The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications

Subhash Saini, Haoqiang Jin, Robert Hood, David Barker, Piyush Mehrotra and Rupak Biswas

NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field, CA 94035-1000, USA

{subhash.saini, haoqiang.jin, robert.hood, david.p.barker, piyush.mehrotra, rupak.biswas}@nasa.gov

Abstract—**Intel provides Hyper-Threading (HT) in processors based on its Pentium and Nehalem micro-architecture such as the Westmere-EP. 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 collect Performance Monitoring Unit (PMU) data (instructions retired; unhalted core cycles; L2 and L3 cache hits and misses; vector and scalar floating-point operations, etc.). We then use the PMU data to calculate 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 (CFD) used by NASA scientists indicate that HT generally improves processor resource utilization efficiency, but does not necessarily translate into overall application performance gain.**

Keywords: Simultaneous Multi-Threading (SMT), Hyper-Threading (HT), Intel's Nehalem micro-architecture, Intel Westmere-EP, Computational Fluid Dynamics (CFD), SGI Altix ICE 8400EX, Performance Tools, Benchmarking, Performance Evaluation

I. INTRODUCTION

Current trends in microprocessor design have made high resource utilization a key requirement for achieving good performance. For example, 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 bottleneck 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. SMT requires only some extra hardware instead of replicating the entire core. Price and performance benefits make it a common design choice as, for example, 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, SMT uses resource sharing to make the parallel implementation economical. With SMT, 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.

The purpose of this paper is to measure the impact of HT on processor utilization. We accomplish this by computing processor efficiency and investigating how various shared resources affect performance of scientific applications in HT mode. Specifically, we present a new metric for processor efficiency to characterize its utilization in single threading (ST) and HT modes for the hex-core Westmere-EP processor used in SGI Altix ICE 8400EX supercomputer. We also investigate the effect of memory hierarchy on the performance of scientific applications in both the modes. We use four production computational fluid dynamics (CFD) applications—OVERFLOW, USM3D, Cart3D, and NCC—that are used extensively by scientists and engineers at NASA and throughout the aerospace industry.

In order to better understand the performance characteristics of these codes, we collect Performance Monitoring Unit (PMU) data (instructions retired; L2 and L3 cache hits and misses; vector and scalar floating-point operations, etc.) in both ST and HT modes. We analyze the results to understand the factors influencing the performance of codes in HT mode.

The remainder of this paper is organized as follows. We present background and related work in the next section. Section III discusses HT in the context of the Nehalem micro-architecture and its Westmere-EP processor. In Section IV, we detail the architecture of the platform used in this study—the SGI Altix ICE 8400EX, based on the Westmere-EP processor. Section V discusses the experimental setup, including the hardware performance counters. In Section VI, we describe the benchmarks and applications used in our study. In Section VII, we discuss metrics used to measure the effectiveness of HT and the utilization of processor resources in both ST and HT modes. Section VIII presents and analyzes the performance results of our experiments. We discuss other factors that influenced the results of this study in Section IX, and end with some conclusions from this work in Section X.

II. BACKGROUND AND RELATED WORK

Intel introduced SMT, called Hyper-Threading (HT), into its product line in 2002 with new models of their Pentium 4 processors [1-3]. 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 HT on application performance [4-6]. Boisseau et al. conducted a performance evaluation of HT on a Dell 2650 dual processor-server based on Pentium 4 using matrix-matrix multiplication and a 256-particle molecular dynamics benchmark written in OpenMP [4]. Haung et al. characterized the performance of Java applications using Pentium 4 processors with HT [5]. Blackburn et al. studied the performance of garbage collection in HT mode by using some of the Pentium 4 performance counters [6]. A key finding of these investigations was that the Pentium 4's implementation of HT was not very advantageous, as the processor had very limited memory bandwidth and issued only two instructions per cycle.

Recently, HT was extended to processors that use Intel's Nehalem micro-architecture [7]. In these processors, memory bandwidth was enhanced significantly by overcoming the front-side bus memory bandwidth bottleneck and by increasing instruction issuance from two to four per cycle. Saini et al. conducted a performance evaluation of HT on small numbers of Nehalem nodes using NPB [8]. Results showed that for one node, HT provided a slight advantage only for LU**.** BT, SP, MG, and LU achieved the greatest benefit from HT 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 HT to measure the relative efficiency *E* of the processor in terms of cycle per instruction using the formula

$$E = 100*(2*CPI_{ST} / CPI_{HT}) - 100$$

where CPI_{ST} and CPI_{HT} are cycle per instruction in ST and HT modes respectively [9].

In this study we focus on the Westmere-EP Xeon processor, which is based on the Nehalem micro-architecture.

The contributions of this 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.
- We analyze 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 HT mode.

III. 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 HT, the Nehalem micro-architecture has several advantages over the Pentium 4. First, the newer design has much more memory bandwidth and larger caches, giving it the ability to get data to the core faster. Second, 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 [2]:

- *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 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 HT 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 behavior of the threads. Some of the shared resources are unaware of HT. 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 HT 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.

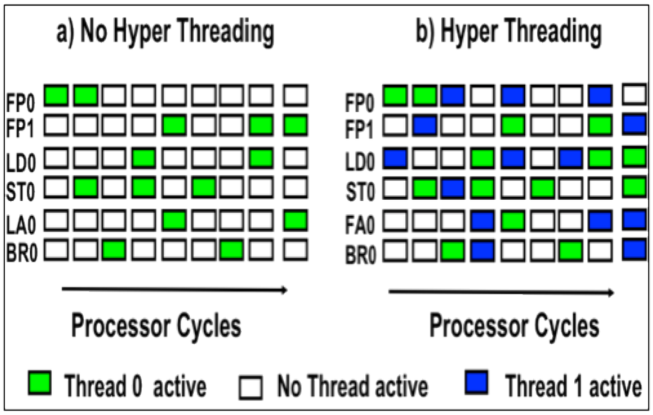

Figure 1. Hyper Threading on the sixteen-stage pipeline Nehalem architecture with six execution units.

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.

IV. COMPUTING PLATFORM

This study was conducted using NASA's Pleiades supercomputer, an SGI Altix ICE 8400EX system located at NASA Ames Research Center. 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 this study, we used the Westmere-EP based nodes [10]. 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 140.64 Gflop/s.

Each Westmere-EP processor has two parts: "core" and "uncore". The core part consists of six cores with per-core L1 and L2 caches. The uncore part has a shared L3 cache, an integrated memory controller, and QuickPath 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.

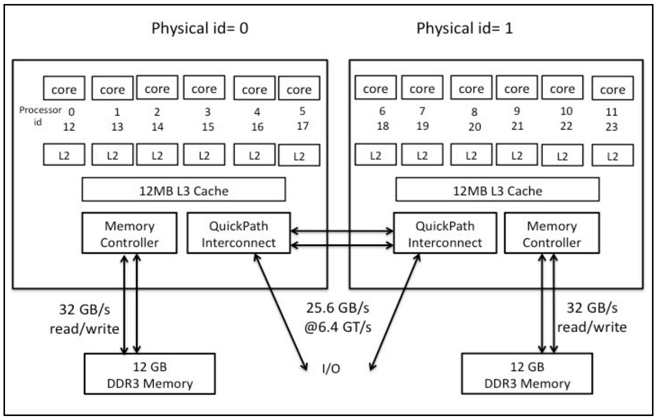

Figure 2. Configuration of an Intel Westmere-EP node.

V. EXPERIMENTAL SETUP AND COUNTERS

In this section we give a brief description of the experimental setup for collecting and analyzing the data based on the hardware performance counters. We also describe the performance counters used in our study.

A. *Experimental Setup*

In this work, we used the SGI Message Passing Toolkit (MPT) version 1.25 and the Intel compiler version 11.1 [12]. We used *op_scope*, a tool developed by Supersmith to collect low-level performance data, e.g. floating-point operations, instruction counts, clock cycles, cache misses/hits, etc. [18]. The tool relies on the Performance Application Programming Interface (PAPI) [13] to access hardware performance counters. In the present study, *op_scope* was built with PAPI version 4.1.0.

The experiments were performed using the same number of physical resources for both ST and HT modes; that is, for a given number of physical cores, say *n*, we used *n* MPI processes in ST mode but doubled it to *2n* MPI processes in HT mode. The main reason for keeping the number of cores used constant while toggling HT was to approximate the situation faced by users: whether or not they should use HT when running an application on a given set of resources.

Note that this approach raises two issues for the remainder of the analysis. First, we are making an implicit assumption that the codes scale perfectly, i.e., there is no performance loss or gain in going from *n* MPI processes (in ST mode) to *2n* MPI processes (in HT mode). Second, this approach also changes the amount of work done per MPI process and the overall communication pattern, which we are not considering. With ST, there are a maximum of 12 MPI processes communicating with other nodes while the number doubles in HT mode, thereby potentially creating a bottleneck at the Host Channel Adaptor (HCA), a physical network card that connects a node to the IB network fabric. A similar communication bottleneck can also occur at the inter IRU links. This effect is more pronounced at a higher number of cores especially for MPI collectives such as

MPI_Allreduce and *MPI_Bcast*. We will not be able to fully address the network-dependent effects in this study since we did not gather any network data for MPI communication. We are currently examining mechanisms for differentiating hardware counter data from inside and outside communication routines.

A tool called *dplace* from SGI was used to bind a related set of processes to specific cores to prevent process migration. In addition, if only a part of the node was used in a run, it was ensured that both ST and HT modes used the same set of cores. Also, in order to reduce the impact of the initialization phase of an application (reading the input data, setting up the computational grid, etc.) on the results, each case was run twice—once for the first iteration only and another for all iterations. Results from the first run were then subtracted from the second run for both timing and hardware counter data.

B. *Westmere-EP Performance Counter Events*

Hardware counter data was collected from the Performance Monitor Unit (PMU) of the Westmere-EP processor. PMU provides seven counters per core; 3 fixed and 4 general-purpose [10]. PAPI users can access 117 native events. We narrowed these 117 down to 8 events that were appropriate for the present study; Table 1 shows the names and descriptions of the events we used. The counter data presented in the later sections is an average of the values collected for all the MPI processes.

TABLE I. **Intel Westmere-EP events.**

Name	Description
UNHALTED_CORE_CYCLES	Clock cycles when not halted
INSTRUCTIONS_RETIRED	Number of instructions retired
FP_COMP_OPS_EXE:SSE_FP_PACKED	Number of packed FP uops executed
FP_COMP_OPS_EXE:SSE_FP_SCALAR	Number of scalar FP uops executed
L2_RQSTS:LD_HIT	Number of loads that hit the L2 cache
L2_RQSTS:LD_MISS	Number of loads that miss the L2 cache
LLC_REFERENCES	Last level cache demand requests from this core
LLC_MISSES	Last level cache demand requests from this core that missed the LLC

VI. APPLICATIONS USED IN THE STUDY

Here is a brief overview of the codes that we used in this study.

Cart3D is an unstructured high fidelity, inviscid CFD application that solves the Euler equations of fluid dynamics [14]. It includes a solver called Flowcart, which uses a second-order, cell-centered, finite-volume upwind spatial discretization scheme, in conjunction with a multi-grid accelerated Runge-Kutta method for steady-state cases. We used the geometry of the Space Shuttle Launch Vehicle (SSLV) for the simulations in this work. The SSLV uses 24 million cells for computation, and the input dataset is 1.8 GB. The application (in this case, the MPI version) requires 16 GB of memory to run.

OVERFLOW is a general-purpose Navier-Stokes solver for CFD problems [15]. The Fortran90 MPI version has 130,000 lines of code. The application uses an overset grid methodology to perform high-fidelity viscous simulations around realistic aerospace configurations. The main computational logic of the sequential code consists of a time loop and a nested grid loop. The code also uses finite differences in space with implicit time stepping, and overset structured grids to accommodate arbitrarily complex moving geometries. The dataset used here is a wing-body-nacelle-pylon geometry (DLRF6), with 23 zones and 36 million grid points. The input dataset is 1.6 GB in size, and the solution file is 2 GB.

NCC, the National Combustion Code, is an unstructured-grid Navier-Stokes CFD application used to develop new physical models for turbulence, chemistry, spray, and turbulence-chemistry, as well as turbulence-spray interactions [16]. It employs a cell-centered finite-volume spatial discretization and pseudo-time preconditioning. An explicit four-stage Runge-Kutta scheme is used to advance the solution in pseudo-time for steady state or time-accurate simulations. Domain decomposition to divide the total computational domain into spatial zones is performed by using the METIS partitioner. Each zone is solved on a separate core and MPI is used for inter-core communication. The test case used is H2C4 fuel injector geometry consisting of seven individual injectors in a radial array each with four gaseous hydrogen injection ports. A 3.49 million element tetrahedral grid is used to model the injector and cylindrical duct used in the experiment. Each NCC run consisted of 350 pseudo-time iterations.

USM3D is a 3-D unstructured tetrahedral, cell-centered, finite-volume Euler and Navier-Stokes flow solver [17]. Spatial discretization is accomplished using an analytical reconstruction process for computing solution gradients within tetrahedral cells. The solution is advanced in time to a steady-state condition by an implicit Euler time-stepping scheme. A single-block, tetrahedral, unstructured grid is partitioned into a user-specified number of contiguous partitions, each containing nearly the same number of grid cells. Grid partitioning is again accomplished using METIS.

The test case used 10 million tetrahedral elements, requiring about 16 GB of memory and 10 GB of disk space.

VII. Performance Metrics

From the user's point of view, the impact of HT is typically measured by calculating the relative speed up attained, e.g. the code sped up by x% using HT. Using PMU counters, we calculate this performance gain as:

$$P = (C_{ST} - C_{HT}) / C_{ST},$$

where C_{ST} and C_{HT} are *UNHALTED_CORE_CYLES* in ST and HT modes, respectively.

From the application point of view, the number unhalted core cycles is an important metric for code optimization as it reflects the total execution time. The goal of any code optimization is to minimize the (a) cycles that are stalled by improving code and data locality, (b) minimizing branches or using more predictable branching, and (c) using vector instructions and/or faster and more efficient algorithms.

In order to measure the core-level effects of HT, we define a quantity called efficiency that reflects the utilization level of the core's execution units. In particular, we calculate the fraction of available micro-operation slots that are being used to completely execute an instruction. If the fraction is high, the execution units are being kept busy doing useful work.

The total number of available micro-operation slots during an execution on a single core is

$$S = \mu \cdot C,$$

where C is the number of cycles executed on the core and μ is the number of micro-operation slots available per cycle, e.g., $\mu = 4$ in the Nehalem microarchitecture. If I is the number of instructions retired by the core during execution, then the efficiency is

$$E = I / S$$

The theoretical maximum value of E is unity and reflects the case where each micro-operation slot is being used to retire an instruction. In practice, however, some instructions will result in multiple micro-operations being issued. In addition, there will often be empty slots because values needed for a micro-operation are not available yet. Thus, typical efficiencies will be less than one.

With our experimental setup, we can use the PMU counters, which are per-thread counters, to calculate efficiency during a single-threaded run as:

$$E_{ST} = \frac{INSTRUCTIONS\_RETIRED}{4 \cdot UNHALTED\_CORE\_CYCLES},$$

When running with HT, we note that since the core is retiring instructions from two threads we need to add the per-thread *INSTRUCTIONS_RETIRED* hardware counter for each thread. The two *UNHALTED_CORE_CYCLES* counters, on the other hand, are usually both incremented for each cycle, as the core is halted relatively infrequently. Thus, either counter can be used to reflect the number of cycles executed on the core. We calculate the efficiency for the whole core in HT mode as:

$$E_{HT} = \frac{2 \cdot INSTRUCTIONS\_RETIRED}{4 \cdot UNHALTED\_CORE\_CYCLES},$$

because the instructions retired counter value reflects the average across all threads in the computation.

VIII. Results

In this section we present results for performance gain and efficiency, and then explain those results in terms of vectorization and memory hierarchy effects.

A. Efficiency and Performance Gain

To begin our analysis of hyper-threading, we examine four metrics for the four applications in the study. Each of the graphs in this section shows four plots:

- HT efficiency (E_{HT} as defined in the last section),
- ST efficiency (E_{ST}),
- efficiency difference ($E_{HT} - E_{ST}$), which is labeled with "Efficiency difference *(HT–ST)*", and
- performance gain (P).

1) NCC

Figure 3 shows the plots for NCC. Efficiency in HT mode is always higher than with ST. Efficiency increases from 35.3% to 40.5% and 39.3% to 44.3% in ST and HT modes, respectively, across the core counts. The difference between the HT and ST efficiency increases from 4% at 24 cores to 5.3% at 96 cores and then decreases to 3.8% at 384 cores. The HT efficiency correlates with the performance gain, which increases with larger core count because data starts fitting in L1 data cache. NCC shows super-linear scaling in ST mode and has enhanced super-scaling in HT mode as data for both threads fits into L2 cache.

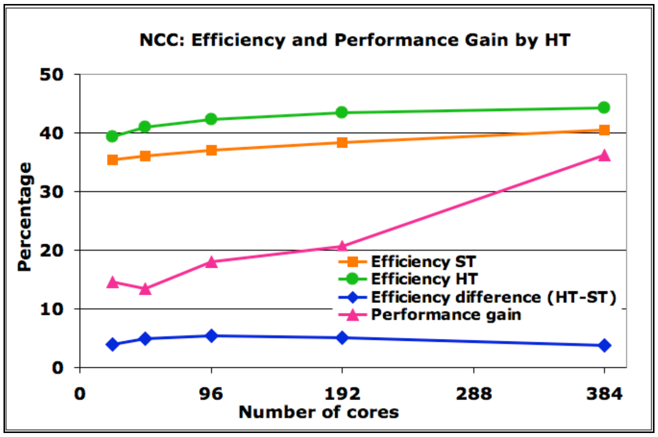

Figure 3. Efficiency and percentage performance gain for NCC.

2) *USM3D*

Figure 4 presents the data for USM3D. We do not show results for 256 cores since we do not have a grid for 512 processes. (The data point requires that the HT run use 512 processes on 256 cores). As with NCC, efficiency in HT mode is always higher than with ST. Efficiency in both ST and HT modes decreases from 32 to 64 cores and then increases to 128 cores. The difference between the HT and ST efficiency decreases from 1.9% at 32 cores to 0.9% at 64 cores and then increases to 3.3% at 128 cores. Performance gain is 11% and remains almost constant.

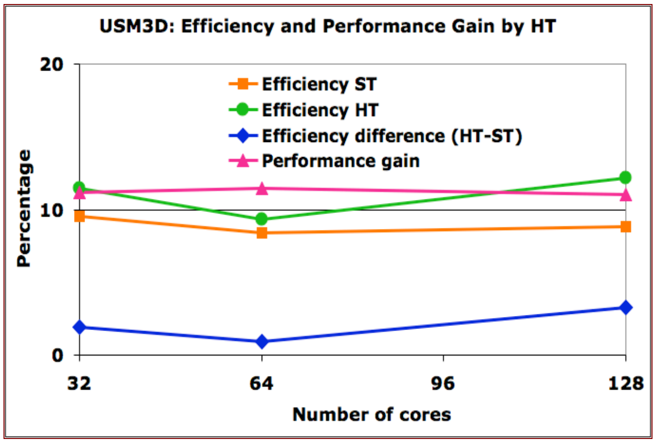

Figure 4. Efficiency and performance gain for USM3D.

3) *Cart3D*

Figure 5 shows the graphs for Cart3D. Again, efficiency in HT mode is always higher than in ST mode. Performance gain is 14%, 7%, 22%, and 15% for 24, 48, 96, and 192 cores, respectively. There is excellent anti-correlation between HT efficiency and performance gain.

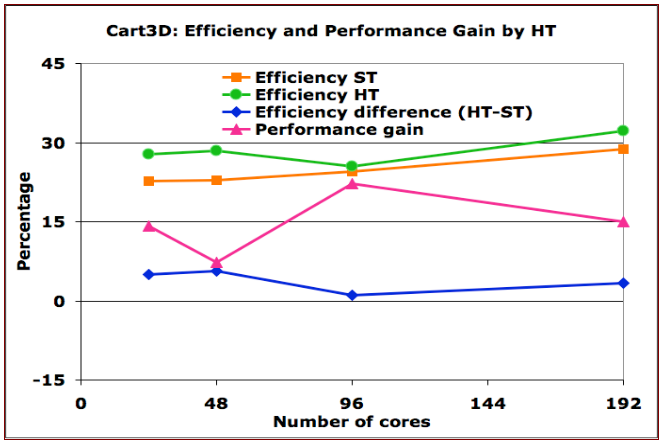

Figure 5. Efficiency and percentage performance gain for Cart3D.

4) *OVERFLOW*

The results for OVERFLOW are plotted in Figure 6. As with the other applications, efficiency in HT mode is always higher than the ST mode. Efficiency in ST and HT modes as well as the difference between the two increases as the number of cores increases. The reason for this improvement is that more data fits into L2 cache with increasing core count. There is a good correlation between efficiency and performance gain. But efficiency does not explain the magnitude of performance gain.

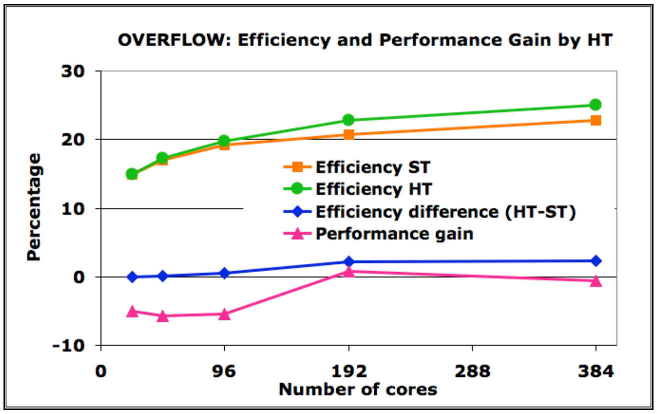

Figure 6. Efficiency and percentage performance gain for OVERFLOW.

Figure 7 recaps the performance gain in HT mode for four applications–NCC, USM3D, OVERFLOW and Cart3D. OVERFLOW is the only application that does not benefit from HT. The other three applications do benefit from HT.

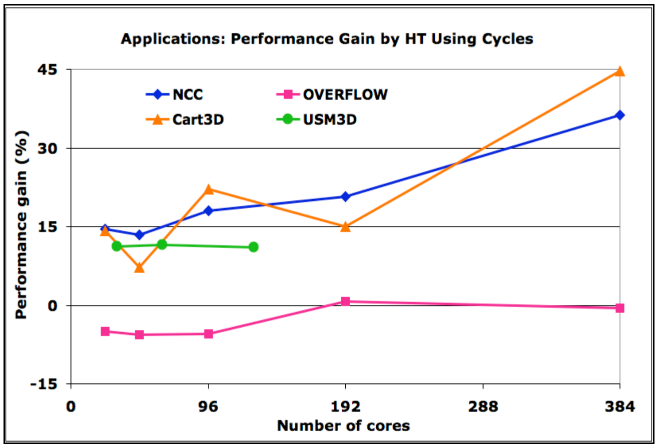

Figure 7. Performance gain in HT mode for applications.

B. *Effect of Code Vectorization*

In our observations, one of the factors influencing the performance of HT is the degree of vectorization in the application code. We compute the percentage of vectorization from two different hardware counters:

- FP_COMP_OPS_EXE:SSE_FP_PACKED that gives the number of vector micro-operations executed, and
- FP_COMP_OPS_EXE:SSE_FP_SCALAR that gives the number of scalar micro-operations executed.

Figure 8 shows the percentage of vector instructions (over all floating point instructions) for the codes used in the study. For each code, this percentage was fairly consistent (within 1%) across the range of core counts. In comparing the vectorization percentage with performance gain, we observe that high vectorization correlates to a negative HT impact, as in the case of OVERFLOW as shown in Figure 7. However, it does not necessarily follow that at lower percentages of vectorization, there is a correlation between the degree of vectorization and performance gain. In particular, NCC shows the best overall gain, but lies between USM3D and Cart3D in vectorization percentage. In order to understand this behavior, we need to explore the implementation of HT more closely.

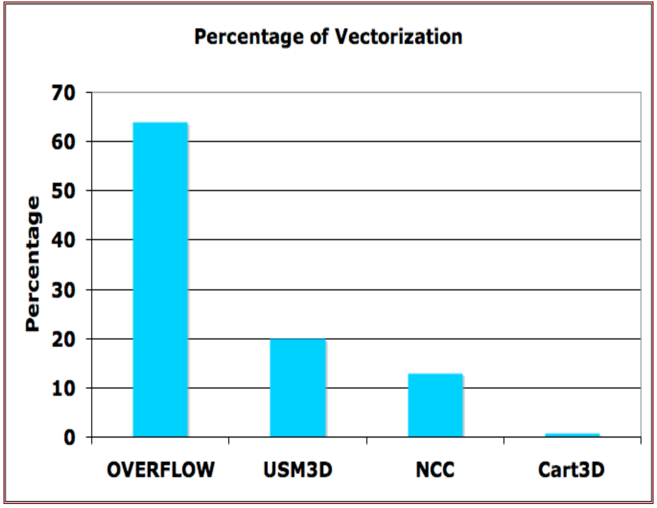

Figure 8. Percentage of vectorization of codes.

The main benefit of HT comes from the ability of execution units in the core, such as the floating-point units (FPU), to handle instructions from more than one thread simultaneously. The FPU is a shared resource that is unaware of the multiple threads. From its perspective, it is merely handling a stream of instructions organized in a pipeline of the six execution units—during each cycle, it can start executing the micro-operation in the next stage. Note that this will often lead to gaps (as shown earlier in Figure 1) where there is no micro-operation to execute. This could be, for example, due to a wait for a load instruction to complete. With HT, such gaps in the FPU's pipeline can be filled with micro-operations from a second thread—thus making for better utilization of the FPU.

Pitted against any potential benefit due to HT is the additional cost of executing with multiple threads. There is almost certainly going to be a time penalty due to increased contention in the memory hierarchy. The bottom line is that we will only see an overall benefit for HT if the time saved by utilizing the idle resources in the pipeline is greater than the extra time needed due to memory hierarchy contention. With a high level of vectorization, the number of execution gaps is very small and there is possibly insufficient opportunity to make up any penalty due to increased contention in HT. With a low level of vectorization there is potential for benefit. Thus, the level of increased memory hierarchy and network contention will determine whether there is any HT benefit.

C. *Effect of Memory Hierarchy*

In this subsection we focus on analyzing the effect of the memory hierarchy on performance when running in HT mode. If sufficient resources (cache and memory bandwidth) are available, sharing them across multiple threads in HT mode will result in better performance than with ST. However, we expect that if such sharing increases contention between the threads to the extent that data needs to be accessed from the next level of cache for each of the threads, there will, in general, be no performance benefit of running in HT mode. Below, we analyze hardware-counter based data related to cache and memory accesses in order to identify application characteristics that can provide a performance boost with HT.

We present two kinds of graphs for each application, namely

1. Percentage of data from each source (L2 cache, L3 cache, and main memory – MM) in ST mode, as:

$$\textit{\% data from L2} = L2H / L2R \cdot 100,$$
$$\textit{\% data from L3} = L3H / L3R \cdot L2M / L2R \cdot 100,$$
$$\textit{\% data from MM} = L3M / L3R \cdot L2M / L2R \cdot 100,$$

2. Percentage difference in ST and HT mode (ST–HT) from each data source as listed above as

$$\textit{Difference of \% data from XX} = \textit{(\% data from XX in ST mode)} - \textit{(\% data from XX in HT mode)},$$

where $XX = L2$, $L3$, or MM.

In the above formulas, L2H (L2 cache hit), L2M (L2 cache miss), L3R (L3 cache reference), and L3M (L3 cache miss) correspond to the following measured counter data, respectively: L2_RQSTS:LD_HIT, L2_RQSTS:LD_MISS, LLC_REFERENCES, and LLC_MISSES. We calculate L2R (L2 cache reference) from L2H+L2M and L3H (L3 cache hit) from L3R-L3M. We assume all the L3 cache misses hit the main memory.

We use these graphs to explain how using HT impacts the four applications. For each application, the first graph allows us to identify the primary source of the data. If the second graph shows that the difference in the ST and HT percentage for the primary data source is high, it implies that in HT mode the two threads have to go to the next level of the memory hierarchy more often, thus incurring extra latency costs. Our proposition is that a low value for this difference should result in a performance gain for HT mode. Our overall proposition is that a code benefits from HT if the primary source of data can accommodate the request in HT mode also.

1) NCC

Figure 9 shows the percentage of data from each source for NCC in ST mode. The amount of data coming from L2 cache, L3 cache, and main memory is 69-79%, 7-13%, and 15-20%, respectively. The percentage of data coming from L2 cache decreases with increasing number of cores because a larger portion of the process's data fits into L1 cache.

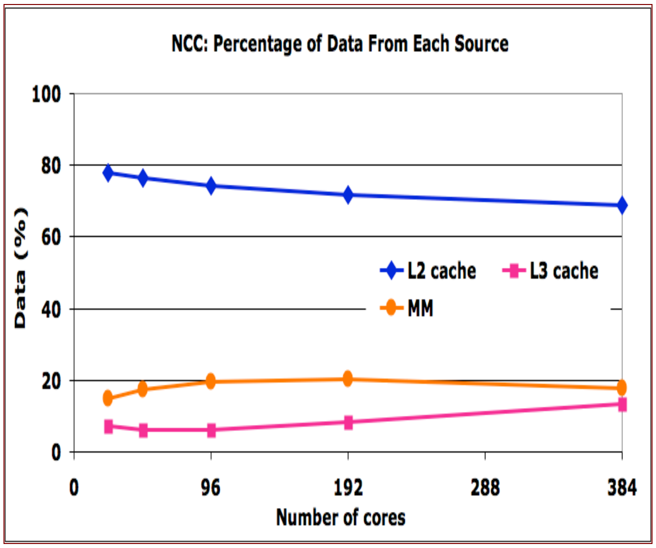

Figure 9. Percentage of data from each source for NCC.

Figure 10 shows the percentage difference between ST and HT modes for the three data sources for NCC, along with the performance gain for HT over ST. The percentage difference between ST and HT for NCC from L2 cache, L3 cache, and main memory is 20% to 50%, -44% to -22%, and -6% to 2%, respectively. In particular, the percentage difference for the primary source of data, L2 cache, steadily decreases from 50% to 20% across the range of cores. There is an anti-correlation between the performance gain and the percentage difference between ST and HT for L2 cache. That is, with increasing cores, the L2 cache can better accommodate both threads in HT mode resulting in more performance gain in this mode.

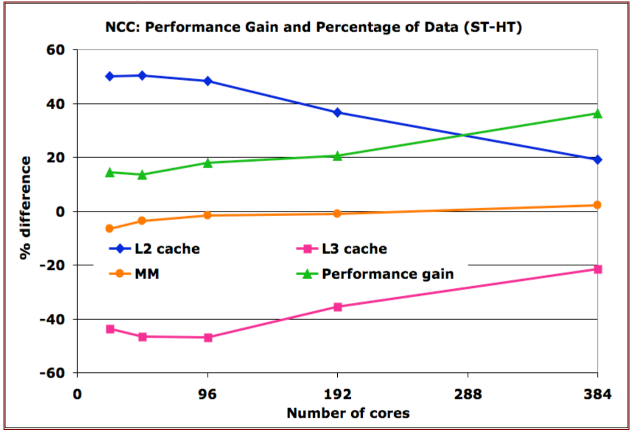

Figure 10. Performance gain and percentage difference between ST and HT for NCC.

2) Cart3D

Figure 11 shows the percentage of data from each source for Cart3D. In ST mode, 61% to 65%, 5% to 8%, and 27% to 34% of the data comes from L2 cache, L3 cache, and main memory, respectively. Hence, the primary source of data is L2 cache.

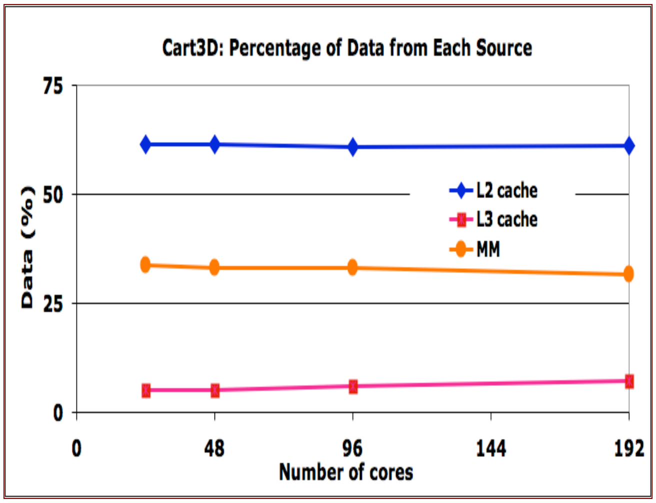

Figure 11. Percentage of data from each source for Cart3D.

Figure 12 shows the percentage difference between ST and HT modes for the three data sources for Cart3D along with the performance gain by HT over ST. As was the case with NCC, there is an anti-correlation between the performance gain and the percentage difference between ST and HT for the primary data source—L2 cache.

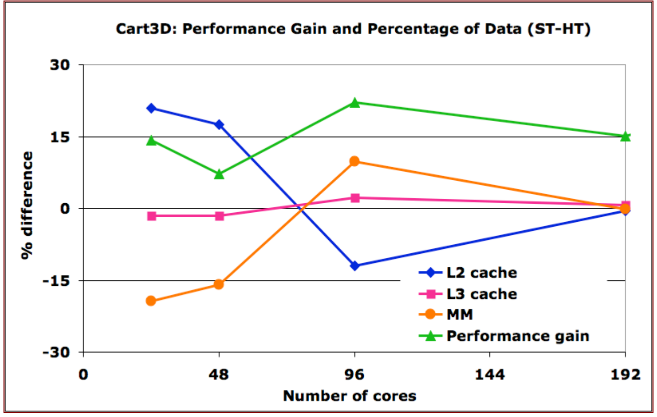

Figure 12. Performance gain and percentage difference between ST and HT for Cart3D.

3) USM3D

Figure 13 shows that main memory (MM) is the source for almost 80% of the data for USM3D across the whole range of cores tested. Note that USM3D is an unstructured code with tetrahedral meshes involving indirect addressing. It usually cannot reuse the data from L2 or L3 cache and thus has to fetch data from main memory. Thus, the various cache levels do not seem to play any significant role for this application.

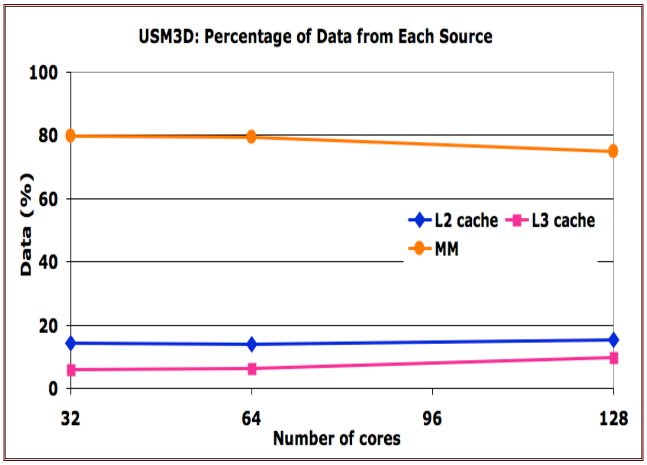

Figure 13. Percentage of data from each source for USM3D.

Figure 14 shows the percentage difference between ST and HT modes for L2 cache, L3 cache and main memory for USM3D. The differences are small, indicating that as we expected, even in HT mode most of data comes from the main memory. Since the code is only 20% vectorized and most of the data (80%) comes from main memory, there is an opportunity to hide memory latency of one thread while the second thread utilizes the floating-point units in the HT mode. This results in better performance gain. As was the case with the two previous applications, there is an anti-correlation between the performance gain and the percentage difference between ST and HT for the primary data source—in this case main memory.

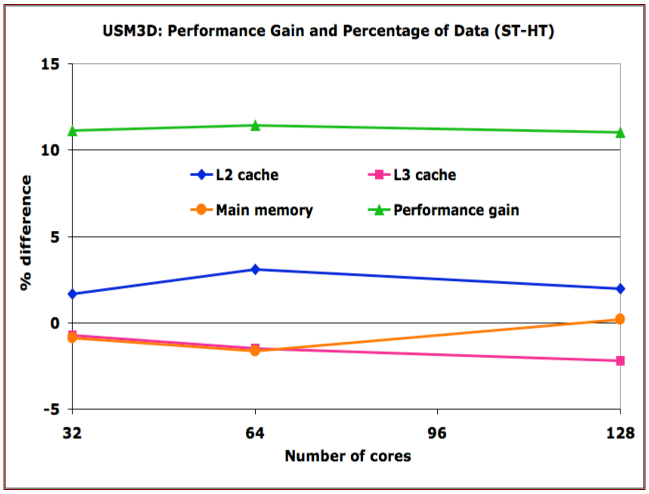

Figure 14. Performance gain and percentage difference between ST and HT for USM3D.

4) *OVERFLOW*

Figure 15 shows the percentage of data from each source for OVERFLOW in ST mode. For ST, the amount of data coming from L2 cache, L3 cache, and main memory is 44-71%, 18-31%, and 12-25%, respectively. As the number of cores increases, more data fits into L2 cache. As a result, a higher percentage of data comes from it.

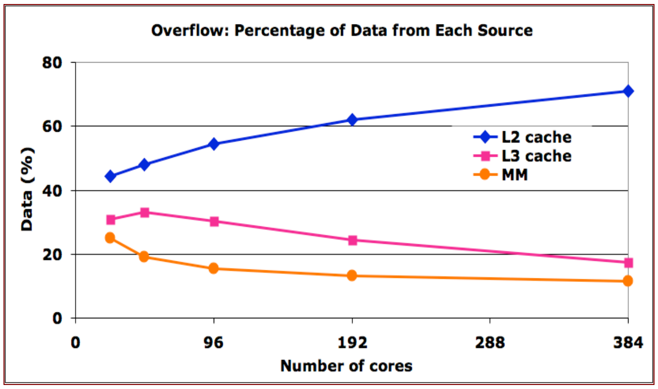

Figure 15. Percentage of data from each source for OVERFLOW.

Figure 16 shows the percentage difference between ST and HT modes for L2 cache, L3 cache, and main memory for OVERFLOW. As was seen in the other applications, there is an anti-correlation between performance gain and the percentage difference between ST and HT for the primary data source—in this case L2 cache. Note also that the secondary data source (L3 cache) has a negative difference between ST and HT. This means that HT causes more L3 requests than ST. Since the L3 latency is higher, this degrades the overall performance.

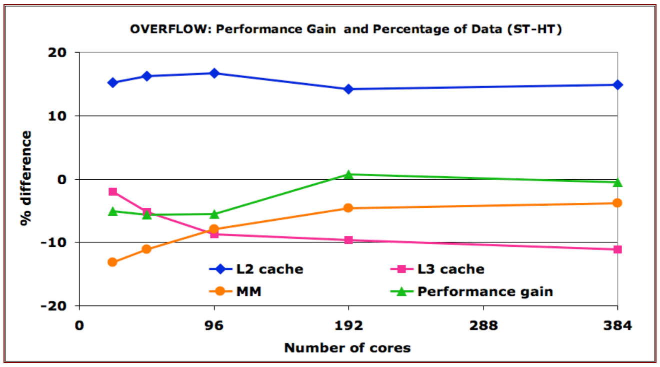

Figure 16. Performance gain and percentage difference between ST and HT for OVERFLOW.

IX. IMPACT OF APPLICATION CHARACTERISTIC

The applications we investigated in this paper fall into two broad classes at the highest level: those utilizing structured meshes (OVERFLOW) and those utilizing unstructured meshes (NCC, Cart3D, and USM3D). The three unstructured applications use a similar strategy for sub-domain partitioning of the grids except that Cart3D uses unstructured Cartesian grids and load-balancing is done via space-filling curves whereas USM3D and NCC use tetrahedral grids and partitioning is accomplished using METIS. In this study we found that unstructured-grid applications benefit by running them in HT mode whereas structured-grid applications do not. In order to understand this performance behavior, we briefly describe the characteristics of these two classes of applications.

Structured applications access adjacent elements of the underlying data structures and this spatial locality of data allow them to be optimized for cache by the compiler. Such codes also tend to be associated with a high degree of vectorization. The success of vectorization puts increased pressure on the memory hierarchy and can result in stalls that lower our measure of efficiency. Adding a second thread to that core with HT increases the demands on memory and communication resources, and does not result in any performance benefits.

Unstructured applications, on the other hand, involve indirect addressing and adjacent elements are often not accessed in sequence. Also, the compiler is usually unable to vectorize the codes, which results in sub-optimal utilization of floating-point execution units and gives opportunities for HT to utilize the resources. Thus, we expect hyper-threading to provide a boost in performance as long as there is no significant increase in the contention for the memory hierarchy or communication resources.

X. CONCLUSIONS

In this paper we have studied the effect of hyper-threading on four applications of interest to NASA: Cart3D, NCC, USM3D, and OVERFLOW. While the first three showed performance boosts from using HT, OVERFLOW did not.

In order to explain the differences in performance that we saw, we introduced an efficiency metric to quantify processor resource utilization. Using the metric, we find that efficiency in hyper-threaded mode is higher than in single-threaded mode across all core counts for all four applications. Since OVERFLOW did not see any improvement from HT, there must be other factors influencing performance. In particular, vectorization plays a key role, as OVERFLOW was by far the most highly vectorized of the codes in the study.

HT increases competition for resources in the memory hierarchy, such as memory bandwidth. Moreover, HT performance is affected by increased communication pressure as additional processes compete for network resources such as HCA chips and IB switches. One factor that affects the results of our experiments is that we conducted a strong scaling study. Also, in the analysis we have assumed that the applications scale perfectly from *n* to *2n* ranks, and thus the entire performance impact in going from ST to HT mode is from the use of hyper-threading. We have not taken into account the changes in communication in our analysis of the results.

We found that unstructured-grid applications like NCC, Cart3D, and USM3D benefit from HT whereas the structured-grid application (OVERFLOW) did not. The unstructured codes usually have a low percentage of vectorization and could get a performance boost from HT provided competition from an additional thread does not cause load instructions to go deeper in the memory hierarchy to be satisfied. We also found an anti-correlation between the performance gain in HT mode and the primary data source for the four applications used in the present study.

As future work, we propose to quantify the impact of scaling and communication in HT mode. We also intend to investigate the impact of power and thermal efficiencies in HT mode.

ACKNOWLEDGMENT

We gratefully acknowledge Sharad Gavali's help with running NCC, and stimulating discussions with Johnny Chang, Jahed Djomehri and Kenichi Taylor.

REFERENCES

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

[2] *Intel Hyper-Threading Technology (Intel HT Technology)*, http://www.intel.com/technology/platform-technology/hyper-threading/

[3] 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

[4] 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

[5] 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,

[6] S. Blackburn, P. Cheng, and K. McKinley. "Myths and Realities: The Performance Impact of Garbage Collection," Proc. SIGMETRICS '04, June 2004.

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

[8] 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.

[9] 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.

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

[11] *SGI Altix ICE 8400:* http://www.sgi.com/products/servers/altix/ice/

[12] *Message Passing Toolkit (MPT) User's Guide,* http://techpubs.sgi.com/library/manuals/3000/007-3773-003/pdf/007-3773-003.pdf

[13] *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.

[15] *OVERFLOW*: http://aaac.larc.nasa.gov/~buning/

[16] 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.

[17] *USM3D,* http://aaac.larc.nasa.gov/tsab/usm3d/usm3d_52_man.html

[18] *op_scope*, Supermith, Inc., Pebble Beach, CA, http://supersmith.com/op_scope