Core™ i7 processor has a 4 component hardware pre-fetcher very similar to that of the Core™ processors. Two components associated with the L2 CACHE and two components associated with the L1 data cache. The 2 components of L2 CACHE hardware pre-fetcher are similar to those in the Pentium™ 4 and Core™ processors. There is a "streaming" component that looks for multiple accesses in a local address window as a trigger and an "adjacency" component that causes 2 lines to be fetched instead of one with each triggering of the "streaming" component. The L1 data cache pre fetcher is similar to the L1 data cache pre-fetcher familiar from the Core™ processors. It has another "streaming" component (which was usually disabled in the bios' for the Core™ processors) and a "stride" or "IP" component that detected constant stride accesses at individual instruction pointers. The Intel® Core™ i7 processor has various improvements in the details of the hardware pattern identifications used in the pre-fetchers.

Normal Memory Subsystem

The normal or "un-core" is essentially a shared last level cache (L3 CACHE), a memory access chipset (Northbridge) , and a socket interconnection interface integrated into the multi-processor package. Cache line access requests (i.e. L2 Cache misses, un-cacheable loads and stores) from the cores are serviced and the multi socket cache line coherency is maintained with the other sockets and the I/O Hub.

There are five basic configurations of the Intel® Core™ i7 processor un-core.

1. Intel® Xeon™ 550 processor has a 3 channel integrated memory controller (IMC), 2 Intel® QuickPath Interconnects to support up to a DP configuration and an 8 MB L3 CACHE. This is the main focus of this document
2. Intel® Core™ i7 processor-HEDT (High End Desk Top) has a 3 channel IMC, 1 Intel® QuickPath Interconnect to access the chipset and an 8 MB L3 CACHE. This is for UP configurations
3. A quad core mainstream configuration with a 2 channel IMC, integrated PCI-e and an 8MB L3 CACHE
4. A dual core mainstream configuration where the memory access is through an off die chipset to enable support of more memory DIMM formats equipped with a 4MB L3 CACHE
5. The 8-core implementation based on the Nehalem microarchitecture will be the MP configuration.

Intel® Xeon™ 5500 Processor

IA block diagram of the Intel® Xeon™ 5500 processor package is shown below

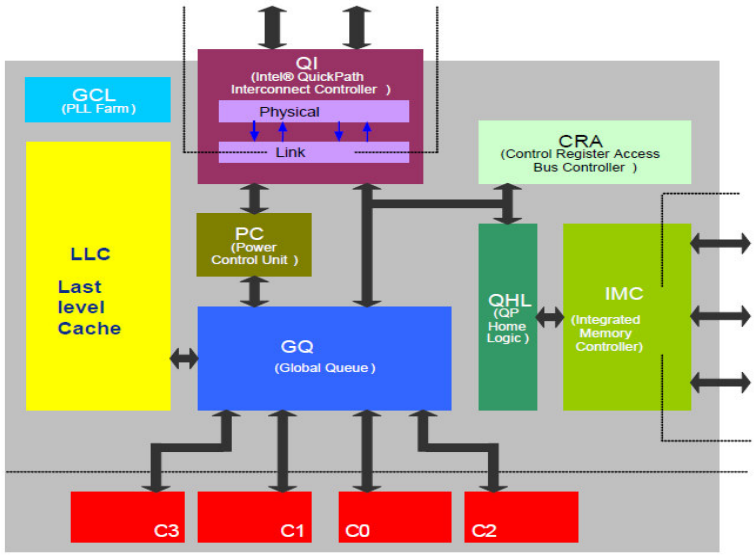

Figure 5 block diagram of the Intel® Xeon™ 5500 processor

Cache line requests from the cores or from a remote package or the I/O Hub are handled by the Intel® Xeon™ 5500 processor Un-core's Global Queue (GQ). The GQ contains 3 request queues for this purpose. One for writes with 16 entries and one of 12 entries for off package requests delivered by the Intel® QuickPath Interconnect and one of 32 entries for load requests from the cores. On receiving a cache-line request from one of the cores, the GQ first checks the Last Level Cache (L3 CACHE) to see if the line is on the package. As the L3 CACHE is inclusive, the answer can be quickly ascertained. If the line is in the L3 CACHE 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 will snoop 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 must send out requests for the line. Since the cache-line could be in the other package, a request through the Intel® QuickPath Interconnect (Intel QPI) to the remote L3 CACHE must be made. As each Intel® Core™ i7 processor package 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 package, then the GQ makes a simultaneous request C3 C1 GQ (Global Queue) IMC (Integrated Memory Controller) **LLC Last level Cache** QI (Intel® QuickPath Interconnect Controller) Link Physical CSI 6.4 GH 1 .4-2 .3 G / C0 C2 QHL (QP Home Logic) PC (Power Control Unit) CRA (Control Register Access Bus Controller) GCL (PLL Farm) Figure 3 12 to the local memory controller, the Integrated memory controller (IMC). If home is identified as belonging to the remote package, the request sent by the QPI will also be used to access the remote IMC.[15],[16]

Core Performance Monitoring Unit (PMU*)*

Each core has its own PMU. They have 3 fixed counters and 4 general counters for each Hyper-Thread. If Hyper Thread is disabled in the bios only one set of counters is available. All then core monitoring events count on a per thread basis with one exception that will be discussed. The PMIs are raised on a per logical core or Hyper Thread basis when Hyper Thread is enabled. There is a significant expansion of the PEBS events with respect to Intel® Core™2 processors. This will be discussed in detail. The Last Branch Record (LBR) has been expanded to hold 16 source/target pairs for the last 16 taken branch instructions.

Un-core Performance Monitoring Unit (PMU*)*

The Un-core has its own PMU for monitoring its activity. It consists of 8 general counters and one fixed counter. The fixed counter monitors the un-core frequency, which is different than the core frequency. In order for the un-core PMU to generate an interrupt it must rely on the core PMUs. If an interrupt on overflow is desired, a bit pattern of which core PMUs to signal to raise a PMI must be programmed. As the un-core events have no knowledge of the core, PID or TID that ultimately generated the event, the most reasonable approach to sampling on un-core events requires sending an interrupt signal to the entire core PMUs and generating one PMI per logical core.

Performance

The Intel Xeon processor family delivers the highest server system performance of any IA-32 Intel architecture processor introduced to date. Initial benchmark tests show up to a 65% performance increase on high-end server applications when compared to the previous-generation Pentium® III Xeon™ processor on 4-way server platforms. A significant portion of those gains can be attributed to Hyper-Threading Technology.

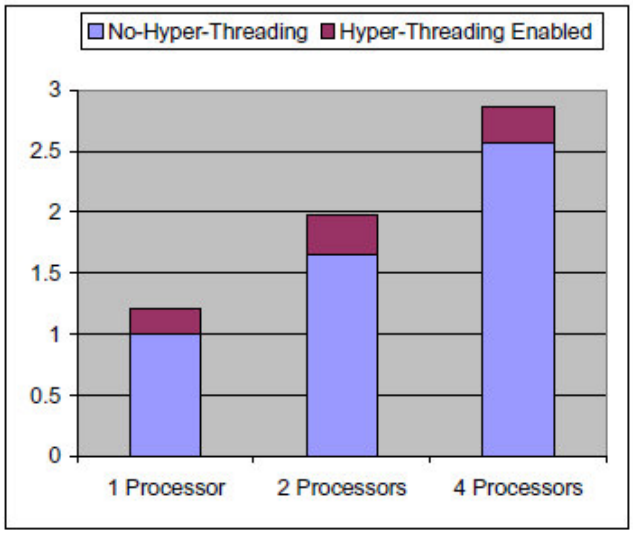

Figure 6: Performance increases HTT on an Online Transaction

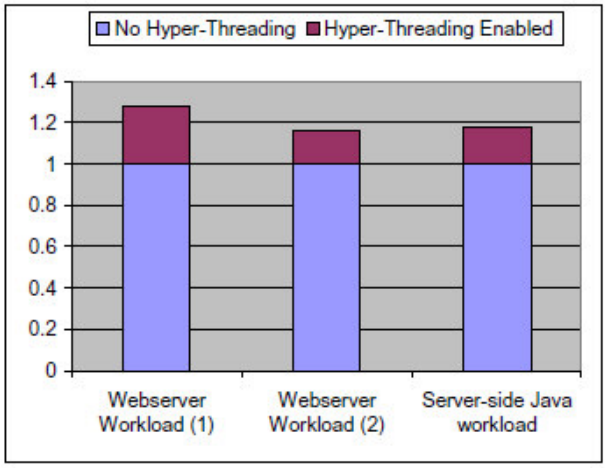

Figure 7 Webserver Benchmark Performance

Microsoft Operating System and Applications

A system with processors that use Hyper-Threading Technology appears to the operating system and application software as having twice the number of processors than it physically has. Operating systems manage logical processors as they do physical processors, scheduling runnable tasks or threads to logical processors. However, for best performance, the operating system should implement two optimizations. The first is to use the HALT instruction if one logical processor is active and the other is not. HALT will allow the processor to transition to either the ST0- or ST1-mode. An operating system that does not use this optimization would execute on the idle logical processor a sequence of instructions that repeatedly checks for work to do. This so-called "idle loop" can consume significant execution resources that could otherwise be used to make faster progress on the other active logical processor. The second optimization is in scheduling software threads to logical processors. In general, for best performance, the operating system should schedule threads to logical processors on different physical processors before scheduling multiple threads to the same physical processor. This optimization allows software threads to use different physical execution resources when possible.

Latency Event

Latency event gives us the best idea for performance measurement; the Intel® Core™ i7 processor has a "latency event" which is very similar to the Itanium® Processor Family Data EAR event. This event samples loads, recording the number of cycles between the execution of the instruction and actual

deliver of the data. If the measured latency is larger than the minimum latency programmed into MSR 0x3f6, bits 15:0, then the counter is incremented. Counter overflow arms the PEBS (Precise Event Based Sampling) mechanism and on the next event satisfying the latency threshold, the measured latency, the virtual or linear address and the data source are copied into 3 additional registers in the PEBS (Precise Event Based Sampling) buffer. Because the virtual address is captured into a known location, the sampling driver could also execute a virtual to physical translation and capture the physical address. The physical address identifies the Non Uniform Memory Architecture home location and in principle allows an analysis of the details of the cache occupancies.

Precise Execution Events

There are a wide variety of precise events monitoring other instructions than load and store instructions. Of particular note are the precise branch events that have been added. All branches, near calls and conditional branches can all be counted with precise events, for both retired and wrongly predicted (and retired) branches of the type selected. For these events, the PEBS (Precise Event Based Sampling) buffer will contain the target of the branch. If the Last Branch Record (LBR) is also captured then the location of the branch instruction can also be determined when the branch is taken the IP value in the PEBS (Precise Event Based Sampling) buffer will also appear as the last target in the LBR. If the branch was not taken (conditional branches only) then it won't and the branch that was not taken and retired is the instruction before the IP in the PEBS (Precise Event Based Sampling) buffer In the case of near calls retired, this means that Event Based Sampling (EBS) can be used to collect accurate function call counts. As this is the primary measurement for driving the decision to inline a function, this is an important improvement. In order to measure call counts, you must sample on calls. Any other trigger introduces a bias that cannot be guaranteed to be corrected properly.

The precise branch events are shown in the table below:

Table 1

Event Name Description unmask Event

BR_INST_RETIRED.CONDITIONAL Retired conditional branch instructions 01 C4

BR_INST_RETIRED.NEAR_CALL Retired near call instructions 02

BR_INST_RETIRED.ALL_BRANCHES Retired branch instructions 04

Shadowing

There is one source of sampling bias associated with precise events. It is due to the time delay between the PMU (Performance Monitoring Unit) counter overflow and the arming of the PEBS (Precise Event Based Sampling) hardware. During this period events cannot be detected due to the timing shadow. To illustrate the effect consider a function call chain where a long duration function, fn, which calls a chain of 3 very short duration functions, fn1 calling fn2 which calls fn3, followed by a long duration function fn4. If the durations of fn1, fn2 and fn3 are less than the shadow period the distribution of PEBS (Precise Event Based Sampling) sampled calls will be severely distorted.

1) If the overflow occurs on the call to fn, the PEBS (Precise Event Based Sampling) mechanism is armed by the time the call to fn1 is executed and samples will be taken showing the call to fn1 from fn.

2) If the overflow occurs due to the call to fn1, fn2 or fn3 however, the PEBS mechanism will not be armed until execution is in the body of fn4. Thus the calls to fn2, fn3 and fn4 cannot appear as PEBS sampled calls

Summary

The general rule that was used here is that we did not provide more VIRTUAL CPUs than the number of PHYISCAL cores the Server been used for testing has. In our scenario here, there are 32 logical processors presented due to hyper-threading, but only 16 physical cores. If we had provisioned more than 16 Virtual CPUs to the Machine it means that execution resources will now be shared for the Machine. Now there are some exceptions here (test your workloads!), but is it generally recommended not to exceed the

number of physical cores for this reason. Measuring the bandwidth for an individual core is complicated on Intel® Core™ i7 processors. And so that was left out

In order not go too deep here, we just concluded that non uniform memory architecture NUMA is a technology designed to assign affinity between CPUs and memory banks in order to optimize memory access times. Virtual non uniform memory architecture (vNUMA) was introduced to allow this technology to be extended down to guest virtual machines. The bottom line here is that the mix of virtual sockets and virtual cores assigned matters. As this article shows, processing latency can be increased if these settings are not optimal. First you'll want to make sure that hot-CPU add is disabled as this disables virtual NUMA in any virtual machine and then you'll want to make sure that your allocation of virtual sockets and virtual cores matches the underlying physical architecture or you could be adding some processing latency to your Machine.

Note there's a setting in firmware called Prefer HT. but it basically changes the preferences in Virtual NUMA. There's no universal answer here as it will vary from application to application, but this setting is a trade- off between additional compute cycles and more efficient access to processor cache and memory via virtual NUMA. If your application needs faster memory access more than it needs compute cycles, you may want to experiment with this setting.

Observations

It actually turned out that all of our settings used for test were optimal and the Operating System used throughout is Microsoft Windows. We had one virtual CPU socket with 16 cores – matching the 16 physical cores on the Server and virtual NUMA enabled. If you are using a Windows guest you can download Coreinfo.exe from Sys-internals and get more detail on how virtual NUMA is configured within your Machine. But that still did not answer the question – why is the Main CPU at 80% when the host is at 41% given 16 physical cores (main host) and 16 virtual cores ? Is it possible that not all the cores are being used? Without breaking down the math, the number of MHz consumed by the Machine divided by the capacity of the host does align with the CORE UTIL% metric.

One thing we could not figure out about this whole thing is why the host shows LESS MHz utilized. There should be no averaging here just raw MHz consumed – so it is bothering us why the host would show less consumed than the virtual (not possible in raw Mhz. what metric do we use to see actual core utilization without factoring for hyper threading? We must confess that we are lost here. Allegedly this metric exists but we couldn't find it anywhere: After some trial and error we did find a CPU Workload % metric which does appear to focus on the cores (no hyper-threading):

Now here's a question that troubles us. The default CPU metrics in vSphere count all the logical cores but look at the peak above. If we looked at the default CPU graph, we think was at 74% when the physical cores were actually at 88%. We can see how averaging across all logical cores can provide a better view of utilization, but it seems to us that the Workload metric (physical cores only) provides a better value for detecting shortcomings. A system with processors that use Hyper-Threading Technology appears to the operating system and application software as having twice the number of processors than it physically has. Operating systems manage logical processors as they do physical Processor Execution Resources

Conclusion

Intel's Hyper-Threading Technology brings the concept of simultaneous multi-threading to the Intel Architecture. This is a significant new technology direction for Intel's future processors. It will become increasingly important going forward as it adds a new technique for obtaining additional performance for lower transistor and power costs. The first implementation of Hyper-Threading Technology was done on the Intel Xeon processor MP. In this implementation there are two logical processors on each physical processor. The logical processors have their own independent architecture state, but they share nearly all the

physical execution and hardware resources of the processor. The goal was to implement the technology at minimum cost while ensuring forward progress on logical processors, even if the other is stalled, and to deliver full performance even when there is only one active logical processor. These goals were achieved through efficient logical processor selection algorithms and the creative partitioning and recombining algorithms of many key resources. Measured performance on the Intel Xeon processor MP with Hyper-
.

Threading Technology shows performance gains of up to 30% on common server application benchmarks for this technology. The potential for Hyper-Threading Technology is tremendous; our current implementation has only just begun to tap into this potential. Hyper-Threading Technology is expected to be viable from mobile processors to servers; its introduction into market segments other than servers is only gated by the availability and prevalence of threaded applications and workloads in those markets

References

[1] A. Agarwal, B.H. Lim, D. Kranz and J. Kubiatowicz, "APRIL: A processor Architecture for Multiprocessing," in Proceedings of the 17th Annual International Symposium on Computer Architectures, pages 104-114, May 1990.

[2] A. Quealy, R. Ryder, A. Norris, and N-S. Liu. "National Combustion Code: Parallel Implementation and Performance," 38th AIAA Aerospace Sciences Mtg., Reno, Nevada, Jan. 2000.

[3] B.J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," in SPIE Real Time Signal Processing IV, Pages 2 241 - 248, 1981.

[4] D. Marr, et al., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal, Volume 06, Issue 01 February 14, 2002. http://www.intel.com/technology/itj/archive/2002.htm

[5] D. J. C. Johnson, "HP's Mako Processor," Microprocessor Forum, October 2001, [6] http://www.cpus.hp.com/technical_references/mpf_2001.pd

[6] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism," in 22nd Annual International Symposium on Computer Architecture, June 1995.

[7] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in 23rd Annual International Symposium on Computer Architecture, May 1996.

[8] Intel Corporation. "IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture,"

[9] Intel® Microarchitecture (Nehalem), www.intel.com/technology/architecture-silicon/next-gen/.

[10] Intel Pentium 4 Processor Extreme Edition Supporting Hyper- Threading Technology, http://www.intel.com/products/processor/pentium4htxe/index.htm

[11] Intel Hyper-Threading Technology (Intel HT Technology), http://www.intel.com/technology/platform-technology/hyperthreading/

[12] Intel Westmere, http://ark.intel.com/ProductCollection.aspx?codeName=33174

[13] J. Boisseau, K. Milfeld, and C. Guiang. "Exploring the Effects of Hyperthreading on Scientific Applications," presented in Technical session number 7B, 45th Cray User Group Conference, , Columbus, Ohio, May 2003. http://www.cug.org/7-archives/previous_conferences/2003/CUG2003/pages/1-program/final_program/20.tuesday.htm

[14] J. M. Tendler, S. Dodson, and S. Fields, "POWER4 System Microarchitecture," Technical White Paper. IBM Server Group, October 2001. Intel Technology Journal Q1, 2002

[15] L. A. Barroso et. al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," in Proceedings of the 27th Annual International Symposium on Computer Architecture, Pages 282 - 293, June 2000.
[16] L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," Computer, 30(9), 79 - 85, September 1997.
[17] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee, "The M-Machine Multicomputer," in 28th Annual International Symposium on Microarchitecture, Nov. 1995.
[18] OVERFLOW: http://aaac.larc.nasa.gov/~buning/
[19] PAPI 4.1.1 Release, http://icl.cs.utk.edu/papi/news/news.html?id=203 [14] D. J. Mavriplis, M. J. Aftosmis, and M. Berger. "High Resolution Aerospace Applications using the NASA Columbia Supercomputer," Proc. ACM/IEEE SC05, Seattle, Washington, Nov. 2005.
[20] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porter, and B. Smith, "The TERA Computer System," in International Conference on Supercomputing, Pages 1 - 6, June 1990.
[21] S. Blackburn, P. Cheng, and K. McKinley. "Myths and Realities: The Performance Impact of Garbage Collection," Proc. SIGMETRICS '04, June 2004.
[22] Software Developers Programming Optimization Guide. Developer's Manual, Volume 3: System Programming Guide," Order number 245472, 2001 http://developer.intel.com/design/Pentium4/manuals.
[23] S. Saini, A. Naraikin, R. Biswas, D. Barkai, and T. Sandstrom, "Early Performance Evaluation of a Nehalem Cluster Using Scientific and Engineering Applications," Proc. ACM/IEEE SC09, Portland, Oregon, Nov. 2009.
[26] S. Saini, P. Mehrotra, K. Taylor, M. Aftosmis, and R. Biswas, "Performance Analysis of CFD Application Cart3D Using MPInside and Performance Monitor Unit Data on Nehalem and Westmere Based Supercomputers," 13th IEEE Intl. Conf. on High Performance Computing and Communications, Banff, Canada, Sep. 2011.
[25]W. Huang, J. Lin, Z. Zhang, and J. M. Chang. "Performance Characterization of Java Applications on SMT Processors," International Symp. on Performance Analysis of Systems and Software (ISPASS), March 2005,