Exploring the Effects of Hyper-Threading on Scientific Applications

Kent F. Milfeld, Chona S. Guiang, Avijit Purkayastha, *and*
John R. Boisseau, *Texas Advanced Computing Center*

ABSTRACT: *A 3.7 teraflops Cray-Dell Linux cluster based on Intel Xeon processors will be installed at the Texas Advanced Computing Center early this summer. It will represent a transition from the T3E line of massively parallel processing systems that served researchers at The University of Texas. Code migration to an IA-32 microarchitecture will benefit from the adoption of new performance-enhancing architectural features within existing codes and algorithms. Intel's Hyper-Threading (HT) Technology is one of the significant performance improvements within the latest IA-32 microarchitecture. HT provides a low-level, simultaneous multithreading parallelism directly within the microarchitectural framework of the processor's core. It is most effective when it can provide additional work via simultaneous instruction execution of two threads per processor. The net effect is to offset stalls due to memory and I/O latencies. This paper will discuss the performance characteristics of HT for scientific applications.*

1. Introduction

In the last decade, the instruction-level parallelism (ILP) in microprocessor architectures has increased substantially, particularly in the form of pipelined and superscalar designs (multiple execution units that can operate simultaneously). Much deeper pipelines[1] in the IA-32 processors have been engineered to allow the processor speeds to reach 3GHz and beyond. While these architectural features boost performance, each new generation of processors must cope with any lag in relative memory speed and increased latency.

The new Intel Xeon and Pentium 4 processors have a hardware feature called Hyper-Threading Technology[2,3] (HT) that makes a single processor appear as two logical processors to the operating system and to user applications. This technology allows two processes or two threads (e.g., MPI tasks or OpenMP threads, respectively) to run simultaneously on a single processor; hence it is a form of simultaneous multi-threading (SMT) implemented in hardware. This means that instructions from each thread or task can execute concurrently in the core. Here, the term "threading" refers to the process of streaming instructions, and does not imply a restriction to execution of programs coded with threads (OpenMP, Multitasking, Pthreads) in the programming sense. Multiple streams of instructions from two different processes, MPI tasks, or OpenMP/PThreads can execute concurrently.

The motivation for incorporating HT in the chip architecture comes primarily from the business server market. In web and database servers, it is common for processors to spend 30 percent or more of their time idle, waiting for resources (especially memory and I/O). The impact of idle time can be offset by overlapping the execution of multiple threads. One hardware approach is to build two (or more) cores per chip to execute different threads, as in the IBM Power4 and future Intel Itanium architectures. HT is a different approach, designed to execute two independent instruction streams concurrently in a single core. In both dual-core and HT simultaneous-multithreading approaches, the memory components and paths from the L2 cache to the memory are shared. In dual-core chips, the L1 caches, the registers, buffers, and the instruction pipelines are duplicated. In HT, some register and buffers are replicated (state registers, queues, etc.), but the instruction pipeline and the execution units are shared. While this might seem prone to contention and congestion, there is a performance benefit when instructions from two different threads can execute simultaneously. This performance benefit is maximized when threads are using different instruction units, *e.g.*, when one thread is performing only integer arithmetic and the other only floating point operations.

For HPC systems that use the new IA-32 processors such as the TACC Cray-Dell cluster, HT capability is built into the processors and can be turned on or off. In the commodity server market a large variety of processes stall on random memory and I/O accesses; therefore, multiple

simultaneous threads on a single processor can provide large performance gains.

While standard benchmarks (SPECfp, Linpack, STREAM, etc.) are a reasonable metric for evaluating performance, benchmarking production applications should be in the strategy for platform evaluation. In addition, application developers should have an understanding of how the technology operates at the instruction-set level (*e.g.*, hyperthreading, multithreading, etc.) for formulating efficient algorithms.

Section 2 presents an outline of the architectural features of Hyper-Threading and an operational description of the technology. Section 3 describes computational kernels that can be used to characterize a memory subsystem, as well as measure memory characteristics that are important to Hyper-Threading (latencies and bandwidth sharing). The HT results of matrix-multiply operations and two simple scientific codes are reported in Section 4. In Section 5 we summarize the results, provide conclusions, and offer insights to application developers.

2. Hyper-Threading

Architecture

Microprocessor performance has improved over the years with branch prediction, out-of-order execution and superscalar (multiple and concurrently operated) execution units. Because the higher clock speeds of the newer processors require deeper pipelines, there are more instructions "in flight"; and cache misses, branch mispredictions, and interrupts become costly. The simultaneous multi-threading of Hyper-Threading allows two threads to execute concurrently in the processor without context switching (moving one instruction stream aside for the other). This is accomplished with a minimal amount of additional circuitry.

Hyper-Threading makes a single processor appear as two logical processors by having two separate architectural states, even though the same physical execution resources are shared. At the application level there are two processors that can be assigned to processes or threads. Figure 1 illustrates the state/core relationship.

The general-purpose registers, control registers, and the Advanced Programmable Interrupt Controller (APIC), as well as some machine state registers have been duplicated to form the two architectural states. The logical processors share the branch prediction, execution and cache trace units, control logic, buses and caches. The circuitry and logic "along" the instruction pipelines were designed to make sure one thread can progress forward even though another may be stalled. This "independent forward progress is guaranteed by making sure that neither thread can consume an inhibiting share of the buffering queue entries. Another design goal was to make sure that a single thread on an HT processor can run as fast as a thread on a processor without this capability.

The execution trace cache (TC) is where the decoded instructions (ops) are stored and retrieved, it has replaced the L1 instruction cache seen in many other microarchitectures. Its operation provides a good example of how the processor pipeline arbitrates between the threads. Two sets of independent pointers are used to point to the "next" instruction of each software thread. Every clock period the thread choice is arbitrated, and any simultaneous access is granted to one thread and followed by the other. If one logical processor is stalled, 100% accessibility is given to the other thread. The TC is 8-way set associative, uses a least recently used (LRU) replacement policy, and is shared by both threads. Each entry is tagged with a processor ID.

The Front-end of the pipeline consists of the ITLB (instruction table lookaside buffer), the prediction logic, streaming buffers, TC and microcode ROM. The duplicated and shared components are illustrated in Figure 2, as described by the Desktop Products Group at Intel Corp[4]. An instruction stream from each thread is guaranteed by reserving one request slot for each thread. The ITLB and their pointers are replicated; the global history array is shared, with each entry tagged with the processor ID. Streaming buffers, as well as the queue before the TC and the op queue after the TC, have been duplicated.

The out-of-order execution engine consists of the allocation, register renaming, scheduling pipe sections and execution units[4]. The partitioned (shared) and separate components are illustrated in Figure 4. The ops from each thread are alternately fetched from the op queue, and stalls on a thread when its resource limit has been met (e.g. store buffer entries, etc.). A single thread will have allocations request occurring every cycle. By imposing a limit (mostly half) on key buffers (126 re-order buffer entries, 128/128 integer and float physical registers an 48/24 load and store buffer entries) for each thread, fairness is preserved and deadlocks are prevented. There are two Register Alias Tables since each thread must keep track of its own state.

After allocation and renaming, the ops are queued into two duplicated buffers (one for memory loads and stored, the other for general operations). Next, five schedulers distribute the ops to the execution units. Up to 6 ops can be dispatched in one CP. The schedulers determine when the ops are ready to be executed; that is when the dependent input register operands are available. Each scheduler has a buffer queue, and there no distinction made between thread ownership of the ops. Deadlock is avoided by limiting the maximum number of queue entries for each

logical processor. The execution is also unaware of logical processor ownership.

The BIOS must be aware of HT, and provide the kernel with information about the additional (logical) processor account. Many vendors are shipping platforms with HT enabled by default; but it can be disabled easily at boot time (e.g. using the boot argument noht). HT enabled processors can be detected by observing the number of CPUs the kernel sees (*e.g.*, using "`cat /proc/cpuinfo`" or running `top` on cluster systems).

3. Measuring Memory Characteristics

Many scientific and engineering HPC applications are particularly sensitive to the memory subsystem. While this was less true years ago because traditional Cray supercomputers (Y-MP, C90, T90, etc.) provided extraordinary memory bandwidth to keep functional units busy, the gap between commodity processor speeds and memory speeds is large—and growing. Therefore, applications are increasingly sensitive to memory latency and bandwidth performance. In this section, we will explore the impact of HT on memory latency and bandwidth. While these are early results on a new technology, they provide insights on how HT will impact the performance of HPC applications (Section 5).

3.1 Memory Latency

Since HT was designed to allow one process to use the processor while another process is stalled (waiting for resources), in particular while waiting on memory or cache data, it is important to know the relative times of memory and cache latencies. The worst case of memory latency impacting code performance is when memory references are random. In an array (IA) containing N array indices (8-byte integers) randomly distributed, the timings of the following loop can be used to report the number of clock periods for single, uncorrelated cache and memory accesses:

```
I1 = IA(1)
DO I = 2,N
   I2 = IA(I1)
   I1 = I2
END DO
```

Figure 4 shows the measured latencies for array sizes between 1KB and 2MB on a Dell 2650 dual-processor server with 2.4 GHz Intel Xeon processors using dual channels to 200 MHz DDR memory. (This system is used for all measurements in this paper; the Cray system to be installed in July 2003 will have faster processors and memory.) Reads (Single CPU data in figure) from L1 (sizes up to 8KB) take approximately 3 clock periods (CP) while reads from L2 (sizes between 8KB and 512KB) take approximately 18 CP, as show in the figure insert. These single-processor latencies are characteristics of the processor since the L1 and L2 caches are on the chip die, and scale with the processor speed. The latencies for memory reads are mainly dependent on the components of the memory subsystem off the chip (northbridge bus speed, as well as the DIMM memory speed and its CLAS rating). Approximately 450 CP are required to fetch a data element from memory.

3.2 Memory Bandwidth

High sustainable memory bandwidth is a desirable capability of an HPC system. There are many different benchmarks, such as STREAM[5], that can characterize the memory performance. Here we use a simple memory-read loop that accesses two different floating point arrays. The memory-read kernel for our measurements consists of a single loop with the potential to support two streams through two independent arrays:

```
DO I = 1,N
   S = S + A(I)
   T = T + B(I)
END DO
```

A minimal but reasonable set of compiler options and loop structures were explored to derive an optimal bandwidth across the caches and memory. (High bandwidths are possible with more elaborate coding, but we wanted to use memory access structures and compiler options that would be found in common codes and makefiles, respectively.) In Figure 5 the memory-read bandwidth is plotted for transfers between 4KB and 2MB. The bandwidth from the L2 cache is constant at about 13 GB/sec (~0.7Word/CP) over about 70 percent of it range (size), and decays to memory speed beyond the "high end" of the cache size. The bandwidth to memory is fairly constant beyond 1MB, with an average value of 2.3 GB/sec (~0.1Word/CP).

3.3 Hyper-Threading and High Latency Memory Accesses

The greatest opportunity for multiple processes to overlap work in the execution units is when processes are stalling on memory requests (or I/O). The same memory-read measurements described above were used with 2, 3 and 4 threads on separate arrays (distributed model) to determine how Hyper-Threading behaves with a simple loop that has the highest degree of memory stalling. Figures 4 shows latencies for a single thread on each processor (Dual CPU curve). As expected, two threads accessing memory randomly exhibit no contention on separate CPUs in the cache region, and only 2% higher wait time from memory.

When three processes are executing on the system, two of the processes share one CPU (using HT) while the third is executed on the second processor. We will refer to these processes as LP0 and LP1 (on the first processor, using HT) and LP2 (on the second processor). Figure 6 shows latency curves for these 'logical processors.' The LP2 curve in the cache region is identical in form and values to those in

Figure 5; in the memory region there is a 26% increase in the latency. For the processor executing both LP0 and LP1 using HT, the latency in the memory region is about 33% larger than in Figure 4. The HT mode has extremely large latencies in the L1 region (see insert in Figure 1); in the L2 region the latencies are also higher, averaging about 25 CP compared to 18 for single-process mode, but for the benefit of two operations instead of one. Also, the effective size of the L2 cache has been reduced significantly, as shown by the vertical lines at 128KB and 384KB. Even though the cache is 512KB, the transition between pure L2 and memory access is not sharp. In HT mode additional data, as well as additional code, must be retained in the L2 cache.

Execution of four tasks (processes) on a dual-processor node is the simplest way to configure Hyper-Threading with MPI applications. Figure 7 shows HT memory latency for 4 threads on independent arrays. (The same behavior is expected in 4 MPI tasks executing the same loops.) The four threads display a common behavior. Latencies to memory are about 700 CP; L2 latencies are about 25CP, and L1 behavior is sporadic, ranging between values seen for HT and non-HT (one thread per processor) modes.

The asymptotic values of the memory accesses in Figures 5 and 7 show that increasing the process count from two to four, to double the "work" of random accesses, increase the time by only a factor of 1.5 (460CP per process for two processes compared to 700CP per logical process for four processes). Also, the work time on L2 cache elements only increases by a factor of 1.4 (18CP compared to ~25CP for 4 threads).

3.4 Hyper-Threading and High Bandwidth Memory Accesses

Random memory loads and stores provide a large opportunity for multiple logical processors to overlap additional memory accesses and operations in the execution units. On the other end of the memory bandwidth spectrum, highly optimized bandwidth-limited applications that stream data into the core usually leave no opportunity for any additional memory accesses. In the case of the (vector) sum used in the bandwidth measurement above, just a single operation (+) is employed on each double precision word fetched from memory. A 2.4GHz system, streaming in an array from memory with a bandwidth of 2.4 GB/sec, has about 7 cycles to perform other arithmetic or logical operations (discounting any pipeline latencies). Therefore, even in bandwidth-limited applications, simple loops may not be using the core units at full capacity. When two processes execute simultaneously on two processors in a bus-based dual-processor system, they must share the bandwidth to memory. [SMP systems such as the AMD Opteron (HyperTransport), SGI Altix (NUMA), and IBM Power4 (Distributed Switch) don't follow this rule.]

For multiple threads, separate arrays were used in the memory-read loop for each thread to mimic a distributed memory application. Figure 8 shows the effect of bandwidth sharing for two threads (tasks) executing on two processors. The cache region bandwidths are identical to those of the single-task executions in Figure 5. This is expected since each processor has its own cache. The memory access region shows a drop in performance of 50% (from approximately 2.2 to 1.1 GB/sec) per processor when two processors read simultaneously. This is also expected since the bus is shared; the total sustained memory bandwidth is essentially 'conserved.'

For 3 and 4 threads executing on the two physical processors (using HT), the aggregate memory access bandwidth drops by 10% (2.2 to 2.0 GB/sec) and 23% (2.2 to 1.7GB/sec) as shown in Figure 9 and Figure 10, compared to only two threads executing on the two processors (Figure 5). Both figures also reveal that cache access bandwidths per process have dropped, but the aggregate bandwidth of the four logical processes has increased to 14.8GB/sec from 12.8 for non-HT (one process per processor) execution, a 16% performance boost.

A comparison of cache regions in Figures 5 and 10 reveals that dual-process executions effectively reduce the cache available to each process. The onset of pure cache access has shifted from 240KB in non-HT mode compared to 160KB in HT mode.

If globally visible arrays are used in the memory-read loop (*i.e.*, shared memory model), the memory performance is quite different from the distributed model with separate arrays described above. The performance of this shared-memory sum in HT mode, on two processors, is shown in Figure 11. The bandwidth per process at the largest array size is 910 MB/sec. At first, this appears incorrect: the sum of the four measured bandwidths is 3.6 GB/s, which is greater than the theoretical peak bandwidth (3.2 GB/s). However, in a shared memory model executing in HT mode, a memory request that is satisfied by one task is available to the other task "free"; the data lines come at a cost of "two for one" if the tasks are synchronized. The synchronization must keep the accesses close enough, in time, so that the second process fulfils its request from cache. At 1.1 GB/sec, it only takes about 500 microseconds to fill up a 512KB cache. In our timing routines, a barrier at the beginning of each timed loop forces the loops to begin within 50 (often 10) microseconds and maintain "cache synchronization" throughout the execution of the loop. The insert in Figure 11 reveals that the effective cache size has been decreased, relative to non-HT execution, as expected. However, the effective cache size in HT mode execution of the shared-memory model is larger than for the distributed-memory model (Figure 10). The decrease in the effective per-process cache size can be attributed to the non-perfect synchronization and extra code (text) used in the 2nd task.

4. Hyper-Threading Performance

Hyper-Threading was developed by Intel to help improve the performance of applications. However, the primary impetus was for increased performance in commercial server applications, which usually execute serially and have very different characteristics compared to HPC applications. The results in the previous section can be used to develop insights into the performance of HPC applications, but we also present a few early results from simple scientific applications to bolster these insights.

4.1 Matrix Multiply

The BLAS library routines, instead of hand-coded loops, are often used in compute-intensive applications to perform basic linear algebra operations. The matrix-matrix multiply routine, DGEMM, is common in many matrix transformations. It is therefore important to measure the benefits of HT on this routine.

Some applications work on a single matrix operation among all the processors, using a shared-memory parallel versions of DGEMM. Figure 12 shows the DGEMM scaling for the Intel MKL 5.1 DGEMM routine throughout a range of matrix sizes, for one to four threads on our dual-processor node. For matrices of order 10 to 1000 the scaling is consistent. All the curves approach an asymptotic limit for large matrix order. The floating point performances for one and two processes are 3.2 Gflops and 5.7 Gflops, respectively, for order 1000. However, the three and four process curves only reach 4.7 Gflops. The scaling from one to two processes is 1.75; the scaling from one process to three or four processes is only 1.5.

4.2 Molecular Dynamics

Molecular dynamics type calculations are often used to model the evolution of proteins, DNA, and other macromolecular structures in a variety of chemical environments. To evaluate the effects of HT on this genre of compute-intensive calculations, the dynamics of a 256-particle argon lattice was calculated for one picosecond, using a Verlet algorithm to advance the atomic positions and velocities. The code was parallelized using OpenMP. Table 1 shows the times for 1-4 threads on our dual-processor system. Even with excellent scaling to two threads, HT provides an additional 12% benefit in scaled performance when four threads are deployed.

Threads:	**1**	**2**	**3 (HT)**	**4 (HT)**
Time (sec)	7.95	4.06	3.89	3.63
Scaling	1	1.96	2.04	2.19

Table 1: MD performance for serial, dual-processor, and HT execution.

4.3 Lithography

Lithographic modeling simulates the behavior of various stages of the lithographic process, such as the photoresist development, using a set of physical process parameters and a set of descriptive models. A modeling program[6] developed at The University of Texas uses Monte Carlo methods and is parallelized with OpenMP. After 10^7 Monte Carlo iterations, each thread outputs the lattice configuration and various Monte Carlo statistics to disk. HT improvements are reported in Table 2.

Threads:	**1**	**2**	**3 (HT)**	**4 (HT)**
Time (sec)	19.9	15.7	13.1	11.5
Scaling	1	1.27	1.52	1.73

Table 2: Monte Carlo lithography simulation times and scaling for serial, dual-processor, and HT execution.

The scaling from one to two processors is not as large as for the MD code. However, significant performance enhancement is achieved using HT—more than for the MD code. The incremental gains with each additional thread are nearly linear. Adding two HT threads provides a performance boost of 36% over the non-HT execution of two threads.

5. Conclusions and Insights

This work represents very early results, since the final system has not been installed by Cray and even the test and evaluation nodes have not been available and operational for long. However, the simple latency and bandwidth measurements conducted can reveal the parameter space for evaluating the performance enhancement of HT on HPC codes. As memory latencies increase, the potential benefit of HT increases as well. In Section 3 we see that applications characterized by random memory access (*e.g.*, employ gather/scatter operations) can overlap their latencies and have the potential for up to 40% increase in performance with HT if execution units are not oversubscribed. This is not unexpected, when considering that HT was designed to benefit server workloads of web or database applications consisting of many random memory references. This technology can also have a positive impact in HPC codes that solve sparse linear algebra problems.

However, HPC applications that use a distributed memory model and also stream data from memory can experience a significant degradation in performance with HT. The results of Section 3 show a degradation up to approximately 25% in memory bandwidth according to our distributed memory bandwidth measurements. This is disturbing, since many HPC applications stream data from memory and most HPC applications—especially those likely to run on Xeon processors—are coded using MPI (or some other distributed programming technique).

The results in Section 3 relevant to a *shared* memory model imply positive impact, however. If the memory accesses can be shared in cache by using shared-memory programming (*e.g.*, OpenMP), there can be a benefit of as much as 60% in memory read performance, especially for applications that need to share a sizeable fraction of the data while also accessing disparate sections of memory. One should keep in mind the additional threads made possible using HT must share the cache, so the effective cache size per thread is decreased. Nevertheless, memory-bound algorithms that use shared data can benefit from the additional threads of HT, provided the sharing is synchronized to access the data by the alternate thread once it is in cache and the code is not heavily cache-optimized already.

The preliminary application results demonstrate a range of possible performance impacts. The highly optimized DGEMM matrix multiply routine degrades by almost 20% for large matrices when one or two additional HT threads are applied. This routine is coded for shared memory, but has been extensively cache-optimized. It is likely that the reduction in effective cache offsets the benefit of using shared data. The MD code, which was coded with OpenMP but in a style which mimics MPI-type parallelism, sees about 12% performance improvements due to HT. This is most likely due to sharing of data again, but now the code is not heavily cache-optimized. Finally, the lithography code sees significant performance enhancement—73%--with HT. This code benefits quite well from HT because it has many random memory references and performs a significant amount of I/O. I/O stalls can be more severe than memory stalls; though we did not explore the impact of HT on I/O intensive codes, it is not surprising that the 60% potential improvement for random memory accesses would be exceeded if there were also significant I/O operations.

In general, compute- or memory-bound applications may not benefit from HT; however, applications that have memory or I/O stalls, and are not extreme in the use of cache/execution units or in memory streaming, can benefit from the Hyper-Threading Technology. We look forward to exploring this further when the large Cray-Dell cluster is installed in July 2003. Future work will include more detailed analysis of HT impact on scientific algorithms and the benefits to I/O-bound applications.

Acknowledgments

The authors would like to thank their colleagues at TACC and in the HPCC Group at Dell for sharing their insights, expertise, and wisdom on Hyper-Threading[7].

References

1. *Increasing Processor Performance, by Implementing Deeper Pipelines*, Eric Sprangle & Doug Carmean, White Paper, Pentium Processor Architecture Group, Intel Corporation (IEEE 2002)
2. http://www.intel.com/technology/hyperthread/index.htm
3. *Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance*, Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, Hideki Saito, Ernesto Su, White Paper, http://developer.intel.com/
4. *.Hyper-ThreadingTechnology Architecture and Microarchitecture*, Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, Michael Upton, White Paper, http://developer.intel.com/
5. *STREAM* benchmarks, www.cs.virginia.edu/stream
6. http://willson.cm.utexas.edu/Research/Sub_Files/Resist_Modeling/collaborations.htm
7. *A Study of Hyper-Threading in High-Performance Computing Clusters*, Tau Long, Rizwan Ali, Jenwei Hsieh, and Christopher Stanton. – www.dell.com/powersolutions

About the Authors

Drs. Kent Milfeld, Chona Guiang, and Avijit Purkayastha are staff members in the HPC Group at TACC. Dr. John ("Jay") R. Boisseau is the manager of the HPC group and the director of TACC. The authors have many years of experience in HPC, in particular using Cray supercomputing systems. All of the authors can be reached at: Texas Advanced Computing Center, CMS 1.154 / PRC (R8700), 10100 Burnet Rd., Austin TX 78758-4497 U.S.A.; or U.S. phone number (512) 475-9411. Dr. Milfeld can be reached by e-mail at milfeld@tacc.utexas.edu.