

Hyper-Threading on the Pentium 4

Vimal K Reddy Ambarish M Sule Aravindh V Anantaraman
{vkreddy,amsule,avananta@unity.ncsu.edu}

December 2002

Abstract

Simultaneous Multi-Threading (SMT) increases processor utilization by fetching and executing instructions from multiple instruction streams, thereby utilizing parallelism present among the multiple streams (also known as thread level parallelism - TLP).

Hyper-Threading is Intel's first version of SMT that has been implemented on the Pentium 4, marking the beginning of commercially available SMT processors. Hyper-Threading poses new challenges to how applications are developed and run. A good workload mix can reap the benefits of a multi-threaded processor. On the contrary a contending workload mix can hinder performance. Also, future versions of Hyper-Threading would aim to increase performance by reducing existing bottlenecks in the micro-architecture and judicious addition of hardware.

In this paper we study the performance aspects of Hyper-Threading in three ways. First, we model key parameters in the Hyper-Threading micro-architecture design space in a simulator. By varying parameters and observing their effects on performance, we explore future design choices for Hyper-Threading that could increase performance.

Second, we analyze performance of different workloads on an actual Hyper-Threading enabled Pentium4 machine. The workloads include hand-coded kernels (microbenchmarks) targeted to utilize particular resources in the pipeline and benchmark programs chosen from BYTE's BYTEmark benchmark. By choosing different "contending" and "complementary" workloads, we analyze resource utilization and suggest workload mixes that could enhance or hinder performance.

Finally, we explore the possibility of software pre-execution on a Hyper-Threading enabled Pentium4 machine. We try to increase performance of memory intensive single-threaded programs by pre-fetching using helper threads. Our results show the difficulties in using such an approach and highlight the need for multi-threading tools for application development.

1. Introduction

Modern day computing has necessitated the growth of high performance processors. Internet and e-Business applications are constantly demanding faster response times. Dynamically scheduled wide superscalar processors have become the order of the day and have helped to quench this need for performance to some extent.

Dynamically scheduled wide superscalar processors depend on two important factors to achieve high performance:

- processor resources : availability of resources ensures that instructions that can be executed in parallel are not stalled due to lack of resources.
- instruction level parallelism : large number of independent instructions allows more work to be done in parallel.

To ensure high performance, it is essential that sufficient processor resources are available and there is enough parallelism in the program to completely utilize the hardware resources available.

Advances in VLSI techniques have enabled hundreds of millions of transistors to be integrated on a single chip. Hence there is an abundance of resources and resource constraints are no longer a bottleneck to improving performance.

Superscalar processors extract instruction level parallelism (ILP) from a single instruction stream to achieve high performance. Modern day compilers do a good job of exposing instruction independence in a program. However there is a limitation to the amount of instruction level parallelism present in most programs as shown by Jouppi and Wall [11]. Available processor resources are not being utilized because of limited ILP. Also there are regions of code where there is abundance of ILP and other regions where ILP is scarce. This bursty nature of ILP in programs does not fully utilize the available resources leading to horizontal and vertical waste.

A possible solution to removing ILP bottlenecks is increasing the granularity of instruction level parallelism to the next level and harvesting the ILP available by running many threads together instead of one single thread. This coarser form of ILP among threads is called thread level parallelism (TLP).

Dean Tullsen et al suggested running multiple threads simultaneously on the same processor [10] by replicating and sharing processor resources among the multiple threads. This innovative approach, known as simultaneous multi-threading (SMT), exposes more parallelism to the processor by fetching and retiring instructions from multiple instruction streams, increasing resource utilization and overall system performance. SMT requires modest hardware additions to a base out-of-order superscalar architecture compared to replication of the entire core in a Chip Multiprocessor (CMP). The economics coupled with its performance benefits make SMT the choice architecture in future micro-processors.

Hyper-Threading is Intel's first version of SMT. It increases processor utilization by fetching and executing instructions from two instruction streams when available. Intel has already made Pentium4 processors with Hyper-Threading features commercially available. With two simultaneously executing contexts, Hyper-Threading brings new capabilities that developers can leverage to increase application performance. However, Hyper-Threading is sensitive to workload mixes and poses new challenges to how applications are scheduled. "Complementary" workloads that utilize unused resources enhance performance whereas "Contending" workloads that contend for shared processor resources can hinder performance.

With the first implementation of SMT already out on the market, more versions of SMT can be expected from Intel in the future based on the current Hyper-Threading technology. We take a step forward in that direction and study the existing Hyper-Threading technology to identify where the future work lies. We also explore how Hyper-Threading could be leveraged to enhance system performance by characterizing workloads that run well together and those that don't. Finally, we explore the possibility of software pre-execution on a Hyper-Threading enabled Pentium4 machine.

To summarize, this paper makes the following contributions:

- *Simulator based study of Intel Hyper-Threading*
Section 3 analyzes the effects of varying key micro-architectural components in the design space of Intel Hyper-Threading technology enabled architecture. A SimpleScalar [5] based timing simulator modified for Hyper-Threading is used for experiments.

- *Study of benchmark behavior on Hyper-Threading enabled Pentium4 machine*
Section 4 describes our benchmark runs on a Pentium4 machine with Hyper-Threading feature. We use small hand-coded kernels (microbenchmarks written in C++) and BYTEmark benchmark programs (previously known as BYTE's Native Mode Benchmarks) in our studies. The kernels are targeted on stressing particular processor resources. By creating various job mixes from the kernels, we study the performance of Hyper-Threading under different workloads. [We then use actual benchmarks provided by BYTEmark benchmark suite to validate our findings.]For instance, we found running 2 memory-intensive kernels that stress the shared data cache leads to data thrashing. [We find similar performance degradation when running the String sort benchmarks that heavily utilize memory.]
- *Speculative pre-execution on Hyper-Threading enabled Pentium4 machine*
Section 5 gives an overview of Speculative Pre-execution or Pre-computation and its possible implementation on the Pentium4. It illustrates the advantages of SP for single threaded memory-intensive programs and difficulties in its implementation.

2. Related Work

Significant amount of research has been done in the field of simultaneous multithreading. Simultaneous multithreading was initially proposed by Dean Tullsen et al [10]. Some parts of our current research directly follow their work. The different fetch policies suggested by them [4] have been used in this paper. The Intel Technology Journal[7] has a wealth of published research on Hyper-Threading. We have closely followed Intel's implementation of the Hyper-threading [1] in our research.

Performance of Hyper-Threading has been tested for multi-threaded kernels and benchmarks [12]. We investigate Hyper-Threading performance when single threaded kernels and benchmarks are co-scheduled. We think many software applications are still single threaded and form most of the workloads in modern day computing.

Related studies in using speculative threads to prefetch data from caches fall into hardware and software methodologies. Slipstream Processors by Sundarmoorthy et al. [15] run redundant copies of the same program, one of which is a reduced version, on duplicate hardware to achieve speedup in single threaded programs. Software based approaches include Zilles and Sohi's Speculative Slices [16] and Luk's Software Controlled Pre-Execution [17]. We attempt to utilize idle hardware thread contexts in Hyper-Threading by software pre-fetching and follow an approach similar to that used by Wang et al. [3].

3. Simulator based study of Intel Hyper-Threading

We analyze the effects of varying key parameters in the design space of Pentium4 Hyper-Threading enabled architecture. Section 3.1 gives a brief introduction to the Pentium4 micro-architecture with Hyper-Threading technology. Section 3.2 describes our methodology of modeling Hyper-Threading in a simulation based environment. Section 3.3 describes our experiments and results.

3.1 Hyper Threading

Hyper threading in the Pentium 4 is limited to two simultaneous threads. Hyper threading provides for two logical processors on one single processor. This is achieved by replicating, partitioning and sharing of resources of the single processor among the two threads to provide the illusion of two logical processors, one per thread.

3.1.1 Replicated resources:

Replicated resources, as the name suggests, a copy of these resources is available per thread. Replicated resources include:

- architectural states of the logical registers
- program counters(a.k.a instruction pointers)
- renaming logic
- return address stack
- instruction translation lookaside buffer (ITLB)

3.1.2 Partitioned resources

Partitioned resources mean that these resources are divided equally among the threads irrespective of need. Partitioned resources include:

- re-order buffer
- load/store buffer
- arithmetic issue queue
- memory issue queue

3.1.2 Shared resources

The two threads use the shared resources as per their need. The shared resources include:

- out-of-order core
- instruction and data caches
- fetch unit

When running only one thread, all resources are freed and are available at the disposal of a single logical processor, ensuring that single thread performance is not degraded. The other logical processor is said to be in idle mode (and hence does not consume any of the resources).

3.2 Simulator Description

3.2.1 ISA

The simulator is based on the SimpleScalar ISA and not on the x86 ISA of the Pentium 4. However since the emphasis of this research is to study first order effects of varying key parameters/policies, it is strongly felt that first order effects would manifest in the same way in both the simulator modeling hyperthreading and the actual pentium 4 machine.

Table 2 indicates the micro-architectural features of the Pentium4 that have been modeled in the simulator.

Table 1: Comparison of features of Pentium4 and simulator

	Features/Resources	Pentium 4	Simulator
	x86 ISA	yes	no
	20 stage misprediction pipeline	yes	yes
	Bandwidth (fetch, dispatch, issue, mem and retire)	yes	yes
Replicated	Architectural state (per CPU)	yes	yes
	program counters	yes	yes
	RAS	yes	yes
	Trace cache	yes	no
Partitioned	Mem IQ and ALU IQ	yes	yes
	Load/store buffers	yes	yes
	Common fetch unit	yes	yes
	Common instruction cache	yes	yes
Shared	Common data cache	yes	yes
	Double speed ALUs/Functional Units	yes	no
	L2 cache	yes	no

3.2.1 Description of hardware resources

The simulator used is a detailed cycle accurate one. The simulator is based on the SimpleScalar ISA [5] and hence it does not model the x86 ISA of the Pentium 4. The base simulator models a dynamically scheduled superscalar processor with a 126 entry re-order buffer, two issue queues totaling 126 entries, a 48 entry load buffer and a 24 entry store buffer. The superscalar bandwidths are specified in Table 1. It is seen that the issue width is twice the other widths to help extract parallelism when bursts of ILP occur.

There are 22 stages in the pipeline to mimic the Pentium 4 pipeline. In the simulator useful work is done in only seven stages. These stages are fetch, dispatch, issue, register read, execute, writeback and retire. The other stages are dummy stages that have been introduced in the pipeline flow to simulate the 20 stage misprediction pipeline of the pentium 4. The fetch and dispatch stage constitute the frontend pipeline and the remaining stages form the back-end pipeline. We now look at how hyperthreading has been modeled in the simulator in terms of hardware resources.

In the frontend pipeline, there is a common fetch unit shared among the threads. The different threads are distinguished using a unique thread identifier (Tid). The global fetch unit has one instruction cache (I- cache) common to all the threads. The trace cache used in the Pentium 4 has not been implemented in the simulator since it is beyond the scope of this project. The L2 cache,

common to the I-cache and the D-cache, used in the Pentium 4 has again not been modeled due to time constraints.

Every cycle, instructions are fetched from either thread based on the fetch policy selected. The different fetch policies that have been implemented are:

- Round-robin - 2, X,X where X stands for the superscalar fetch width. This means that every cycle a total of X instructions can be fetched from two threads. The priority among the threads alternates every cycle.
- ICOUNT - the thread with the least number of instructions in the fetch, dispatch and issue stages of the pipeline is given priority.
- BRCOUNT - the thread with the least number of unresolved branches is given priority.
- MISSCOUNT - the thread with the least number of outstanding cache misses is given priority.

While running multiple threads, the fetch policy is modeled such that it does not fetch from threads that have completed.

Name of pipeline stage	bandwidth	latency
Fetch	3	4
Dispatch	3	4
Issue	6	1
Execute	5 + 2	variable
Mem1	3	variable
Retire	3	1

Table 2: Important pipeline stages, bandwidth and latencies

Branch prediction is performed using a 2 level GShare predictor and a 2K-entry branch target buffer. The branch predictor table and the branch target buffer are shared among the threads. However the branch history register and the CTI queue have been replicated.

At dispatch stage, a global limit on the total number of physical registers used by the threads is enforced. The total number of free checkpoints is also checked to see that enough checkpoints are available. No modifications have been done to the renamer class and each thread has its own renamer. A new function that returns the number of physical registers and number of checkpoints used by that renamer has been implemented. Thus no changes have to be made to the original implementation of exception and branch misprediction handling since the renamers are independent of each other.

A global limit on the total number of load-store queue entries, used by all the threads, is also enforced. In this simulator each thread has its own active list (reorder buffer ROB). There are two issue queues that are modeled in the simulator, similar to the implementation in the Pentium 4. One of the issue queues services all ALU operations and the other issue queue services all memory operations.

The issue stage and the register read stage are shared by all the threads. Note that each thread accesses its own renamer in the register read stage.

In the agen stage, there are two address generation functional units, one for loads and the other for stores. These are shared by all the threads.

In the execute stage, there are seven functional units. Five of these functional units are integer and two of them are floating point. One of the five integer functional units is slow and handles only shift/rotate operations. In the actual Pentium 4, there are two double speed ALUs and one slow ALU. The two double speed ALUs are mimicked in the simulator by using 4 normal speed ALUs. This implementation has been chosen over others because of time considerations.

The memory interface has a load and a store buffer that have been partitioned equally among the two threads. The load buffer can hold upto 48 instructions (24 entries per thread) and the store buffer is limited to 24 instructions (12 entries per thread).

3.4 Experiments and results

In this section, the experiments performed on the simulator to validate our results and to explore other design choices for Hyper-Threading are described. We discuss the behavior observed in the results. The following benchmarks from the SPECInt95 suite were chosen for our experiments:

- GCC
- PERL
- GO
- COMPRESS (COMP)

3.4.1 SMT validation

Experiments were run on the simulator to validate the SMT implementation. Separate runs with one, two, three and four threads of GCC were performed. The results are plotted in figure (). Normal SMT refers to a superscalar SMT processor that does not model Hyper-Threading. Hyper-Threading refers to a HT enabled simulator with architectural parameters close to the Pentium4. It is seen that the results for both cases conform to the results published by Tullsen et al. [4]. IPC increases with the number of threads but diminishing returns sets in as the number of threads increase and then the IPC saturates.

It is also noticed that the IPC of the HT enabled simulator is less than the normal SMT. This behaviour can be caused due to a variety of factors. The fetch, dispatch, mem and retire widths of the HT model are 3 compared to 4 of the normal SMT. The fetch and dispatch latency of the HT enabled processor are 4 each compared to 1 each in the normal SMT. Thus the HT enabled simulator has 6 extra pipeline stages (3 in fetch and 3 in dispatch). This can cause a big penalty in terms of branch mispredictions. Also the HT enabled simulator has a restricted number of functional units (5 integer, 2 floating point and 2 agen functional units) compared to normal SMT.

The IPC varies from 2.28 for 1 thread to almost 3.5 for 4 threads in normal SMT and from 1.3 for 1 thread to 2.21 for 4 threads in Hyper-Threading. Note that we also study the effects of having a logical processor for every thread. This is an extension of the existing Hyper-Threading technology to multiple threads without increasing the shared resources. An increase in IPC on increasing the number of threads shows that some of the resources remain under-utilized in a HT enabled processor running two threads.

3.4.2 Varying Fetch policies

The next set of experiments involve studying the effects of different fetch policies in a HT enabled simulator. The different fetch policies are: Round-Robin, ICOUNT, BRCOUNT and MISSCOUNT. Previous research has studied the effects of these policies on a normal SMT processor[1]. We attempt to extend these results for a HT enabled processor.

Figure 2 summarizes the results for different fetch policies (RR for round-robin, ICNT for ICOUNT, BCNT for BRCOUNT and MCNT for MISSCOUNT). Since Hyper-Threading is limited to two threads, we limit our experiments to running only two threads. The X axis in Figure 2 shows the different benchmark pairs. It can be seen that for two of the benchmark pairs (GCC-Go and COMP-GCC), the results conform to previously published work [1]. However, for GCC-PERL and GO-PERL, MISSCOUNT and BRCOUNT seem to be performing better than ICOUNT. We attribute this behavior to the highly parallel nature of the PERL benchmark.

We observe that the different fetch policies do not greatly affect the performance in a Hyper-Threaded processor. However, since our simulator does not model HT in all respects, we cannot conclude that fetch policies do not affect performance. This is left as future work. Future work also involves running more benchmark pairs and observing their behavior.

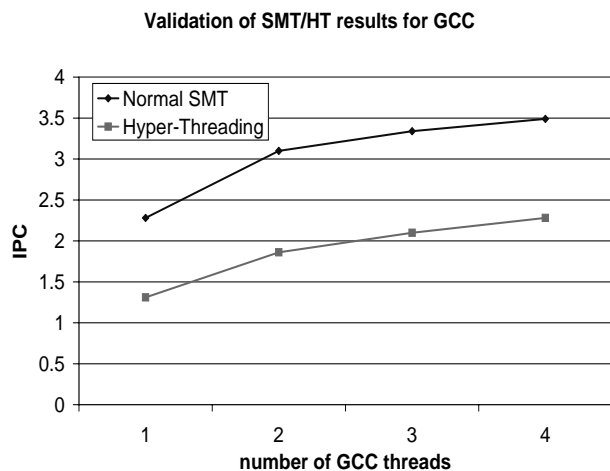


Figure 1

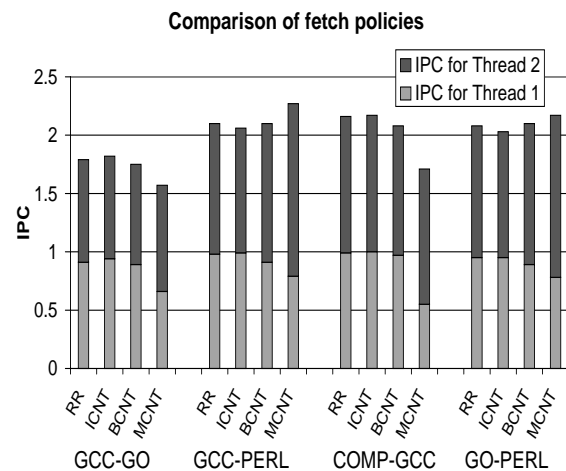


Figure 2

3.4.4 Varying IQ sizes

The HT enabled Pentium 4 has two issue queues: one for memory instructions (which we will refer to as MEM IQ) and one for non-memory operations (called ALU IQ). We now study the effects of varying the relative size of the individual queues and of varying the total number of queue entries itself.

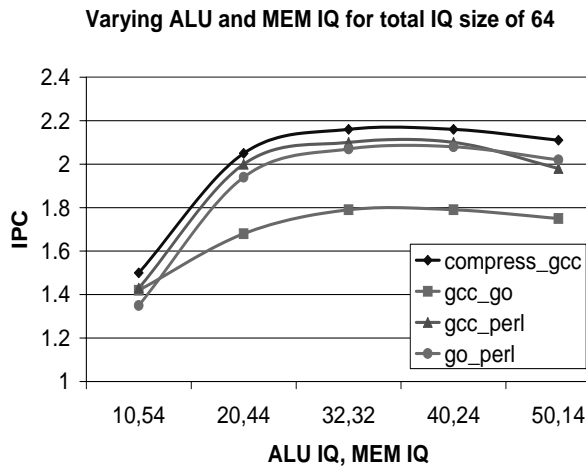


Figure 3

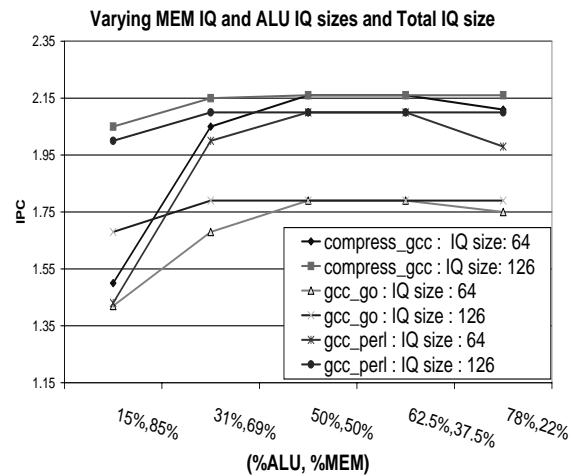


Figure 4

Figure 3 shows the results of increasing the relative size of the ALU IQ and the MEM IQ. The sum of the MEM IQ and the ALU IQ entries is restricted to 64. The X axis shows the different IQ size pairs, the first number indicating the number of ALU IQ entries and the second number indicating the number of MEM IQ entries.

It is seen that performance improves as the relative sizes become equal. The worst performance is seen for an ALU IQ size of 10 and MEM IQ size of 54. It is also seen that performance is affected when the size of the MEM IQ is 14 and the ALU IQ is 50. However since the ratio of the number of memory instructions to the total number of instructions is less (overall and at any given point of time in the pipeline), the small size of the MEM IQ does not drastically affect performance as is the case with the small ALU IQ. It is also seen that IPC remains constant for relatively equal MEM IQ and ALU IQ sizes. This indicates that issue queues are no longer a bottleneck to performance at that stage and performance is limited by some other factor.

The next set of experiments involves studying the effect of the total number of issue queue entries on performance. Figure 4 summarizes the results of these experiments. The total number of issue queue entries have been kept at 126 and 64. The X axis represents the ratios of the MEM IQ and ALU IQ to the total number of issue queue entries. It is seen that curves for both the runs follow similar trends. It is seen that performance increases by 50% for a total issue queue size of 126 entries in the end case (ALU IQ being small compared to MEM IQ).

Another interesting observation is that for relatively equal sizes of the MEM IQ and ALU IQ, the performance of both the models (126 total IQ entries and 64 total IQ entries) match. Thus increasing the total number of issue queues for relatively equal sizes of the MEM IQ and ALU IQ does not result in an increase in performance.

3.4.5 Varying LSQ sizes

The Pentium 4 has separate load and store buffers that are partitioned among the logical processors. In the Pentium 4, the load buffer has 48 entries (24 per logical processor) and the store buffer has 24 entries (12 per processor). We attempt to study the effects of varying the sizes of the load and store buffers.

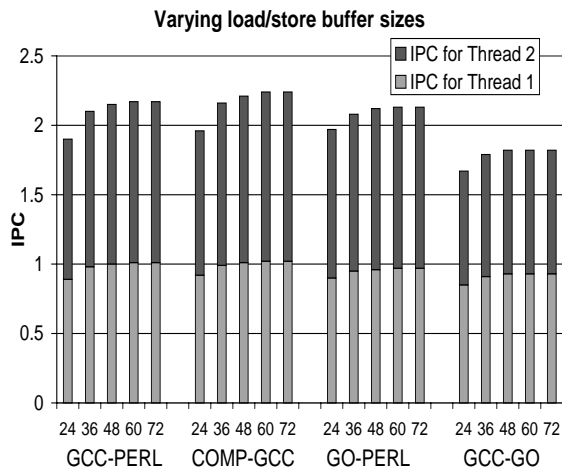


Figure 5

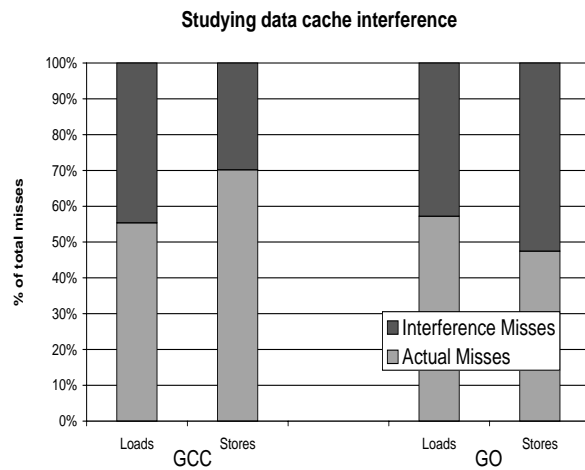


Figure 6

Figure 5 shows the results for various total load/store buffer sizes per logical processor. The load buffer constitutes two-thirds of the total size and the store buffer constitutes one-third of the total. The reason for using total number of entries and not separate numbers for load and store buffers is implementation specific to the simulator.

It is seen that as the size of the load and store buffers are increased, performance increases. But diminishing returns are seen even though the size of the load and store buffers are increased after a certain point. We conclude that the current size of 24 for the load buffer and 12 for the store buffer is sufficient.

3.4.2 Data cache interference

SMT processors have shared instruction and data caches. In this paper, we take a look at the penalty of using a shared data cache. The cache modeled in our simulator is 64KB, 4 way set associative with a line size of 64 bytes. The cache used in the Pentium4 has a similar configuration.

We classify all misses into two categories

- Actual misses: these are misses that would have been incurred even if the processor had a dedicated cache (e.g: cold misses)
- Interference misses: these are misses that are caused because of one thread evicting the data of another thread from the cache.

We measure interference by maintaining separate dedicated caches and a miss is counted as an interference miss iff it misses in the shared cache but hits in the dedicated cache. Figure 6 shows the ratio of actual misses and interference misses for two threads (GCC and GO) when run with Hyper-Threading. Loads and stores are shown separately. It is seen that interference misses contribute from 28% upto 52 % of total misses. A point to be noted is that we have not modeled an L2 cache that is a part of the Pentium 4. Partitioning the cache is an option that could be investigated. The other option is having dedicated caches per logical processor. However power considerations would come into play. Future work involves integrating the L2 cache into the simulator.

4. Study of benchmark behaviour on Hyper-Threading enabled Pentium4 machine

Hyper-Threading provides the capability of co-executing two programs by keeping separate hardware contexts for each program or separate logical processors as Intel refers to them. The operating system sees the two logical processors as two separate processors and schedules jobs accordingly. Multithreaded applications would readily benefit from this as shown by Magro et al. [1]. In this section, we investigate the performance of Hyper-Threading when single threaded programs are co-scheduled on an actual Pentium4 machine with Hyper-Threading. Section 4.1 describes the microbenchmarks and benchmarks we used. Section 4.2 discusses our experiments and results.

4.1 Microbenchmarks and Benchmarks

The microbenchmarks we used were hand-coded kernels (microbenchmarks in C++) written to target particular processor resources. The kernels were compiled with g++ version 3.2 without any optimizations (-O0). We used four kernels:

Floating point intensive (float_math) - It performs floating point add, subtract, multiply and divide operations a large number of times. The kernel targets the single FP execution unit and utilizes it heavily.

Integer intensive (integer_math) - It performs integer add, subtract and shift operations a large number of times. The kernel has very few branch mispredictions. The kernel targets and utilizes the integer units (2 double speed and 1 slow integer unit) heavily.

Memory intensive small (mem_intense_small) - It dynamically allocates a large linked list that does not fit in the cache. Then it parses through the linked list till the end. Each linked list node is separately allocated with C++ new operator. This makes sure they are not contiguous in memory and more page faults occur when the linked list nodes are accessed. The kernel targets the entire memory hierarchy as such. But we are interested in the shared data cache which is repeatedly accessed, especially when running two memory intensive kernels.

Memory intensive big (mem_intense_big) - It is same as mem_intense_small except that it runs for a longer time. It does so by parsing the linked list multiple times, stressing the data cache more.

The mem_intense_small kernel saved us time when co-scheduled with another memory-intensive kernel (mem_intense_small itself) while getting the results we wanted.

[Describe the actual benchmarks used here and write what we are using in the Benchmark suite to compare our results.]

4.2 Experiments and results

Our programs were run on a Pentium4 “Northwood” 2.53-2.66 Ghz. machine with Hyper-Threading technology. We used BIOS settings to enable and disable Hyper-Threading. The operating system was Linux running RedHat's SMP kernel version 2.4.18-17.8.0smp. We timed our runs using the operating system provided timer. We used PERL scripts to fork different processes at the same time. We used the Linux “top” utility to monitor processor and memory utilization by the programs.

In our results we use the following notation to indicate the kernels:

- floating point benchmark – flt
- integer benchmark – int
- memory intensive small – mem_s
- memory intensive big - mem

We indicate combinations of kernels by appending the notations. For instance, a job mix of 2 floating point kernels is indicated by “fltflt”.

As our first experiment we analyzed single thread performance with Hyper-Threading (HT). We ran each of the kernels with HT on and HT off. Figure 7 shows the execution times of the kernels. The execution times are almost the same with and without HT. This validates that Hyper-Threading does not degrade single thread performance.

Next, we create different job mixes or workloads comprising of the kernels. These workloads are run in two scenarios: with “HT off” and “HT on”. In each case, the programs in a job mix are spawned as separate processes. The execution time of a job mix with “HT off” and “HT on” is the time from start of the run up to the finish of all kernels in the job mix. The “back-to-back” execution time of a job mix is the sum of single thread run times of all kernels in a job mix. A thing to be noted about running more than two jobs with “HT on” (more than the number of hardware contexts) is the role played by the operating system scheduler. The operating system we used scheduled one of the multiple jobs to an available logical processor. It shared the other logical processor by context switching between the other jobs.

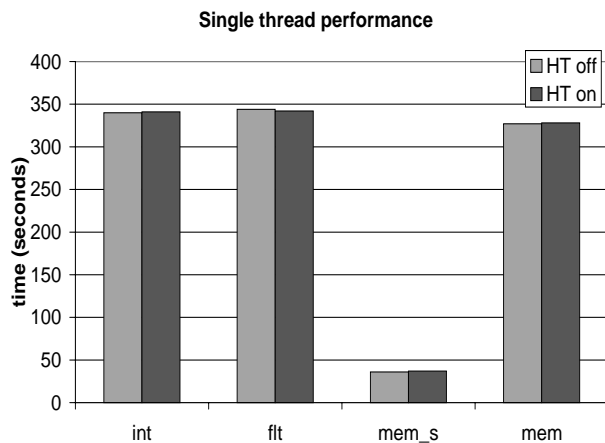


Figure 7

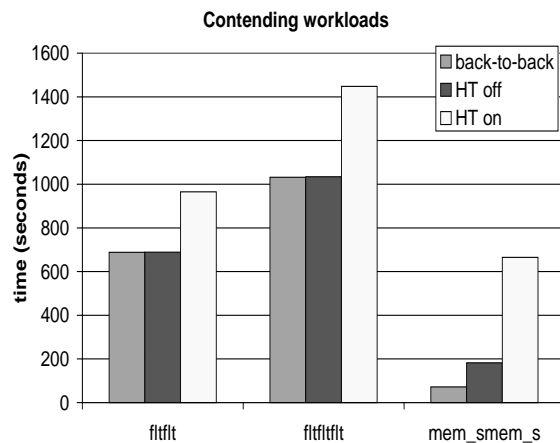


Figure 8

We characterize our workloads into two categories based on their behavior when Hyper-Threading is on: Contending workloads and Complementary workloads. Contending workloads comprise of those kernels that would compete for processor resources and hence would hinder each other's progress and degrade performance. Complementary workloads comprise of those kernels that would utilize unused processor resources and increase performance.

The execution times for the different job mixes are shown in the graphs in Figures 2 and 3. Based on whether a performance gain is achieved with “HT on” over “HT off” we separate the results into “Contending” and “Complementary” workloads respectively.

Figure 8 shows the results with contending workloads. Job mixes comprising only of floating point kernels (fltflt and fltfltflt) show a 40% increase in execution times with “HT on” over “back-

to-back.” The execution times almost remain the same when run with “HT off.” In fltflt job mix, instructions from both kernels compete for the FP execution unit and hence are stalled. This in turn clogs the pipeline and degrades performance. Similar trend is found in the fltfltflt job mix.

The mem_smem_s job mix performs worst with Hyper-Threading on. It increases execution time from 72 secs when the kernels are run “back-to-back” to 665 secs with “HT on.” This is because both the kernels have huge memory foot prints and they contend for the same data cache with “HT on.” Hence they end up thrashing each other from the cache. It should, however, be noted that the kernels we developed are synthetic and made to stress the memory hierarchy. But it still highlights the effect of data cache interference in a Hyper-Threaded processor and represents an extreme case of performance degradation.

In contrast the execution time for “HT off” is 182 secs. It shows that it is better for each process to have full ownership of the data cache and be context switched by the operating system than being run simultaneously. The increase in execution time with “HT off” is attributed to context switching between the two processes.

Figure 9 shows the results with complementary workloads. All the workloads, excepting one, show consistent decrease in execution times. The intint workload provides a 40% improvement in execution time with “HT on” over running it “back-to-back.” Though integer instructions from the two kernels have similar resource requirement, presence of 3 integer units (2 of them clocking at twice the processor speed), leads to increased resource utilization instead of contention for units. In fact, the intintint workload also provides 40% increase in performance over “back-to-back” runs. This shows Hyper-Threading performs really well for integer workloads.

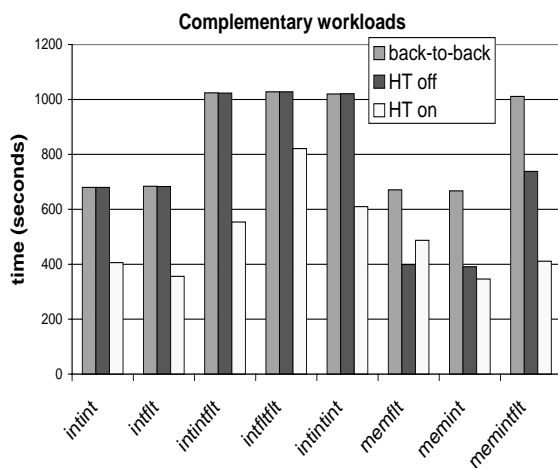


Figure 9

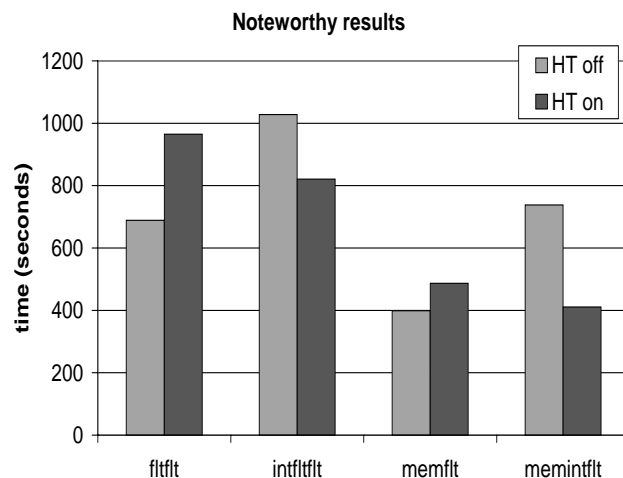


Figure 10

The intflt workload increases resource utilization by keeping both the FP and Integer functional units busy. It is a case where instructions from both kernels have complementary requirements and don't clog the pipeline waiting long times for functional units. This is noticed by execution time gains in the range of 45-50% while running the intflt and intintflt job mixes.

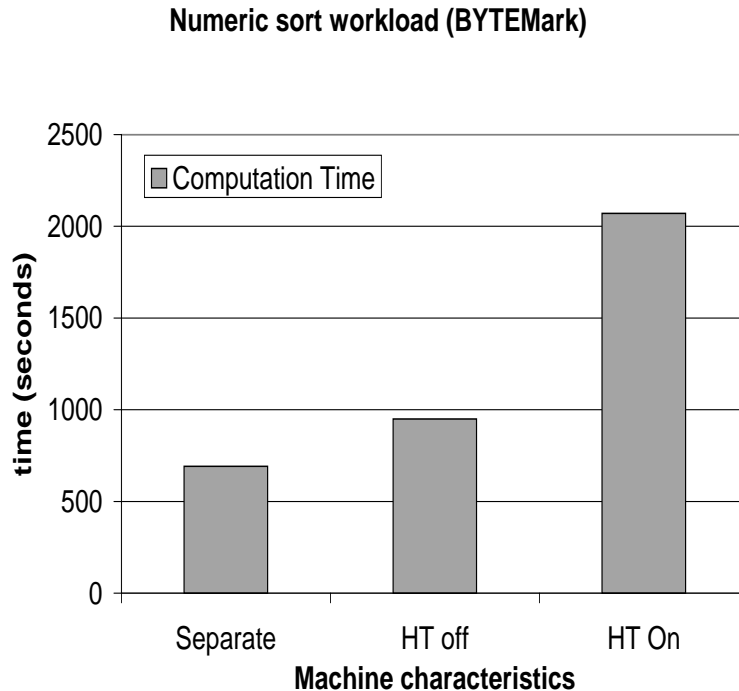
The combination of two floating point kernels with an integer kernel in the intfltflt job mix produces a noteworthy result, as also shown in Figure 4. The job mix produces an instruction stream with instructions contending for the FP execution unit and the integer units. The operating system plays a crucial role here by selecting a “symbiotic” workload as defined by Snavley et al

[2]. It assigns one logical processor to the floating point kernel and shares the other logical processor between the integer and the second floating point kernel. This increases performance in two ways. First, contention is reduced when the integer and floating point kernels are co-scheduled. The floating point kernel makes more progress than when it is co-scheduled with the second floating point kernel. Second, this leads to faster completion of the floating point kernel that completely owns one of the logical processors. This frees up resources for the second floating point kernel leading to its faster completion. In fact the integer kernel finished at about the same time as the first floating point kernel. This not only freed up the single FP execution unit, but also the entire pipeline for the second floating point kernel. The performance benefit is shown by a 20% gain in execution time. This is significant considering that the job mix has contending floating point kernels.

The memory and float kernels (memflt) forms an interesting workload. Though they do not contend for functional units or the data cache they end up doing badly with “HT on” than with “HT off.” This is due to a combination of two factors. First, the memory intensive kernel suffers a lot of cache misses. This leads to its instructions waiting long times in the pipeline. Second, the clogging due to data cache misses also holds up floating point instructions in the pipeline. And when the FP instructions make it to the issue queue, there is only a single FP execution unit. The scarcity of FP execution units and contention for the issue queue leads to an increase of the execution time by 22% with “HT on” compared to “HT off.” However, the workload still performs better with “HT on” than running it “back-to-back” due to better resource utilization.

The combination of memory and integer kernels (memint) overcomes some of the problems faced in the memflt workload. As more integer units are available, the integer kernel flows more quickly through the pipeline. This is shown by gains in execution times with “HT on” compared to “HT off” and “back-to-back.” Of course, higher floating point execution latencies could also be a limiting factor in the memflt job mix.

Another noteworthy result, as shown in Figure 4, is that the addition of an integer kernel to the memflt workload shows performance gains. This is attributed to the operating system scheduling a symbiotic workload [2]. In this case one of the logical processors is used completely by the FP kernel. The other logical processor is shared by the memory and integer kernels through context switching. This allows the floating point kernel to flow easily through the pipeline when co-scheduled with the integer kernel than it did with the memory kernel. This highlights the importance of operating system scheduling in a Hyper-Threaded processor, especially when there are more jobs than the available logical processors.



We have compared the results obtained using our benchmarks with those obtained using the commercially available BYTE's BYTEMark workload. These results showed similar trends in the changes of the execution times when Hyperthreading was turned on and off. As an example, we are presenting above the results obtained when running 2 memory intensive benchmarks together.

5. Speculative pre-computation

Speculative Precomputation (SP) is a technique to speedup the execution of single-threaded applications by utilizing idle multithreading hardware resources to perform data prefetches [3]. It can be used to mask the huge latencies of major performance degrading cache misses from the program (main thread) by prefetching the required data ahead of time using a spare thread of the processor. The spare helper thread tries to use the idle memory bandwidth of the processor to prefetch the data before the main thread needs it. By doing so, it bears the brunt of the cache miss latency, while the main thread sees a cache hit when it wants to access the data. Since the spare thread and memory bandwidth were idle anyway, there is ideally no performance penalty from this approach for the main program. Thus, a large number of cache misses (delinquent loads) could potentially be overlapped with useful work, thereby decreasing the memory latency from the critical path in the original program.

SP is not helpful for programs which are memory-intensive and cause a small amount of memory accesses. Typical applications for SP would be pointer-chasing programs accessing a large amount of data structures, which do not fit in the cache. SP requires static or dynamic construction of the helper threads using Software or Hardware mechanisms. The Hardware mechanism tries to dynamically identify a program's delinquent loads and generate precomputation slices to prefetch them. This is done at run-time without any Software support. We have used the

Software Speculative Precomputation (SSP) approach to perform precomputation. This is a static approach, not requiring any additional hardware support in a Hyperthreading-enabled processor.

The SSP code could be hand generated for delinquent loads identified using profiling techniques. As per our experience, this is not a very easy task. It requires careful understanding of the main thread and a great deal of synchronization between the helper slice and the main program. The speculative threads performing the prefetch should run neither behind nor too far ahead of the main non-speculative thread.

For most practical purposes, an automated tool would be preferred to do the job. Hong Wang et al [3] indicates the development of such an automated post-pass compilation tool, This tool analyzes existing single threaded programs to identify opportunities for prefetching, embeds triggering points for the helper threads in the binary code and creates a new executable which includes these prefetch slices.

We have used POSIX threads (or pthreads for short) to implement spawning and simultaneous execution of the main and helper threads. Fork() does not work for our case, as forking produces 2 threads having separate memory spaces. Since the prime requirement is memory sharing, we have used POSIX threads. Synchronization between the 2 threads is done using mutexes and conditional variables, all part of the POSIX standard. Care has been taken to ensure that the helper thread does not go too far ahead of the main thread.

We were unable to cause any major speedup on our pointer chasing benchmark. We attribute this to difficulties in the synchronization between the 2 threads and finding the right amount of CPU non-memory processing "work" for the main thread to hide the cache miss latency for the helper slice. The uncertainty of load scheduling/balancing done by the OS and other processes running at the time also contributed to the difficulties. We propose to continue work in this direction.

6. Conclusion and Future Work

This paper targets the Hyper-Threading technology implemented on the Pentium 4. By modeling Hyper-Threading on a simulator and studying the first order effects on varying key parameters, we build the framework for a system which could help researchers and developers alike in making intelligent design choices for future multithreaded systems.

In our study of workload characteristics on an actual Hyper-Threading enabled Pentium 4 machine we identify "Contending" workloads that degrade performance and "Complementary" workloads that enhance performance. One important outcome was that memory intensive job mixes perform badly on Hyper-Threading. We also verified our results with a popular benchmark. Other interesting results include better performance due to O/S's "symbiotic" scheduling in job mixes that would otherwise be contending workloads. This shows that the OS scheduler can play a crucial role in getting performance on Hyper-Threading and points to needed research in this area.

We also look at a concept called speculative precomputation, wherein a helper thread is used to prefetch data into the cache, thereby improving the performance of a single threaded application. We have listed various bottlenecks that hindered us from using speculative pre-computation successfully to get a performance enhancement. This would help future researchers to be aware of

bottlenecks that would be encountered when adapting speculative precomputation to improve performance.

Future work involves complete modeling of the Hyper-Threading technology in the simulator and studying the effects of other parameters like the pipeline depth, L2 cache etc. We do not anticipate that the simulator would be ported to the x86 ISA in the near future. We expect our simulator, when complete would be an invaluable tool for multithreading research.

Future work also involves studying benchmarks which are of industry standards and characterizing them. There is a need of tools which facilitate easier development of multi-threaded applications. Future research in this area is also anticipated.

Acknowledgments

We would like to thank Dr. Eric Rotenberg for being a constant source of encouragement throughout the course of this project. He was available whenever we needed him to clear our doubts and help us make progress. We would like to thank Intel for providing NC State University with the HT enabled Pentium 4 machine.

Our sincere thanks to Karthik Sundaramoorthy for helping us install the linux kernel on the P4 machine. We would also like to acknowledge Ali El-Haj-Mahmoud and Mazen Kharbutli, Bhaskar Bharat and Lashminarayan Venkatesan for their valuable suggestions.

REFERENCES

- [1] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. Intel Technical Journal, Q1 2001.
- [2] Marr, D.; Binns, F.; Hill, D.; Hinton, G.; Koufaty, D.; Miller, J.; Upton, M. "Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History." Intel Technology Journal. <http://developer.intel.com/technology/itj/2002/volume06issue01/> (Feb 2002).
- [3] Hong Wang. Speculative Precomputation: Exploring the Use of Multithreading for Latency. Intel Technology Journal 1 st Quarter 2002 Volume 6 Issue1.
- [4] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. 23rd Annual International Symposium on Computer Architecture, pp. 191-202, May 1996.
- [5] www.simplescalar.com
- [6] Eric Rotenberg, Course Notes, ECE 792E Advanced Microarchitecture, Fall 2002 NC State University.
- [7] <http://developer.intel.com/itj>
- [8] <http://www.intel.com/technology/hyperthread/>

- [9] <http://www.tux.org/~mayer/linux/bmark.html>
- [10] Dean Tullsen, Susan Eggers, and Henry Levy, Simultaneous Multithreading: Maximizing On-Chip Parallelism Proceedings of the 22rd Annual International Symposium on Computer Architecture, June 1995, pages 392-403.
- [11] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989.
- [12] W. Magro, P.Petersen and Sanjiv Shah , "Hyper-Threading Technology: Impact on Compute-Intensive Workloads," Intel Technology Journal, Q1, 2002
- [13] A. Snaveley and D.M.Tullsen., "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," 9th International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000
- [14] <http://www.byte.com/bmark/>
- [15] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. "Slipstream Processors: Improving both Performance and Fault Tolerance". 9th International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.
- [16] C. B. Zilles and G. S. Sohi. Execution-based Prediction Using Speculative Slices. ISCA-2001, July 2001
- [17] Chi-Keung,Luk "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processor"28th ISCA June 2001