Intel Technology Journal

Volume 06 Issue 01 Published February 14, 2002 ISSN 1535766X

Hyper-Threading Technology
Table of Contents

Articles

Preface q1. 2002

By Lin Chao, Publisher Intel Technology Journal

This February 2002 issue of the *Intel Technology Journal* (ITJ) is full of new things. First, there is a new look and design. This is the first big redesign since the inception of the ITJ on the Web in 1997. The new design, together with inclusion of the ISSN (International Standard Serial Number), makes it easier to index articles into technical indexes and search engines. There are new "subscribe," search ITJ, and "e-mail to a colleague" features in the left navigation tool bar. Readers are encouraged to subscribe to the ITJ. The benefit is subscribers are notified by e-mail when a new issue is published.

The focus of this issue is Hyper-Threading Technology, a new microprocessor architecture technology. It makes a single processor look like two processors to the operating system. Intel's Hyper-Threading Technology delivers two logical processors that can execute different tasks simultaneously using shared hardware resources. Hyper-Threading Technology effectively looks like two processors on a chip. A chip with this technology will not equal the computing power of two processors; however, it will seem like two, as the performance boost is substantial. Chips enabled with Hyper-Threading Technology will also be cheaper than dual-processor computers: one heat sink, one fan, one cooling solution, and one chip are what are necessary.

The six papers in this issue of the *Intel Technology Journal* discuss this new technology. The papers cover a broad view of Hyper-Threading Technology including the architecture, microarchitecture, pre-silicon validation and performance impact on media and compute-intensive applications. Also included is an intriguing paper on speculative precomputation, a technique that improves the latency of single-threaded applications by utilizing idle multithreading hardware resources to perform long-range data prefetches.

ITJ Foreword Q1, 2002

Intel® Hyper-Threading Technology

By Robert L. Cross

Multithreading Technologies Manager

Performance–affordable performance, relevant performance, and pervasively available performance–continues to be a key concern for end users. Enterprise and technical computing users have a never-ending need for increased performance and capacity. Moreover, industry analysts continue to observe that complex games and rich entertainment for consumers, plus a wide range of new business uses, software, and components, will necessitate growth in computing power.

Processor resources, however, are often underutilized and the growing gap between core processor frequency and memory speed causes memory latency to become an increasing performance challenge. Intel's Hyper-Threading Technology brings Simultaneous Multi-Threading to the Intel Architecture and makes a single physical processor appear as two logical processors with duplicated architecture state, but with shared physical execution resources. This allows two tasks (two threads from a single application or two separate applications) to execute in parallel, increasing processor utilization and reducing the performance impact of memory latency by overlapping the memory latency of one task with the execution of another. Hyper-Threading Technology-capable processors offer significant performance improvements for multi-threaded and multi-tasking workloads without sacrificing compatibility with existing software or single-threaded performance. Remarkably, Hyper-Threading Technology implements these improvements at a very low cost in power and processor die size.

The papers in this issue of the *Intel Technology Journal* discuss the design, challenges, and performance opportunities of Intel's first implementation of Hyper-Threading Technology in the Intel® Xeon processor family. Hyper-Threading Technology is a key feature of Intel's enterprise product line and will be integrated into a wide variety of products. It marks the beginning of a new era: the transition from instruction-level parallelism to thread-level parallelism, and it lays the foundation for a new level of computing industry innovation and end-user benefits.

Hyper-Threading Technology Architecture and Microarchitecture

Deborah T. Marr, Desktop Products Group, Intel Corp.
Frank Binns, Desktop ProductsGroup, Intel Corp.
David L. Hill, Desktop Products Group, Intel Corp.
Glenn Hinton, Desktop Products Group, Intel Corp.
David A. Koufaty, Desktop Products Group, Intel Corp.
J. Alan Miller, Desktop Products Group, Intel Corp.
Michael Upton, CPU Architecture, Desktop Products Group, Intel Corp.

Index words: architecture, microarchitecture, Hyper-Threading Technology, simultaneous multi-threading, multiprocessor

ABSTRACT

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

This paper describes the Hyper-Threading Technology architecture, and discusses the microarchitecture details of Intel's first implementation on the Intel® Xeon™ processor family. Hyper-Threading Technology is an important addition to Intel's enterprise product line and will be integrated into a wide variety of products.

INTRODUCTION

The amazing growth of the Internet and telecommunications is powered by ever-faster systems demanding increasingly higher levels of processor performance. To keep up with this demand we cannot rely entirely on traditional approaches to processor design. Microarchitecture techniques used to achieve past processor performance improvement–super-pipelining, branch prediction, super-scalar execution, out-of-order execution, caches–have made microprocessors increasingly more complex, have more transistors, and consume more power. In fact, transistor counts and power are increasing at rates greater than processor performance. Processor architects are therefore looking for ways to improve performance at a greater rate than transistor counts and power dissipation. Intel's Hyper-Threading Technology is one solution.

Processor Microarchitecture

Traditional approaches to processor design have focused on higher clock speeds, instruction-level parallelism (ILP), and caches. Techniques to achieve higher clock speeds involve pipelining the microarchitecture to finer granularities, also called super-pipelining. Higher clock frequencies can greatly improve performance by increasing the number of instructions that can be executed each second. Because there will be far more instructions in-flight in a super-pipelined microarchitecture, handling of events that disrupt the pipeline, e.g., cache misses, interrupts and branch mispredictions, can be costly.

ILP refers to techniques to increase the number of instructions executed each clock cycle. For example, a super-scalar processor has multiple parallel execution units that can process instructions simultaneously. With super-scalar execution, several instructions can be executed each clock cycle. However, with simple in-order execution, it is not enough to simply have multiple execution units. The challenge is to find enough instructions to execute. One technique is out-of-order execution where a large window of instructions is simultaneously evaluated and sent to execution units, based on instruction dependencies rather than program order.

Accesses to DRAM memory are slow compared to execution speeds of the processor. One technique to reduce this latency is to add fast caches close to the processor. Caches can provide fast memory access to frequently accessed data or instructions. However, caches can only be fast when they are small. For this reason, processors often are designed with a cache hierarchy in which fast, small caches are located and operated at access latencies very close to that of the processor core, and progressively larger caches, which handle less frequently accessed data or instructions, are implemented with longer access latencies. However, there will always be times when the data needed will not be in any processor cache. Handling such cache misses requires accessing memory, and the processor is likely to quickly run out of instructions to execute before stalling on the cache miss.

The vast majority of techniques to improve processor performance from one generation to the next is complex and often adds significant die-size and power costs. These techniques increase performance but not with 100% efficiency; i.e., doubling the number of execution units in a processor does not double the performance of the processor, due to limited parallelism in instruction flows. Similarly, simply doubling the clock rate does not double the performance due to the number of processor cycles lost to branch mispredictions.

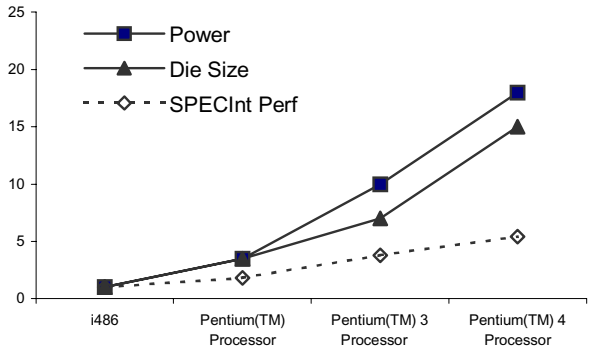


Figure 1: Single-stream performance vs. cost

Figure 1 shows the relative increase in performance and the costs, such as die size and power, over the last ten years on Intel processors[1]. In order to isolate the microarchitecture impact, this comparison assumes that the four generations of processors are on the same silicon process technology and that the speed-ups are normalized to the performance of an Intel486™ processor. Although we use Intel's processor history in this example, other high-performance processor manufacturers during this time period would have similar trends. Intel's processor performance, due to microarchitecture advances alone, has improved integer performance five- or six-fold[1]. Most integer applications have limited ILP and the instruction flow can be hard to predict.

Over the same period, the relative die size has gone up fifteen-fold, a three-times-higher rate than the gains in integer performance. Fortunately, advances in silicon process technology allow more transistors to be packed into a given amount of die area so that the actual measured die size of each generation microarchitecture has not increased significantly.

The relative power increased almost eighteen-fold during this period[1]. Fortunately, there exist a number of known techniques to significantly reduce power consumption on processors and there is much on-going research in this area. However, current processor power dissipation is at the limit of what can be easily dealt with in desktop platforms and we must put greater emphasis on improving performance in conjunction with new technology, specifically to control power.

[1] These data are approximate and are intended only to show trends, not actual performance.

™ Intel486 is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Thread-Level Parallelism

A look at today's software trends reveals that server applications consist of multiple threads or processes that can be executed in parallel. On-line transaction processing and Web services have an abundance of software threads that can be executed simultaneously for faster performance. Even desktop applications are becoming increasingly parallel. Intel architects have been trying to leverage this so-called thread-level parallelism (TLP) to gain a better performance vs. transistor count and power ratio.

In both the high-end and mid-range server markets, multiprocessors have been commonly used to get more performance from the system. By adding more processors, applications potentially get substantial performance improvement by executing multiple threads on multiple processors at the same time. These threads might be from the same application, from different applications running simultaneously, from operating system services, or from operating system threads doing background maintenance. Multiprocessor systems have been used for many years, and high-end programmers are familiar with the techniques to exploit multiprocessors for higher performance levels.

In recent years a number of other techniques to further exploit TLP have been discussed and some products have been announced. One of these techniques is chip multiprocessing (CMP), where two processors are put on a single die. The two processors each have a full set of execution and architectural resources. The processors may or may not share a large on-chip cache. CMP is largely orthogonal to conventional multiprocessor systems, as you can have multiple CMP processors in a multiprocessor configuration. Recently announced processors incorporate two processors on each die. However, a CMP chip is significantly larger than the size of a single-core chip and therefore more expensive to manufacture; moreover, it does not begin to address the die size and power considerations.

Another approach is to allow a single processor to execute multiple threads by switching between them. Time-slice multithreading is where the processor switches between software threads after a fixed time period. Time-slice multithreading can result in wasted execution slots but can effectively minimize the effects of long latencies to memory. Switch-on-event multi-threading would switch threads on long latency events such as cache misses. This approach can work well for server applications that have large numbers of cache misses and where the two threads are executing similar tasks. However, both the time-slice and the switch-on-event multi-threading techniques do not achieve optimal overlap of many sources of inefficient resource usage, such as branch mispredictions, instruction dependencies, etc.

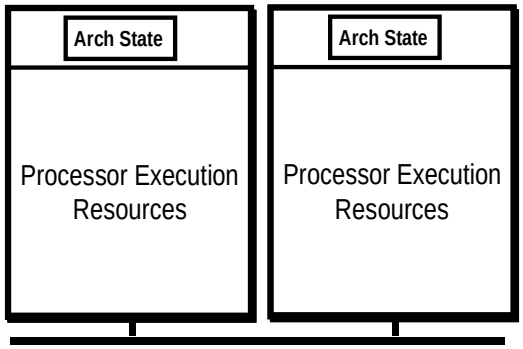
Finally, there is simultaneous multi-threading, where multiple threads can execute on a single processor without switching. The threads execute simultaneously and make much better use of the resources. This approach makes the most effective use of processor resources: it maximizes the performance vs. transistor count and power consumption.

Hyper-Threading Technology brings the simultaneous multi-threading approach to the Intel architecture. In this paper we discuss the architecture and the first implementation of Hyper-Threading Technology on the Intel® Xeon™ processor family.

HYPER-THREADING TECHNOLOGY ARCHITECTURE

Hyper-Threading Technology makes a single physical processor appear as multiple logical processors [11, 12]. To do this, there is one copy of the architecture state for each logical processor, and the logical processors share a single set of physical execution resources. From a software or architecture perspective, this means operating systems and user programs can schedule processes or threads to logical processors as they would on conventional physical processors in a multi-processor system. From a microarchitecture perspective, this means that instructions from logical processors will persist and execute simultaneously on shared execution resources.

Figure 2: Processors without Hyper-Threading Tech



®Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

As an example, Figure 2 shows a multiprocessor system with two physical processors that are not Hyper-Threading Technology-capable. Figure 3 shows a multiprocessor system with two physical processors that are Hyper-Threading Technology-capable. With two copies of the architectural state on each physical processor, the system appears to have four logical processors.

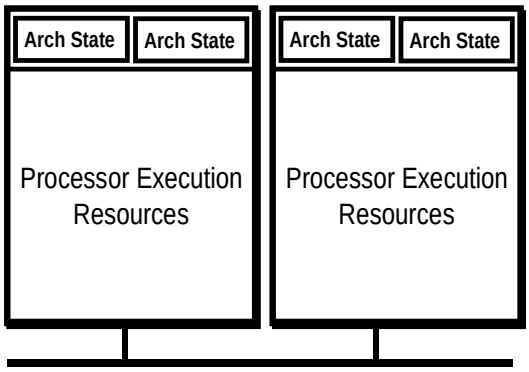


Figure 3: Processors with Hyper-Threading Technology

The first implementation of Hyper-Threading Technology is being made available on the Intel® Xeon™ processor family for dual and multiprocessor servers, with two logical processors per physical processor. By more efficiently using existing processor resources, the Intel Xeon processor family can significantly improve performance at virtually the same system cost. This implementation of Hyper-Threading Technology added less than 5% to the relative chip size and maximum power requirements, but can provide performance benefits much greater than that.

Each logical processor maintains a complete set of the architecture state. The architecture state consists of registers including the general-purpose registers, the control registers, the advanced programmable interrupt controller (APIC) registers, and some machine state registers. From a software perspective, once the architecture state is duplicated, the processor appears to be two processors. The number of transistors to store the architecture state is an extremely small fraction of the total. Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic, and buses.

Each logical processor has its own interrupt controller or APIC. Interrupts sent to a specific logical processor are handled only by that logical processor.

FIRST IMPLEMENTATION ON THE INTEL XEON PROCESSOR FAMILY

Several goals were at the heart of the microarchitecture design choices made for the Intel® Xeon™ processor MP implementation of Hyper-Threading Technology. One goal was to minimize the die area cost of implementing Hyper-Threading Technology. Since the logical processors share the vast majority of microarchitecture resources and only a few small structures were replicated, the die area cost of the first implementation was less than 5% of the total die area.

A second goal was to ensure that when one logical processor is stalled the other logical processor could continue to make forward progress. A logical processor may be temporarily stalled for a variety of reasons, including servicing cache misses, handling branch mispredictions, or waiting for the results of previous instructions. Independent forward progress was ensured by managing buffering queues such that no logical processor can use all the entries when two active software threads[2] were executing. This is accomplished by either partitioning or limiting the number of active entries each thread can have.

A third goal was to allow a processor running only one active software thread to run at the same speed on a processor with Hyper-Threading Technology as on a processor without this capability. This means that partitioned resources should be recombined when only one software thread is active. A high-level view of the microarchitecture pipeline is shown in Figure 4. As shown, buffering queues separate major pipeline logic blocks. The buffering queues are either partitioned or duplicated to ensure independent forward progress through each logic block.

® Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™ Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

®Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™ Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[2] Active software threads include the operating system idle loop because it runs a sequence of code that continuously checks the work queue(s). The operating system idle loop can consume considerable execution resources.

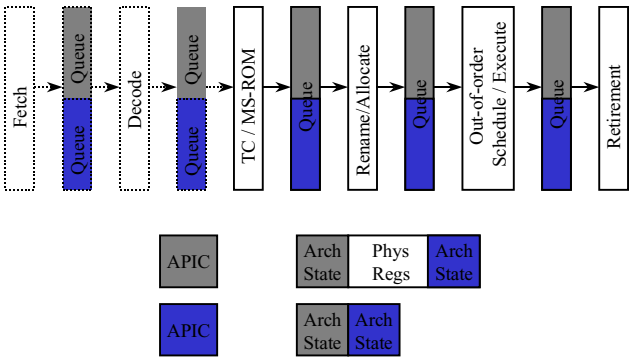


Figure 4 Intel® Xeon™ processor pipeline

In the following sections we will walk through the pipeline, discuss the implementation of major functions, and detail several ways resources are shared or replicated.

FRONT END

The front end of the pipeline is responsible for delivering instructions to the later pipe stages. As shown in Figure 5a, instructions generally come from the Execution Trace Cache (TC), which is the primary or Level 1 (L1) instruction cache. Figure 5b shows that only when there is a TC miss does the machine fetch and decode instructions from the integrated Level 2 (L2) cache. Near the TC is the Microcode ROM, which stores decoded instructions for the longer and more complex IA-32 instructions.

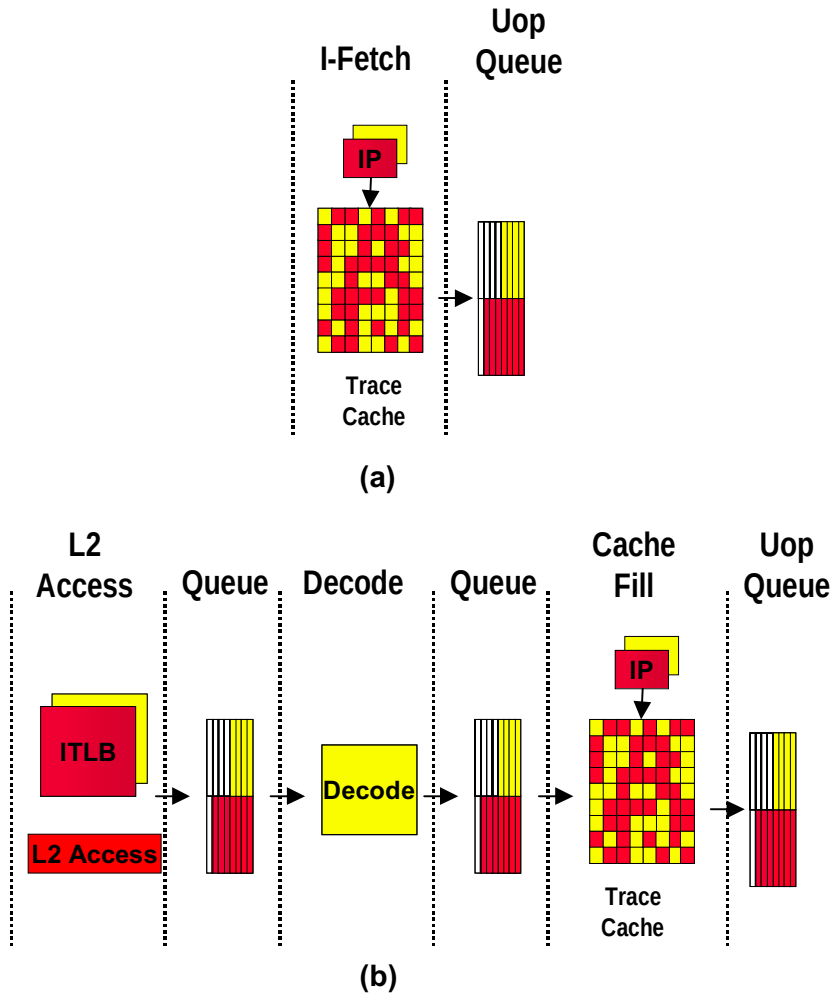


Figure 5: Front-end detailed pipeline (a) Trace Cache Hit (b) Trace Cache Miss

Execution Trace Cache (TC)

The TC stores decoded instructions, called micro-operations or "uops." Most instructions in a program are fetched and executed from the TC. Two sets of next-instruction-pointers independently track the progress of the two software threads executing. The two logical processors arbitrate access to the TC every clock cycle. If both logical processors want access to the TC at the same time, access is granted to one then the other in alternating clock cycles. For example, if one cycle is used to fetch a line for one logical processor, the next cycle would be used to fetch a line for the other logical processor, provided that both logical processors requested access to the trace cache. If one logical processor is stalled or is unable to use the TC, the other logical processor can use the full bandwidth of the trace cache, every cycle.

The TC entries are tagged with thread information and are dynamically allocated as needed. The TC is 8-way set associative, and entries are replaced based on a least-recently-used (LRU) algorithm that is based on the full 8 ways. The shared nature of the TC allows one logical processor to have more entries than the other if needed.

Microcode ROM

When a complex instruction is encountered, the TC sends a microcode-instruction pointer to the Microcode ROM. The Microcode ROM controller then fetches the uops needed and returns control to the TC. Two microcode instruction pointers are used to control the flows independently if both logical processors are executing complex IA-32 instructions.

Both logical processors share the Microcode ROM entries. Access to the Microcode ROM alternates between logical processors just as in the TC.

ITLB and Branch Prediction

If there is a TC miss, then instruction bytes need to be fetched from the L2 cache and decoded into uops to be placed in the TC. The Instruction Translation Lookaside Buffer (ITLB) receives the request from the TC to deliver new instructions, and it translates the next-instruction pointer address to a physical address. A request is sent to the L2 cache, and instruction bytes are returned. These bytes are placed into streaming buffers, which hold the bytes until they can be decoded.

The ITLBs are duplicated. Each logical processor has its own ITLB and its own set of instruction pointers to track the progress of instruction fetch for the two logical processors. The instruction fetch logic in charge of sending requests to the L2 cache arbitrates on a first-come first-served basis, while always reserving at least one request slot for each logical processor. In this way, both logical processors can have fetches pending simultaneously.

Each logical processor has its own set of two 64-byte streaming buffers to hold instruction bytes in preparation for the instruction decode stage. The ITLBs and the streaming buffers are small structures, so the die size cost of duplicating these structures is very low.

The branch prediction structures are either duplicated or shared. The return stack buffer, which predicts the target of return instructions, is duplicated because it is a very small structure and the call/return pairs are better predicted for software threads independently. The branch history buffer used to look up the global history array is also tracked independently for each logical processor. However, the large global history array is a shared structure with entries that are tagged with a logical processor ID.

IA-32 Instruction Decode

IA-32 instructions are cumbersome to decode because the instructions have a variable number of bytes and have many different options. A significant amount of logic and intermediate state is needed to decode these instructions. Fortunately, the TC provides most of the uops, and decoding is only needed for instructions that miss the TC.

The decode logic takes instruction bytes from the streaming buffers and decodes them into uops. When both threads are decoding instructions simultaneously, the streaming buffers alternate between threads so that both threads share the same decoder logic. The decode logic has to keep two copies of all the state needed to decode IA-32 instructions for the two logical processors even though it only decodes instructions for one logical processor at a time. In general, several instructions are decoded for one logical processor before switching to the other logical processor. The decision to do a coarser level of granularity in switching between logical processors was made in the interest of die size and to reduce complexity. Of course, if only one logical processor needs the decode logic, the full decode bandwidth is dedicated to that logical processor. The decoded instructions are written into the TC and forwarded to the uop queue.

Uop Queue

After uops are fetched from the trace cache or the Microcode ROM, or forwarded from the instruction decode logic, they are placed in a "uop queue." This queue decouples the Front End from the Out-of-order

Execution Engine in the pipeline flow. The uop queue is partitioned such that each logical processor has half the entries. This partitioning allows both logical processors to make independent forward progress regardless of front-end stalls (e.g., TC miss) or execution stalls.

OUT-OF-ORDER EXECUTION ENGINE

The out-of-order execution engine consists of the allocation, register renaming, scheduling, and execution functions, as shown in Figure 6. This part of the machine re-orders instructions and executes them as quickly as their inputs are ready, without regard to the original program order.

Allocator

The out-of-order execution engine has several buffers to perform its re-ordering, tracing, and sequencing operations. The allocator logic takes uops from the uop queue and allocates many of the key machine buffers needed to execute each uop, including the 126 re-order buffer entries, 128 integer and 128 floating-point physical registers, 48 load and 24 store buffer entries. Some of these key buffers are partitioned such that each logical processor can use at most half the entries.

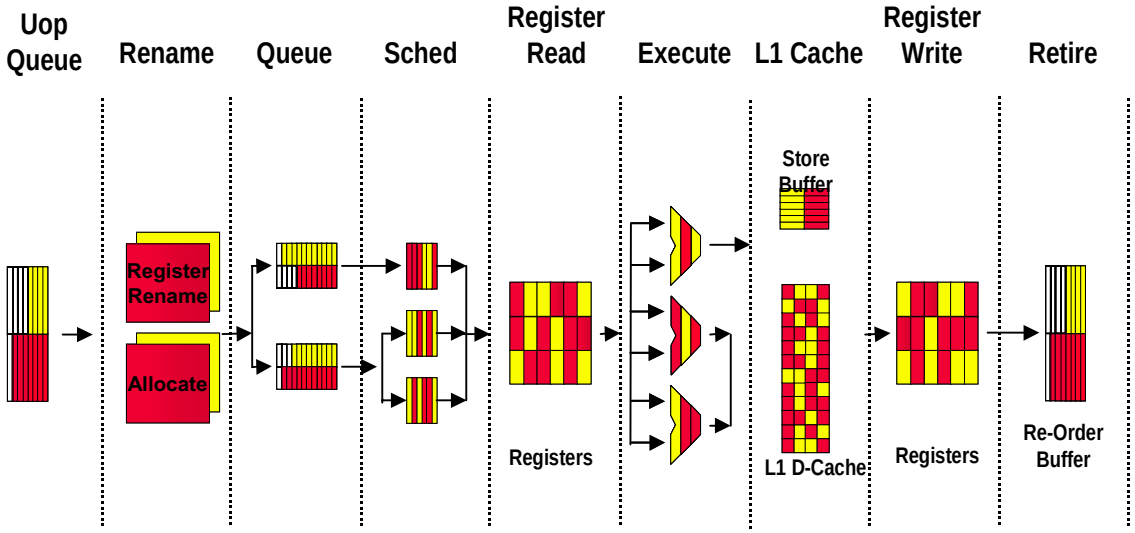


Figure 6: Out-of-order execution engine detailed pipeline

Specifically, each logical processor can use up to a maximum of 63 re-order buffer entries, 24 load buffers, and 12 store buffer entries.

If there are uops for both logical processors in the uop queue, the allocator will alternate selecting uops from the logical processors every clock cycle to assign resources. If a logical processor has used its limit of a needed resource, such as store buffer entries, the allocator will signal "stall" for that logical processor and continue to assign resources for the other logical processor. In addition, if the uop queue only contains uops for one logical processor, the allocator will try to assign resources for that logical processor every cycle to optimize allocation bandwidth, though the resource limits would still be enforced.

By limiting the maximum resource usage of key buffers, the machine helps enforce fairness and prevents deadlocks.

Register Rename

The register rename logic renames the architectural IA-32 registers onto the machine's physical registers. This allows the 8 general-use IA-32 integer registers to be dynamically expanded to use the available 128 physical registers. The renaming logic uses a Register Alias Table (RAT) to track the latest version of each architectural register to tell the next instruction(s) where to get its input operands.

Since each logical processor must maintain and track its own complete architecture state, there are two RATs, one for each logical processor. The register renaming process is done in parallel to the allocator logic described above, so the register rename logic works on the same uops to which the allocator is assigning resources.

Once uops have completed the allocation and register rename processes, they are placed into two sets of

queues, one for memory operations (loads and stores) and another for all other operations. The two sets of queues are called the memory instruction queue and the general instruction queue, respectively. The two sets of queues are also partitioned such that uops from each logical processor can use at most half the entries.

Instruction Scheduling

The schedulers are at the heart of the out-of-order execution engine. Five uop schedulers are used to schedule different types of uops for the various execution units. Collectively, they can dispatch up to six uops each clock cycle. The schedulers determine when uops are ready to execute based on the readiness of their dependent input register operands and the availability of the execution unit resources.

The memory instruction queue and general instruction queues send uops to the five scheduler queues as fast as they can, alternating between uops for the two logical processors every clock cycle, as needed.

Each scheduler has its own scheduler queue of eight to twelve entries from which it selects uops to send to the execution units. The schedulers choose uops regardless of whether they belong to one logical processor or the other. The schedulers are effectively oblivious to logical processor distinctions. The uops are simply evaluated based on dependent inputs and availability of execution resources. For example, the schedulers could dispatch two uops from one logical processor and two uops from the other logical processor in the same clock cycle. To avoid deadlock and ensure fairness, there is a limit on the number of active entries that a logical processor can have in each scheduler's queue. This limit is dependent on the size of the scheduler queue.

Execution Units

The execution core and memory hierarchy are also largely oblivious to logical processors. Since the source and destination registers were renamed earlier to physical registers in a shared physical register pool, uops merely access the physical register file to get their destinations, and they write results back to the physical register file. Comparing physical register numbers enables the forwarding logic to forward results to other executing uops without having to understand logical processors.

After execution, the uops are placed in the re-order buffer. The re-order buffer decouples the execution stage from the retirement stage. The re-order buffer is partitioned such that each logical processor can use half the entries.

Retirement

Uop retirement logic commits the architecture state in program order. The retirement logic tracks when uops from the two logical processors are ready to be retired, then retires the uops in program order for each logical processor by alternating between the two logical processors. Retirement logic will retire uops for one logical processor, then the other, alternating back and forth. If one logical processor is not ready to retire any uops then all retirement bandwidth is dedicated to the other logical processor.

Once stores have retired, the store data needs to be written into the level-one data cache. Selection logic alternates between the two logical processors to commit store data to the cache.

MEMORY SUBSYSTEM

The memory subsystem includes the DTLB, the low-latency Level 1 (L1) data cache, the Level 2 (L2) unified cache, and the Level 3 unified cache (the Level 3 cache is only available on the Intel® Xeon™ processor MP). Access to the memory subsystem is also largely oblivious to logical processors. The schedulers send load or store uops without regard to logical processors and the memory subsystem handles them as they come.

DTLB

The DTLB translates addresses to physical addresses. It has 64 fully associative entries; each entry can map either a 4K or a 4MB page. Although the DTLB is a shared structure between the two logical processors, each entry includes a logical processor ID tag. Each logical processor also has a reservation register to ensure fairness and forward progress in processing DTLB misses.

L1 Data Cache, L2 Cache, L3 Cache

The L1 data cache is 4-way set associative with 64-byte lines. It is a write-through cache, meaning that writes are always copied to the L2 cache. The L1 data cache is virtually addressed and physically tagged.

The L2 and L3 caches are 8-way set associative with 128-byte lines. The L2 and L3 caches are physically addressed. Both logical processors, without regard to which logical processor's uops may have initially

brought the data into the cache, can share all entries in all three levels of cache.

Because logical processors can share data in the cache, there is the potential for cache conflicts, which can result in lower observed performance. However, there is also the possibility for sharing data in the cache. For example, one logical processor may prefetch instructions or data, needed by the other, into the cache; this is common in server application code. In a producer-consumer usage model, one logical processor may produce data that the other logical processor wants to use. In such cases, there is the potential for good performance benefits.

BUS

Logical processor memory requests not satisfied by the cache hierarchy are serviced by the bus logic. The bus logic includes the local APIC interrupt controller, as well as off-chip system memory and I/O space. Bus logic also deals with cacheable address coherency (snooping) of requests originated by other external bus agents, plus incoming interrupt request delivery via the local APICs.

From a service perspective, requests from the logical processors are treated on a first-come basis, with queue and buffering space appearing shared. Priority is not given to one logical processor above the other.

Distinctions between requests from the logical processors are reliably maintained in the bus queues nonetheless. Requests to the local APIC and interrupt delivery resources are unique and separate per logical processor. Bus logic also carries out portions of barrier fence and memory ordering operations, which are applied to the bus request queues on a per logical processor basis.

For debug purposes, and as an aid to forward progress mechanisms in clustered multiprocessor implementations, the logical processor ID is visibly sent onto the processor external bus in the request phase portion of a transaction. Other bus transactions, such as cache line eviction or prefetch transactions, inherit the logical processor ID of the request that generated the transaction.

SINGLE-TASK AND MULTI-TASK MODES

To optimize performance when there is one software thread to execute, there are two modes of operation referred to as single-task (ST) or multi-task (MT). In MT-mode, there are two active logical processors and some of the resources are partitioned as described earlier. There are two flavors of ST-mode: single-task logical processor 0 (ST0) and single-task logical processor 1 (ST1). In ST0- or ST1-mode, only one logical processor is active, and resources that were partitioned in MT-mode are re-combined to give the single active logical processor use of all of the resources. The IA-32 Intel Architecture has an instruction called HALT that stops processor execution and normally allows the processor to go into a lower-power mode. HALT is a privileged instruction, meaning that only the operating system or other ring-0 processes may execute this instruction. User-level applications cannot execute HALT.

On a processor with Hyper-Threading Technology, executing HALT transitions the processor from MT-mode to ST0- or ST1-mode, depending on which logical processor executed the HALT. For example, if logical processor 0 executes HALT, only logical processor 1 would be active; the physical processor would be in ST1-mode and partitioned resources would be recombined giving logical processor 1 full use of all processor resources. If the remaining active logical processor also executes HALT, the physical processor would then be able to go to a lower-power mode.

In ST0- or ST1-modes, an interrupt sent to the HALTed processor would cause a transition to MT-mode. The operating system is responsible for managing MT-mode transitions (described in the next section).

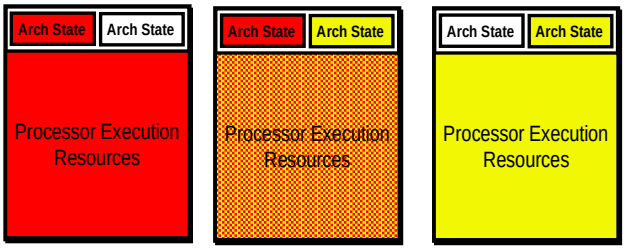


Figure 7: Resource allocation

Figure 7 summarizes this discussion. On a processor with Hyper-Threading Technology, resources are allocated to a single logical processor if the processor is in ST0- or ST1-mode. On the MT-mode, resources are shared between the two logical processors.

OPERATING SYSTEM AND APPLICATIONS

A system with processors that use Hyper-Threading Technology appears to the operating system and application software as having twice the number of processors than it physically has. Operating systems manage logical processors as they do physical

processors, scheduling runnable tasks or threads to logical processors. However, for best performance, the operating system should implement two optimizations.

The first is to use the HALT instruction if one logical processor is active and the other is not. HALT will allow the processor to transition to either the ST0- or ST1-mode. An operating system that does not use this optimization would execute on the idle logical processor a sequence of instructions that repeatedly checks for work to do. This so-called "idle loop" can consume significant execution resources that could otherwise be used to make faster progress on the other active logical processor.

The second optimization is in scheduling software threads to logical processors. In general, for best performance, the operating system should schedule threads to logical processors on different physical processors before scheduling multiple threads to the same physical processor. This optimization allows software threads to use different physical execution resources when possible.

PERFORMANCE

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

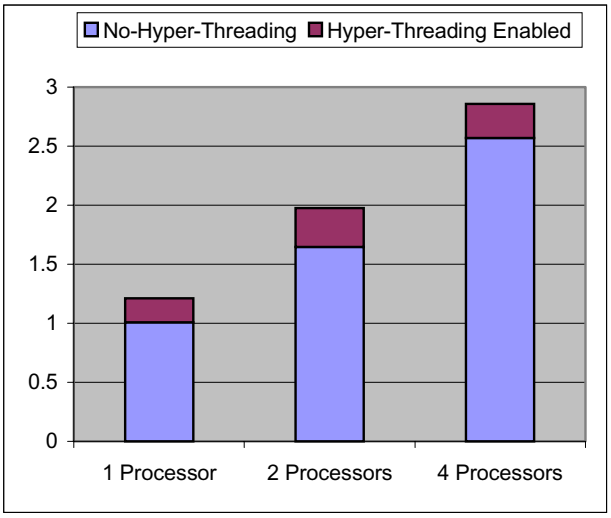


Figure 8: Performance increases from Hyper-Threading Technology on an OLTP workload

Figure 8 shows the online transaction processing performance, scaling from a single-processor configuration through to a 4-processor system with Hyper-Threading Technology enabled. This graph is normalized to the performance of the single-processor system. It can be seen that there is a significant overall performance gain attributable to Hyper-Threading Technology, 21% in the cases of the single and dual-processor systems.

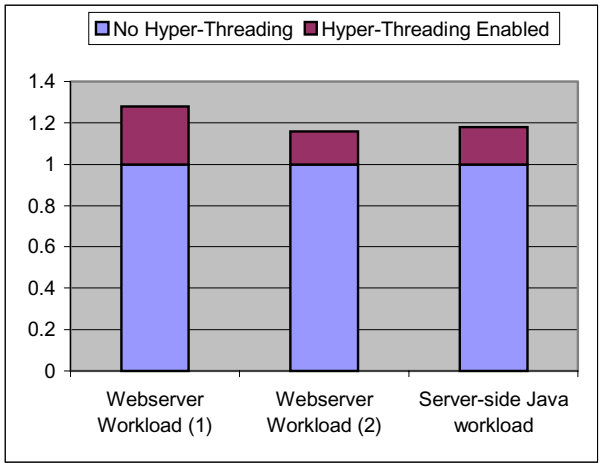


Figure 9: Web server benchmark performance

Figure 9 shows the benefit of Hyper-Threading Technology when executing other server-centric benchmarks. The workloads chosen were two different benchmarks that are designed to exercise data and Web server characteristics and a workload that focuses on exercising a server-side Java environment. In these cases the performance benefit ranged from 16 to 28%.

®Intel and Pentium are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

™Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

All the performance results quoted above are normalized to ensure that readers focus on the relative performance and not the absolute performance.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to www.intel.com/procs/perf/limits.htm or call (U.S.) 1-800-628-8686 or 1-916-356-3104

CONCLUSION

Intel's Hyper-Threading Technology brings the concept of simultaneous multi-threading to the Intel Architecture. This is a significant new technology direction for Intel's future processors. It will become increasingly important going forward as it adds a new technique for obtaining additional performance for lower transistor and power costs.

The first implementation of Hyper-Threading Technology was done on the Intel® Xeon™ processor MP. In this implementation there are two logical processors on each physical processor. The logical processors have their own independent architecture state, but they share nearly all the physical execution and hardware resources of the processor. The goal was to implement the technology at minimum cost while ensuring forward progress on logical processors, even if the other is stalled, and to deliver full performance even when there is only one active logical processor. These goals were achieved through efficient logical processor selection algorithms and the creative partitioning and recombining algorithms of many key resources.

Measured performance on the Intel Xeon processor MP with Hyper-Threading Technology shows performance gains of up to 30% on common server application benchmarks for this technology.

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

ACKNOWLEDGMENTS

Making Hyper-Threading Technology a reality was the result of enormous dedication, planning, and sheer hard work from a large number of designers, validators, architects, and others. There was incredible teamwork from the operating system developers, BIOS writers, and software developers who helped with innovations and provided support for many decisions that were made during the definition process of Hyper-Threading Technology. Many dedicated engineers are continuing to work with our ISV partners to analyze application performance for this technology. Their contributions and hard work have already made and will continue to make a real difference to our customers.

REFERENCES

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

R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porter, and B. Smith, "The TERA Computer System," in *International Conference on Supercomputing*, Pages 1 - 6, June 1990.

L. A. Barroso et. al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Pages 282 - 293, June 2000.

M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee, "The M-Machine Multicomputer," in *28th Annual International Symposium on Microarchitecture*, Nov. 1995.

L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *Computer*, 30(9), 79 - 85, September 1997.

D. J. C. Johnson, "HP's Mako Processor," *Microprocessor Forum*, October 2001, http://www.cpus.hp.com/technical_references/mpf_2001.pdf

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

J. M. Tendler, S. Dodson, and S. Fields, "POWER4 System Microarchitecture," *Technical White Paper. IBM Server Group*, October 2001.

® Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

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

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

Intel Corporation. "IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture," Order number 245472, 2001 http://developer.intel.com/design/Pentium4/manuals

Intel Corporation. "IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide," Order number 245472, 2001 http://developer.intel.com/design/Pentium4/manuals

AUTHORS' BIOGRAPHIES

Deborah T. Marr is the CPU architect responsible for Hyper-Threading Technology in the Desktop Products Group. Deborah has been at Intel for over ten years. She first joined Intel in 1988 and made significant contributions to the Intel 386SX processor, the P6 processor microarchitecture, and the Intel® Pentium® 4 Processor microarchitecture. Her interests are in high-performance microarchitecture and performance analysis. Deborah received her B.S. degree in EECS from the University of California at Berkeley in 1988, and her M.S. degree in ECE from Cornell University in 1992. Her e-mail address is debbie.marr@intel.com.

Frank Binns obtained a B.S. degree in electrical engineering from Salford University, England. He joined Intel in 1984 after holding research engineering positions with Marconi Research Laboratories and the Diamond Trading Company Research Laboratory, both of the U.K. Frank has spent the last 16 years with Intel, initially holding technical management positions in the Development Tool, Multibus Systems and PC Systems divisions. Frank's last eight years have been spent in the Desktop Processor Group in Technical Marketing and Processor Architecture roles. His e-mail is frank.binns@intel.com.

Dave L. Hill joined Intel in 1993 and was the quad pumped bus logic architect for the Pentium® 4 processor. Dave has 20 years industry experience primarily in high-performance memory system microarchitecture, logic design, and system debug. His e-mail address is david.l.hill@intel.com.

Glenn Hinton is an Intel Fellow, Desktop Platforms Group and Director of IA-32 Microarchitecture Development. He is responsible for the microarchitecture development for the next-generation IA-32 design. He was appointed Intel Fellow in January 1999. He received bachelor's and master's degrees in Electrical Engineering from Brigham Young University in 1982 and 1983, respectively. His e-mail address is glenn.hinton@intel.com.

David A. Koufaty received B.S. and M.S. degrees from the Simon Bolivar University, Venezuela in 1988 and 1991, respectively. He then received a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1997. For the last three years he has worked for the DPG CPU Architecture organization. His main interests are in multiprocessor architecture and software, performance, and compilation. His e-mail address is david.a.koufaty@intel.com.

John (Alan) Miller has worked at Intel for over five years. During that time, he worked on design and architecture for the Pentium® 4 processor and proliferation projects. Alan obtained his M.S. degree in Electrical and Computer Engineering from Carnegie-Mellon University. His e-mail is alan.miller@intel.com.

Michael Upton is a Principal Engineer/Architect in Intel's Desktop Platforms Group, and is one of the architects of the Intel Pentium® 4 processor. He completed B.S. and M.S. degrees in Electrical Engineering from the University of Washington in 1985 and 1990. After a number of years in IC design and CAD tool development, he entered the University of Michigan to study computer architecture. Upon completion of his Ph.D. degree in 1994, he joined Intel to work on the Pentium® Pro and Pentium 4 processors. His e-mail address is mike.upton@intel.com.

Pre-Silicon Validation of Hyper-Threading Technology

David Burns, Desktop Platforms Group, Intel Corp.

Index words: microprocessor, validation, bugs, verification

ABSTRACT

Hyper-Threading Technology delivers significantly improved architectural performance at a lower-than-traditional power consumption and die size cost. However, increased logic complexity is one of the trade-offs of this technology. Hyper-Threading Technology exponentially increases the micro-architectural state space, decreases validation controllability, and creates a number of new and interesting micro-architectural boundary conditions. On the Intel Xeon processor family, which implements two logical processors per physical processor, there are multiple, independent logical processor selection points that use several algorithms to determine logical processor selection. Four types of resources: Duplicated, Fully Shared, Entry Tagged, and Partitioned, are used to support the technology. This complexity adds to the pre-silicon validation challenge.

Not only is the architectural state space much larger (see "Hyper-Threading Technology Architecture and Microarchitecture" in this issue of the *Intel Technology Journal*), but also a temporal factor is involved. Testing an architectural state may not be effective if one logical processor is halted before the other logical processor is halted. The multiple, independent, logical processor selection points and interference from simultaneously executing instructions reduce controllability. This in turn increases the difficulty of setting up precise boundary conditions to test. Supporting four resource types creates new validation conditions such as cross-logical processor corruption of the architectural state. Moreover, Hyper-Threading Technology provides support for inter- and intra-logical processor store to load forwarding, greatly increasing the challenge of memory ordering and memory coherency validation.

This paper describes how Hyper-Threading Technology impacts pre-silicon validation, the new validation challenges created by this technology, and our strategy for pre-silicon validation. Bug data are then presented and used to demonstrate the effectiveness of our pre-silicon Hyper-Threading Technology validation.

INTRODUCTION

Intel IA-32 processors that feature the Intel NetBurst microarchitecture can also support Hyper-Threading Technology or simultaneous multi-threading (SMT). Pre-silicon validation of Hyper-Threading Technology was successfully accomplished in parallel with the Pentium® 4 processor pre-silicon validation, and it leveraged the Pentium 4 processor pre-silicon validation techniques of Formal Verification (FV), Cluster Test Environments (CTEs), Architecture Validation (AV), and Coverage-Based Validation.

THE CHALLENGES OF PRE-SILICON HYPER-THREADING TECHNOLOGY VALIDATION

The main validation challenge presented by Hyper-Threading Technology is an increase in complexity that manifested itself in these major ways:

Project management issues

An increase in the number of operating modes: MT-mode, ST0-mode, and ST1-mode, each described in "Hyper-Threading Technology Architecture and Microarchitecture" in this issue of the *Intel Technology Journal*.

Hyper-Threading Technology squared the architectural state space.

A decrease in controllability.

An increase in the number and complexity of microarchitectural boundary conditions.

New validation concerns for logical processor starvation and fairness.

Microprocessor validation already was an exercise in the intractable engineering problem of ensuring the correct functionality of an immensely complex design with a limited budget and on a tight schedule. Hyper-Threading Technology made it even more intractable. Hyper-Threading Technology did not demand entirely new validation methods and it did fit within the already planned Pentium 4 processor validation framework of formal verification, cluster testing, architectural validation, and coverage-based microarchitectural validation. What Hyper-Threading Technology did require, however, was an increase in validation staffing and a significant increase in computing capacity.

Project Management

The major pre-silicon validation project management decision was where to use the additional staff. Was a single team, which focused exclusively on Hyper-Threading Technology validation, needed? Should all the current functional validation teams focus on Hyper-Threading Technology validation? The answer, driven by the pervasiveness, complexity, and implementation of the technology, was both. All of the existing pre-silicon validation teams assumed responsibility for portions of the validation, and a new small team of experienced engineers was formed to focus exclusively on Hyper-Threading Technology validation. The task was divided as follows:

Coverage-based validation [1] teams employed coverage validation at the microcode, cluster, and full-chip levels. Approximately thirty percent of the coded conditions were related to Hyper-Threading Technology. As discussed later in this paper, the use of cluster test environments was essential for overcoming the controllability issues posed by the technology.

The *Architecture Validation (AV)* [1] team fully explored the IA-32 Instruction Set Architecture space. The tests were primarily single-threaded tests (meaning the test has only a single thread of execution and therefore each test runs on one logical processor) and were run on each logical processor to ensure symmetry.

The *Formal Verification (FV)* team proved high-risk logical processor-related properties. Nearly one-third of the FV proofs were for Hyper-Threading Technology [1].

The *MT Validation (MTV)* team validated specific issues raised in the Hyper-Threading Technology architecture specification and any related validation area not covered by other teams. Special attention was paid to the cross product of the architectural state space, logical processor data sharing, logical processor forward progress, atomic operations and self-modifying code.

Operating Modes

Hyper-Threading Technology led to the creation of the three operating modes, MT, ST0, and ST1, and four general types of resources used to implement Hyper-Threading Technology. These resources can be categorized as follows:

Duplicated. This is where the resources required to maintain the unique architectural state of each logical processor are replicated.

Partitioned. This is where a structure is divided in half between the logical processors in MT-mode and fully utilized by the active logical processor in ST0- or ST1-mode.

Entry Tagged. This is where the overall structure is competitively shared, but the individual entries are owned by a logical processor and identified with a logical processor ID.

Fully Shared. This is where logical processors compete on an equal basis for the same resource.

Examples of each type of resource can be found in "Hyper-Threading Technology Architecture and Microarchitecture" in this issue of the *Intel Technology Journal*. Consider the operating modes state diagram shown in Figure 1.

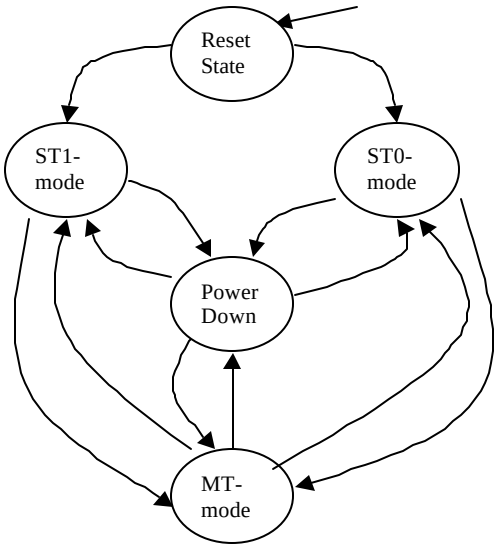


Figure 1: Operating Mode State Diagram

It can be used to illustrate test cases involving the three operating modes and how they affect the four types of resources. At the start of test, both logical processors are reset. After reset, the logical processors vie to become the boot serial processor. Assume logical processor 0 wins and the operating mode is now ST0. All non-duplicated resources are fully devoted to logical processor 0. Next, logical processor 1 is activated and MT-mode is entered. To make the transition from ST0- or ST1-mode to MT-mode, the partitioned structures, which are now fully devoted to only one logical processor, must be drained and divided between the logical processors. In MT-mode, accidental architectural state corruption becomes an issue, especially for the entry-tagged and shared resources. When a logical processor runs the hlt instruction, it is halted, and one of the ST-modes is entered. If logical processor 0 is halted, then the transition is made from MT-mode to ST1-mode. During this transition, the partitioned structures must again be drained and then recombined and fully devoted to logical processor 1. MT-mode can be re-entered if, for example, an interrupt or non-maskable interrupt (NMI) is sent to logical processor 0. The Power Down state is entered whenever the STP_CLK pin is asserted or if both logical processors are halted.

Now contrast this to a non-Hyper-Threading Technology-capable processor like the Intel Pentium 4 processor. For the Pentium 4 processor, there are only three states: Reset, Power Down, and Active, and four state transitions to validate. In addition, there is no need to validate the transitioning of partitioned resources from divided to and from combined.

Architectural State Space

The creation of three operating modes has a material impact on the amount of architectural state space that must be validated. As mentioned earlier, the AV team develops single-threaded tests that fully explore the IA-32 Instruction Set Architecture and architectural space. A single-threaded test has just one thread of execution, meaning that it can run on only one logical processor. A multi-threaded test has two or more threads of execution, meaning that it can run and use two or more logical processors simultaneously.

To validate both ST0-mode and ST1-mode, all AV tests need to be run on both logical processors. A possible solution to validating the micro-architectural state space might be to take all AV tests and simulate all combinations of them in MT-mode. This proved to be impractical and insufficient because one AV test might be much shorter than the other test so a logical processor is halted and an ST-mode is entered before the MT-mode architectural state is achieved. The practical problem is that while simulating all AV tests in one of the ST modes can be done regularly, simulating the cross-product of all AV tests was calculated to take nearly one thousand years [3]!

The solution was to analyze the IA-32 architectural state space for the essential combinations that must be validated in MT-mode.

A three-pronged attack was used to tackle the challenge of Hyper-Threading Technology micro-architectural state space:

All AV tests would be run at least once in both ST0- and ST1-mode. This wasn't necessarily a doubling of the required simulation time, since the AV tests are normally run more than once during a project anyway. There was just the additional overhead of tracking which tests had been simulated in both ST modes.

A tool, Mtmerge, was developed that allowed single-threaded tests to be merged and simulated in MT-mode. Care was taken to adjust code and data spaces to ensure the tests did not modify each other's data and to preserve the original intentions of the single-threaded tests.

The MTV team created directed-random tests to address the MT-mode architectural space. Among the random variables were the instruction stream types: integer, floating-point, MMX, SSE, SSE2, the instructions within the stream, memory types, exceptions, and random pin events such as INIT,

SMI, STP_CLK and SLP. The directed variables that were systematically tested against each other included programming modes, paging modes, interrupts, and NMI.

CONTROLLABILITY

The implementation of Hyper-Threading Technology used multiple logical processor selection points at various pipeline stages. There was no requirement that all selection points picked the same logical processor in unison. The first critical selection point is at the trace cache, which sends decoded micro-operations to the out-of-order execution engine. This selection point uses an algorithm that considers factors such as trace cache misses and queue full stalls. Hence, controllability can be lost even before reaching the out-of-order execution engine. In addition to the logical processor selection points, controllability is lost because uops from both logical processors are simultaneously active in the pipeline and competing for the same resources. The same test run twice on the same logical processor, but with different tests on the other logical processor used during both simulations, can have vastly different performance characteristics.

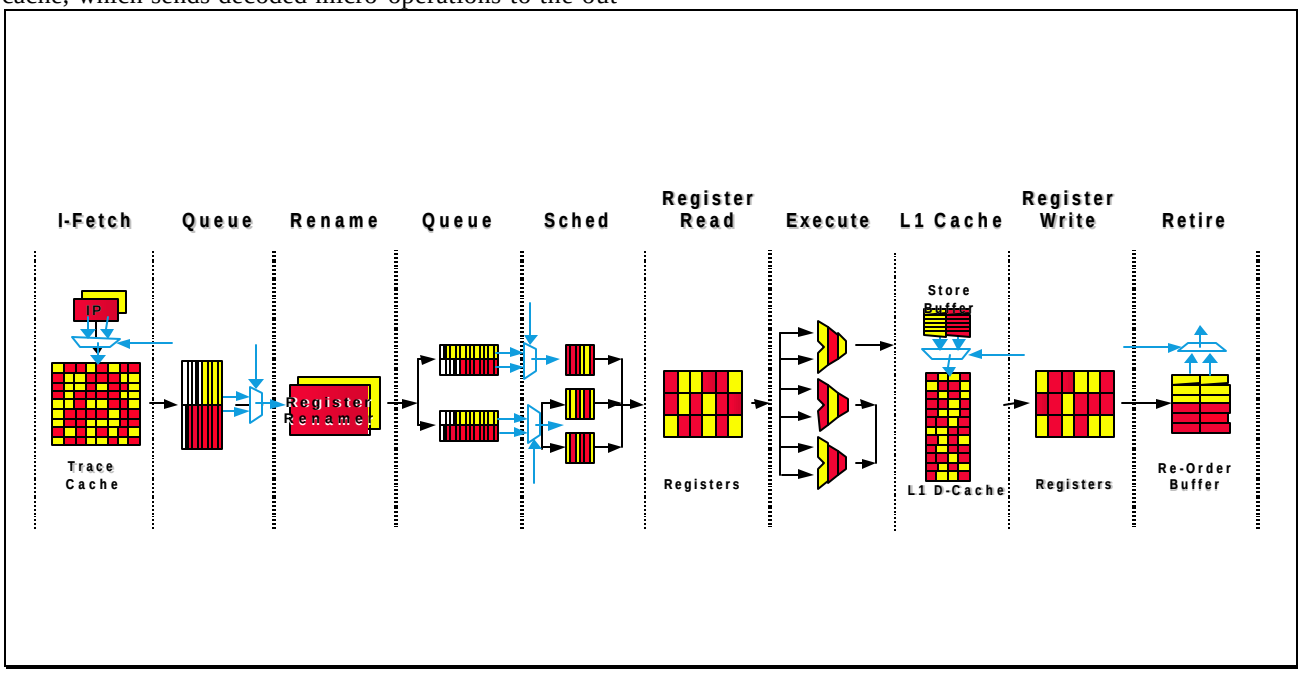


Figure 2: Logical processor selection point

Figure 2 shows some of the critical logical processor selection points and provides a glimpse into how interacting logical processors can affect their performance characteristics. The independent selection points coupled with the out-of-order, speculative execution, and speculative data nature of the microarchitecture obviously resulted in low controllability at the full-chip level. The solution to the low controllability was the use of the Cluster Test Environment [1] coupled with coverage-based validation at the CTE and full-chip levels.

The Cluster Test Environments allow direct access to the inputs of a cluster that helps alleviate controllability issues, especially in the backend memory and bus clusters. However, logical processor selection points and other complex logic are buried deep within the clusters. This meant that coverage-based validation coupled with directed-random testing was needed to ensure all interesting boundary conditions had been validated. Naturally, cluster interactions can be validated only at the full-chip level and again coverage-based validation and directed-random testing were used extensively.

Boundary Conditions

Hyper-Threading Technology created boundary conditions that were difficult to validate and had a large impact on our validation tool suite. Memory ordering validation was made more difficult since data sharing between logical processors could occur entirely within the same processor. Tools that looked only at bus traffic to determine correct memory ordering between logical processors were insufficient. Instead, internal RTL information needed to be conveyed to architectural state checking tools such as Archsim-MP, an internal tool provided by Intel Design Technology.

While ALU bypassing is a common feature, it becomes more risky when uops from different logical processors are executing together. Validation tested that cross-logical-processor ALU forwarding never occurred to avoid corruption of each logical processor's architectural state.

New Validation Concerns

Hyper-Threading Technology adds two new issues that need to be addressed: logical processor starvation and logical processor fairness. Starvation occurs when activity on one logical processor prevents the other from fetching instructions. Similar to starvation are issues of logical processor fairness. Both logical processors may want to use the same shared resource. One logical processor must not be allowed to permanently block the other from using a resource. The validation team had to study and test for all such scenarios.

BUG ANALYSIS

The first silicon with Hyper-Threading Technology successfully booted multi-processor-capable operating systems and ran applications in MT-mode. The systems ranged from a single physical processor with two logical processors, to four-way systems running eight logical processors. Still, there is always room for improvement in validation. An analysis was done to review the sources of pre-silicon and post-silicon bugs, and to identify areas for improving pre-silicon Hyper-Threading Technology validation.

To conduct the analysis of Hyper-Threading Technology bugs, it was necessary to define what such a bug is. A Hyper-Threading Technology bug is a bug that broke MT-mode functionality. While a seemingly obvious definition, such tests were found to be very good at finding ST-mode bugs. The bugs causing most MT-mode test failures were actually bugs that would break both ST-mode and MT-mode functionality. They just happened to be found first by multi-threaded tests. Every bug found from MT-mode testing was studied to understand if it would also cause ST-mode failures. The bugs of interest for this analysis were those that affected only MT-mode functionality. The bug review revealed the following:

Eight percent of all pre-silicon SRTL bugs were MT-mode bugs.

Pre-silicon MT-mode bugs were found in every cluster and microcode.

Fifteen percent of all post-silicon SRTL bugs were MT-mode bugs.

Two clusters [2] did not have any MT-mode post-silicon SRTL bugs.

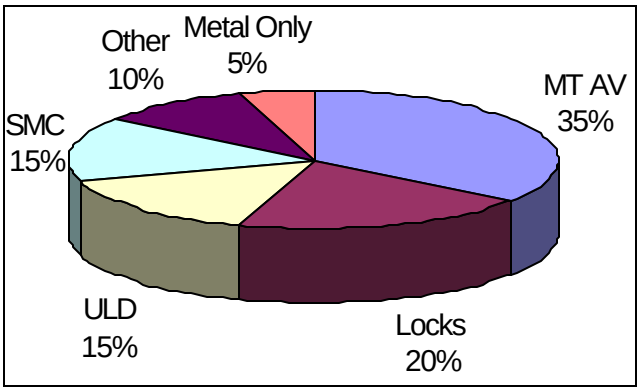


Figure 3: Breakdown of post-silicon MT-Mode bugs

Figure 3 categorizes the post-silicon MT-mode bugs into the functionality that they affected [2, 3]. Multi-Threading Architectural Validation (MT AV) bugs occurred where a particular combination of the huge cross product of IA-32 architectural state space did not function properly. Locks are those bugs that broke the functionality of atomic operations in MT-mode. ULD represents bugs involving logical processor forward progress performance degradation. Self-Modifying Code (SMC) bugs were bugs that broke the functionality of self or cross-logical processor modifying code. Other is the category of other tricky micro-architectural boundary conditions. Metal Only is an interesting grouping. We found that post-silicon MT-mode bugs were difficult to fix in metal only steppings and often required full layer tapeouts to fix successfully. Metal Only are the bugs caused by attempting to fix known bugs in Metal Only tapeouts.

IMPROVING MULTI-THREADING VALIDATION

Clearly, with MT-mode bugs constituting nearly twice the number of post-silicon bugs, 15% versus 8% of the pre-silicon bugs, coupled with the high cost of fixing post-silicon MT bugs (full layer versus metal tapeouts), there is an opportunity for improving pre-silicon validation of future MT-capable processors. Driven by the analysis of pre- and post-silicon MT-mode bugs [2, 3], we are improving pre-silicon validation by doing the following:

Enhancing the Cluster Test Environments to improve MT-mode functionality checking.

Increasing the focus on microarchitecture validation of multi-cluster protocols such as SMC, atomic operations, and forward progress mechanisms.

Increasing the use of coverage-based validation techniques to address hardware/microcode interactions in the MT AV validation space.

Increasing the use of coverage-based validation techniques at the full-chip level to track resource utilization.

Done mainly in the spirit of continuous improvement, enhancing the CTEs to more completely model adjacent clusters and improve checking will increase the controllability benefits of CTE testing and improve both ST- and MT-mode validation. Much of the full-chip microarchitecture validation (uAV) had focused on testing of cluster boundaries to complement CTE testing. While this continues, additional resources have been allocated to the multi-cluster protocols mentioned previously.

The MTV team is, for the first time, using coverage-based validation to track architectural state coverage. For example, the plan is to go beyond testing of interrupts on both logical processors by skewing a window of interrupt occurrence on both logical processors at the full-chip level. In addition, this will guarantee that both logical processors are simultaneously in a given architectural state.

The MTV team is also increasing its use of coverage to track resource consumption. One case would be the filling of a fully shared structure, by one logical processor, that the other logical processor needs to use. The goal is to use coverage to ensure that the desired traffic patterns have been created.

Nevertheless, these changes represent fine-tuning of the original strategy developed for Hyper-Threading Technology validation. The use of CTEs proved essential for overcoming decreased controllability, and the division of MT-mode validation work among the existing functional validation teams proved an effective and efficient way of tackling this large challenge. The targeted microarchitecture boundary conditions, resource structures, and areas identified as new validation concerns were all highly functional at initial tapeout. Many of the bugs that escaped pre-silicon validation could have been caught with existing pre-silicon tests if those tests could have been run for hundreds of millions of clock cycles or involved unintended consequences from rare interactions between protocols.

CONCLUSION

The first Intel microprocessor with Hyper-Threading Technology was highly functional on A-0 silicon. The initial pre-silicon validation strategy using the trinity of coverage-based validation, CTE testing, and sharing the validation work was successful in overcoming the complexities and new challenges posed by this technology. Driven by bug data, refinements of the original validation process will help ensure that Intel Corporation can successfully deploy new processors with Hyper-Threading Technology and reap the benefits of improved performance at lower die size and power cost.

ACKNOWLEDGMENTS

The work described in this paper is due to the efforts of many people over an extended time, all of whom deserve credit for the successful validation of Hyper-Threading Technology.

REFERENCES

[1] Bentley, B. and Gray, R., "Validating The Intel Pentium 4 Processor," *Intel Technology Journal Q1, 2001*at http://developer.intel.com/technology/itj/q12001/articles/art_3.htm.

[2] Burns, D., "Pre-Silicon Validation of the Pentium 4's SMT Capabilities," *Intel Design and Test Technology Conference, 2001,* Intel internal document.

[3] Burns, D., "MT Pre-Silicon Validation,`" *IAG Winter 2001 Validation Summit,* Intel internal document.

AUTHOR'S BIOGRAPHY

David Burns is the Pre-Silicon Hyper-Threading Technology Validation Manager for the DPG CPU Design organization in Oregon. He has more than 10 years of experience with microprocessors, including processor design, validation, and testing in both pre- and post-silicon environments. He has a B.S. degree in Electrical Engineering from Northeastern University. His e-mail address is david.w.burns@intel.com.

Speculative Precomputation: Exploring the Use of Multithreading for Latency

Hong Wang, Microprocessor Research, Intel Labs
Perry H. Wang, Microprocessor Research, Intel Labs
Ross Dave Weldon, Logic Technology Development Group, Intel Corporation
Scott M. Ettinger, Microprocessor Research, Intel Labs
Hideki Saito, Software Solution Group, Intel Corporation
Milind Girkar, Software Solution Group, Intel Corporation
Steve Shih-wei Liao, Microprocessor Research, Intel Labs
John P. Shen, Microprocessor Research, Intel Labs

Index words: cache misses, memory prefetch, precomputation, multithreading, microarchitecture

ABSTRACT

Speculative Precomputation (SP) is a technique to improve the latency of single-threaded applications by utilizing idle multithreading hardware resources to perform aggressive long-range data prefetches. Instead of trying to explicitly parallelize a single-threaded application, SP does the following:

- Targets only a small set of static load instructions, called *delinquent loads*, which incur the most performance degrading cache miss penalties.
- Identifies the dependent instruction slice leading to each delinquent load.
- Dynamically spawns the slice on a spare hardware thread to speculatively precompute the load address and perform data prefetch.

Consequently, a significant amount of cache misses can be overlapped with useful work, thus hiding the memory latency from the critical path in the original program.

Fundamentally, contrary to conventional wisdom that multithreading microarchitecture techniques can be used to only improve the throughput of multitasking workloads or the performance of multithreaded programs, SP demonstrates the potential to leverage multithreading hardware resources to exploit a form of implicit thread-level parallelism and significantly speed up single-threaded applications. Most desktop applications in the traditional PC environment are not otherwise easily parallelized to take advantage of multithreading resources.

This paper chronicles the milestones and key lessons from Intel's research on SP, including an initial simulation-based evaluation of SP for both in-order and out-of-order multithreaded microarchitectures. We also look at recent experiments in applying software-based SP (SSP) to significantly speed up a set of pointer-intensive applications on a pre-production version of Intel Xeon processors with Hyper-Threading Technology.

INTRODUCTION

Memory latency has become the critical bottleneck to achieving high performance on modern processors. Many large applications today are memory intensive, because their memory access patterns are difficult to predict and their working sets are becoming quite large. Despite continued advances in cache design and new developments in prefetching techniques, the memory bottleneck problem still persists. This problem worsens when executing *pointer-intensive* applications, which tend to defy conventional stride-based prefetching techniques.

One solution is to overlap memory stalls in one program with the execution of useful instructions from another program, thus effectively improving system performance in terms of overall *throughput*. Improving throughput of multitasking workloads on a single processor has been the primary motivation behind the emerging simultaneous multithreading (SMT) techniques [1][2][3]. An SMT processor can issue instructions from multiple hardware contexts, or *logical processors* (sometimes also called *hardware threads*), to the functional units of a super-scalar processor in the same cycle. SMT achieves higher overall throughput by increasing overall instruction-level parallelism available to the architecture via the exploitation of the natural parallelism between independent threads during each cycle.

However, this traditional use of SMT does not directly improve performance in terms of *latency* when only a single thread is executing. Since the majority of desktop applications in the traditional PC environment are single-threaded code, it is important to investigate if and how SMT techniques can be used to enhance single-threaded code performance by reducing latency.

At Intel Labs, extensive microarchitecture research efforts have been dedicated to discover and evaluate innovative hardware and software techniques to leverage multithreaded hardware resources to speed up single-threaded applications. One of the techniques is called Speculative Precomputation (SP), a novel thread-based cache prefetching mechanism. The key idea behind SP is to utilize otherwise idle hardware thread contexts to execute speculative threads on behalf of the main (non-speculative) thread. These speculative threads attempt to trigger future cache-miss events far enough in advance of access by the non-speculative thread that the memory miss latency can be masked. SP can be thought of as a special prefetch mechanism that effectively targets load instructions that exhibit unpredictable irregular or data-dependent access patterns. Traditionally, these loads have been difficult to handle via either hardware prefetchers [5][6][7] or software prefetchers [8].

In this paper, we chronicle several milestones we have reached including initial simulation-based evaluations of SP for both in-order and out-of-order multithreaded research processors [9][10][11][12][13][14], and highlight recent experiments in successfully applying software-based SP (SSP) to significantly speed up a set of pointer-intensive benchmarks on a pre-production version of Intel Xeon processors with the Hyper-Threading Technology.

We first recount the motivation for SP, and we introduce the basic algorithmic ingredients and key optimizations, such as chaining triggers, which ensure the effectiveness of SP. We then compare SP with out-of-order execution, the traditional latency tolerance technique, and shed light on the effectiveness of combining both techniques. We follow with a discussion of the trade-offs for hardware-based SP and software-based SP (SSP), and in particular, highlight an automated post-pass binary adaptation tool for SSP. This tool can achieve performance gains comparable to that of implementing SSP using hand optimization. We then describe recent experiments where SSP is applied to speed up a set of applications on a pre-production version of Intel Xeon processors with the Hyper-Threading Technology. Finally, we review related work.

SPECULATIVE PRECOMPUTATION: KEY IDEAS

Chronologically, the key ideas for Speculative Precomputation (SP) were developed prior to the arrival of silicon for the Intel® Xeon™ processors with Hyper-Threading Technology. Our initial research work on SP was conducted on a simulation infrastructure modeling a range of research Itanium™ processors that support Simultaneous Multithreading (SMT) with a pipeline configurable to be either in-order or out-of-order. Before we discuss the trade-offs for hardware- vs. software-based implementations of SP, our discussion will assume the research processor model described below in

Table 1. We use a set of benchmarks selected from SPEC2000 and the Olden suite, including *art*, *equake*, *gzip*, *mcf*, *health* and *mst*.

Table 1: Details of the research Itanium processor models

Pipeline Structure	In-order: 8-12-stage pipeline. Out-of-order: 12-16-stage pipeline.
Fetch	2 bundles from 1 thread, or 1 bundle from each of 2 threads.
Branch pred	2K-entry GSHARE. 256 entry 4-way

Expansion	Private, per-thread, in-order 8 bundle expansion queue
Register Files	Private, per-thread register files. 128 integer registers, 128 FP registers, 64 predicate registers, 128 application registers
Execute Bandwidth	In-order: 6 instructions from one thread or 3 instructions from each of 2 threads Out-of-order: 18-instruction schedule window
Cache Structure	L1 (separate I and D): 16K 4-way, 8-way banked, 1-2-cycle L2 (shared): 256K 4-way, 8-way banked, 7-14-cycle L3 (shared): 3072K 12-way, 1-way banked, 15-30-cycle Fill buffer (MSHR): 16 entries. All caches: 64-byte lines
Memory	115-230 cycle latency, TLB Miss Penalty 30 cycles.

Delinquent Loads

For most programs, only a small number of static loads are responsible for the vast majority of cache misses [15]. Figure 1 shows the cumulative contributions to L1 data cache misses by the top 50 static loads for the processor models in

Table 1 running benchmarks to completion. It is evident that a few poorly behaved static loads dominate cache misses in these programs. We call these loads *delinquent loads*.

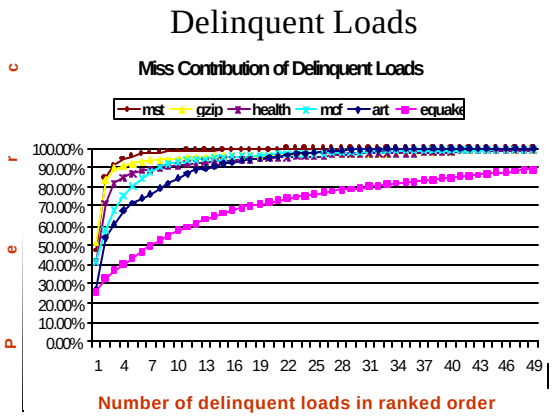


Figure 1: Cumulative L1 data cache misses due to delinquent loads

In order to gauge the impact of these loads on performance, Figure 2 compares the performance of a perfect memory subsystem, where all loads hit in the L1, to that of a memory subsystem that assumes the worst 10 delinquent loads always hitting in the L1 cache. In most cases, eliminating performance losses from only the top delinquent loads yields most of the speed-up achievable by the ideal memory. These data suggest that significant improvements can be achieved by just focusing latency-reduction techniques on the delinquent loads.

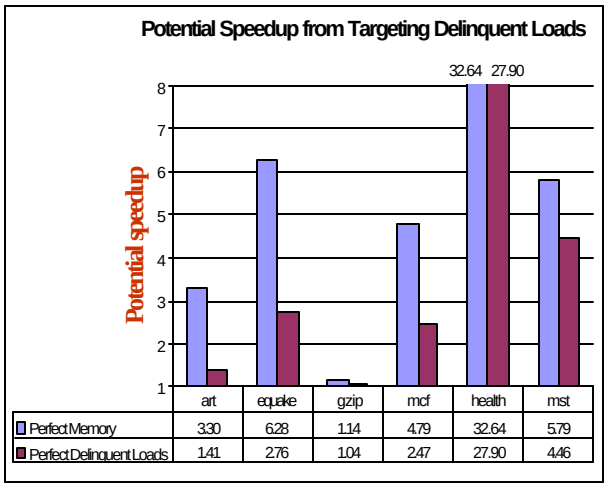


Figure 2: Speed-up when 10 delinquent loads are assumed to always hit in cache

SP Overview

To perform effective prefetch for delinquent loads, SP requires the construction of the *precomputation slices*, or p-slices, which consist of dependent instructions that compute the addresses accessed by delinquent loads. When an event triggers the invocation of a p-slice, a speculative thread is spawned to execute the p-slice. The speculatively executed p-slice then prefetches for the delinquent load that will be executed later by the main thread. Speculative threads can be spawned under one of two conditions: when encountering a *basic trigger*, which occurs when a designated instruction in the non-speculative thread is retired, or when encountering a *chaining trigger*, which occurs when a speculative thread explicitly spawns another.

Spawning a speculative thread entails allocating a hardware thread context, copying necessary live-in values into its register file, and providing the thread context with the address of the first instruction of the p-slice. If a free hardware context is not available, the spawn request is ignored.

Necessary live-in values are always copied into the thread context when a speculative thread is spawned. This eliminates the possibility of inter-thread hazards, where a

register is overwritten in one thread before a child thread has read it. Fortunately, as shown in Table 2, the number of live-in values that must be copied is very small.

Table 2: Slice statistics

Benchmark	Slices (#)	Average size (#inst)	Average # live-in
art	2	4	3.5
equake	8	12.5	4.5
gzip	9	9.5	6.0
mcf	6	5.8	2.5
health	8	9.1	5.3
mst	8	26	4.7

When spawned, a speculative thread occupies a hardware thread context until the speculative thread completes execution of all instructions in the p-slice. Speculative threads are not allowed to update the architectural state. In particular, stores in a p-slice are not allowed to update any memory state. For the benchmarks studied in this research, however, none of the p-slices include any store instructions.

SP Tasks

Several steps are necessary to employ SP: identification of the set of delinquent loads, construction of p-slices for these loads, and the establishment of triggers. In addition, upon dynamic execution with SP, proper control is necessary to ensure that the precomputation can generate timely and accurate prefetches. These steps can be performed by a variety of approaches including compiler assistance, hardware support, and a hybrid of both software and hardware approaches. These steps can be applied to any processor supporting SMT, regardless of differences in instruction set architectures (ISA) or pipeline organization. Different manifestations of SP are further discussed later in the paper.

Identify Delinquent Loads

The set of delinquent loads that contribute the majority of cache misses is determined through memory access profiling, performed either by the compiler or a memory access simulator [15], or by dedicated profiling tools for real silicon, such as the VTune Performance Analyzer [16]. From such profile analysis, the loads that have the largest impact on performance (i.e., incurring long latencies) are selected as delinquent loads. The total number of L1 cache misses can be used as the criterion to select delinquent loads, while other filters (e.g., L2 or L3 misses or total memory latency) could also be used. For example, in our simulation-based study, we use the L1 cache misses to identify the delinquent loads, while for our experiment on a pre-production version of the Intel Xeon processor with the Hyper-Threading Technology, we use L2 cache miss profiling from the VTune analyzer instead.

Construct and Optimize P-Slices

In this phase, a p-slice is created for each delinquent load. Depending upon the environment, the p-slice can be constructed by hand, via a simulator [11][13], by a compiler [14], or directly by hardware [12]. For example, a p-slice with a basic trigger can be captured via traditional backward slicing [17] within a window of dynamic instruction traces. By eliminating instructions that delinquent loads do not depend on, the resulting p-slices are typically of very small sizes, typically 5 to 15 instructions per p-slice. For p-slices with chaining triggers, a more elaborate construction process is required.

P-slices containing chaining triggers typically have three parts–a prologue, a spawn instruction for spawning another copy of the p-slice, and an epilogue. The prologue consists of instructions that compute values associated with a loop-carried dependency, i.e., those values produced in one loop iteration and used in the next loop iteration, such as updates to a loop induction variable. The epilogue consists of instructions that produce the address for the targeted delinquent load. The goal behind chaining trigger construction is for the prologue to be executed as quickly as possible, so that additional speculative threads can be spawned as quickly as possible.

To add chaining triggers to p-slices targeting delinquent loads within loops, the algorithm for capturing p-slices using basic triggers can be augmented to track the distance between different instances of a delinquent load. If two instances of the same p-slice are consistently spawned within a fixed-sized window of instructions, we create a new p-slice that includes a chaining trigger that targets the same delinquent load. Instructions from one

slice that modify values used in the next p-slice are added to the prologue. Instructions that are necessary to produce the address loaded by the delinquent load are added to the epilogue. Between the prologue and epilogue, a spawn instruction is inserted to spawn another copy of this same p-slice.

Condition Precomputation

To be effective, SP-based prefetches must be accurate and timely. By accuracy, we mean a p-slice upon spawning should use mostly valid live-in values to produce a correct prefetch address. By timeliness, we mean the speculative threads performing the SP prefetch thread should run neither behind nor too far ahead of the main non-speculative thread.

For accuracy, if spawning of the speculative thread is done only after its corresponding trigger reaches the commit stage of the processor pipeline, then the live-in values of the associated p-slice are usually guaranteed to be architecturally correct, thus ensuring precomputation will produce the correct prefetch address. An alternative policy might attempt to spawn as soon as the trigger instruction is detected at the decode stage of the pipeline. The drawback of such an early spawning scheme is that both the trigger and the live-in values are speculative and prefetching from the wrong address can occur.

For timeliness, basic trigger by definition is more sensitive to how far it is between the trigger and the target delinquent load and how long the p-slice is, since the thread spawning is tightly coupled to progress made by the main thread. Any overhead associated with thread spawning will not only reduce the headroom for prefetch but also incur additional latency on the main thread.

The use of chaining, while decoupling thread spawning from progress made by the main thread, could potentially be overly aggressive in getting too far ahead and evicting useful data from the cache before the main thread has accessed them. To condition the run-ahead distance between the main thread and the SP threads, a structure called an *Outstanding Slice Counter* (OSC), is introduced to track, for a subset of distinct delinquent loads, the number of speculative threads that have been spawned relative to the number of instances of a delinquent load that have not yet been retired by the non-speculative thread. Each entry in the OSC tracking structure contains a counter, the instruction pointer (IP) of a delinquent load and the address of the first instruction in a p-slice, which identifies the p-slice. This counter is decremented when the non-speculative thread retires the corresponding delinquent load, and is incremented when the corresponding p-slice is spawned. When a speculative thread is spawned for which the entry in the OSC is negative, the resulting speculative thread is forced to wait in the pending state until the counter becomes positive, during which time it is not considered for assignment to a hardware thread context.

As we will see later, the controlling mechanism can also be implemented entirely in software as part of the speculative thread.

SP Trade-offs

One of the key findings in our SP research is that the chaining trigger, assuming fairly conservative hardware support but with a proper conditioning mechanism, can be much more effective than the basic trigger even assuming ideal hardware support. The trade-offs between the basic trigger and the chaining trigger can be summarized as follows.

Basic Trigger With Ideal Hardware Assumption

Figure 3 shows the performance gains achieved through two rather ideal SP configurations. One is more aggressive in that speculative threads are spawned from the non-speculative thread at the rename stage, but only by an instruction on the correct control flow path using oracle knowledge. The other is a less aggressive one, in that speculative threads are spawned only at the commit stage, when the instruction is guaranteed to be on the correct path. In both cases, we assume aggressive and ideal hardware support for directly copying live-in values from the non-speculative parent thread's context to its child thread's context, i.e., one-cycle *flash-copy* of live-in values. This allows the speculative thread to begin execution of a p-slice just one cycle after it is spawned.

For each benchmark, results are grouped into three pairs, corresponding to, from left to right, 2, 4, and 8 total hardware thread contexts. Within each pair, the configuration on the left corresponds to spawning speculative threads in the rename stage, while the configuration on the right corresponds to spawning in the commit stage as described above.

Basic Trigger Without Ideal Hardware Assumption

We propose a more realistic implementation of SP, which performs thread spawning after the trigger instruction is retired and assumes overhead, such as potential pipeline flush and multiple-cycle transfer of live-in values across threads via memory. This approach differs from the idealized hardware approach in two ways. First, spawning a thread is no longer instantaneous. It will slow down the non-speculative thread, due to the need to invoke and execute the handler code to check hardware thread availability and copy out live-in values to memory to prepare for cross-thread transfer. At the very minimum, invoking this handler requires a pipeline flush. The

second difference is that p-slices must be modified with a prologue to first load their live-in values from the transfer memory buffer, thus delaying the beginning of precomputation.

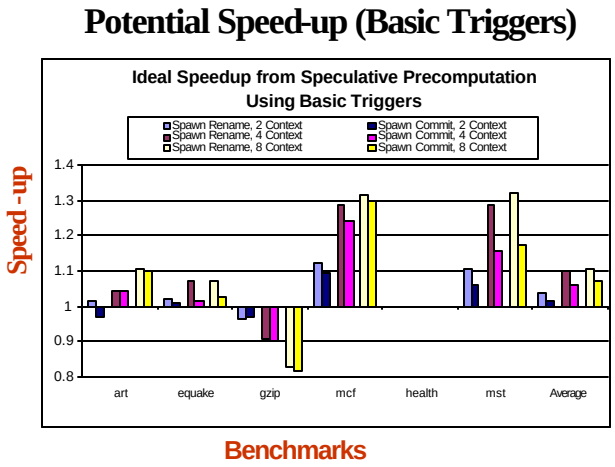


Figure 3: SP speed-up with basic trigger and ideal hardware assumptions

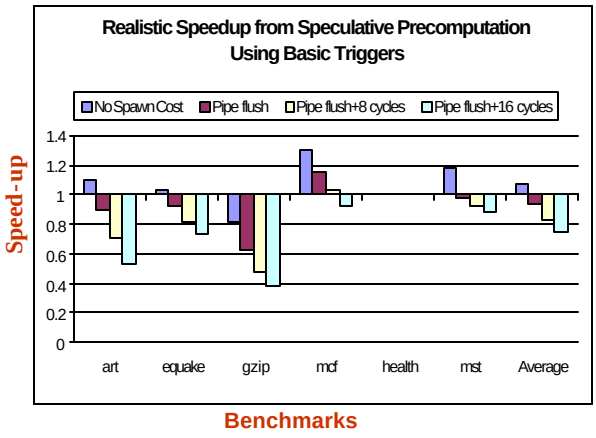


Figure 4: SP speed-up with basic trigger and realistic hardware

Figure 4 shows the performance speed-ups achieved when this more realistic hardware is assumed for a processor with eight hardware thread contexts. Four processor configurations are shown, each corresponding to differing thread-spawning costs. The leftmost configuration is given for reference, in which speculative threads are spawned with no penalty for the non-speculative thread, but must still perform a sequence of load instructions to read their live-in values from the memory transfer buffer. This configuration yields the highest possible performance because the main thread is still instantaneous in spawning a speculative thread. In the other three configurations, spawning a speculative thread causes the non-speculative thread's instructions following the trigger to be flushed from the pipeline. In the configuration second from the left, this pipeline flush is the only penalty, while in the third and fourth configurations, an additional penalty of 8 and 16 cycles, respectively, is assumed for the cost of executing the handler code to perform the live-in transfer.

Comparing these results to the performance of SP with ideal hardware (see Figure 3), the results for realistic SP in Figure 4 are rather disappointing. The primary reason that this performance falls short of that in the ideal case is the overhead incurred when the non-speculative thread spawns speculative threads. Specifically, the penalty of pipeline flush and the cost of performing live-in spill instructions in the handler both negatively affect the performance of the non-speculative thread.

Chaining Trigger

Figure 5 shows the speed-up achieved from realistic SP using chaining triggers as the number of thread contexts is varied. We assume that a thread spawning incurs a pipeline flush and an additional penalty of 16 cycles. Chaining triggers make effective use of available thread contexts when sufficient memory parallelism exists, resulting in impressive average performance gains of 51% with four threads and 76% with eight threads.

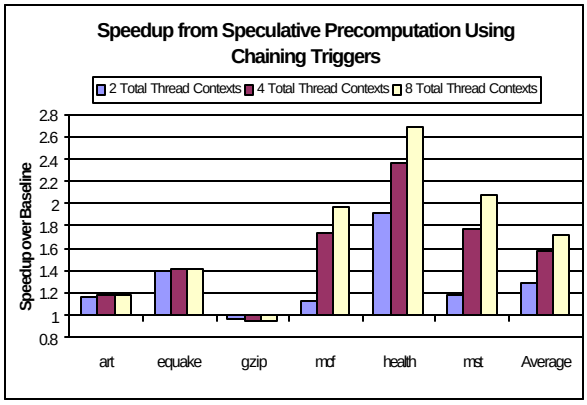


Figure 5: SP speed-up with chaining trigger and realistic hardware

Most noticeable is *health*. Though it does not benefit significantly from basic triggers (as shown in Figure 4) the speed-up is boosted to 169%, when using chaining triggers.

Figure 6 shows which level of the memory hierarchy is accessed by delinquent loads under three processor configurations: the baseline processor without use of SP, a processor with 8 thread contexts that uses basic triggers,

and a processor with 8 thread contexts that uses both basic and chaining triggers.

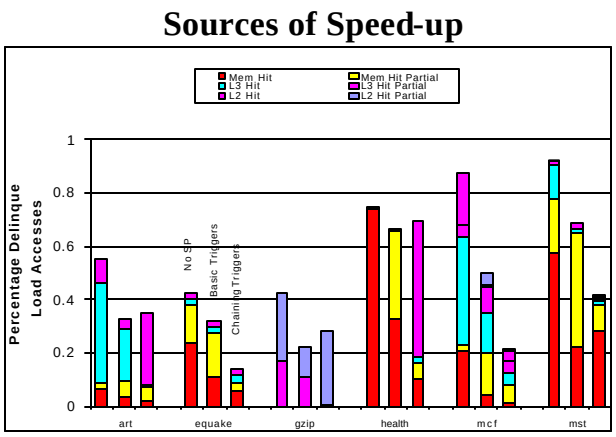


Figure 6: Reduction of cache misses in the memory hierarchy via SP-based prefetching

In general, basic triggers provide high accuracy but fail to significantly impact the number of loads that require access to the main memory. Even though basic triggers can be effective in targeting delinquent loads with relatively low latency, such as L1 misses, they are not likely to significantly help prefetch cache misses to main memory in a timely manner.

Chaining triggers, however, can achieve higher coverage and prefetch data in a much more timely manner, even for data that require access to the main memory. This is due to the chaining trigger's ability to effectively target delinquent loads and perform prefetches significantly far ahead of the non-speculative thread.

MEMORY LATENCY TOLERANCE: SP VS. OOO

Before the advent of thread-based prefetch techniques like SP, out-of-order (OOO) execution [18][19][20] has been the primary microarchitecture technique to tolerate cache miss latency. With the register renamer and reservation stations, an OOO processor is able to dynamically schedule the in-flight instructions, and execute those instructions independent of the missing loads, while the misses are being served.

Fundamentally, both OOO and SP aim to hide memory latency by overlapping instruction execution with the service to outstanding cache misses. OOO tries to overlap the outstanding cache-miss cycles by finding independent instructions after the missing load and executing them as early as possible, while SP prefetches for the delinquent loads far ahead of the non-speculative thread, thus overlapping future cache misses with the current execution of the non-speculative thread.

While both SP and OOO can reduce the data cache miss penalty incurred on the program's critical path, they differ in the targeted memory access instructions and the effectiveness for different levels of the cache hierarchy. On the one hand, while OOO can potentially hide the miss penalty for all load and store instructions to all layers of the cache hierarchy, it is most effective in tolerating L1 miss penalties. But for misses on L2 or L3, OOO may have difficulty in finding sufficient independent instructions to execute and overlap the much longer cache-miss latency. On the other hand, SP by design targets only a small set of delinquent loads that incur cache misses all the way to the memory.

To quantify the difference between SP and OOO, using the research processor models in Table 1, we evaluate two sets of benchmarks, one representing CPU-intensive workloads, including *gap*, *gzip* and *parser*, from SPEC2000Int, and the other representing memory-access-intensive workloads, including *equake* from SPEC2000fp, *mcf* from SPEC2000int, and *health* from the Olden suite.

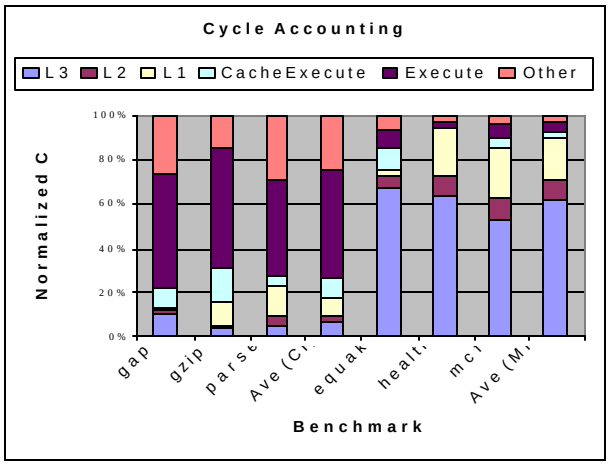


Figure 7: Characteristics of CPU-intensive vs. memory-intensive workloads on an in-order machine

Figure 7 depicts the cycle breakdown of these benchmarks on the in-order baseline processor. A cycle is assigned to *L1*, *L2*, and *L3* when the memory system is busy servicing the miss at the respective layer of cache hierarchy. *Execute* indicates that the processor issues an instruction for execution while the memory system is idle. Finally, *CacheExecute* shows the overlapping of cache misses with instruction execution. Clearly, the compute-intensive benchmarks spend most of their time in *Execute* while the memory-intensive benchmarks spend their time in waiting for cache misses.

Figure 8 shows speed-ups over the baseline model achieved by each of the two memory-tolerance techniques and by a combination of the two. The OOO processor model has four additional pipe stages to account for the increased complexity. Furthermore, SP assumes the use of chaining triggers and support for conditional precomputation.

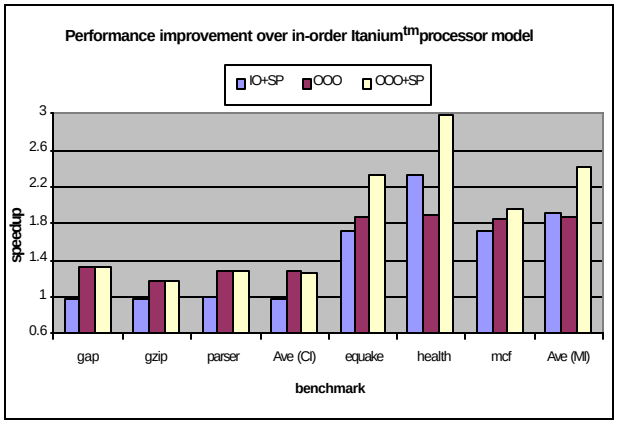


Figure 8: Speed-ups of in-order+SP, OOO, OOO+SP over in-order

Figure 9 further shows the cycle breakdown normalized to the in-order execution. This allows us to dissect where the speed-ups come from in terms of contributions leading to latency reduction.

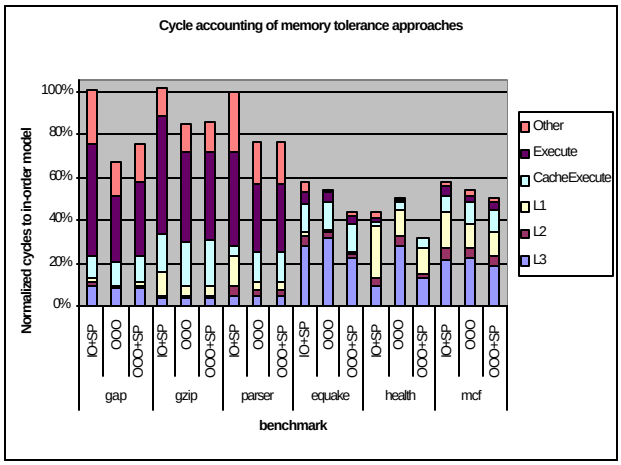


Figure 9: Cycle breakdown of in-order+SP, OOO and OOO+SP relative to in-order (100%)

The key findings can be summarized as follows.

OOO vs. SP

As shown in Figure 8, for memory-intensive workloads, the SP-enabled in-order SMT processor, albeit targeting only up to the top ten most delinquent loads that miss frequently in the L2 or L3 caches, can achieve slightly better speed-up than OOO. As shown in Figure 9, the speed-up is due to the reduction of the miss penalty at different levels of the cache hierarchy. For example, for *health*, OOO reduces the L3 cycle count from 62% in the baseline in-order to 28%, while SP achieves an even bigger reduction, down to 9%.

However, for compute-intensive benchmarks, SP can actually degrade performance. This is because for these benchmarks, almost all the delinquent loads that miss L1 hit in L2 and leave little headroom for the SP threads to run ahead and produce timely prefetches. In addition, spawning threads increase resource contention with the main thread and potentially can induce slowdowns in the main thread as well.

However, OOO is able to tolerate cache misses at all levels of the cache hierarchy and tolerate long latency executions on functional units. For instance, for *parser*, OOO can achieve a 10% reduction in the L1 cache stall cycles, and an even larger reduction of 12% in the execution cycles accounted by *Execute*. Furthermore, *CacheExecute*, the portion accounting for overlapping between cache servicing and execution, also increases by 9%.

Combination of OOO and SP

As shown in Figure 8, for compute-intensive benchmarks, SP does not bring about any speed-up beyond using OOO alone.

For memory-intensive benchmarks, however, the effectiveness of combining SP with OOO depends on the benchmarks. For *health*, if used individually, the OOO and SP approaches can achieve about a 131% and 90% speed-up, respectively. Together the two approaches achieve a near additive speed-up of 198%, demonstrating a potential complementary effect between the two approaches. Data in Figure 9 further shed light on the cause behind this effect. For *health*, SP alone can reduce *L3* cycles to 9% without improving *L1*, and OOO alone can reduce *L1* to 11% with a relatively smaller reduction in *L3*. By attacking both L1 and L3 cache misses, SP and OOO used in combination can achieve an overall reduction for both *L1* and *L3*. This is the root of the complementary effect between OOO and SP, where each covers cache misses at relatively disjointed levels of the cache hierarchy. Another interesting observation is that on the SP-enabled OOO processor, almost all instruction executions are overlapped with memory accesses, a desired effect of memory tolerance techniques.

For *mcf*, comparing the SP-enabled in-order execution (a.k.a. in-order+SP) with OOO in Figure 9, a relatively smaller difference exists between cycle counts in each

corresponding category. This is a clear indication of overlapping, whose root cause is the fact that SP and OOO redundantly cover the delinquent loads in the loop body.

A key to effectively utilizing SP on an OOO processor is to avoid overlapping the efforts of these two approaches. In particular, in typical memory-intensive loops, lengthy loop control that contains pointer chasing usually is on the critical path for the OOO processor. Loop control consists of instructions that resolve loop-carried dependencies and compute the induction variables for the next loop iteration. Once such a computation in the loop control is completed, independent instructions across multiple iterations can be effectively executed to tolerate cache misses incurred in the loop body of a particular iteration. A good combination of SP and OOO is to judiciously apply SP to perform prefetches for the critical loads in the loop control while letting OOO handle delinquent loads in the loop body. Then complementary benefits can be achieved, as shown in the case of *health*.

HARDWARE-ONLY SPECULATIVE PRECOMPUTATION VS. SOFTWARE-ONLY SPECULATIVE PRECOMPUTATION

The basic steps and algorithmic ingredients for Speculative Precomputation (SP) can be implemented in a gamut of techniques ranging from a hardware-only [12] approach to a software-only approach [14], in addition to the hybrid approaches originally studied in [11][13].

At one end of the spectrum, in close collaboration with Professor Dean Tullsen's research team at the University of California at San Diego, we investigated the hardware-only approach, called Dynamic Speculative Precomputation (DSP), a run-time technique that employs hardware mechanisms to identify a program's delinquent loads and generate precomputation slices to prefetch them. Like thread-based prefetching, the prefetch code is decoupled from the main program, allowing much more flexibility than traditional software prefetching. Like hardware prefetching, DSP works on legacy code and does not sacrifice software compatibility with future architectures and can operate on dynamic information rather than static to initiate prefetching and to evaluate the effectiveness of a prefetch. But unlike the software approaches, speculative threads on DSP are constructed, spawned, enhanced, and possibly removed by hardware. Both basic trigger- and chaining trigger-based p-slices can be efficiently constructed using a back-end structure off the critical path. Even with minimal p-slice optimization, a speed-up of 14% can be achieved on a set of various memory-limited benchmarks. More aggressive p-slice optimizations yield an average speed-up of 33%.

Interestingly, even in a multiprogramming environment where multiple non-speculative threads execute, if SP is applied to the worst behaving loads in the machine, regardless of which thread they belong to, the overall throughput can actually be improved, even if only one of the threads benefits directly from SP. In other words, though SP is originally intended to reduce the latency of a single-threaded application, it can also contribute to throughput improvement in a multiprogramming environment.

At the other end of the spectrum, we developed a post-pass compilation tool [14] that facilitates the automatic adaptation of existing single-threaded binaries for SSP on a multithreaded target processor without requiring any additional hardware mechanisms. This tool has been implemented in Intel's IPF production compiler infrastructure and is able to accomplish the following tasks:

1) Analyze an existing single-thread binary to generate prefetch threads.
2) Identify and embed triggering points in the original binary code.
3) Produce a new binary that has the prefetch threads attached, which can be spawned at run time.

The execution of the new binary spawns the prefetch threads, which are executed concurrently with the main thread. Initial results indicate that the prefetching performed by the speculative threads can achieve significant speed-ups on an in-order processor, ranging from 16% to 104%, on pointer-intensive benchmarks. Furthermore, the speed-ups achieved using the automated binary-adaptation tool loses at most 18% of the speed-up relative to that produced by hand-generated SSP code on the same processor.

To our knowledge, this is the first time that such an automated binary-adaptation tool has been implemented and shown to be effective in accomplishing the entire process of extracting dependent instructions leading to target operation, identifying proper spawning points, and managing inter-thread communication to ensure timely pre-execution leading to effective prefetches.

SPECULATIVE PRECOMPUTATION ON THE INTEL® XEON® PROCESSOR WITH HYPER-THREADING TECHNOLOGY

With the arrival of silicon for the Intel Xeon processor with Hyper-Threading Technology, it is of great interest to try out our Speculative Precomputation (SP) ideas on a real physical computer, since, thus far, our techniques have been primarily developed on simulation-based research processor models. Within just a few weeks of getting a system with a pre-production version of the Intel Xeon processor with Hyper-Threading Technology, we were able to come up with a crucial set of insights and innovative techniques to successfully apply software-only SP (SSP) to a small set of pointer-intensive benchmarks via hand adaptation of the original code. As shown in Table 3, significant performance boosts were achieved. The range of speed-ups per benchmark is due to the use of different inputs. This result was first disclosed in the 2001 Microprocessor Forum [2] where the details of Intel's Hyper-Threading Technology were originally introduced.

Benchmark	Description	Speed-up
Synthetic	Graph traversal in large random graph simulating large database retrieval	22% - 45%
MST (Olden)	Minimal Spanning Tree algorithm used for data clustering	23% - 40%
Health (Olden)	Hierarchical database modeling health care system	11% - 24%
MCF (SPEC2000int)	Integer programming algorithm used for bus scheduling	7.08 %

Table 3: Initial performance data: SP on a pre-production version of an Intel® Xeon™ processor with Hyper-Threading Technology

The silicon used in our experiment is the first generation implementation of Hyper-Threading Technology. The chip provides two hardware thread contexts and runs under Microsoft's Windows XP Operating System optimized for Hyper-Threading Technology. The two hardware contexts are exposed to the user as two symmetric multiprocessing logical processors. The on-chip cache hierarchy has the same configuration as the commercially available Intel Pentium® 4 processor in the 2001 timeframe. The entire on-chip cache hierarchy is shared between two hardware threads. There is no special hardware support for SP on this chip. In the following subsections, we use a pseudo-code of the synthetic benchmark in Table 3 as an example to highlight the methodology of applying SSP.

Figure 10 shows the pseudo-code for this microbenchmark. Figure 11 and Figure 12 illustrate the pseudo-code for both the main thread and the SP prefetch worker thread.

```
1 main()
  {
2   n = NodeArray[0]
3   while(n and remaining)
    {
4     work()
5     n->i = n->next->j + n->next->k + n->next->l
6     n = n->next
7     remaining--
    }
  }
```

Line 4: 49.47% of total execution time
Line 5: 49.46% of total execution time
Line 5: 99.95% of total L2 misses

Figure 10: Pseudo-code for single-thread code and the delinquent load profile

Like the general SP tasks described earlier, our experiment consists of methodologies for identification of delinquent loads, construction of SP threads, embedding of SP triggers, and a mechanism enabling live-in state transfer between the main thread and the speculative thread.

The identification of delinquent loads can be performed with the help of Intel's VTune™ Performance Analyzer 6.0 [16]. For instance, as shown in Figure 10, the pointer de-referencing loads originated at Line 5 are identified as delinquent with regard to L2 misses, and they incur significant latency.

Without explicit hardware support for thread spawning, for inter-thread communication and state transfer, we use standard Win32 thread APIs. **`CreateThread()`** is used to create the SP thread at initialization, **`SetEvent()`** is used to embed a basic trigger inside the main thread, and **`WaitForSingleObject()`** is used in the SP prefetch thread to implement the event-driven activation inside the corresponding speculative thread. In addition, we use global variables as a medium to explicitly implement inter-thread state transfer, where the main thread is responsible for copying out the live-in values before signaling a trigger event (using **`SetEvent()`**). The SP prefetch thread is responsible for copying in the live-in values prior to performing pointer-chasing prefetches.

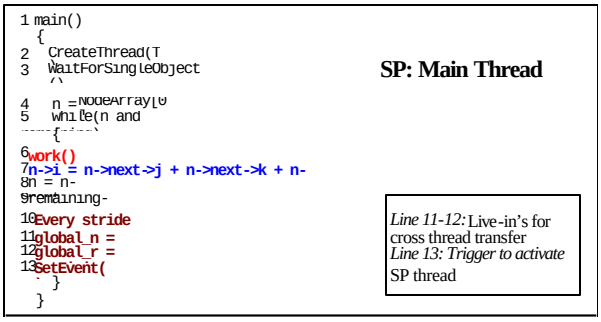


Figure 11: SP main thread pseudo-code

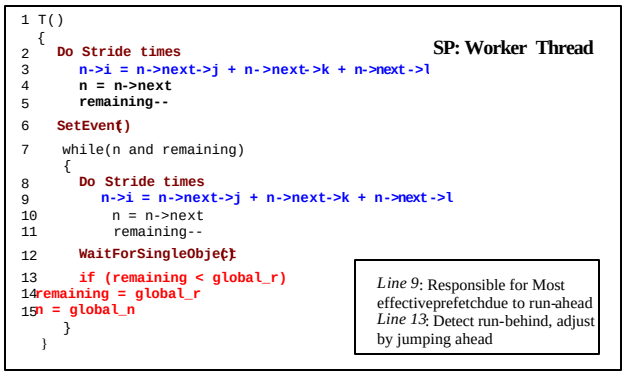


Figure 12: SP Prefetch worker thread–pseudo-code

Furthermore, as shown in Figure 12, a simple yet extremely important mechanism is used to implement SP conditioning inside the SP prefetch worker thread. This mechanism effectively ensures the SP prefetch worker thread performs the following two essential steps.

1. Upon each activation, it always runs a set of "stride" iterations of pointer chasing independent from the main thread.

   It is important to note that the pointer chasing loop bounded by "`stride`" effectively realizes a chaining trigger mechanism, since the progress can be made across multiple iterations independent of the main thread's progress.

2. After completing each set of "`stride`" iterations, it always monitors the progress made by the main thread to determine whether it is behind.

   If running behind, the SP thread will try to catch up with the main thread by synchronizing the global pointer.

   In addition, conditioning code can be introduced to detect if the SP thread is running too far ahead. The thread local variables "`remaining`" within both the main thread and the SP worker thread, are essentially trip counts recording their respective progress.

It is interesting to note that the SP worker thread uses only a regular load instruction and achieves effective prefetch for the main thread without actually using any literal prefetch instruction.

To do a fair comparison of performance, we use the Win32 API routine **`timeGetTime()`** to measure and compare the absolute wall clock execution time of the original code and the SSP-enabled code, both built for maximum speed optimizations using the Intel IA-32 C/C++ compiler [35]. For the example microbenchmark, Figure 13 summarizes the reason why SSP-enabled code runs faster, using profiling information from the VTune Performance Analyzer 6.0 [16]. In short, the SP thread is able to prefetch successfully most cache misses for the identified delinquent loads. This optimization brings about a 22% – 45% speed-up for a range of input sizes.

Main Thread
Line 7 corresponds to *Line 5* of single thread code
- Execution time:
  19% vs 49.46% in single-thread code
- L2 miss:
  0.61% vs 99.95% in single-thread code

SP worker thread:
Line 9:
- Execution time:
  26.21%
- L2 miss:
  97.61%

SP successful in shouldering most L2 cache misses

Figure 13: Why SSP-enabled code runs faster

Other brands may be claimed as the property of others.

This successful experiment not only serves to corroborate insights and benefits of SP learned from our earlier studies, which were based on simulation, but also convincingly demonstrates an alternative way to effectively use multithreading processor resources, i.e., exploit a pseudo form of "thread-level parallelism" within the single-threaded application, and use multithreading hardware to reduce its latency.

RELATED WORK

Earlier ideas exploring speculative threads to achieve benefits of cache prefetches include Chappel et al., Simultaneous Subordinate Microthreading (SSMT) [27]; Sundaramoorthy et al., Slipstream Processors [28]; and Song et al., Assisted Execution [29].

Along with our research on SP [9][10][11][12][13][14], several thread-based prefetch paradigms have recently been proposed, including Zilles and Sohi's Speculative Slices [21], Roth and Sohi's Data Driven Multithreading [22], Luk's Software Controlled Pre-Execution [23], Annavaram et al., Data Graph Precomputation [24], and Moshovos et al., Slice-Processors [25]. Most of these techniques are equivalent to the basic trigger SP mechanism.

As pointed out by Roth et al. in [30], these thread-based prefetching approaches are in effect performing a logical form of access execute decoupling as originally envisioned by Smith in [31] and further studied in [32][33][34]. Instead of assuming a dedicated decoupled memory access engine, the access function is carried out by the prefetching SP threads. Using the post-pass SSP tool, special "access" threads are attached to the original code. "Access" and "Execute" threads are performed and overlapped ("pipelining") on distinct hardware thread contexts in a general-purpose SMT or CMP processor.

What distinguishes our research from other research in this area includes the discovery of the chaining trigger mechanism; in-depth analysis of trade-offs between different memory tolerance techniques, especially SP and OOO; a fully automated post-pass compilation tool for binary adaptation to enable SSP; and the physical experiment successfully demonstrating that using SSP on real hardware enabled with Hyper-Threading Technology can bring about significant speed-up for single-threaded benchmarks.

CONCLUSION

In this paper we examine key milestones from Intel's research on Speculative Precomputation (SP), a technique that allows a multithreaded processor to use spare hardware contexts to spawn speculative threads to prefetch data well in advance of the main thread. Fundamentally, our research demonstrates Simultaneous Multithreading (SMT) processor resources can be used effectively to reduce the latency and enhance the performance of single-threaded applications.

Instead of relying on the existence of a multitasking or multiprogramming workload environment in which many threads run simultaneously on SMT processors to achieve better throughput, SP is geared towards latency reduction by extracting assist threads out of the targeted single-threaded application itself. One insight about SP is that the potential performance gain is dictated by the reduction of cache miss latency (which is likely to get worse as clock frequency increases) and not by the increased instruction execution throughput in an SMT processor. Executing a small number of instructions of an SP thread can result in latency reduction far greater than the latency required to execute the SP thread. In traditional multithreading of an application, the potential speed-up is bounded by the number of instructions that can be executed in the additional thread context.

As explained in [2], the arrival of Intel's Hyper-Threading Technology on the Intel Xeon processor marks the beginning of a new era: the transition from instruction-level parallelism (ILP) to thread-level parallelism (TLP). Multithreading techniques can help both power and complexity efficiency in future microarchitecture designs. It is of great interest to us to continue to look for alternate (and potentially better) use of multithreading resources. To summarize: Speculative Precomputation (SP) in effect leverages resources intended for thread-level parallelism (TLP) to achieve greater memory-level parallelism (MLP). This in turn significantly improves the effective instruction-level parallelism (ILP) of traditional single-threaded applications.

ACKNOWLEDGMENTS

The Speculative Precomputation research team has received tremendous support from Justin Rattner, Intel Fellow and Director of Microprocessor Research in Intel Labs (formerly MRL), and Richard Wirt, Intel Fellow and general manager, Software Solution Group (SSG). Individuals from various organizations who have provided critical support include Kevin J. Smith, David Sehr, Wilfred

Pinfold, Jesse Fang, George K Chen, Gerolf Hoflehner, Dan Lavery, Wei Li, Xinmin Tian, Sanjiv Shah, Ernesto Su, Paul Grey, Ralph Kling, Jim Dundas, Wen-hann Wang, Yong-fong Lee, Ed Grochowski, Gadi Ziv, Shalom Goldenberg, Shai Satt, Ido Shamir, Per Hammarlund, Debbie Marr, John Pieper, Bo Huang, Young Wang, Rob Portman, Pete Andrews, Tony Martinez, Oren Gershon, Jonathan Beimel, Ady Tal, Yigal Zemach, and Leonid Baraz.

Professor Dean M. Tullsen, and his research team at UC San Diego, have been our closest collaborators throughout our research into Speculative Precomputation. In particular, Mr. Jamison D Collins, a Ph.D. candidate of Professor Tullsen's, has made significant contributions.

Finally, we thank the referees of this paper for their useful suggestions. We are grateful to Lin Chao, Judith Anthony and Marian Lacey for their extremely careful editing.

REFERENCES

[1] J. Emer, "Simultaneous Multithreading: Multiplying Alpha's Performance," *Microprocessor Forum*, Oct 1999.

[2] G. Hinton and J. Shen, "Intel's Multi-Threading Technology," *Microprocessor Forum*, October 2001.

[3] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *22nd ISCA*, June 1995.

[4] D. M. Tullsen and J. A. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreading Processor," in *Micro-34*, Dec. 2001, pp. 318-327.

[5] T. Chen, "An Effective Programmable Prefetch Engine for On-chip Caches," In *Micro-28*, pp 237-242, Dec. 1995.

[6] N. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fully associative Cache and Prefetch Buffers," in *ISCA-17*, pp. 364-373, May 1990.

[7] D. Joseph and D. Grunwald, "Prefetching using Markov Predictors," in *ISCA-24*, pp. 252-263, June 1997.

[8] T. Mowry and A. Gupta, "Tolerating Latency through Software-controlled Prefetching in Shared-memory Multiprocessors," in *Journal of Parallel and Distributed Computing*, pp. 87-106, June 1991.

[9] H. Wang, et al., "A Conjugate Flow Processor," in *Docket No. 884.225US1*, Patent Pending, May 2000.

[10] H. Wang, et al., "Software-based Speculative Precomputation and Multithreading," in *Docket No. 042390.P10811*, Patent Pending, March 2001.

[11] J. Collins, H. Wang, D. Tullsen, C, Hughes, Y-F Lee, D. Lavery, J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads," in *28th ISCA*, July 2001.

[12] J. Collins, D. Tullsen, H. Wang, J. Shen, "Dynamic Speculative Precomputation," in *Micro-34*, pp. 306-317, December 2001.

[13] P. Wang, H. Wang, J. Collins, E. Grochowski, R. Kling, J. Shen, "Memory latency-tolerance approaches for Itanium processors: out-of-order execution vs. speculative precomputation," in *Proceedings of the 8th IEEE HPCA*, Feb 2002.

[14] S. S. W. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen, "Post-pass Binary Adaptation for Software-based Speculative Precomputation," Accepted for publication at *PLDI'02*.

[15] S. G. Abraham and B. R. Rau, "Predicting Load Latencies using Cache Profiling," in *HP Lab Technical Report HPL-94-110*, Dec 1994.

[16] Intel Corp. VTune Performance Analyzer. http://developer.intel.com/software/products/VTune/index.htm

[17] C. Zilles and G. Sohi, "Understanding the backward slices of performance degrading instructions," in *27th ISCA*, pp. 172-181, June 2000.

[18] D. P. Bhandarkar, *Alpha Implementations and Architecture*, Digital Press, Newton, MA, 1996.

[19] J. Heinrich, *MIPS R10000 Microprocessor User's Manual*, MIPS Technologies Inc., Sept 1996.

[20] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*. Q1, 2001.

[21] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices, in *28th ISCA*, July 2001.

[22] A. Roth and G. Sohi, "Speculative Data-Driven Multithreading," in *7th HPCA*, Jan 2001.

[23] C. K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," in *28th ISCA*, June 2001.

[24] M. Annavaram, J. Patel, and E. Davidson, "Data Prefetching by Dependence Graph Precomputation," in *ISCA-28*, pp 52-61, July 2001.

[25] A. Moshovos, D. Pnevmatikatos, A. Baniasadi, "Slice processors: an implementation of operation-based prediction, in *International Conference on Supercomputing*, June 2001.

[26] A. Roth, A. Moshovos, and G. Sohi, "Dependence-based prefetching for linked data structures," in *ASPLOS-98*, pp. 115-126, Oct. 1998.

[27] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, "Simultaneous Subordinate Microthreading (SSMT)," in *26th International Symposium on Computer Architecture*, May 1999.

[28] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving both Performance and Fault Tolerance," in *9th ASPLOS*, Nov. 2000.

[29] Y. Song and M. Dubois, *Assisted Execution. Technical Report #CENG 98-25, Department of EE-Systems*, University of Southern California, Oct. 1998.

[30] A. Roth, C. B. Zilles, G. S. Sohi, "Microarchitectural Miss/Execute Decoupling," in *MEDEA Workshop*, Oct. 2000.

[31] J. E. Smith. "Decoupled Access/Execute Computer Architecture," in *9th ISCA*, July 1982.

[32] M. K. Farrens, P. Ng, and P. Nico, "A Comparison of Superscalar and Decoupled Access/Execute Architectures," in *26th Micro*, Nov. 1993.

[33] G. P. Jones and N. P. Topham, "A Limitation Study into Access Decoupling," in *3rd Euro-Par Conference*, Aug. 1997.

[34] J. M. Parcerisa and A. Gonzalez, "The Synergy of Multithreading and Access/Execute Decoupling," in *5th HPCA*, Jan. 1999.

[35] A. Bik, M. Girkar, P. Grey and X. Tian, "Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems," in *Intel Technology Journal*, Q1, 2001. http://www.intel.com/technology/itj/q12001/articles/art_6.htm

AUTHORS' BIOGRAPHIES

Hong Wang is a senior staff engineer in Microprocessor Research in Intel Labs. His current interests are in discovering unorthodox ideas and applying them to future Intel processor designs. Hong joined Intel in 1995 and earned a Ph.D. degree in electrical engineering from the University of Rhode Island in 1996. His e-mail is hong.wang@intel.com.

Perry H. Wang joined Intel in 1995. He is with the Microprocessor Research Group in Intel Labs. His technical interests are in advanced microarchitectures and compiler optimizations. He received a B.S. degree in engineering physics and an M.S. degree in computer science from the University of Michigan in Ann Arbor. His e-mail is perry.wang@intel.com

R. David Weldon joined Intel in 2000. He currently works in the Logic Technology Development group, designing future IA-32 processors. David has a BSEE degree from the University of Washington and a Masters degree from Cornell University. His technical interests include processor microarchitecture and hardware/software co-design. His e-mail is ross.d.weldon@intel.com

Scott M. Ettinger joined Intel in 2001. His technical interests are in computer architecture and multimedia signal processing. He received a B.S. and an M.S. degree in electrical engineering from the University of Florida. His e-mail is scott.m.ettinger@intel.com

Hideki Saito received a B.E. degree in Information Science in 1993 from Kyoto University, Japan, and a M.S. degree in Computer Science in 1998 from University of Illinois at Urbana-Champaign, where he is currently a Ph.D. candidate. He joined Intel Corporation in June 2000 and has been working on multithreading and performance analysis. He is a member of the OpenMP Parallelization group. His e-mail is hideki.saito@intel.com

Milind Girkar received a B.Tech. degree from the Indian Institute of Technology, Mumbai, an M.S. degree from Vanderbilt University, and a Ph.D. degree from the University of Illinois at Urbana-Champaign, all in computer science. Currently, he manages the IA-32 compiler development team in Intel's Software Solution Group. Before joining Intel, he worked on a compiler for the UltraSPARC platform at Sun Microsystems. His e-mail is milind.girkar@intel.com.

Steve Shih-wei Liao received a B.S. degree in computer science from National Taiwan University, and M.S. and Ph.D. degrees in electrical engineering from Stanford University. His research interests are in program analyses and optimizations, computer architectures, and programming environments. He currently works in the Microprocessor Research Group at Intel Labs. His e-mail is shih-wei.liao@intel.com.

John P. Shen currently directs Microarchitecture Research in Intel Labs. Prior to joining Intel in 2000, he was on the faculty of the Electrical and Computer Engineering Department of Carnegie Mellon University for over 18 years. He is an IEEE Fellow and is currently writing a textbook on "Fundamentals of Superscalar Processor Design" which will be published by McGraw-Hill in 2002. His e-mail is john.shen@intel.com.

Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance

Xinmin Tian, Software Solutions Group, Intel Corporation
Aart Bik, Software Solutions Group, Intel Corporation
Milind Girkar, Software Solutions Group, Intel Corporation
Paul Grey, Software Solutions Group, Intel Corporation
Hideki Saito, Software Solutions Group, Intel Corporation
Ernesto Su, Software Solutions Group, Intel Corporation

Index words: Hyper-Threading Technology, OpenMP, Optimization, Parallelization, Vectorization

ABSTRACT

In the never-ending quest for higher performance, CPUs become faster and faster. Processor resources, however, are generally underutilized by many applications. Intel's Hyper-Threading Technology is developed to resolve this issue. This new technology allows a single processor to manage data as if it were two processors by executing data instructions from different threads in parallel rather than serially. Processors enabled with Hyper-Threading Technology can greatly improve the performance of applications with a high degree of parallelism. However, the potential gain is only obtained if an application is multithreaded, by either manual, automatic, or semi-automatic parallelization techniques. This paper presents the compiler techniques of OpenMP pragma- and directive-guided parallelization developed for the high-performance Intel C++/Fortran compiler. We also present a performance evaluation of a set of benchmarks and applications.

INTRODUCTION

Intel processors have a rich set of performance-enabling features such as the Streaming-SIMD-Extensions (SSE and SSE2) in the IA-32 architecture [11], large register files, predication, and control and data speculation in the Itanium-based architecture [8]. These features allow the compiler to exploit parallelism at various levels. Intel's newest Hyper-Threading Technology [14], a simultaneous multithreading design, allows one physical processor to manage data as if it were two logical processors by handling data instructions in parallel rather than serially. The Hyper-Threading Technology-enabled processors can significantly increase the performance of application programs with a high degree of parallelism. These potential performance gains are only obtained, however, if an application is efficiently multithreaded, either manually or by automatic or semi-automatic parallelization techniques. The Intel C++/Fortran high-performance compiler supports several such techniques. One of those techniques, automatic loop parallelization, was presented in [3]. In addition to automatic loop level parallelization, Intel compilers support OpenMP directive- and pragma-guided parallelization as well, which significantly increase the domain of various applications amenable to effective parallelism. For example, users can use OpenMP parallel sections to develop an application where *section-1* calls an integer-intensive routine and where *section-2* calls a floating-point intensive routine. Higher performance is obtained by scheduling *section-1* and *section-2* onto two different logical processors that share the same physical processor to fully utilize processor resources based on the Hyper-Threading Technology. The OpenMP standard API [12, 13] supports a multi-platform, shared-memory, parallel programming paradigm in C++/C/Fortran95 on all Intel architectures and popular operating systems such as

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Windows NT , Linux*, and Unix*. OpenMP directives and pragmas have emerged as the de facto standard of expressing parallelism in various applications as they substantially simplify the notoriously complex task of writing multithreaded programs.

The Intel compilers support the OpenMP pragmas and directives in the languages C++/C/Fortran95, on Windows* and Linux platforms and on IA-32 and IPF architectures. The Intel OpenMP implementation in the compiler strives to (i) generate multithreaded code which gains a speed-up due to Hyper-Threading Technology over optimized uniprocessor code, (ii) integrate parallelization tightly with advanced scalar and loop optimizations such as intra-register vectorization [4] and memory optimizations [1, 10] to achieve better cache locality and efficiently exploit multi-level parallelism, and (iii) minimize the overhead of data-sharing among threads.

This paper focuses on the design and implementation of OpenMP pragma- and directive-guided parallelization in the Intel® C++/Fortran compilers. We also present performance results of a number of applications (Micro-benchmark, Image processing library functions, OpenMP benchmarks from [2]) that exhibit performance gains due to Hyper-Threading Technology when such programs are multithreaded through the OpenMP directives or pragmas and compiled with Intel C++/Fortran compilers.

The remainder of this paper is organized as follows. We first give a high-level overview of the architecture of the Intel C++/Fortran compiler with OpenMP support. We then present the Multi-Entry Threading (MET) technique that is the key technique developed for multithreaded code generation in the Intel compilers. We go on to describe the local static data-sharing and privatization methods for minimizing overhead of data sharing among threads. We briefly explain how OpenMP parallelization interacts with advanced optimizations such as constant propagation, interprocedural optimization, and partial redundancy elimination. We also briefly describe how multi-level parallelism is exploited by combining parallelization with intra-register vectorization to take advantage of the Intel Pentium 4 processor SIMD-Streaming-Extensions (SSE and SSE2). Finally, we show the performance results of several OpenMP benchmarks and applications when such programs are multithreaded by the Intel OpenMP C++/Fortran compilers.

HIGH-LEVEL COMPILER OVERVIEW

A high-level overview of the Intel® OpenMP C++/Fortran compiler is shown in Figure 1. The compiler incorporates many well-known and advanced optimization techniques that are designed and extended to fully leverage Intel processor features for higher performance. The Intel compiler has a common intermediate representation for C++, C and Fortran95 languages, so that the OpenMP directive- or pragma-guided parallelization and a majority of optimization techniques are applicable through a single high-level code transformation, irrespective of the source language. Throughout the rest of this paper, we refer to Intel OpenMP C++ and Fortran compilers for IA-32 and Itanium processor family architectures collectively as "the Intel compiler."

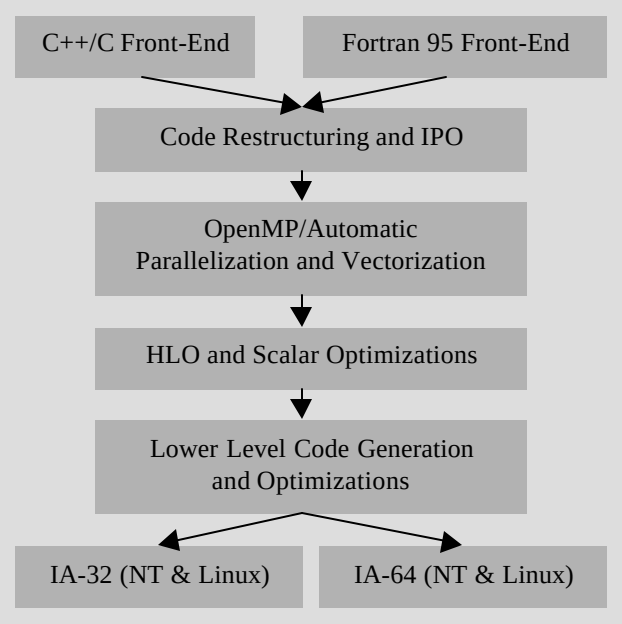


Figure 1: Compiler architecture overview

The code transformations and optimizations in the Intel compiler can be classified into (i) code restructuring and interprocedural optimizations (IPO); (ii) OpenMP-based and automatic parallelization and vectorization; (iii) high-level optimizations (HLO) and scalar optimizations including memory optimizations such as loop control and data transformations, partial redundancy elimination (PRE) [7], and partial dead store elimination (PDSE); and (iv) low-level machine code generation and optimizations such as register allocation and instruction scheduling.

Parallelization [3, 4, 10] guided by OpenMP directives or pragmas or derived by automatic data dependency and control-flow analysis is a high-level code transformation that exploits both medium- and coarse-grained parallelism for Intel processor and multiprocessor systems enabled with Hyper-Threading Technology to achieve better performance and higher throughput. The Intel compiler has a common intermediate code representation (called IL0) into which C++/C and Fortran95 programs are translated by the language front-ends. Many optimization phases in the compiler work on the IL0 representation. The IL0 has been extended to express the OpenMP directives and pragmas. Implementing the OpenMP phase at the IL0 level allows the same implementation to be used across languages (C++/C, Fortran95) and architectures (IA-32 and IPF). The Intel compiler-generated code has references to a high-level multithreaded library API; this allows the compiler OpenMP transformation phase to be independent of the underlying operating systems. This also facilitates our "one-for-all" design philosophy.

A primary goal of the Intel compiler is to have OpenMP parallelization tightly integrated with advanced compiler optimizations for generating efficient multithreaded code that gains a speed-up over optimized uniprocessor code. Therefore, an effective optimization phase ordering has been designed in the Intel compiler to make sure that all optimizations, such as IPO inlining, code restructuring; Igoto optimizations, and constant propagation, which are effectively enabled before the OpenMP parallelization, preserve legal OpenMP program semantics and necessary information for parallelization. It also ensures that all optimizations after the OpenMP parallelization, such as automatic vectorization, loop transformation, PRE, and PDSE, can effectively kick in to achieve a better cache locality and to minimize the number of computations and the number of references to memory. For example, given a double-nested OpenMP parallel loop, the parallelization methods are able to generate multithreaded code for the outer loop, while maintaining the loop structure, memory reference behavior, and symbol table information for the innermost loop. This enables subsequent intra-register vectorization of the innermost loop to fully leverage the Hyper-Threading Technology and SIMD Streaming Extension features of Intel processors. Exploiting multi-level parallelism is described later in this paper.

OpenMP parallelization in the Intel compiler includes (i) a pre-pass that transforms OpenMP parallel sections and worksharing sections into a parallel loop and worksharing loop, respectively; (ii) a work-region graph builder that builds a region hierarchical graph based on the OpenMP-aware control-flow graph; (iii) a loop analysis phase for building the loop structure that consists of loop control variable, loop lower-bound, loop upper-bound, loop pre-header, loop header, and control expression; (iv) a variable classification phase that performs analysis of shared and private variables; (v) a multithreaded code generator that generates multithreaded code at compiler intermediate code level based on Guide, a multithreaded run-time library API that is provided by the Intel KAI Software Laboratory (KSL); (vi) a privatizer that performs privatization to handle firstprivate, private, lastprivate, and reduction variables; and (vii) a post-pass that generates code to cache in thread local storage for handling threadprivate variables. There are a number of compiler techniques developed for parallelization in the Intel compiler. The following sections describe some of these techniques in detail.

MULTI-ENTRY THREADING

A well-known conventional technology, which was named outlining [5, 6], has been used by existing parallelizing compilers for generating multithreaded codes. The basic idea of outlining is to generate a separate subroutine for a parallel region or loop. All threads in a team call this routine with necessary data environment. In contrast to the outlining technology, we developed and implemented a new compiler technology called *Multi-Entry Threading* (MET). The rationale behind MET is that the compiler does not create a separate compilation unit (or routine) for a parallel region or loop. Instead, the compiler generates a threaded entry and a threaded return for a given parallel region and loop [3]. Based on this idea, we introduced three new graph nodes in the Region-based graph, built on top of the control-flow graph. These graph nodes are *T-entry* (threaded entry), *T-ret* (threaded return), and *T-region* (threaded code region). A detailed description of these graph nodes is given as follows:

> T-entry indicates the entry point of a multithreaded code region and has a list of firstprivate, lastprivate, shared and/or reduction variables for communication among the threads in a team.
>
> T-ret indicates the exit point of a multithreaded code region and guides the lower-level target machine code generator to adjust stack offset properly and give the control to the caller inside the runtime library. T-region represents a multithreaded code region that is attached inside the original user routine.

The main concept of the MET compilation model is to keep all newly generated multithreaded codes, which are captured by *T-entry*, *T-region* and *T-ret* nodes, intact or inlined within the same user-defined routine without splitting them into independent subroutines. This method provides later compiler optimizations with more

opportunities for performing optimization. Example (E1-I) is an OpenMP program sample.

Given the parallel program with OpenMP pragmas above, its region-based hierarchical graph is shown in Figure 2. As we see, the first *T-region* represents the OpenMP parallel sections and the second *T-region* represents the OpenMP parallel loop in the routine '*parfoo*.' Each *T-region* contains a *T-entry* node and a *T-ret* node. With OpenMP data attribute clauses, the variables '*w*' and '*y*' are marked as *shared* and the arrays '*x*' and '*z*' are marked as *shared* as well in the parallel sections clause. For the parallel loop, the loop control variable 'm' is marked as *private*, and the variables '*y*' and '*w*' and the array '*z*' are marked as *shared*. The *guided* scheduling type is specified for the parallel loop. The generated pseudo-multithreaded code is shown below in (E1-II). As mentioned previously, the Intel KSL Guide runtime library API has been adopted for thread creation, synchronization and scheduling.

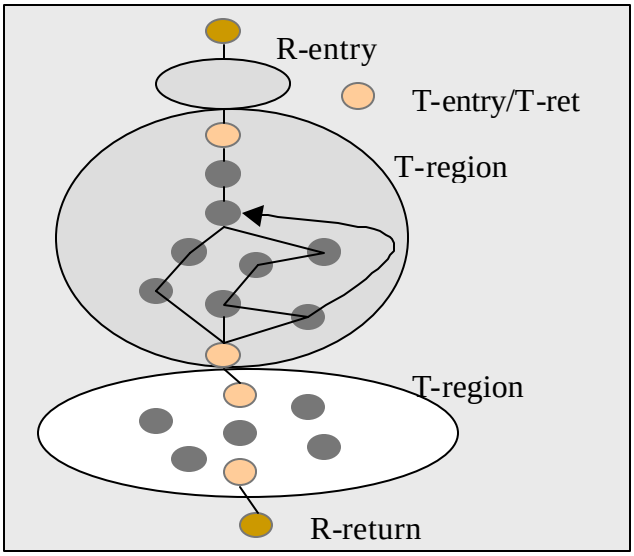


Figure 2: Region-based hierarchical graph

(E1-I) An OpenMP Parallel Sections and Loop Example

```
void parfoo( )
{   int m, y, x[5000];
    float w, z[3000];
#pragma omp parallel sections shared(w, z, y, x)
   {
        w = floatpoint_foo(z, 3000);
    #pragma omp section
        y = myinteger_goo(x, 5000) ;
   }
#pragma omp parallel for private(m) shared(y, z, w)
schedule(guided)
   for (m=0; m<3000; m++) {
        z[m] = z[m] * w * y;
   }
}
```

Essentially, the multithreaded code generator inserts the thread invocation call *__kmpc_fork_call*(…) with *T-entry* point and data environment (source line information *loc*, thread number *tid*, etc.) for each parallel loop, parallel sections or parallel region, and transforms a serial loop, sections, or region to a multithreaded loop, sections, or region, respectively. In this example, the *pre-pass* first converts a *parallel section* to a *parallel loop*. Then, the multithreaded code generator localizes loop lower-bound and upper-bound, privatizes the section id variable, and generates runtime initialization and synchronization code such as the call *__kmpc_static_init*(…) and the call *__kmpc_static_fini*(…) for the T-region marked with [T_entry, T-ret] nodes. For the parallel loop in the routine "*parfoo*" with the scheduling type *guided*, the OpenMP parallelization involves (i) generating a runtime dispatch and initialization routine (*__kmpc_dispatch_init*) call to pass necessary information to the runtime system; (ii) generating an enclosing while loop to dispatch *loop-chunk* at runtime through the *__kmpc_dispatch_next* routine in the library; (iii) localizing the loop lower-bound, upper-bound, and privatizing the loop control variable '*m*.'

(E1-II) Pesudo Multithreaded Code after Parallelization

```
R-entry void parfoo( )
{   int m, y, x[5000];
    float w, z[3000];
   __kmpc_fork_call(loc, 4, T-entry(_parfoo_psection_0),
&w,z,x,&y)
   goto L1:
   T-entry _parfoo_psection_0(loc, tid, *w, z[], *y, x[]) {
      lower_pid = 0;
      upper_pid = 1;
     __kmpc_static_init(loc, tid, STATIC, &lower_pid,
&upper_pid...);
     for (pid=lower_pid, pid<=upper_pid; pid++) {
        if (pid == 0) {
           *w = floatpoint_foo(z, 3000);
        } else if (pid == 1) {
           *y = myinteger_goo(x, 5000);
        }
     }
    __kmpc_static_fini(loc, tid);
    T-ret;
  }
L1:
   __kmpc_fork_call(loc, 3, T-entry(_parfoo_ploop_1), &w, z,
&y);
  goto L2:
  T-entry _parfoo_ploop_1(loc, tid, *w, z[], *y) {
     lower = 0;
     upper = 3000;
       __kmpc_dispatch_init(loc, tid, GUIDED, &lower, &upper,
…);
      while (__kmpc_dispatch_next(loc, tid, &lower, &upper, …))
{
         for (prv_m=lower; prv<upper; prv_m++) {
            z[prv_m] = z[prv_m] * (*w) * (*y);
         }
```

```
    }
    T-ret;
  }
L2:
  R-return;
}
```

There are four well-defined properties of the T-region graph model of the *Multi-Entry Threading* technique:

1. T-region is a sub-graph on top of the pragma-aware control-flow graph, which is identified by the T-entry and T-ret node for a parallel region, sections, or loop.
2. T-region can be nested to present the hierarchy of nested parallelism, e.g., T-region(*k*) = [T-entry(*m*), T-region(*m*), T-ret(*m*)], where the '*k*' and '*m*' are the unique id of a T-region inside this routine.
3. T-region shares the same local memory locations of all local static variables of R-entry (Routine entry). In other words, the local static variables are visible by every T-region associated with this routine.
4. Multiple T-regions are permitted to represent multiple parallel constructs at the same nesting level.

With the T-region graph representation of the Multi-Entry Threading technique, the OpenMP parallelizer and low-level code generators do not generate a separate routine (or compilation unit) for a parallel region or parallel loop. All newly generated multithreaded code blocks (T-regions) for parallel loops are still kept inlined within the same compilation unit. The code transformations are done in a natural way.

From a compiler engineering point of view, the Multi-Entry Threading technique greatly reduces the complexity of generating separate routines in the Intel compiler. In addition, this technique minimizes the impact of OpenMP parallelization on well supported optimizations in the Intel compiler such as constant propagation, vectorization, PRE, PDSE, scalar replacement, loop transformation, interprocedural optimization, and profile-feedback guided optimization (PGO). This meets one of the design goals, namely, to tightly incorporate parallelization with all well-known and advanced compiler optimizations in the Intel compiler.

DATA-SHARING AND PRIVATIZATION

When a routine calls another routine, the communication between them is through global variables and through arguments of the called routine (or callee). This argument-passing across the routine boundary introduces some overhead. The more arguments are passed, the more overhead is introduced. If the call-by-reference method is used for associating actual and dummy arguments, the caller passes to the callee the storage address of the actual argument, and the reference to the dummy argument in the callee becomes an indirect reference. Many optimizations could become disabled by this memory de-referencing. Given the Guide run-time library API, with the *outlining* technology, the parallelizer needs to create a separate routine for a parallel construct, which means the address of each local static variable has to be passed to the outlined routine, since the local static variables in a routine are not visible to other routines. Thus, there are three drawbacks with the *outlining* technique [5, 6]: (i) it adds extra overhead due to argument-passing to outlined routine for sharing local static variables among threads; (ii) it causes less efficient memory access due to memory de-referencing in the outlined routine; and (iii) it may disable some optimizations such as Intra-Register Vectorization and Partial Redundancy Elimination (PRE).

In our implementation of OpenMP parallelization, we are able to overcome these drawbacks based on our Multi-Entry Threading technique. The advances of our technique are: (i) the extra overhead of sharing local static variables is reduced to zero; (ii) no extra memory de-referencing is introduced for accessing local static shared variables; and (iii) later scalar optimizations on local static variables are preserved. The following (E2-I) example has local static variables '*w*,' '*z*,' '*y*,' and '*x*' that are marked as *shared*.

(E2-I) An OpenMP Parallel Sections Example

```
void staticparfoo( )
{  int m;
   static int    y, x[5000];
   static float  w, z[5000];
#pragma omp parallel sections shared(w, z, y, x)
  {
      w = floatpoint_foo(z, 5000);
   #pragma omp section
      y = myinteger_goo(x, 5000) ;
  }
  return;
}
```

In (E2-II), we show the C-like pseudo-multithreaded code generated by the parallelizer. As we see, there are no extra arguments on the *T-entry* node for sharing local static variable '*w*,' '*z*,' '*y*,' and '*x*,' and there is no pointer de-referencing inside the *T-region* for sharing those local static variables among all threads in the team.

(E2-II) Pesudo Multithreaded Code after Parallelization

```
R-entry void staticfoo( )
{   int m;
    static int    y, x[5000];
    static float  w, z[5000];
    __kmpc_fork_call(loc, 0, T-entry(_staticfoo_psection_0))
    goto L1:
    T-entry _parfoo_psection_0(loc, tid) {
       lower_pid = 0; upper_pid = 1;
```

```
      __kmpc_static_init(loc,  tid,  STATIC,  &lower_pid,
&upper_pid...);
      for (pid=lower_pid,  pid<=upper_pid; pid++)  {
          if (pid == 0) {
              w = floatpoint_foo(z, 5000);
          } else if (pid == 1) {
              y = myinteger_goo(x, 5000);
          }
      }
      __kmpc_static_fini(loc, tid);
      T-ret;
   }
L1:    R-return;
}
```

It is well known that the privatization technique can break cycles in a dependence graph and eliminate loop-carried dependencies, so parallelization can be enabled effectively. Actually, privatization removes memory de-references as well. There are three privatization clauses: *firstprivate*, *lastprivate* and *private*, defined in the OpenMP Fortran and C++ standard. Given an OpenMP Fortran example in (E3-I), we see that variables '*x*' and '*y*' are marked as *firstprivate*. The intermediate code before parallelization contains memory de-references *(F32 *)*x* and *(F32 *)*y* (where 'F32' indicates 32-bit floating-point data type) for accessing the variables '*x*' and '*y*' in terms of the call-by-reference argument-passing method used in the Fortran language, as shown in (E3-II).

(E3-I) An OpenMP Fortran Example

```
      subroutine privatefoo(x, y)
      real x, y
      real, save :: a(100)
!$omp parallel do firstprivate(x,y) shared(a)
      do k=1, 100
        a(k) = x + y*k
      end do
      return
      end
```

(E3-II) Pesudo Intermediate Code before Parallelization

```
R-entry void  privatefoo(x, y)
{   … … …
    DIR_OMP_PARALLEL_LOOP      FIRSTPRIVATE(x,      y)
SHARED(a)
    k = 1;
 L3:
    a[k] = *(F32*)x + *(F32*)y * k;
    k = k + 1;
    if (k <= 100 )  { goto L3;  }
    DIR_OMP_END_PARALLEL_LOOP
    R-return
}
```

(E3-III) Pesudo Multithreaded Code after Parallelization

```
R-entry void  privatefoo(x, y)
{    … … …
    __kmpc_fork_call(loc, 2, T-entry(_privatefoo_ploop_0), x, y)
    goto L1:
    T-entry _privatefoo_ploop_0(loc, tid, *x, *y) {
        lower = 0;
        upper = 99;
        prv_x = *(F32 *)x;
        prv_y = *(F32 *)y;
       __kmpc_static_init(loc, tid, STATIC, &lower, &upper, ...);
        prv_k = lower;
        L4:
            a[prv_k] = prv_x + prv_y * prv_k;
            prv_k = prv_k + 1
            if (prv_k <= upper) { goto L4: }
      __kmpc_static_fini(loc, tid);
      T-ret;
   }
L1:    R-return;
}
```

As we can see from (E3-III), privatization has eliminated the memory de-references **x* and **y* inside the parallel loop through the pre-load and pre-copy into the local stack variables '*prv_x*' and '*prv_y*' created by the privatizer. Obviously, this transformation improves the performance by lifting memory de-references outside the loop.

ADVANCED OPTIMIZATIONS

In order to fully leverage advanced scalar optimizations before and after the OpenMP parallelization phase, an optimization phase ordering is carefully designed and implemented in the Intel compiler. In this section, we discuss our design relative to advanced optimizations such as Inter-Procedural Optimization (IPO) [9] and Partial Redundancy Elimination (PRE).

The IPO phase is enabled before OpenMP parallelization at the higher optimization level, so that the Profile-feedback Guided Optimization (PGO), inlining, partial inlining, and forward-substitution can use and benefit from all heuristic and profiling information without any disturbance from multithreaded code generated by the OpenMP parallelizer. In this way, the parallelization is done based on the optimized code. See example E4-I.

(E4-I) An OpenMP Example for Using IPO

```
float w;
void floatpoint_add(float z[ ], int n)
{  int k;
#pragma omp for reduction(+: w) private(k)
   for (k =0; k < n; k++) {
       w = w + z[k];
   }
}

void  inlinefoo( )
{   static float  w, z[5000];
#pragma omp parallel shared(w, z)
   {
       floatpoint_add(z, 5000);
   }
}
```

(E4-II) Pesudo intermediate code after IPO

```
float  w;
R-entry void  inlinefoo( )
{  static float  w, z[5000];
#pragma omp parallel shared(w, z)
   {    int k;
#pragma omp for reduction(+: w) private(k)
        for (k =0; k < 5000; k++) {
           w = w + z[k];
        }
   }
   R-return;
}
```

With IPO inlining and forward-substitution optimization, the subroutine '*floatpoint_add*' is inlined to the subroutine '*inlinefoo*,' the variable '*n*' is substituted with the constant 5000. If the IPO is enabled after OpenMP parallelization, then inlining and forward-substitution may not be able to kick in due to extensive code transformation within the '*floatpoint_add*' and '*inlinefoo*' by the parallelization phase, and due to the changes of the profiling information.

The PRE phase was implemented based on the algorithm in [7] and runs after OpenMP parallelization. Given the example E5-I, the expression '*x+y*k*' is redundant and only needs to be evaluated once for each iteration, and '*x*y*' can be lifted outside the parallel loop.

(E5-I) An OpenMP for Using PRE

```
int b[200], c[200];
void prefoo(int x, int y)  /* x=1 and y=2  in caller */
{ int a[100], k;

#pragma omp parallel for private(k) shared(a, b, c, x,  y)
   for (k = 0; k < 100; k++) {
       a[k] = b[x + y*k] + c[x+y k] + x*y;
  }
  return;
}
```

(E5-II) Pesudo Multithreaded Code from Parallelization and PRE

```
R-entry void prefoo(int x, int y)
{    … … …
    __kmpc_fork_call(loc, 2, T-entry(_prefoo_ploop_0), &x, &y)
    goto L1:
    T-entry _privatefoo_ploop_0(loc, tid, *x, *y) {
       lower = 0;
       upper = 99;
       prv_x = *(SI32 *)x;
       prv_y = *(SI32 *)y;
       t0 = prv_x * prv_y;
      __kmpc_static_init(loc, tid, STATIC, &lower, &upper, ...);
       L3:
            t1 = prv_x + prv_y * prv_k;
            a[prv_k] = b[t1] + c[t1] + t0;
            prv_k = prv_k + 1
            if (prv_k <= upper) {
                goto L3:
            }
      __kmpc_static_fini(loc, tid);
      T-ret;
   }
L1:
   R-return;
}
```

Redundancy is removed through saving the value of the redundant expression in a temporary variable and later reusing that value instead of reevaluating the expression. However, we must be careful with moving code around parallel constructs, since it could generate an unsafe insertion of code for a lifted common expression without knowing the parallel region or parallel loop boundary. Our solution is to apply PRE within each *T-region* after OpenMP parallelization. This guarantees that the correct code is generated. In the code example shown above, we see that '*t0*' and '*t1*' are created as register temporary variables. The 't0' is lifted outside the parallel loop, but it is inserted within the *T-region* and only evaluated once for each thread. The '*t1*' is only evaluated once for each loop iteration. In our experience, there is almost no difference between this and applying PRE optimization to sequential code. There are many more design and implementation details related to incorporating advanced optimizations with parallelization. In the next section, we discuss how the OpenMP parallelization incorporates intra-register vectorization to effectively exploit multi-level parallelism.

MULTI-LEVEL PARALLELISM

The SIMD extensions to the Intel Architecture provide an alternative way to utilize data parallelism in multi-media and scientific applications. These extensions let multiple functional units operate simultaneously on packed data elements, i.e., relatively short vectors that reside in memory or registers. The Pentium 4 processor features the streaming-SIMD-extensions (SSE and SSE2) that support floating-point operations on 4 packed single-precision and 2 packed double-precision floating-point numbers, as well as integer operations on 16 packed bytes, 8 packed words and 4 packed dwords. The Intel compiler supports the automatic conversion of serial loops into SIMD form, a transformation that we refer to as intra-register vectorization [3,4].

Combining intra-register vectorization with parallelization for hyper- or multithreading enables the exploitation of multi-level parallelism, i.e., using the different forms of parallelism that are present in a code fragment to obtain high performance. Take, for instance, the code for matrix-vector multiplication shown in example (E6-I).

(E6-I) An OpenMP-Vector Loop Example

```
double a[N][N], x[N], y[N];
…
#pragma omp parallel for private(k,j)
 for (k = 0; k < N; k++) { /* parallel loop */
   double d = 0.0;
   for (j = 0; j < N; j++) { /* vector loop */
       d += a[k][j] * y[j];
   }
   x[k] = d;
 }
…
```

(E6-II) Pseudo code after Parallelization and Vectorization

```
    __kmpc_fork_call(loc, 0, T-entry(_ompvec_ploop_0),  … )
    goto L1:
    T-entry _ompvec_ploop_0(loc, tid) {
       lower = 0;
       upper = N;
      __kmpc_static_init(loc, tid, STATIC, &lower, &upper, ...);
       prv_k = lower;
L2:
       xorpd    xmm0, xmm0            ; reset accumulator
L3:
       movapd   xmm1, _a[ecx+edx]  ; load 2 DP  from a
       mulpd    xmm1, _y[edx]         ; mult 2 DP  from y
       addpd    xmm0, xmm1            ; add  2 DP  into accumulator
       add      edx, 16               ;
       cmp      edx, eax               ;
       jl        L3                    ; looping logic

       movapd    xmm1, xmm0          ;
       unpckhpd xmm1, xmm1           ;
       addsd     xmm0, xmm1          ; compute final sum

        store result in  x[prv_k]

        prv_k = prv_k + 1
        if (prv_k <= upper) goto L2;

       __kmpc_static_fini(loc, tid);
       T-ret;
    }
L1:  … …
```

In this example, parallelism appears at multiple levels. The iterations of the outermost *k-loop* may execute independently, as has been made explicit with an OpenMP pragma. The reduction performed in the innermost *j-loop* provides yet another level of parallelism. This loop can be implemented by accumulating partial sums in SIMD style, followed by code that constructs the final sum. In (E6-II), we illustrate how these two levels of parallelism can be exploited (where we assume that all access patterns in the vector loop are aligned at a 16-byte boundary).

If the alignment of memory references cannot be determined at compile-time, the Intel compiler has at its disposal several alignment optimizations (such as run-time loop peeling) to avoid performance penalties that are usually associated with unaligned memory accesses. Dynamic data dependence testing is used to allow the compiler to proceed with vectorization in situations where analysis has failed to prove independence statically. These advanced techniques (and others) have been discussed in detail in previous work [4].

PERFORMANCE EVALUATION

The performance study of SPEC OpenMP benchmarks is carried out on a pre-production 1-CPU Hyper-Threading Technology-enabled Intel Xeon processor system running at 1.7GHz, with 512M memory, an 8K L1-Cache, and a 256K L2-Cache. All benchmarks and applications studied in this paper are compiled by the Intel OpenMP C++/Fortran compiler. For the performance study, we chose a subset of SPEC OMPM2001 benchmarks to demonstrate the performance effect of Hyper-Threading Technology. The SPEC OMPM2001 is a benchmark suite that consists of a set of scientific applications. Those SPEC OpenMP benchmarks target small and medium scale (2- to 16-way) SMP multiprocessor systems and the memory footprint reaches 1.6GB for several very large application programs.

The performance scaling is derived from serial execution (SEQ) with Hyper-Threading Technology disabled, and multithreaded execution under one thread and two threads with Hyper-Threading Technology disabled and enabled. In Figure 3, we show the normalized speed-up of the chosen OpenMP benchmarks compared to the serial execution with Hyper-Threading Technology disabled. The OMP1 and OMP2 denote the multithreaded code generated by the Intel OpenMP C++/Fortran compiler executing with one thread and two threads, respectively.

As we see, the multithreaded code generated by the Intel compiler on a Hyper-Threading Technology-enabled Intel Xeon processor 1-CPU system achieved a performance improvement of 4% to 34% (OMP2 w/ HT). The 320.*equake* obtained a 14% performance gain from scalar optimizations enabled by OpenMP (OMP1 w/o HT).

Another 20% performance improvement was achieved by the second thread running on the second logical processor, resulting in a 34% performance gain overall (OMP2 w/ HT). The multithreaded code of the 330.*art* does not show OpenMP overhead, and obtained an 8% speed-up. A 23% slowdown was observed from the 332.*ammp* due to the overhead of thread creation, forking, synchronization, scheduling at run-time, and memory access de-referencing for sharing local stack variables (OMP1 w/o HT), but the second thread running on the second logical processor contributed to the overall 4% performance improvement (OMP2 w/ HT).

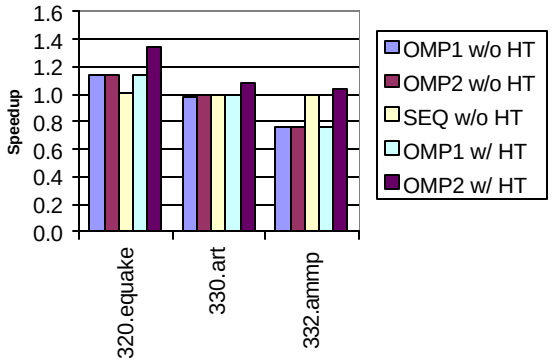


Figure 3: Performance of OpenMP benchmarks

In Figure 4, we show the performance speed-up of three image-processing functions taken from the OpenMP version IPPI library developed by the Intel Performance Library group. The performance speed-up ranges from 1.26x to 1.41x (image size 720x480) on a pre-production Hyper-Threading Technology-enabled Intel Xeon processor 1-CPU system running at 1.8GHz, with 512M of memory, an 8K L1-Cache and a 256K L2-Cache.

As far as we know, there are around 300 image-processing and JPEG functions multithreaded by OpenMP directives in the Intel IPPI performance library. An average speedup of 1.4x was reported when compared with the serial execution of those routines on a pre-production Intel 1.8GHz Hyper-Threading Technology -enabled Intel Xeon Processor 1-CPU system.

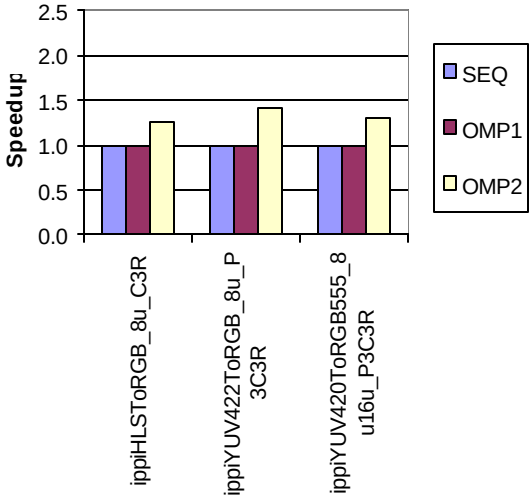


Figure 4: Performance of image processing functions

In Figure 5, we show some performance results for the matrix-vector multiplication kernel discussed earlier on a pre-production Hyper-Threading Technology-enabled Intel Xeon processor dual-CPU system running at 1.5GHz with 512MB of memory, an 8K L1-Cache and a 256K L2-Cache. This graph shows speed-ups (relative to serial execution) for varying matrix sizes for vector execution (VEC), multithreaded execution using two threads and four threads, (OMP2) and (OMP4), respectively, and vector-multithreaded execution using two and four threads, (OMP2+VEC) and (OMP4+VEC), respectively.

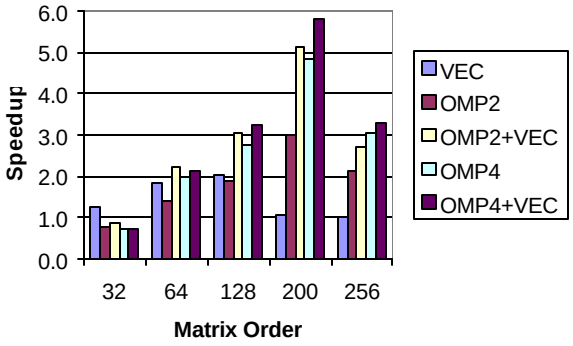


Figure 5: Performance of Matrix x Vector kernel

Timings were obtained by calling the kernel many times and dividing the total execution time accordingly, which implies that for the data sets that completely fit in cache, the kernel is computationally bound. In these cases, intra-register vectorization alone obtains a speed-up of up to 2x. For the larger data sets, where the kernel becomes more memory bound, the improvements of merely intra-register vectorization become less evident. As we have seen

before, the overhead associated with multithreading causes a slight slowdown for the matrix size 32x32. For the larger matrices ranging from 64x64 to 256x256, the relative overhead introduced by parallelization becomes negligible and observed speed-up ranges from 1.4x to 5.8x.

The difference between (OMP2) and (OPM4) for matrix size 200x200 reveals a 1.6x performance gain. For the same matrix size, the performance gain from the versions that are optimized with intra-register vectorization, (OMP2+VEC) and (OMP4+VEC), is 1.2x. The best performance gains are obtained when all levels of parallelism (SIMD parallelism and parallelism due to Hyper-Threading Technology and multithreading) are exploited simultaneously, yielding a speed-up of up to 5.8x with four threads (OMP4+VEC) and a speed-up of 5.1x with two threads (OMP2+VEC).

CONCLUSION

With the growing processor-memory performance gap, memory latency becomes a major bottleneck for achieving high performance for various applications. There are a number of multithreading techniques proposed to hide memory latency. Intel's Hyper-Threading Technology is a very promising technology that allows a single processor to manage data as if it were two processors by executing data instructions in parallel rather than serially. With this new technology, the performance of applications can be greatly improved by exploiting thread-level parallelism. The potential gains are only obtained, however, if an application program is multithreaded. The Intel OpenMP C++/Fortran compiler has been designed to leverage the rich set of performance enabling features, such as Hyper-Threading Technology and the Streaming-SIMD-Extensions (SSE and SSE2), this is achieved by tightly integrating OpenMP directive- or pragma-guided parallelization with other well-known and advanced optimizations to generate efficient multithreaded code for exploiting parallelism at various levels. The results of performance measurement show that OpenMP applications compiled with the Intel C++/Fortran compiler can achieve great performance gains on Intel single and multiprocessor systems that are enabled with Hyper-Threading Technology.

ACKNOWLEDGMENTS

The authors thank the other members of the compiler team for their great work in implementing the Intel high-performance C++/Fortran compiler. In particular, we thank Max Domeika for the OpenMP C++/C front-end support, Michael L. Ross and Bhanu Shankar for the OpenMP Fortran front-end support, Knud J. Kirkegaard for IPO support, and Zia Ansari for PCG support. Special thanks go to Sanjiv Shah and the compiler group at KSL for providing the Guide runtime library, and to the INNL library team for providing the Short Vector Mathematical Library. Both libraries are currently part of the Intel C++/Fortran compiler. Many thanks go to Boris Sabanin for providing the performance numbers for Intel IPPI Image processing functions.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers–Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Boston, Massachusetts, 1986.

[2] Vishal Aslot, et .al., "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," in Proceedings of WOMPAT 2001, *Workshop on OpenMP Applications and Tools, Lecture Notes in Computer Science*, 2104, pages 1-10, July 2001.

[3] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian, "Efficient Exploitation of Parallelism on Pentium® III and Pentium® 4 Processor-Based Systems," *Intel Technology Journal*, Q1 2001, http://intel.com/technology/itj/q12001/articles/art_6.htm.

[4] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian, "Automatic Intra-Register Vectorization for the Intel® Architecture," accepted by the *International Journal of Parallel Programming*, December, 2001.

[5] C. Brunschen and M. Brorsson, "OdinMP/CCp–A Portable Implementation of OpenMP for C," in *Proceedings of the First European Workshop on OpenMP (EWOMP)*, September. 1999.

[6] Jyh-Herng Chow, Leonard E. Lyon, and Vivek Sarkar, "Automatic Parallelization for Symmetric Shared-Memory Multiprocessors, in *Proceedings of CASCON'96: 76-89*, Toronto, ON, November 12-14, 1996.

[7] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu, "A new algorithm for partial redundancy elimination based on SSA form," in *Proceedings of the ACM*

SIGPLAN '97 Conference on Programming Language Design and Implementation, June 1997, pp. 273-286.

[8] Carole Dulong, "The IA-64 Architecture at Work," *IEEE Computer*, July 1998, pp. 24-32.

[9] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam, "Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler," in *Proceedings of Supercomputing*, San Diego, California, Dec. 1995.

[10] Michael J. Wolfe, *High Performance Compilers for Parallel Computers*, Addison-Wesley Publishing Company, Redwood City, California, 1996.

[11] Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, Intel Corporation, 2001, http://developer.intel.com/

[12] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface," Version 1.0, October 1998, http://www.openmp.org

[13] OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface," Version 2.0, November 2000, http://www.openmp.org

[14] Debbie Marr, et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, Q1 2002.

AUTHORS' BIOGRAPHIES

Xinmin Tian is currently working in the vectorization and parallelization group at Intel Corp. where he works on compiler parallelization and optimization. He manages the OpenMP Parallelization group. He holds B.Sc., M.Sc., and Ph.D. degrees in Computer Science from Tsinghua University. He was a postdoctoral researcher in the School of Computer Science at McGill University, Montreal. Before joining Intel Corp., he worked on a parallelizing compiler, code generation, and performance optimization at IBM. His e-mail is xinmin.tian@intel.com

Aart Bik received his M.Sc. degree in Computer Science from Utrecht University, The Netherlands, in 1992 and his Ph.D. degree from Leiden University, The Netherlands, in 1996. In 1997, he was a postdoctoral researcher at Indiana University, Bloomington, Indiana, where he conducted research in high-performance compilers for Java*. In 1998, he joined Intel Corporation where he is currently working in the vectorization and parallelization group. His e-mail is aart.bik@intel.com

Milind Girkar received a B.Tech. degree from the Indian Institute of Technology, Mumbai, an M.Sc. degree from Vanderbilt University, and a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in Computer Science. Currently, he manages the IA-32 Compiler Development group. Before joining Intel Corp., he worked on an optimizing compiler for the UltraSPARC platform at Sun Microsystems. His e-mail is milind.girkar@intel.com

Paul Grey did his B.Sc. degree in Applied Physics at the University of the West Indies and his M.Sc. degree in Computer Engineering at the University of Southern California. Currently he is working at Intel Corp. on compiler optimizations for parallel computing. Before joining Intel, he worked on parallel compilers, parallel programming tools, and graphics system software at Kuck and Associates, Inc., SUN, and SGI. He is interested in optimizing compilers, advanced microarchitecture, and parallel computers. His e-mail is paul.grey@intel.com

Hideki Saito received a B.E. degree in Information Science in 1993 from Kyoto University, Japan, and a M.S. degree in Computer Science in 1998 from University of Illinois at Urbana-Champaign, where he is currently a Ph.D. candidate. He joined Intel Corporation in June 2000 and has been working on multithreading and performance analysis. He is a member of the OpenMP Parallelization group. His e-mail is hideki.saito@intel.com

Ernesto Su received a B.S. degree from Columbia University, and M.S. and Ph.D degrees from the University of Illinois at Urbana-Champaign, all in Electrical Engineering. He joined Intel Corp. in 1997 and is currently working in the OpenMP Parallelization group. His research interests include compiler performance optimizations, parallelizing compilers, and computer architectures. His e-mail is ernesto.su@intel.com.

*Other brands and names are the property of their respective owners.

Media Applications on Hyper-Threading Technology

Yen-Kuang Chen, Microprocessor Research, Intel Labs
Matthew Holliman, Microprocessor Research, Intel Labs
Eric Debes, Microprocessor Research, Intel Labs
Sergey Zheltov, Microprocessor Research, Intel Labs
Alexander Knyazev, Microprocessor Research, Intel Labs
Stanislav Bratanov, Microprocessor Research, Intel Labs
Roman Belenov, Microprocessor Research, Intel Labs
Ishmael Santos, Software Solutions Group, Intel Corporation

Index words: Hyper-Threading Technology, multithreading, multimedia, MPEG, performance analysis

ABSTRACT

This paper characterizes selected workloads of multimedia applications on current superscalar architectures, and then it characterizes the same workloads on Intel Hyper-Threading Technology. The workloads, including video encoding, decoding, and watermark detection, are optimized for the Intel® Pentium® 4 processor. One of the workloads is even commercially available and it performs best on the Pentium 4 processor. Nonetheless, due to the inherently sequential constitution of the algorithms, most of the modules in these well-optimized workloads cannot fully utilize all the execution units available in the microprocessor. Some of the modules are memory-bounded, while some are computation-bounded. Therefore, Hyper-Threading Technology is a promising architecture feature that allows more CPU resources to be used at a given moment.

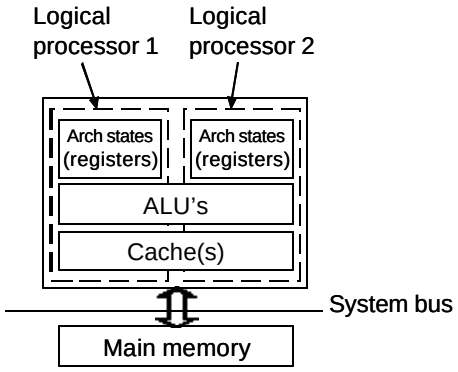
Our goal, in this paper, is to better explain the performance improvements that are possible in multimedia applications using Hyper-Threading Technology. Our initial studies show that there are many unexplored issues in algorithms and applications for Hyper-Threading Technology. In particular, there are many techniques to develop better software for multithreading systems. We demonstrate different task partition/scheduling schemes and discuss their trade-offs so that a reader can understand how to develop efficient applications on processors with Hyper-Threading Technology.

INTRODUCTION

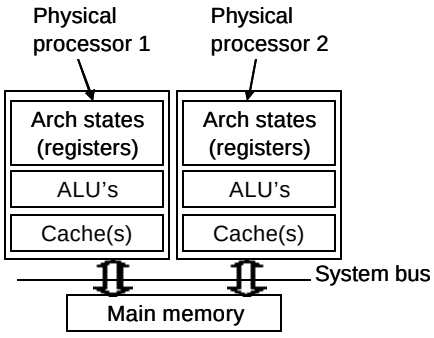
To date, computational power has typically increased over time because of the evolution from simple pipelined designs to the complex speculation and out-of-order execution of many of today's deeply-pipelined superscalar designs. While processors are now much faster than they used to be, the rapidly growing complexity of such designs also makes achieving significant additional gains more difficult. Consequently, processors/systems that can run multiple software threads have received increasing attention as a means of boosting overall performance. In this paper, we first characterize the workloads of video decoding, encoding, and watermarking on current superscalar architectures, and then we characterize the same workloads using the recently-announced Hyper-Threading Technology. Our goal is to provide a better understanding of performance improvements in multimedia applications on processors with Hyper-Threading Technology.

Figure 1 shows a high-level view of Hyper-Threading Technology and compares it to a dual-processor system. In the first implementation of Hyper-Threading Technology, one physical processor exposes two logical processors. Similar to a dual-core or dual-processor system, a processor with Hyper-Threading Technology appears to an application as two processors. Two applications or threads can be executed in parallel. The major difference between systems that use Hyper-Threading Technology and dual-processor systems is the

different amounts of duplicated resources. In today's Hyper-Threading Technology, only a small set of the microarchitecture state is duplicated[1], while the front-end logic, execution units, out-of-order retirement engine, and memory hierarchy are shared. Thus, compared to processors without Hyper-Threading Technology, the die-size is increased by less than 5% [7]. While sharing some resources may increase the latency of some single-threaded applications, the overall throughput is higher for multi-threaded or multi-process applications.



(a)



(b)

Figure 1: High-level diagram of (a) a processor with Hyper-Threading Technology and (b) a dual-processor system

This paper is organized as follows. First, we provide a brief review of the basic principles behind most current video codecs, describing the overall application behavior of video decoding/encoding/watermarking and the implications of the key kernels for current and emerging architectures. Then, we show the multi-threaded software architectures of our applications, including data-domain and functional decomposition. Additionally, we describe some potential pitfalls when developing software on processors with Hyper-Threading Technology and our techniques to avoid them. Finally, we provide some performance numbers and our observations.

MULTIMEDIA WORKLOADS

This section describes the workload characterization of selected multimedia applications on current superscalar architectures. Although the workloads are well optimized for Pentium® 4 processors, due to the inherent constitution of the algorithms, most of the modules in these workloads cannot fully utilize all the execution resources available in the microprocessor. The particular workloads we target are video decoding, encoding, and watermark detection[2], which are key components in both current and many future applications and are representative of many media workloads.

MPEG Decoder and Encoder

The Moving Pictures Expert Group (MPEG) is a standards group founded in 1988. Since its inception, the group has defined a number of popular audio and video compression standards, including MPEG-1, MPEG-2, and MPEG-4 [3]. The standards incorporate three major compression techniques: (1) predictive coding; (2) transform-based coding; and (3) entropy coding. To implement these, the MPEG encoding pipeline consists of motion estimation, Discrete Cosine Transform (DCT), quantization, and variable-length coding. The MPEG decoding pipeline consists of the counterpart operations of Variable-Length Decoding (VLD), Inverse Quantization (IQ), Inverse Discrete Cosine Transform (IDCT), and Motion Compensation (MC), as shown in Figure 2.

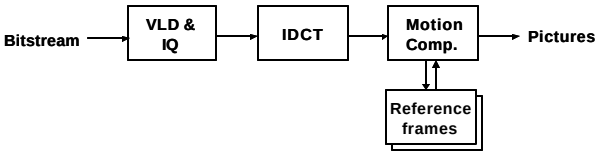


Figure 2: Block diagram of an MPEG decoder

[1] Nearly all the architectural state is duplicated, however.

[2] A digital video watermark, which is invisible and hard to alter by others, is information embedded in the video content. A watermark can be made by slightly changing the video content according to a secret pattern. For example, when just a few out of the millions of pixels in a picture are adjusted, the change is imperceptible to the human eye. A decoder can detect and retrieve the watermark by using the key that was used to create the watermark.

Table 1: MPEG decoding kernel characterization on 2 GHz Pentium® 4 processors (9 Mb/s MPEG-2, 720x480)

Kernel	IPC	UPC	MMX/SSE/SSE-2 per instructions	Cond. Branch/ instr.	Mispred. Cond./ Instr.	Mispred. Cond./ Clock	L1 misses/ Instr.	FSB activity
VLD	0.76	0.99	0.074	1/9	1/120	**1/158**	1/92	11.1%
IDCT	0.59	0.89	**0.90**	1/141	1/2585	1/4381	1/193	2.4%
MC	0.24	0.40	0.42	1/17	1/142	1/592	1/11	**30.3%**

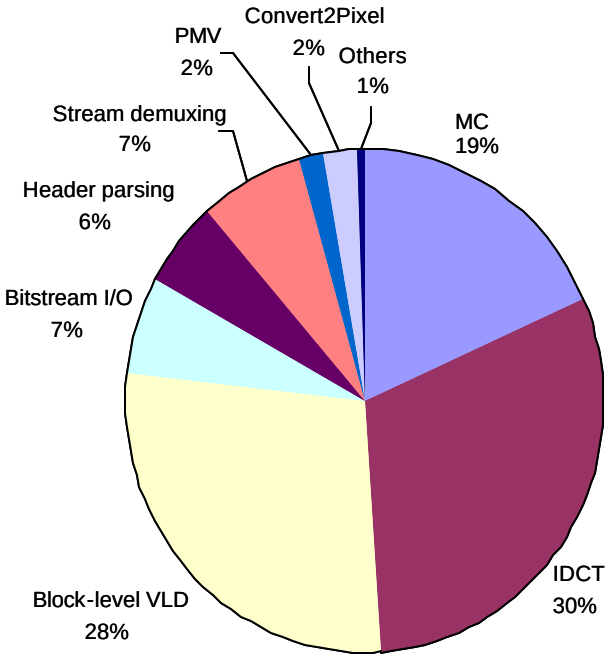


Figure 3: MPEG-2, 720x480 decoding breakdown by time on 2GHz Pentium® 4 processors

The behavior of the MPEG decoder can be highly dependent on the characteristics of the video stream being decoded. Figure 3 shows an example of the CPU time breakdown of our MPEG decoder for a typical DVD resolution video sequence. VLD, IDCT, and MC are the main components in the process. The decoder used in the study is part of the Intel Media Processing Library (MPL)[3], which was developed by Intel Labs. The software was analyzed using the Intel VTune Performance Analyzer on an Intel® Pentium 4 processor with a 400 MHz system bus, an 8 KB first-level data cache, a 256 KB second-level shared instruction/data cache, and 512 MB of main memory. We run our applications on Windows XP.

Table 1 shows a high-level analysis of the MPEG-2 decoder. The first stage of the decoding pipeline, VLD/IQ, is characterized by substantial data dependency, limiting opportunities for instruction, data, and thread-level parallelism. The kernel is entirely computation-bound, and it shows excellent performance scaling over increasing frequencies on the Pentium 4 processor. The next stage, IDCT, is also completely computation-bound. The kernel is dominated by MMX/SSE/SSE2 (Streaming SIMD Extension) operations, with interspersed register-to-register moves and stores; e.g., a sequence of `movaps`, `addps`, and `subps` is a typical recurring theme, corresponding to the well-known butterfly operation, surrounded by associated prescaling/multiply operations. Because 90% of the instructions are executed in the MMX/SSE/SSE2 unit, the integer execution unit is idle most of the time in the IDCT module[4]. The final stage of the decoding pipeline, MC, is memory intensive compared to the other modules in the pipeline. The front-side bus is busy 30% of the time in this module. Although the out-of-order execution core in the Pentium 4 processor can tolerate some memory latencies, the module shows an equal distribution of time between computation and memory latency because there are too many memory operations. All these modules are well-optimized, but still cannot utilize 100% of the execution units available in the microprocessors. While the Pentium 4 processor can execute multiple uops in one cycle, the uops retired per cycle (UPC) is only 0.74 in the MPL decoder.

MPEG encoders, similar to the decoder, consist of some MMX/SSE/SSE2 intensive modules (e.g., motion estimation, DCT) and some data-dependent modules (e.g., variable-length coding). All these modules are well optimized, but a UPC of 1.05 again indicates that the

[3] More information about the MPL can be found at http://www.intel.com/research/mrl/research/mpl/. Additionally, the MPL MPEG-2 decoder is commercially available as part of the Ligos* GoMotion* SDK.

VTune is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Pentium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Other brands and names may be claimed as the property of others.

[4] See Figure 4 in [4], integer operations and floating-point/MMX/SSE/SSE2 operations are executed in different units.

encoder cannot fully utilize all the execution units available in the microprocessor.

Video Watermarking

Another application that we studied is video watermark detection [1]. Our watermark detector has two basic stages: video decoding and image-domain watermark detection. The application is optimized with MPL (as the video decoder) and the Intel IPL (for the image manipulations used during watermark detection) [5]. A UPC of 1.01 also indicates that there is room for improvement.

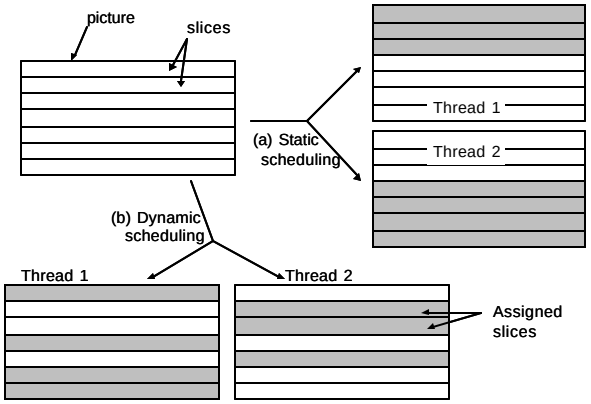


Figure 4: Two slice-based task partitioning schemes between two threads: (a) half-and-half dispatching (static scheduling); and (b) slice-by-slice scheduling (dynamic scheduling)

TASK PARTITIONING AND SCHEDULING

In general, multimedia applications, such as video encoding and decoding, exhibit not only data- and instruction-level parallelism, but also the possibility for substantial thread-level parallelism. Such workloads are good candidates for speed-up on a number of different multithreading architectures. This section discusses the trade-offs of different software multithreading methods.

Data-Domain Decomposition–Slice-Based Dispatching

As shown in Figure 4, a picture in a video bit stream can be divided into slices of macroblocks. Each slice, consisting of blocks of pixels, is a unit that can be decoded independently. Here we compare two methods to decode the pictures in parallel:

1. Half-and-half (aka static partitioning): In this method, one thread is statically assigned the first half of the picture, while another thread is assigned the other half of the picture (as shown in Figure 4 (a)). Assuming that the complexity of the first half and second half is similar, these two threads will finish the task at roughly the same time. However, some areas of the picture may be easier to decode than others. This may lead to one thread being idle while the other thread is still busy.

2. Slice-by-slice (aka dynamic partitioning): In this method, slices are dispatched dynamically. A new slice is assigned to a thread when the thread has finished its previously assigned slice. In this case, we don't know which slices will be assigned to which thread. Instead, the assignment depends on the complexity of the slices assigned. As a result, one thread may decode a larger portion of the picture than the other if its assignments are easier than those of the other thread. The execution time difference between two threads, in the worst case, is the decoding time of the last slice.

In both cases, each thread performs Variable-Length Decoding (VLD), Inverse Discrete Cosine Transform (IDCT), and Motion Compensation (MC) in its share of the pictures, macroblock by macroblock. While one thread is working on MC (memory intensive), the other thread may work on VLD or IDCT (less memory intensive). Although the partitioning does not explicitly interleave computations and memory references, on average, it better balances the use of resources.

Functional Decomposition of Video Watermark Detection

Besides data-domain decomposition, an application can also be partitioned functionally into multiple threads. For example, our video watermark detector consists of two basic stages: video decoding and watermark detection. Hence, we assign different threads to decode the video and to detect the watermark, as shown in Figure 5.

One method is to use two threads: one for video decoding and another for watermark detection, as shown in Figure 5 (b). However, this method does not have very good load balance. This is because in our video watermark detector, video decoding takes roughly one-third of the CPU time, while watermark detection takes two-thirds of the CPU time.

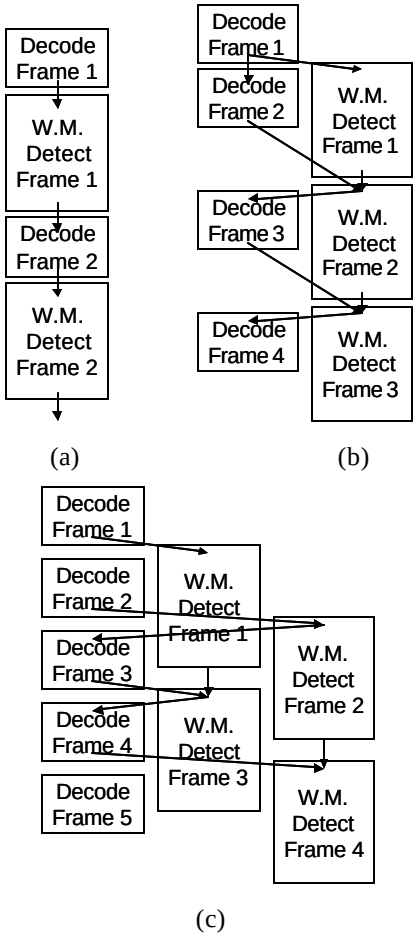


Figure 5: Three threading methods for video watermark detection: (a) single-threaded mode; (b) two-threaded mode; and (c) three-threaded mode

Because watermark detection takes twice as much computation time as video decoding, we use two threads for watermark detection for better load balancing, as shown in Figure 5 (c). While one thread decodes the video sequence, two threads work on watermark detection. The lines in the figure indicate the dependency between functional blocks. We can see that at any moment, there are at least two threads running in the three-threaded mode. In contrast to the data-domain video decoding decomposition described above, threads in this implementation are assigned to different functions.

IMPLICATIONS OF SOFTWARE DESIGN FOR HYPER-THREADING TECHNOLOGY

During the implementation of our applications on processors with Hyper-Threading Technology, we had a number of observations. In this section, we discuss some general software techniques to help readers design their applications better on systems with Hyper-Threading Technology.

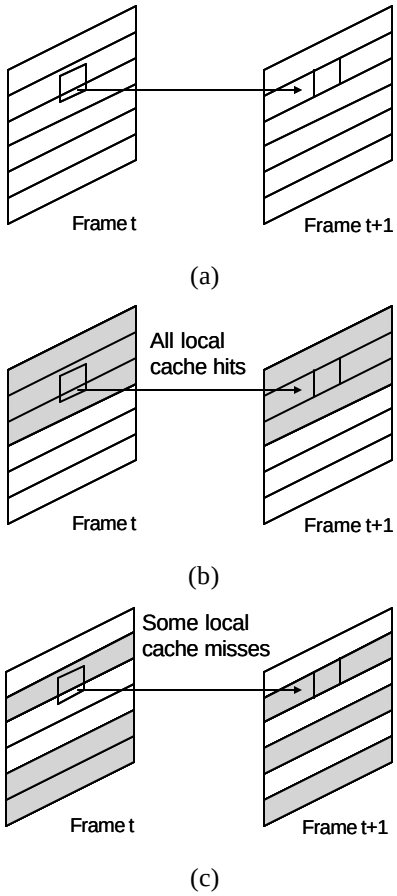


Figure 6: Cache localities, during; (a) motion compensation, in; (b) static partitioning, and in; (c) dynamic partitioning

Using Hyper-Threading Technology, performance can be lost when the loads are not balanced. Because two logical processors share resources on one physical processor with Hyper-Threading Technology, each logical processor does not get all the resources a single processor would get. When only a single thread of the application is actively working and the other thread is waiting (especially, spin-waiting), this portion of the application could have less than 100% of the resources when compared to a single processor, and it might run slower on a processor with simultaneous multithreading capability than on processors without simultaneous multithreading capability. Thus, it is important to reduce the portion in which only one thread is actively working. For better performance, effective load balancing is crucial.

The foremost advantage of the dynamic scheduling scheme (see Figure 4) is its good load balance between the two threads. Because some areas of the picture may be easier to decode than others, one thread under the static

partitioning scheme may be idle while another thread still has a lot of work to do. In the dynamic partitioning scheme, we have very good load balance. As we assign a new slice to a thread only when it has finished its previous slice, the execution time difference between the two threads, in the worst case, is the decoding time of a slice.

Because two logical processors share one physical processor, the effective sizes of the caches for each logical processor are roughly one half of the original size. Thus, it is important for multithreaded applications to target one half of the caches for each application thread. For example, when considering code size optimization, excessive loop unrolling should be avoided.

While sharing caches may be a drawback for some applications running on processors with Hyper-Threading Technology, it can provide better cache locality between the two logical processors for other applications. For example, Wang *et al*. use one logical processor to prefetch data into the shared caches to reduce a substantial amount of the memory latency of the application in the other logical processors [8]. We now illustrate the advantage of sharing caches in our application.

On dual-processor systems, each processor has a private cache. Thus, there may be a drawback to dynamic partitioning in terms of cache locality. Figure 6 illustrates the cache locality in multiple frames of video. During motion compensation, the decoder uses part of the previous picture, the referenced part of which is roughly co-located in the previous reference frame, to reconstruct the current frame. It is faster to decode the picture when the co-located part of the picture is still in the cache. In the case of a dual-processor system, each thread is running on its own processor, each with its own cache. If the co-located part of the picture in the previous frame is decoded by the same thread, it is more likely that the local cache will have the pictures that have just been decoded. Since we dynamically assign slices to different threads, it is more likely that the co-located portion of the previous picture may not be in the local cache when each thread is running on its own physical processor and cache, as shown in Figure 6 (c). Thus, dynamic partitioning may incur more bus transactions[5]. In contrast, the cache is shared between logical processors on a processor with Hyper-Threading Technology, and thus, cache localities are preserved. We obtain the best of both worlds with dynamic scheduling: there is load balancing between the threads, and there is the same effective cache locality as for static scheduling on a dual-processor system.

RESULTS

This shows some performance numbers and analysis of our applications on multithreading architectures. In general, our results show that Hyper-Threading Technology offers a cost-effective performance improvement (7%-18%) for multithreading without doubling hardware cost (see Figure 7) as in dual-processor systems.

Our Hyper-Threading Technology system has an experimental 1.7GHz Intel Pentium 4 processor with Hyper-Threading Technology capability, which is a pre-production prototype, running Windows XP. The processor has a 512KB second-level cache, but no third-level cache. To contrast the performance with single-thread performance on the system experimentally in lab setting, we disable the support of Hyper-Threading Technology from the CPU, motherboard, BIOS, and the operating system. Our dual-processor system has two 1.7GHz Intel Xeon processors, each of which has a 256KB second-level cache and a 1MB third-level cache, running Windows XP. To measure single-thread performance on the dual-processor system, we disable one physical processor and run a single-thread version of the application. The relative speed between Hyper-Threading Technology systems and dual-processor systems is not measured in our experiment.

To measure the performance of the encoder, we use five 720x480 YVU 4:2:0 benchmark sequences. To measure the performance of the decoder, we use one 640x480, three 704x480, three 720x480, one 1280x720, and two 1920x1080 MPEG-2 sequences. Moreover, three 704x480 MPEG-2 sequences are used to measure the performance of the video watermark detectors. The speed-ups are sequence dependent, but within a small variation. We report only the average numbers in Figure 7.

[5] On dual-processor systems, an alternative method of keeping cache locality in dynamic scheduling is to dispatch slices to one thread from top-down and slices to the other thread from bottom-up. However, it is hard to generalize the method for four-way or eight-way multi-processor systems. In this paper, we did not show the results of this method.

Data-Domain Decomposition

This section describes the performance of the data-domain decomposition of the video decoding described earlier.

First, Figure 7 (b) shows that it is better to use the dynamic scheduling method than the static scheduling method on a processor with Hyper-Threading Technology, as it is very important to have a balanced load. Because resources are shared between the logical processors, the relative performance of each logical processor can be less than 1.0 compared to an equivalent processor without simultaneous multithreading capability. When only one thread is busy, the overall throughput is less than that of a single processor. To have the best performance, it is important to have a balanced workload between threads. Hence, the dynamic scheme is better than static scheduling.

On the other hand, Figure 7 (b) shows that the static scheduling method is better than the dynamic scheduling method on a dual-processor system. It is faster to decode the picture when the co-located parts of the pictures are still in the cache. As mentioned earlier, although dynamic scheduling has better load balance, co-located parts of the pictures may not be decoded by the same processor when using dynamic scheduling. This scheduling scheme incurs more bus transactions, as shown in Table 2, with the result that the overall speed using dynamic scheduling is slower.

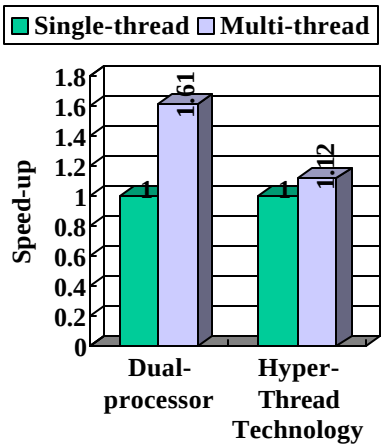
Compared to dual-processor systems, processors with Hyper-Threading Technology have the advantage of sharing the second-level cache between two logical processors. Even when the same logical processor does not decode the co-located part of the reference picture, that part of the picture can still be read from the shared second-level cache. Table 3 shows that the numbers of bus activities are similar between static scheduling and dynamic scheduling. In this case, the overall speed of dynamic scheduling is faster because the workload is

Table 2: The numbers of front-side bus (FSB) data activities per second between static scheduling and dynamic scheduling on a dual-processor system

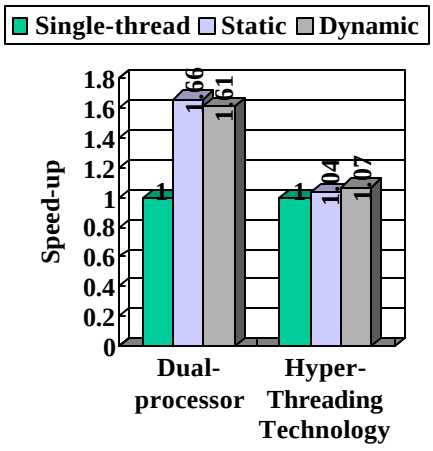
Event	Static scheduling	Dynamic scheduling
FSB_data_activity	8,604,511	12,486,051

Table 3: The numbers of FSB data activities per second between static scheduling and dynamic scheduling on a processor with Hyper-Threading Technology

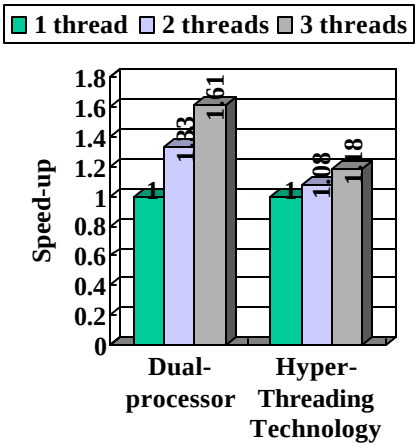
Event	Static scheduling	Dynamic scheduling
FSB_data_activity	8,474,022	8,536,838



(a)



(b)



(c)

Figure 7: Performance of; (a) our video encoder; (b) our video decoder; and (c) our watermarking detection with software configurations

Table 4: The workload characteristics of our applications on single-threaded processors and processors with Hyper-Threading Technology

Event	MPEG encoding		MPEG decoding		Video watermarking	
	Single-thread	Hyper-threading	Single-thread	Hyper-threading	Single-thread	Hyper-threading
Clockticks (Millions)	13,977	11,688	7,467	6,687	23,942	20,162
Instructions retired (Millions)	11,253	11,674	3,777	3,921	17,728	17,821
Uops retired (Millions)	14,735	15,539	5,489	5,667	24,120	24,333
MMX/SIMD uops retired (Millions)	6,226	6,220	1,119	1,120	5,334	5,341
IPC (instructions per clock)	0.80	1.00	0.51	0.59	0.74	0.88
UPC (uops per clock)	1.05	1.33	0.74	0.85	1.01	1.21
Trace cache misses (Millions)	20.8	29.0	13.3	24.1	7.6	13.3
First-level cache misses (Millions)	132	145	132	166	510	638
Bus utilization	8.5%	8.5%	14.7%	16.4%	14.2%	22.3%

better balanced.

Functional Decomposition

Here, we describe the performance of the video watermark detection functional decomposition described earlier in Figure 5. Figure 7 (c) shows the performance comparisons. 2-thread denotes one video-decoding thread and one watermark detection thread, and 3-thread denotes one video-decoding thread and two watermarking threads (see Figure 5 (c)). Similar to the results of the video decoder, better performance is obtained with better balanced workloads.

Overall Performance Characteristics

As mentioned earlier, different modules have been interleaved in the application to utilize more execution resources in the machine at a given time. Hence, it is hard to break down the workload characteristics in individual modules. Rather, it is better to consider the application as a whole.

As shown in Table 4, although the numbers of instructions retired and cache misses (e.g., trace and first-level) increase in both applications after threading, because of threading overhead and capacity misses in each thread, the overall application performance still increases. To verify that resource utilization is better balanced on a processor with Hyper-Threading Technology, we compare UPC for single-threaded and multi-threaded applications. UPC increases from 1.05 to 1.33 in video encoding, from 0.78 to 0.85 in video decoding, and from 1.01 to 1.21 in watermark detection, confirming the more efficient resource utilization possible with Hyper-Threading Technology. (These numbers include the overhead of thread synchronization; however, this overhead is relatively small, being on the order of 0.5% for watermark detection, approximately 3-4% for video decoding, and 4-5% for video encoding.)

POWER CONSUMPTION ISSUES

In this paper, we have mainly discussed methods to improve the application throughput on processors with Hyper-Threading Technology. In addition to throughput, power consumption is also an important performance factor for the next generation of processors. This is especially true for battery-run mobile systems, in which the average power consumption for a given fixed application is a crucial parameter to consider for the evaluation of the overall performance of the system.

In this section, we show that Hyper-Threading Technology can not only improve system throughput but can also save energy for applications with fixed duties. As an introduction to this new research topic, we give some hints on how to design "power-aware" applications on processors with Hyper-Threading Technology and we show the first results of this ongoing work.

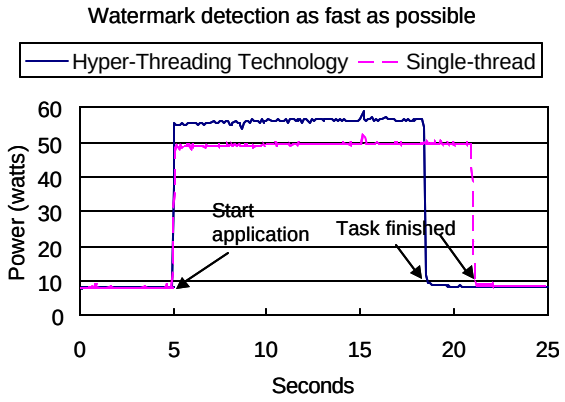


Figure 8: Measured power consumption of our watermark detector on a processor with Hyper-Threading Technology and a normal system at the same frequency and voltage

In various situations, Hyper-Threading Technology consumes additional power while improving the performance, as shown in Figure 8. When idle, the execution units in Intel Pentium 4 processors consume less power because of clockgating [2]. Hyper-Threading Technology makes the execution units busier, and thus, they consume slightly more power. The graphs also show that the task finishes earlier on a system with Hyper-Threading Technology. Because the task finishes in fewer cycles, the overall energy consumption is slightly less on a system with Hyper-Threading Technology even with the same voltage and frequency. This is because powering up additional execution units for two simultaneous threads is more economical than powering the whole pipeline with fewer execution units to run serial threads.

In the case of real-time applications[6], where we need only a fixed amount of throughput, we can reduce the frequency and the voltage. As Hyper-Threading Technology increases the throughput, and we have more spare cycles, we can further reduce the frequency and the voltage. Because the active power consumption is proportional to frequency*(voltage)^2, we can have a cubic effect on energy saving.

Nonetheless, a common thread scheduling pitfall in multithreading real-time applications can reduce the overall energy gain on the system with Hyper-Threading Technology. Figure 9 (a) shows a common, but less than optimal, multithreading method of the watermark detection application–the watermark detector is active immediately after the video frame is decoded. Due to a large cycle period, there may be no overlapping between two threads (see Figure 5 (b)). While Figure 9 (b) has the same cycle period as Figure 9 (a), by delaying the starting time of the second thread, we increase the overlapping period of two threads. That is, we queue the tasks and dispatch together to maximize the overlap. In this case, the halted period in CPU is increased. Because powering up additional execution units for two simultaneous threads is more economical and the physical processor consumes less power when it is halted (or when both logical processors are halted), Figure 9 (b) consumes less energy. (In our real-time watermark detector, the measured CPU power is 22.8 watts vs. 23.6 watts[7].) The key is to overlap the busy cycles of one logical processor with those of the other.

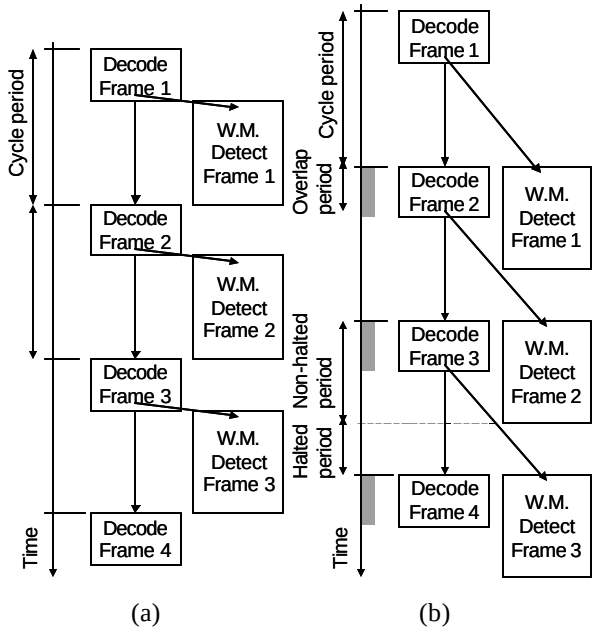


Figure 9: Two different methods of multithreading real-time applications. (a) uses more energy than (b)

CONCLUSION

In this paper we explained how typical media applications can benefit from Hyper-Threading Technology. From the increases in UPCs, we have observed that Hyper-Threading Technology can increase the utilization of processor resources by 15 to 27%, even for well-optimized multimedia applications. The results given in this paper also show that it is possible to benefit from Hyper-Threading Technology to save power when executing a fixed task.

Moreover, it has been shown that it is crucial to reach an optimal load balancing for an efficient implementation on Hyper-Threading Technology. This can usually be done for media applications exploiting both data and functional decompositions. Such partitioning, especially with a dynamic scheduling scheme, benefits in most cases from the fact that, unlike in symmetric multiprocessor systems,

[6] Real-time in this work means that applications need to perform some tasks periodically, while throughput-oriented applications just finish all the tasks as fast as possible.

[7] Here, we use average power as the indicator for energy saving. In real-time applications, power saving and energy saving can be used interchangeably.

threads share the cache in a processor with Hyper-Threading Technology.

Finally, the results show that for complex media applications running on Hyper-Threading Technology, in which multiple threads typically interact together and access memory concurrently, the thread synchronization issues and the overall data and functional partitioning are more important than the individual function characteristics.

ACKNOWLEDGMENTS

We acknowledge the exceptional efforts of the people at the Intel Nizhny Novgorod Lab in developing the encoder/decoder used in this study, especially Valery Kuriakin. Additionally, we thank Doug Carmean, Mike Upton, Per Hammarlund, Russell Arnold, Shihjong Kuo, George K. Chen, and Stephen Gunther for their help in setting up our Hyper-Threading Technology hardware and software environments and for valuable discussions during this work.

REFERENCES

[1] E. Debes, M. Holliman, W. Macy, Y.-K. Chen, and M. Yeung, "Computational Analysis and System Implications of Video Watermarking Applications," in *Proceedings of SPIE Conference on Security and Watermarking of Multimedia Contents IV*, Jan. 2002.

[2] S. Gunther, F. Binns, D. Carmean, and J. Hall, "Managing the Impact of Increasing Microprocessor Power Consumption," *Intel Technology Journal, Q1 2001*.

[3] B. G. Haskell, A. Puri, and A. N. Netravali, *Digital Video: An Introduction to MPEG-2*, MA: Kluwer, 1997.

[4] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium® 4 Processor," *Intel Technology Journal, Q1 2001*.

[5] Intel Corp., *Intel® Performance Library Suite*, (available on-line: http://developer.intel.com/software/products/perflib/index.htm)

[6] Intel Corp., *Intel® Pentium® 4 Processor Optimization Reference Manual*, Order Number: 248966 (also available on-line: http://developer.intel.com/design/pentium4/manuals/24896604.pdf)

[7] D. Marr, et al., "Hyper-Threading Technology Microarchitecture and Performance," *Intel Technology Journal, Q1 2002*.

[8] H. Wang, P. Wang, R. D. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, and J. Shen, "Speculative Precomputation: Exploring the Use of Multithreading Technology for Latency," *Intel Technology Journal, Q1 2002*.

AUTHORS' BIOGRAPHIES

Yen-Kuang Chen is a researcher in the Media Systems Lab, Microprocessor Research, Intel Labs. His research interests include video compression and processing, architecture and algorithm design in multimedia computing, video and graphics hardware design, and performance evaluation. He received a Ph.D. in electrical engineering from Princeton University. His e-mail is yen-kuang.chen@intel.com.

Matthew J. Holliman is a researcher in the Media Systems Lab, Microprocessor Research, Intel Labs. His research interests include media and Internet technology, focusing on content delivery and protection. His e-mail is matthew.holliman@intel.com.

Eric Debes is a researcher in the Media Systems Lab, Microprocessor Research, Intel Labs. His research interests include media coding, processing, communications and content protection as well as microarchitecture design and parallelism in computer architecture. He received an M.S. degree in electrical and computer engineering from Supélec, France, an M.S. degree in electrical engineering from the Technical University Darmstadt, Germany and a Ph.D. degree from the Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland. His e-mail is Eric.Debes@intel.com.

Sergey Zheltov is a project manager in Microprocessor Research, Intel Labs. His research interests include media compression and processing, software and platforms architecture, signal processing, high-order spectra. He received a Diploma in radio-physical engineering and MS degree in theoretical and mathematical physics from Nizhny Novgorod State University. His e-mail is Sergey.Zheltov@intel.com.

Alexander Knyazev is a software engineer in Microprocessor Research, Intel Labs. His research interests include video compression and processing, multimedia software architecture, platforms architecture, test, rough set and fuzzy logic theories. He received a Master's Degree of Applied Mathematics and Computer Science from Nizhny Novgorod State University. His e-mail is Alexander.Knyazev@intel.com.

Stanislav Bratanov is a software engineer in Microprocessor Research, Intel Labs. His research interests include multi-processor software platforms, operating system environments, and platform-dependent media data coding. He graduated from Nizhny Novgorod

State University, Russia. His e-mail is Stanislav.Bratanov@intel.com.

Roman Belenov is a software engineer in Microprocessor Research, Intel Labs. His research interests include video compression and processing, multimedia software architecture and wireless networking. He received a Diploma in physics from Nizhny Novgorod State University. His e-mail is Roman.Belenov@intel.com

Ishmael Santos is a Hardware Engineer for Power and Trace Technologies in Software Solutions Group, Intel Corporation. His interests include computer architecture with an emphasis on microprocessor power consumption and performance. Ishmael received his B.S. in Electrical Engineering and Computer Science from the University of California, Los Angeles. His e-mail is Ishmael.F.Santos@intel.com.

Hyper-Threading Technology: Impact on Compute-Intensive Workloads

William Magro, Software Solutions Group, Intel Corporation
Paul Petersen, Software Solutions Group, Intel Corporation
Sanjiv Shah, Software Solutions Group, Intel Corporation

Index words: SMP, SMT, Hyper-Threading Technology, OpenMP, Compute Intensive, Parallel Programming, Multi-Threading

ABSTRACT

Intel's recently introduced Hyper-Threading Technology promises to increase application- and system-level performance through increased utilization of processor resources. It achieves this goal by allowing the processor to simultaneously maintain the context of multiple instruction streams and execute multiple instruction streams or *threads*. These multiple streams afford the processor added flexibility in internal scheduling, lowering the impact of external data latency, raising utilization of internal resources, and increasing overall performance.

We compare the performance of an Intel Xeon processor enabled with Hyper-Threading Technology to that of a dual Xeon processor that does not have Hyper-Threading Technology on a range of compute-intensive, data-parallel applications threaded with OpenMP[1]. The applications include both real-world codes and hand-coded "kernels" that illustrate performance characteristics of Hyper-Threading Technology.

The results demonstrate that, in addition to functionally decomposed applications, the technology is effective for many data-parallel applications. Using hardware performance counters, we identify some characteristics of applications that make them especially promising candidates for high performance on threaded processors.

Finally, we explore some of the issues involved in threading codes to exploit Hyper-Threading Technology, including a brief survey of both existing and still-needed tools to support multi-threaded software development.

[1] OpenMP is an industry-standard specification for multi-threading data-intensive and other highly structured applications in C, C++, and Fortran. See www.openmp.org for more information.

INTRODUCTION

While the most visible indicator of computer performance is its clock rate, overall system performance is also proportional to the number of instructions retired per clock cycle. Ever-increasing demand for processing speed has driven an impressive array of architectural innovations in processors, resulting in substantial improvements in clock rates and instructions per cycle.

One important innovation, super-scalar execution, exploits multiple execution units to allow more than one operation to be in flight simultaneously. While the performance potential of this design is enormous, keeping these units busy requires super-scalar processors to extract independent work, or instruction-level parallelism (ILP), directly from a single instruction stream.

Modern compilers are very sophisticated and do an admirable job of exposing parallelism to the processor; nonetheless, ILP is often limited, leaving some internal processor resources unused. This can occur for a number of reasons, including long latency to main memory, branch mis-prediction, or data dependences in the instruction stream itself. Achieving additional performance often requires tedious performance

analysis, experimentation with advanced compiler optimization settings, or even algorithmic changes. Feature sets, rather than performance, drive software economics. This results in most applications never undergoing performance tuning beyond default compiler optimization.

An Intel processor with Hyper-Threading Technology offers a different approach to increasing performance. By presenting itself to the operating system as two logical processors, it is afforded the benefit of simultaneously scheduling two potentially independent instruction streams [1]. This explicit parallelism complements ILP to increase instructions retired per cycle and increase overall system utilization. This approach is known as simultaneous multi-threading, or SMT.

Because the operating system treats an SMT processor as two separate processors, Hyper-Threading Technology is able to leverage the existing base of multi-threaded applications and deliver immediate performance gains.

To assess the effectiveness of this technology, we first measure the performance of existing multi-threaded applications on systems containing the Intel® Xeon processor with Hyper-Threading Technology. We then examine the system's performance characteristics more closely using a selection of hand-coded application kernels. Finally, we consider the issues and challenges application developers face in creating new threaded applications, including existing and needed tools for efficient multi-threaded development.

APPLICATION SCOPE

While many existing applications can benefit from Hyper-Threading Technology, we focus our attention on single-process, numerically intensive applications. By numerically intensive, we mean applications that rarely wait on external inputs, such as remote data sources or network requests, and instead work out of main system memory. Typical examples include mechanical design analysis, multi-variate optimization, electronic design automation, genomics, photo-realistic rendering, weather forecasting, and computational chemistry.

A fast turnaround of results normally provides significant value to the users of these applications through better quality products delivered more quickly to market. The data-intensive nature of these codes, paired with the demand for better performance, makes them ideal candidates for multi-threaded speed-up on shared memory multi-processor (SMP) systems.

We considered a range of applications, threaded with OpenMP, that show good speed-up on SMP systems. The applications and their problem domains are listed in Table 1. Each of these applications achieves 100% processor utilization from the operating system's point of view. Despite external appearances, however, *internal* processor resources often remain underutilized. For this reason, these applications appeared to be good candidates for additional speed-up via Hyper-Threading Technology.

Table 1: Applications type

Code	Description
A1	Mechanical Design Analysis (finite element method) This application is used for metal-forming, drop testing, and crash simulation.
A2	Genetics A genetics application that correlates DNA samples from multiple animals to better understand congenital diseases.
A3	Computational Chemistry This application uses the self-consistent field method to compute chemical properties of molecules such as new pharmaceuticals.
A4	Mechanical Design Analysis This application simulates the metal-stamping process.
A5	Mesoscale Weather Modeling This application simulates and predicts mesoscale and regional-scale atmospheric circulation.
A6	Genetics This application is designed to generate Expressed Sequence Tags (EST) clusters, which are used to locate important genes.
A7	Computational Fluid Dynamics This application is used to model free-surface and confined flows.
A8	Finite Element Analysis This finite element application is specifically targeted toward geophysical engineering applications.
A9	Finite Element Analysis This explicit time-stepping application is used for crash test studies and computational fluid dynamics.

One might suspect that, for applications performing very similar operations on different data, the instruction streams might be too highly correlated to share a

threaded processor's resources effectively. Our results show differently.

METHODOLOGY

To assess the effectiveness of Hyper-Threading Technology for this class of applications, we measured the performance of existing multi-threaded executables, with no changes to target the threaded processor specifically.

We measured the elapsed completion time of stable, reproducible workloads using operating-system-provided timers for three configurations:

1. single-threaded execution on an single-processor SMT system
2. dual-threaded execution on a single-processor SMT system
3. dual-threaded execution on a dual-processor, non-SMT system

We then computed application speed-up as the ratio of the elapsed time of a single-threaded run to that of a multi-threaded run. Using the Intel VTune Performance Analyzer, we gathered the following counter data directly from the processor during a representative time interval of each application[2]:

Clock cycles

Instructions retired

Micro-operations retired

Floatingpoint instructions retired

From this raw data, we evaluated these ratios:

Clock cycles per instruction retired (CPI)

Clock cycles per micro-operation retired (CPu)

Fractional floating-point instructions retired (FP%)

APPLICATION RESULTS

Naturally, highly scalable applications; that is, those that speed up best when run on multiple, physical processors, are the best candidates for performance improvement on a threaded processor. We expect less scalable applications to experience correspondingly smaller potential benefits.

As shown in Figure 1, this is generally the case, with all of the applications, except application A1, receiving a significant benefit from the introduction of Hyper-Threading Technology. It is important to note that the applications realized these benefits with little to no incremental system cost and no code changes.

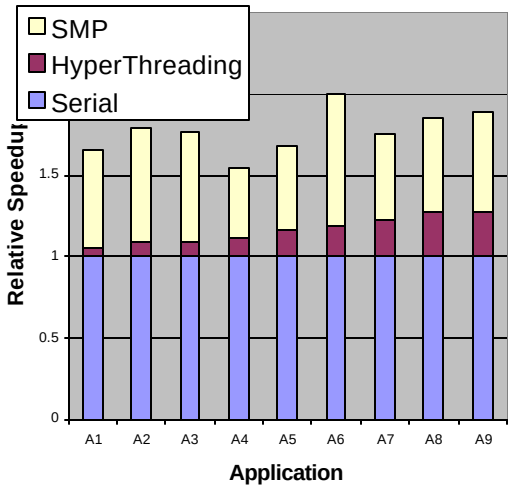


Figure 1: Application relative speed-up

Because the Intel Xeon processor is capable of retiring up to three micro-operations per cycle, the best-case value of clocks per micro-op (CPu) is 1/3. Table 2 shows counter data and performance results for the application experiments. The comparatively high CPI and CPu values indicate an individual stream does not typically saturate internal processor resources. While not sufficient, high CPu is a necessary condition for good speed-up in an SMT processor.

VTune is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[2] The counters and their significance are described in the Appendix.

Application	Cycles/instruction	Cycles/uop	FP%	SMT speedup	SMP Speedup
A1	2.04	1.47	29	1.05	1.65
A2	1.11	0.89	4.6	1.09	1.79
A3	1.69	0.91	16	1.09	1.77
A4	1.82	1.29	20	1.11	1.54
A5	2.48	1.45	36	1.17	1.68
A6	2.54	1.60	0.1	1.19	2.00
A7	2.80	2.05	10	1.23	1.75
A8	1.69	1.27	19	1.28	1.85
A9	2.26	1.76	20	1.28	1.89

Table 2: Counter data and performance results

Exactly which resources lie idle, however, is not clear. The fraction of floating-point instructions (FP%) gives one indication of the per-stream instruction mix. For the chosen applications, the FP% ranges from zero, for application A6, to the range of 4.6% to 35.5% for the remaining applications. It may seem unusual that the FP% of these numerically intensive applications is so low; however, even in numerically intensive code, many other instructions are used to index into arrays, manage data structures, load/store to memory, and perform flow control. The result can be a surprisingly balanced instruction mix.

Even though the instruction mix *within* a stream may be varied, a data parallel application typically presents pairs of similar or even identical instruction streams that could compete for processor resources at each given moment. The performance results, however, show that Hyper-Threading Technology is able to overlap execution of even highly correlated instruction streams effectively. To understand how this can occur, consider two threads consisting of identical instruction streams. As these threads execute, spatial correlation exists only with particular temporal alignments; a slight shift in the timing of the streams can eliminate the correlation, allowing a more effective interleaving of the streams and their resource demands. The net result is that two identical but time-shifted instruction streams can effectively share a pool of resources.

By reducing the impact of memory latency, branch mis-prediction penalties, and stalls due to insufficient ILP, Hyper-Threading Technology allows the Xeonprocessor to more effectively utilize its internal resources and increase system throughput.

TEST KERNEL RESULTS

To examine these effects more closely, we developed four test kernels. The first two kernels (int_mem and dbl_mem) illustrate the effects of latency hiding in the memory hierarchy, while the third kernel (int_dbl) attempts to avoid stalls due to low ILP. The fourth kernel (matmul) and a corresponding, tuned library function illustrate the interplay between high ILP and SMT speed-up. The performance results of all the kernels are shown in Table 3. The int_mem kernel, shown in Figure 2, attempts to overlap cache misses with integer operations. It first creates a randomized access pattern into an array of cache-line-sized objects, then indexes into the objects via the randomized index vector and performs a series of addition operations on the cache line.

```
#pragma omp for
    for (i = 0; i < buf_len; ++i) {
        j = index[ i ];
        for (k = 0; k < load; ++k) {
            buffer[ j ][ 0 ] += input;
            buffer[ j ][ 1 ] += input;
            buffer[ j ][ 2 ] += input;
            buffer[ j ][ 3 ] += input;
        }
    }
```

Figure 2: The int_mem benchmark

Table 3: Kernel performance results

Code	Benchmark	CPI	CPuops	FP%	SMT Speed-up
M1	int_mem (load=32)	1.99	0.94	0.0%	1.08
M2	int_mem (load=4)	6.91	3.61	0.0%	1.36
M3	dbl_mem (load=32)	1.81	1.47	23.2%	1.90
M4	int_dbl	3.72	1.63	9.8%	1.76
M5	matmul	2.17	1.60	34.5%	1.64
M6	dgemm	1.63	1.58	58.0%	1.00

We tested two variants (M1 and M2). In the first, we assigned a value of 32 to the parameter "load"; in the second test, "load" was 4. The larger value of "load" allows the processor to work repeatedly with the same data. Cache hit rates are consequently high, as is integer unit utilization. Smaller values of "load" cause the code to access second-level cache and main memory more often, leading to higher latencies and increased demand on the memory subsystem. Provided these additional accesses do not saturate the memory bus bandwidth, the processor can overlap the two threads' operations and effectively hide the memory latency. This point is demonstrated by the inverse relationship between clocks per instruction and speed-up.

The dbl_mem kernel is identical to int_mem, but with the data variables changed to type "double." The results with "load" equal to 32 (M3) demonstrate the same effect, instead interleaving double-precision floating-point instructions with cache misses. In addition, the floating-point operations can overlap with the supporting integer instructions in the instruction mix to allow the concurrent use of separate functional units resulting in near-linear speed-up.

The int_dbl kernel (M4), shown in Figure 3, calculates an approximation to Pi via a simple Monte Carlo method. This method uses an integer random number generator to choose points in the x-y plane from the range [-1…1]. It then converts these values to floating point and uses each point's distance from the origin to determine if the point falls within the area of the unit radius circle. The fraction of points that lies within this circle approximates Pi/4. Like dbl_mem, this kernel achieves excellent speed-up, but for a different reason: the different functional units inside the processor are utilized simultaneously.

```
#pragma omp for reduction(+:count)
    for (i = 0; i < NPOINTS; ++i) {
        double x, y;
        //  guess  returns  a  pseudo-random
number
        x = guess(&seed, 2.0)-1.0;
        y = guess(&seed, 2.0)-1.0;
        if ( sqrt(x*x + y*y) <= 1.0 ) {
            /* The current point is
               inside the circle... */
            ++count;
        }
    }
```

Figure 3: The int_dbl benchmark

The "matmul" kernel (M5), shown in Figure 4, computes the product of two 1000 x 1000 matrices using a naï ve loop formulation written in FORTRAN. Comparing its absolute performance and speed-up to that of a functionally equivalent, but hand-tuned library routine illustrates the effect of serial optimization on the effectiveness of Hyper-Threading Technology. The naï ve loop formulation (M5) has comparatively poor absolute performance, executing in 3.4 seconds, but achieves good SMT speed-up. The hand-optimized dgemm (M6) library routine executes in a fraction of the time (0.6s), but the speed-up vanishes. The highly tuned version of the code effectively saturates the processor, leaving no units idle[3].

```
!$omp parallel do
      DO  26  J = 1,N
          DO  24  K = 1,N
              DO  22  I = 1,N
                   C(I,J) = C(I,J) + A(I,K)
* B(K,J)
   22         CONTINUE
   24     CONTINUE
   26 CONTINUE
```

[3] Note that the FP% for M6 is due to SIMD packed double precision instructions, rather than the simpler x87 instructions used by the other test codes.

Figure 4: The Matmul kernel

MEMORY HIERARCHY EFFECTS

Depending on the application characteristics, Hyper-Threading Technology's shared caches [1] have the potential to help or hinder performance. The threads in data parallel applications tend to work on distinct subsets of the memory, so we expected this to halve the effective cache size available to each logical processor. To understand the impact of reduced cache, we formulated a very simplified execution model of cache-based system.

In a threaded microprocessor with two logical processors, the goal is to execute both threads with no resource contention issues or stalls. When this occurs, two fully independent threads should be able to execute an application in half the time of a single thread. Likewise, each thread can execute up to 50% more slowly than the single-threaded case and still yield speed-up.

Figure 5 exhibits the approximate time to execute an application on a hypothetical system with a three-level memory hierarchy consisting of registers, cache, and main memory.

Given:

N = Number of instructions executed
F_{memory} = Fraction of N that access memory
G_{hit} = Fraction of loads that hit the cache
T_{proc} = #cycles to process an instruction
T_{cache} = #cycles to process a hit
T_{memory} = #cycles to process a miss

T_{exe} = Execution time

Then:

$$T_{exe}/N = (1 - F_{memory})\, T_{proc} + F_{memory}\, [G_{hit}\, T_{cache} + (1 - G_{hit})\, T_{memory}]$$

Figure 5: Simple performance model for a single-level cache system

While cache hit rates, G_{hit}, cannot be easily estimated for the shared cache, we can explore the performance impact of a range of possible hit rates. We assume F_{memory}=20%, T_{proc}=2, T_{cache}=3, and T_{memory}=100. For a given cache hit rate in the original, single-threaded execution, Figure 6 illustrates the effective miss rate, T'_{miss}, which would cause the thread to run twice as slowly as in serial. Thus, any hit rate that falls in the shaded region between the curves should result in overall speed-up when two threads are active.

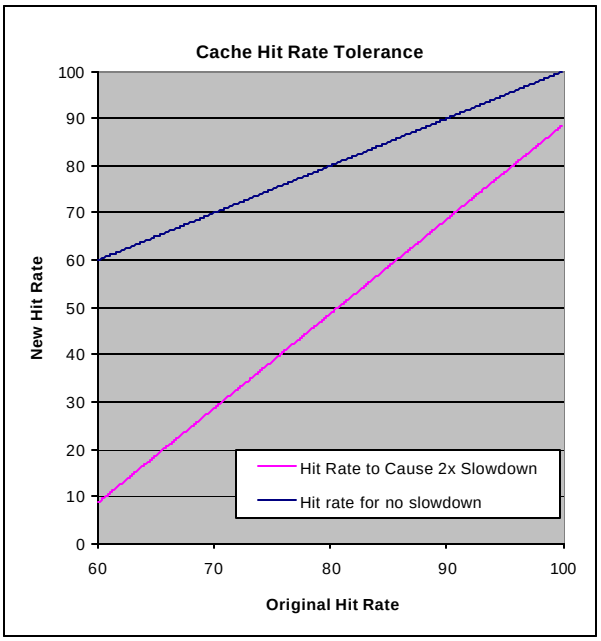


Figure 6: Hit rate tolerance for 2x slowdown in performance

The shaded region narrows dramatically as the original cache hit rate approaches 100%, indicating that applications with excellent cache affinity will be the least tolerant of reduced effective cache size. For example, when a single-threaded run achieves a 60% hit rate, the dual-threaded run's hit rate can be as low as 10% and still offer overall speed-up. On the other hand, an application with a 99% hit rate must maintain an 88% hit rate in the smaller cache to avoid slowdown.

TOOLS FOR MULTI-THREADING

It is easy to see that the existence of many multi-threaded applications increases the utility of Hyper-Threading Technology. In fact, every multi-threaded application can potentially benefit from SMT without modification. On the other hand, if no applications were multi-threaded, the only obvious benefits from SMT would be throughput benefits from multi-process parallelism. Shared memory parallel computers have existed for more than a decade, and much of the performance benefits of multi-threading have been available, yet few multi-threaded applications exist. What are some of the reasons for this lack of multi-threaded applications, and how might SMT technology change the situation?

First and foremost among these reasons is the difficulty of building a correct and well-performing multi-threaded application. While it is not impossible to build such applications, it tends to be significantly more difficult

than building sequential ones. Consequently, most developers avoid building multi-threading applications until their customers demand additional performance. The following constraints often drive performance requirements:

> Real-time requirements to accomplish some computing task that cannot be satisfied by a single processor, e.g., weather forecasting, where a 24-hour forecast has value only if completed and published in well under 24 hours.
>
> Throughput requirements, usually in interactive applications, such that users are not kept waiting for too long, e.g., background printing while editing in a word processor.
>
> Turnaround requirement, where job completion time materially impacts the design cycle, e.g., computational fluid dynamics used in the design of aircraft, automobiles, etc.

Most software applications do not have the above constraints and are not threaded. A good number of applications do have the throughput requirement, but that particular one is easier to satisfy without particular attention to correctness or performance.

Another reason for the lack of many multi-threaded applications has been the cost of systems that can effectively utilize multiple threads. Up until now, the only kinds of systems that could provide effective performance benefits from multiple threads were expensive multiple-processor systems. Hyper-Threading Technology changes the economics of producing multi-processor systems, because it eliminates much of the additional "glue" hardware that previous systems needed.

Economics alone cannot guarantee a better computing experience via the efficient utilization of Hyper-Threading Technology. Effective tools are also necessary to create multi-threaded applications. What are some of the capabilities of these tools? Do such tools already exist in research institutions?

One of the difficulties is the lack of a good programming language for multi-threading. The most popular multi-threading languages are the POSIX threads API and the Windows* Threads API. However, these are the threading equivalent of assembly language, or C at best. All the burden of creating high-level structures is placed upon the programmer, resulting in users making the same mistakes repeatedly. Modern programming languages like Java and C# include threading as a part of the language, but again few high-level structures are available for programmers. These languages are only marginally better than the threading APIs. Languages like OpenMP [3,5] do offer higher-level constructs that address synchronous threading issues well, but they offer little for asynchronous threading. Even for synchronous threading, OpenMP [3,5] has little market penetration outside the technical computing market. If OpenMP [3,5] can successfully address synchronous threading outside the technical market, it needs to be deployed broadly to ease the effort required to create multi-threaded applications correctly. For asynchronous threading, perhaps the best model is the Java- and C#-like threading model, together with the threading APIs.

Besides threaded programming languages, help is also needed in implementing correct threaded programs. The timing dependencies among the threads in multi-threaded programs make correctness validation an immense challenge. However, race detection tools have existed in the research community for a long time, and lately some commercial tools like Visual Threads [7] and Assure [6] have appeared that address these issues. These tools are extremely good at finding bugs in threaded programs, but they suffer from long execution times and large memory-size footprints. Despite these issues, these tools are a very promising start for ensuring the correctness of multi-threaded programs and offer much hope for the future.

After building a correct multi-threaded program, a tool to help with the performance analysis of the program is also required. There are some very powerful tools today for analysis of sequential applications, like the VTune Performance Analyzer. However, the equivalent is missing for multi-threaded programs. Again, for OpenMP [3,5], good performance-analysis tools do exist in the research community and commercially. These tools rely heavily on the structured, synchronous OpenMP [3,5] programming model. The same tools for asynchronous threading APIs are non-existent, but seem necessary for the availability of large numbers of multi-threaded applications. Hyper-Threading Technology presents a unique challenge for performance-analysis tools, because the processors share resources and neither processor has all of the resources available at all times. In order to create a large pool of multi-threaded applications, it seems clear that effective tools are necessary. It is also clear that such tools are not yet

available today. To exploit Hyper-Threading Technology effectively, multi-threaded applications are necessary, and tools to create those are key.

CONCLUSION

High clock rates combined with efficient utilization of available processor resources can yield high application performance. As microprocessors have evolved from simple single-issue architectures to the more complex multiple-issue architectures, many more resources have become available to the microprocessor. The challenge now is effective utilization of the available resources. As processor clock frequencies increase relative to memory access speed, the processor spends more time waiting for memory accesses. This gap can be filled using extensions of techniques already in use, but the cost of these improvements is often greater than the relative gain. Hyper-Threading Technology uses the explicit parallel structure of a multi-threaded application to complement ILP and exploit otherwise wasted resources. Under carefully controlled conditions, such as the test kernels presented above, the speed-ups can be quite dramatic.

Real applications enjoy speed-ups that are more modest. We have shown that a range of existing, data-parallel, compute-intensive applications benefit from the presence of Hyper-Threading Technology with no source code changes. In this suite of multi-threaded applications, every application benefited from threading in the processor. Like assembly language tuning, Hyper-Threading Technology provides another tool in the application programmer's arsenal for extracting more performance from his or her computer system. We have shown that high values of clock cycles per instruction and per micro-op are indicative of opportunities for good speed-up.

While many existing multi-threaded applications can immediately benefit from this technology, the creation of additional multi-threaded applications is the key to fully realizing the value of Hyper-Threading Technology. Effective software engineering tools are necessary to lower the barriers to threading and accelerate its adoption into more applications.

Hyper-Threading Technology, as it appears in today's Intel Xeon processors, is just the beginning. The fundamental ideas behind the technology apply equally well to larger numbers of threads sharing additional resources. Just as the number of distinct lines in a telephone network grows slowly relative to the number of customers served, Hyper-Threading Technology has the potential to modestly increase the number of resources in the processor core and serve a large numbers of threads. This combination has the potential to hide almost any latency and utilize the functional units very effectively.

APPENDIX: PERFORMANCE METRICS

Using the VTune™ Performance Analyzer, one can collect several execution metrics *in situ* as the application runs. While the Intel Xeon™ processor contains a host of counters, we focused on the following set of raw values and derived ratios.

Clock Cycles

The number of clock cycles used by the application is a good substitute for the CPU time required to execute the application. For a single threaded run, the total clock cycles multiplied by the clock rate gives the total running time of the application. For a multithreaded application on a Hyper-Threading Technology-enabled processor, the process level measure of clock cycles is the sum of the clocks cycles for both threads.

Instructions Retired

When a program runs, the processor executes sequences of instructions, and when the execution of each instruction is completed, the instructions are retired. This metric reports the number of instructions that are retired during the execution of the program.

Clock Cycles Per Instruction Retired

CPI is the ratio of clock cycles to instructions retired. It is one measure of the processor's internal resource utilization. A high value indicates low resource utilization.

Micro-Operations Retired

Each instruction is further broken down into micro-operations by the processor. This metric reports the number of micro-operations retired during the execution of the program. This number is always greater than the number of instructions retired.

Clock Cycles Per Micro-Operations Retired

This derived metric is the ratio of retired micro-operations to clock cycles. Like CPI, it measures the processor's internal resource utilization. This is a finer measure of utilization than CPI because the execution engine operates directly upon micro-ops rather than instructions. The Xeon processor core is capable of retiring up to three micro-ops per cycle.

Percentage of Floating-Point Instructions

This metric measures the percentage of retired instructions that involve floating-point operations. To what extent the different functional units in the processor are busy can be determined by the instruction type mix because processors typically have multiple floating-point, integer, and load/store functional units. The percentage of floating-point instructions is an important indicator of whether the program is biased toward the use of a specific resource, potentially leaving other resources idle.

REFERENCES

[1] D. Marr, et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, Q1, 2002.

[2] Intel® Pentium® 4 Processor Optimization Reference Manual.

[3] http://developer.intel.com/software/products/compilers/

[4] http://www.openmp.org/

[5] http://developer.intel.com/software/products/kappro/

[6] http://developer.intel.com/software/products/assure/

[7] http://www.compaq.com/products/software/visualthreads/

AUTHORS' BIOGRAPHIES

William Magro manages the Intel Parallel Applications Center, which works with independent software vendors and enterprise developers to optimize their applications for parallel execution on multiple processor systems. He holds a B.Eng. degree in Applied and Engineering Physics from Cornell University and M.S. and Ph.D. degrees in Physics from the University of Illinois at Urbana-Champaign. His e-mail is bill.magro@intel.com.

Paul Petersen is a Principal Engineer at Intel's KAI Software Lab. He currently works with software development tools to simplify threaded application development. He has been involved in the creation of the OpenMP parallel programming language and tools for the performance and correctness evaluation of threaded applications. He holds a B.S. degree from the University of Nebraska and M.Sc. and Ph.D. degrees from the University of Illinois at Urbana-Champaign, all in Computer Science. His e-mail is paul.petersen@intel.com

Sanjiv Shah co-manages the compiler and tools groups at Intel's KAI Software Lab. He has worked on compilers for automatic parallelization and vectorization and on tools for software engineering of parallel applications. He has been extensively involved in the creation of the OpenMP specifications and serves on the OpenMP board of directors. Sanjiv holds a B.S. degree in Computer Science with a minor in Mathematics and an M.S. degree in Computer Science from the University of Michigan. His e-mail is sanjiv.shah@intel.com.