



Hyper-Threading Technology Table of Contents

Articles

Preface	2
Foreword	3
Hyper-Threading Technology Architecture and Microarchitecture	4
Pre-Silicon Validation of Hyper-Threading Technology	16
Speculative Precomputation: Exploring the Use of Multithreading for Latency Tools	22
Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance	36
Media Applications on Hyper-Threading Technology	47
Hyper-Threading Technology: Impact on Compute-Intensive Workloads	58

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.

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.

Hyper-Threading Technology Architecture and Microarchitecture

Deborah T. Marr, Desktop Products Group, Intel Corp.

Frank Binns, Desktop Products Group, 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.

®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.

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.

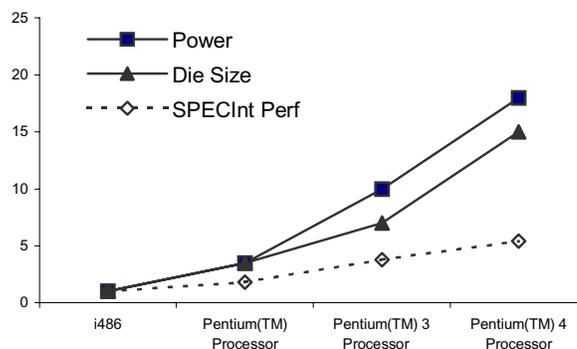


Figure 1: Single-stream performance vs. cost

Figure 1 shows the relative increase in performance and the costs, such as die size and power, over the last ten years on Intel processors¹. 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-fold¹. 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 period¹. 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 multithreading 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.

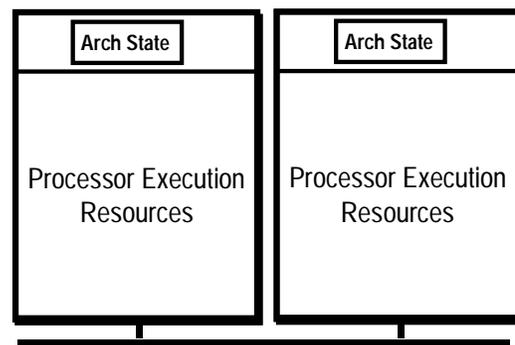
Finally, there is simultaneous multi-threading, where multiple threads can execute on a single processor without switching. The threads execute simultaneously and make much better use of the resources. This approach makes the most effective use of processor resources: it maximizes the performance vs.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

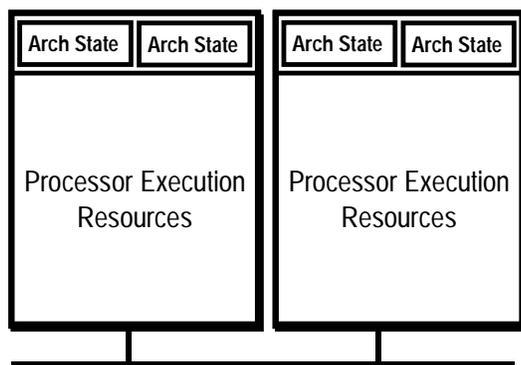


Figure 3: Processors with Hyper-Threading Technology

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,

[®] 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.

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 threads² 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.

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.

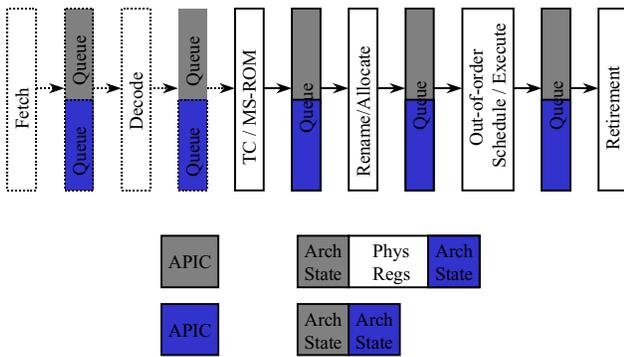


Figure 4 Intel® Xeon™ processor pipeline

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.

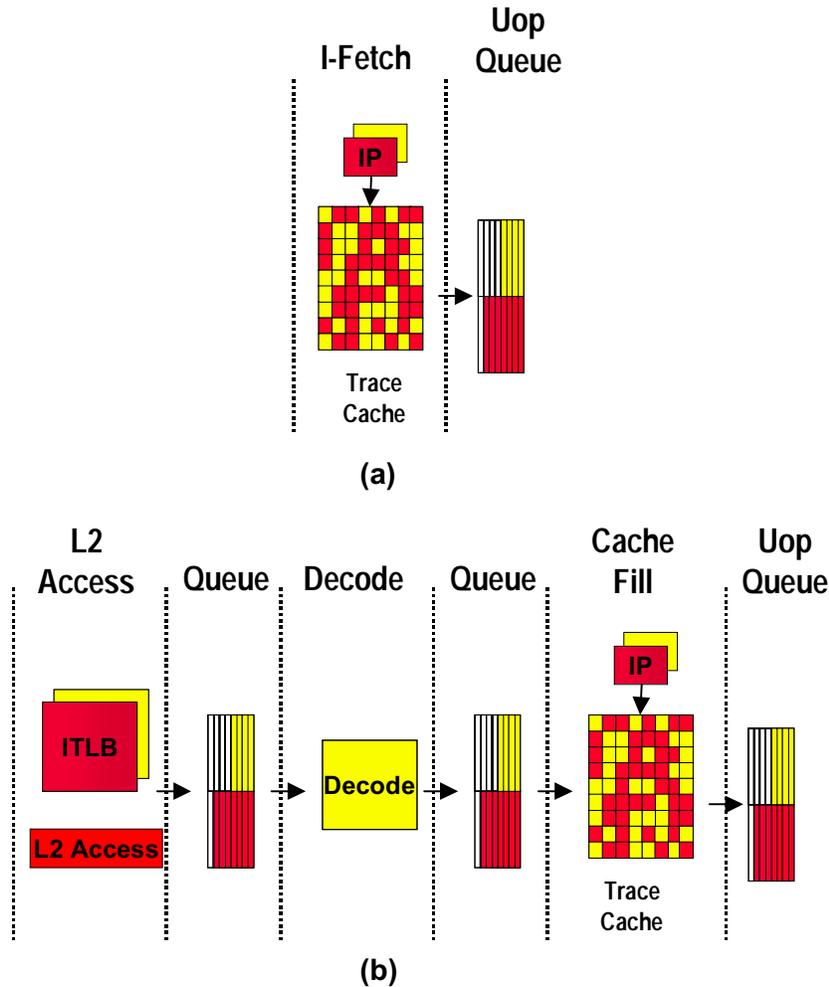


Figure 5: Front-end detailed pipeline (a) Trace Cache Hit (b) Trace Cache Miss

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.

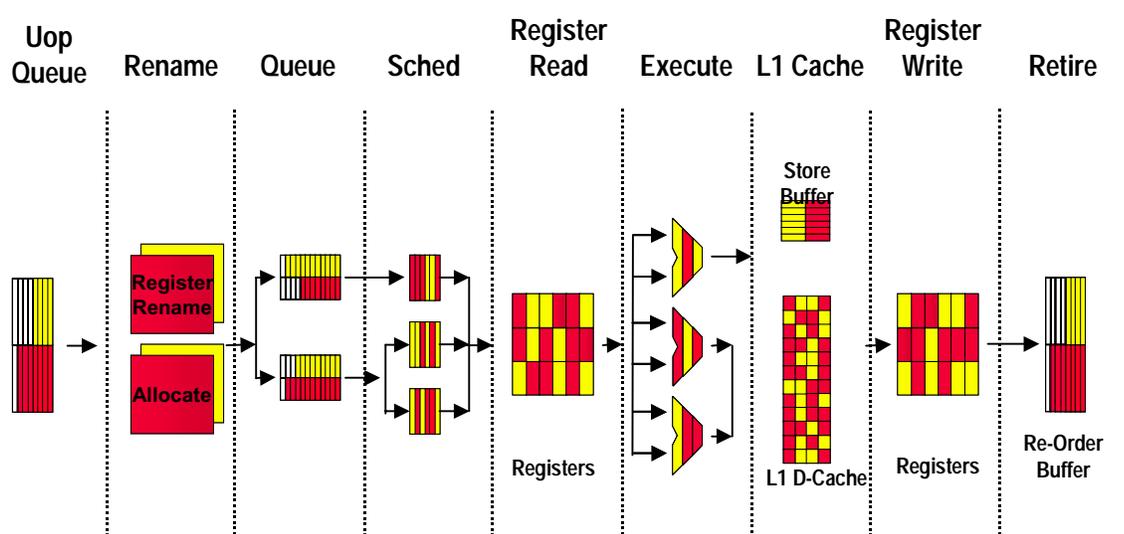


Figure 6: Out-of-order execution engine detailed pipeline

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

®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.

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 HALT processor would cause a transition to MT-mode. The operating system is responsible for managing MT-mode transitions (described in the next section).

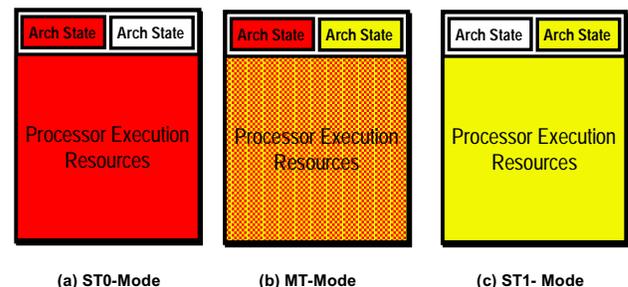


Figure 7: Resource allocation

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.

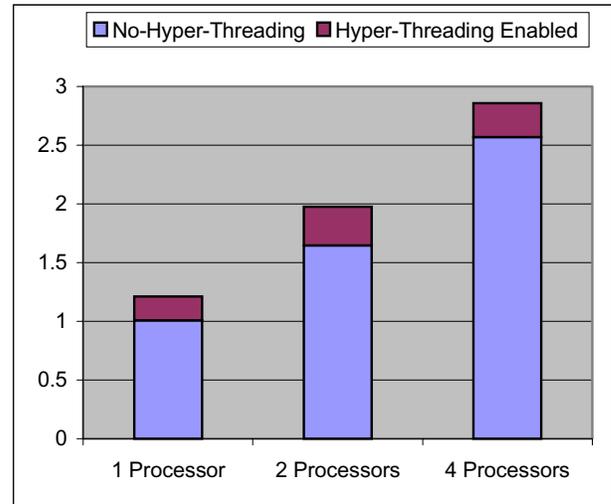


Figure 8: Performance increases from Hyper-Threading Technology on an OLTP workload

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.

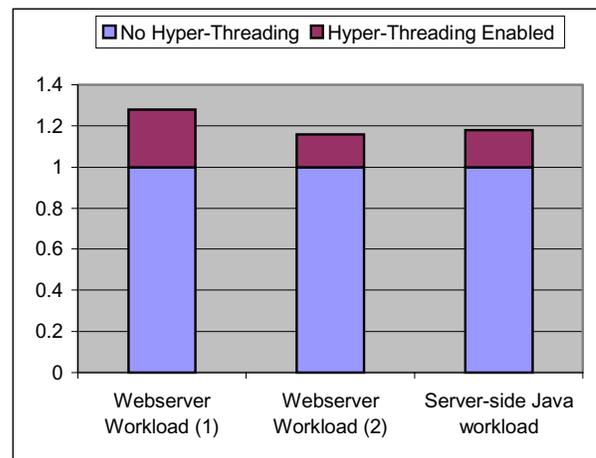


Figure 9: Web server benchmark performance

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. Kubiawicz, "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.

Copyright © Intel Corporation 2002.

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

This publication was downloaded from
<http://developer.intel.com/>

Legal notices at
<http://developer.intel.com/sites/corporate/tradmarx.htm>.

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.

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.

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*.

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

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

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.

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

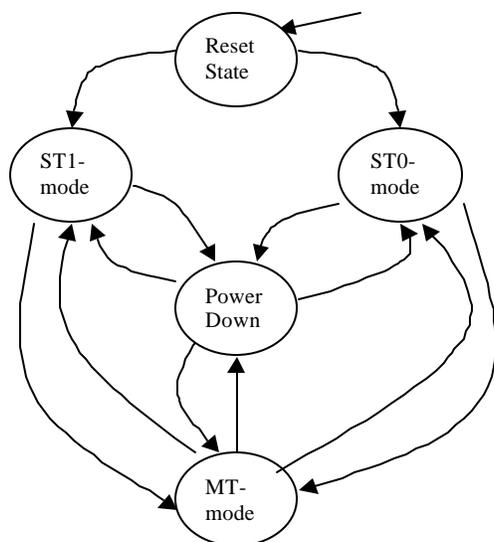


Figure 1: Operating Mode State Diagram

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

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

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.

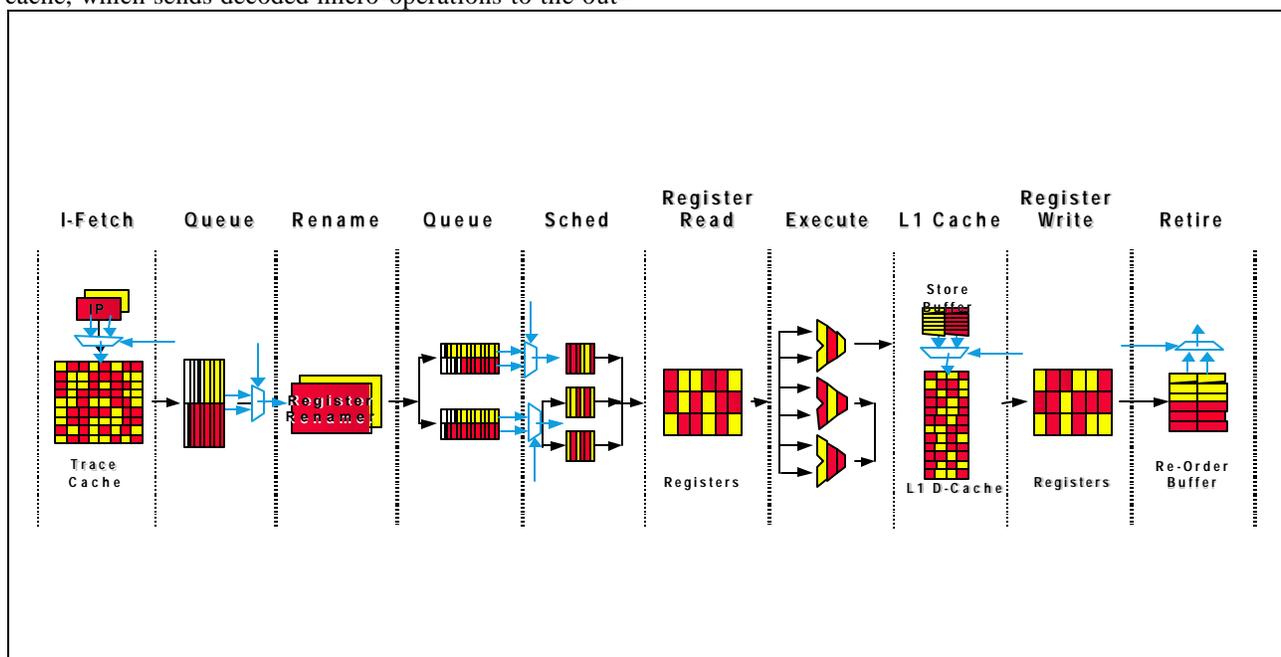


Figure 2: Logical processor selection point

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.

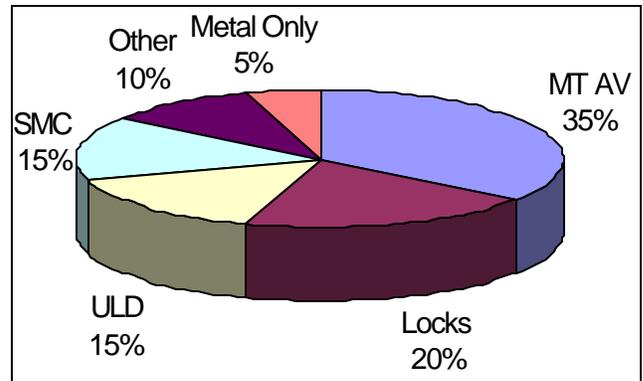


Figure 3: Breakdown of post-silicon MT-Mode bugs

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/article/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

Copyright © Intel Corporation 2002. Other names and brands may be claimed as the property of others.

This publication was downloaded from <http://developer.intel.com/>

Legal notices at <http://developer.intel.com/sites/corporate/tradmarx.htm>

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

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.

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.

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 superscalar 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

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

Xeon and Itanium are trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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*.

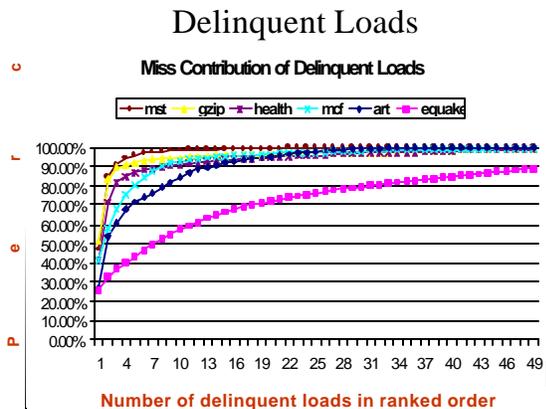


Figure 1: Cumulative L1 data cache misses due to delinquent loads

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.

Performance Impact of D-Loads

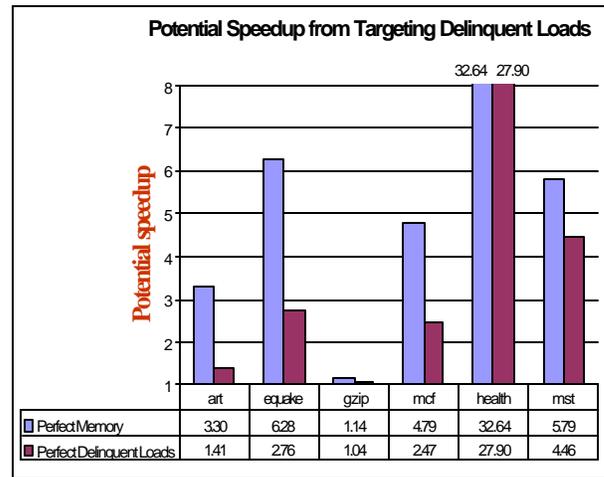


Figure 2: Speed-up when 10 delinquent loads are assumed to always hit in cache

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
<i>art</i>	2	4	3.5
<i>equake</i>	8	12.5	4.5
<i>gzip</i>	9	9.5	6.0
<i>mcf</i>	6	5.8	2.5
<i>health</i>	8	9.1	5.3
<i>mst</i>	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

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

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.

Potential Speed-up (Basic Triggers)

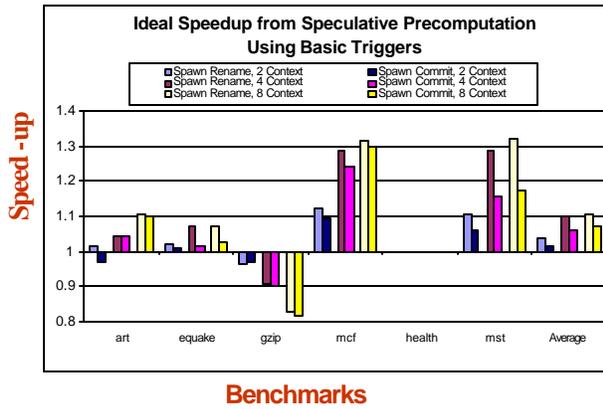


Figure 3: SP speed-up with basic trigger and ideal hardware assumptions

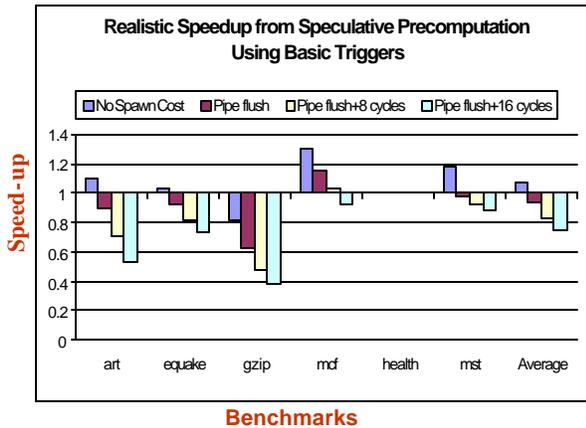


Figure 4: SP speed-up with basic trigger and realistic hardware

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.

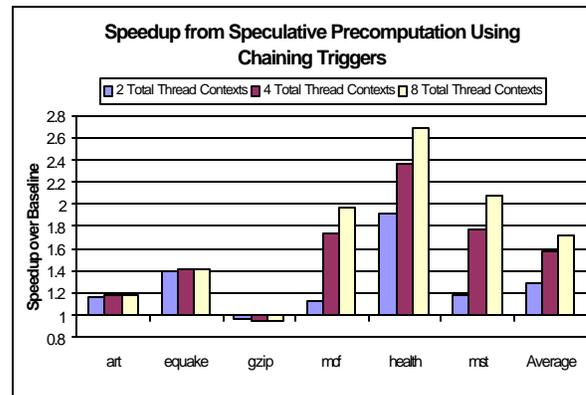


Figure 5: SP speed-up with chaining trigger and realistic hardware

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.

Sources of Speed-up

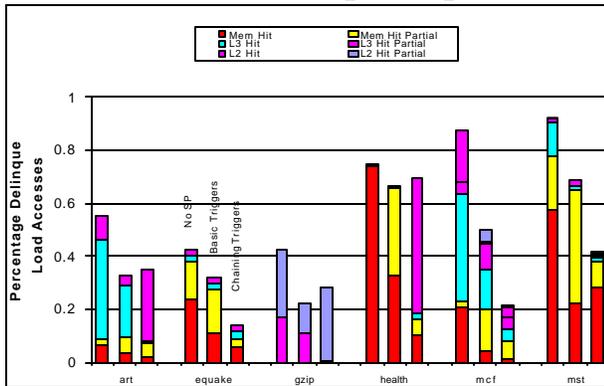


Figure 6: Reduction of cache misses in the memory hierarchy via SP-based prefetching

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.

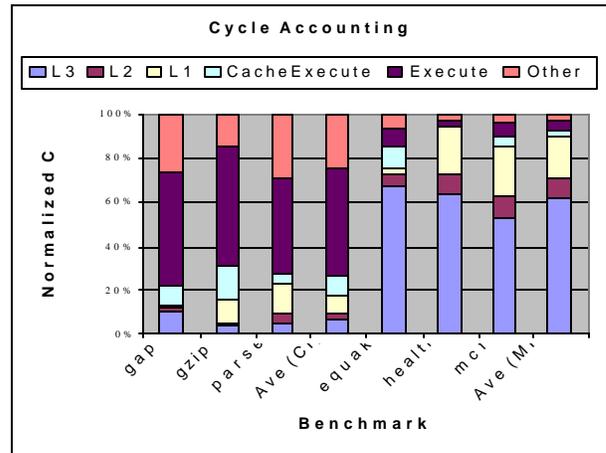


Figure 7: Characteristics of CPU-intensive vs. memory-intensive workloads on an in-order machine

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.

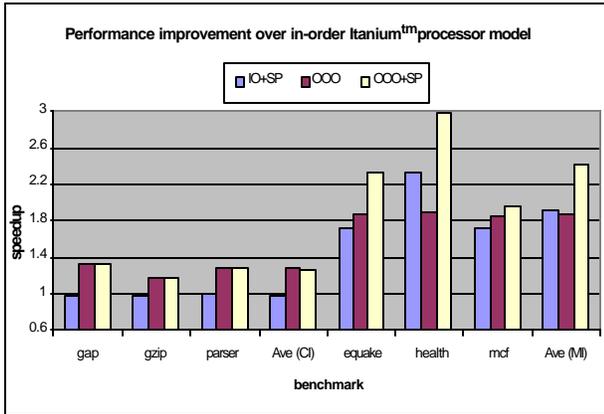


Figure 8: Speed-ups of in-order+SP, OOO, OOO+SP over in-order

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.

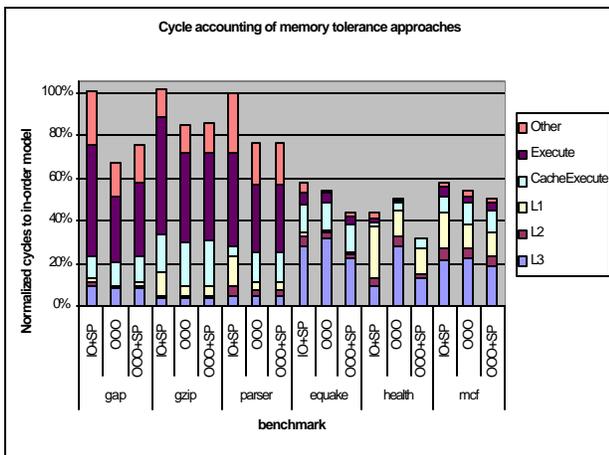


Figure 9: Cycle breakdown of in-order+SP, OOO and OOO+SP relative to in-order (100%)

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
<i>Synthetic</i>	Graph traversal in large random graph simulating large database retrieval	22% - 45%
<i>MST</i> (Olden)	Minimal Spanning Tree algorithm used for data clustering	23% - 40%
<i>Health</i> (Olden)	Hierarchical database modeling health care system	11% - 24%
<i>MCF</i> (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

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

Xeon and VTune are trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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()
  {
22  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 dereferencing loads originated at Line 5 are identified as delinquent with regard to L2 misses, and they incur significant latency.

Other brands may be claimed as the property of others.

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.

```

1 main()
2 {
3   CreateThread(T
4   WaitForSingleObject
5
6   n = NodeArray[0
7   while(n and
8   {
9     work()
10    n->i = n->next->j + n->next->k + n-
11    n = n-
12    remaining-
13    every stride
14    global_n =
15    global_r =
16    SetEvent(
17  }
18 }

```

SP: Main Thread

Line 11-12: Live-in's for cross thread transfer
Line 13: Trigger to activate SP thread

Figure 11: SP main thread pseudo-code

```

1 T()
2 {
3   Do Stride times
4   n->i = n->next->j + n->next->k + n->next->l
5   n = n->next
6   remaining--
7   SetEvent()
8   while(n and remaining)
9   {
10    Do Stride times
11    n->i = n->next->j + n->next->k + n->next->l
12    n = n->next
13    remaining--
14    WaitForSingleObject
15    if (remaining < global_r)
16    remaining = global_r
17    n = global_n
18  }
19 }

```

SP: Worker Thread

Line 9: Responsible for Most effective prefetch due to run-ahead
Line 13: Detect run-behind, adjust by jumping ahead

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.

<p>Main Thread Line 7 corresponds to Line 5 of single thread code oExecution time: 19% vs 49.46% in single-thread code oL2 miss: 0.61% vs 99.95% in single-thread code</p> <p>SP worker thread: Line 9, oExecution time: 26.21% oL2 miss: 97.61%</p> <p style="text-align: center;">SP successful in shouldering most L2 cache misses</p>
--

Figure 13: Why SSP-enabled code runs faster

Other brands may be claimed as the property of others.

