# Introduction to Multithreading, Superthreading and Hyperthreading

## Introduction

Back in the dual-Celeron days, when symmetric multiprocessing (SMP) first became cheap enough to come within reach of the average PC user, many hardware enthusiasts eager to get in on the SMP craze were asking what exactly (besides winning them the admiration and envy of their peers) a dual-processing rig could do for them. It was in this context that the PC crowd started seriously talking about the advantages of multithreading. Years later when Apple brought dual-processing to its PowerMac line, SMP was officially mainstream, and with it multithreading became a concern for the mainstream user as the ensuing round of benchmarks brought out the fact you really needed multithreaded applications to get the full benefits of two processors.

Even though the PC enthusiast SMP craze has long since died down and, in an odd twist of fate, Mac users are now many times more likely to be sporting an SMP rig than their x86-using peers, multithreading is once again about to increase in importance for PC users. Intel's next major IA-32 processor release, codenamed Prescott, will include a feature called **simultaneous multithreading** (SMT), also known as **hyper-threading**. To take full advantage of SMT, applications will need to be multithreaded; and just like with SMP, the higher the degree of multithreading the more performance an application can wring out of Prescott's hardware.

Intel actually already uses SMT in a shipping design: the Pentium 4 Xeon. Near the end of this article we'll take a look at the way the Xeon implements hyper-threading; this analysis should give us a pretty good idea of what's in store for Prescott. Also, it's rumored that the current crop of Pentium 4's actually has SMT hardware built-in, it's just disabled. (If you add this to the rumor about x86-64 support being present but disabled as well, then you can get some idea of just how cautious Intel is when it comes to introducing new features. I'd kill to get my hands on a 2.8 GHz P4 with both SMT and x86-64 support turned on.)

SMT, in a nutshell, allows the CPU to do what most users think it's doing anyway: run more than one program at the same time. This might sound odd, so in order to understand how it works this article will first look at how the current crop of CPUs handles multitasking. Then, we'll discuss a technique called superthreading before finally moving on to explain hyper-threading in the last section. So if you're looking to understand more about multithreading, symmetric multiprocessing systems, and hyper-threading then this article is for you.

As always, if you've read some of my previous tech articles you'll be well equipped to understand the discussion that follows. From here on out, I'll assume you know the basics of pipelined execution and are familiar with the general architectural division between a processor's front end and its execution core. If these terms are mysterious to you, then you might want to reach way back and check out my "Into the K7" article, as well as some of my other work on the P4 and G4e.

## Conventional multithreading

Quite a bit of what a CPU does is illusion. For instance, modern out-of-order processor architectures don't actually execute code sequentially in the order in which it was written. I've covered the topic of out-of-order execution (OOE) in previous articles, so I

won't rehash all that here. I'll just note that an OOE architecture takes code that was written and compiled to be executed in a specific order, reschedules the sequence of instructions (if possible) so that they make maximum use of the processor resources, executes them, and then arranges them back in their original order so that the results can be written out to memory. To the programmer and the user, it looks as if an ordered, sequential stream of instructions went into the CPU and identically ordered, sequential stream of computational results emerged. Only the CPU knows in what order the program's instructions were actually executed, and in that respect the processor is like a black box to both the programmer and the user.

The same kind of sleight-of-hand happens when you run multiple programs at once, except this time the operating system is also involved in the scam. To the end user, it appears as if the processor is "running" more than one program at the same time, and indeed, there actually are multiple programs loaded into memory. But the CPU can *execute* only one of these programs at a time. The OS maintains the illusion of concurrency by rapidly switching between running programs at a fixed interval, called a **time slice**. The time slice has to be small enough that the user doesn't notice any degradation in the usability and performance of the running programs, and it has to be large enough that each program has a sufficient amount of CPU time in which to get useful work done. Most modern operating systems include a way to change the size of an individual program's time slice. So a program with a larger time slice gets more actual execution time on the CPU relative to its lower priority peers, and hence it runs faster. (On a related note, this brings to mind one of my favorite .sig file quotes: "A message from the system administrator: 'I've upped my priority. Now up yours.'")

## Clarification of terms: "running" vs. "executing," and "front end" vs. "execution core."

For our purposes in this article, "running" does not equal "executing." I want to set up this terminological distinction near the outset of the article for clarity's sake. So for the remainder of this article, we'll say that a program has been launched and is "running" when its code (or some portion of its code) is loaded into main memory, but it isn't actually *executing* until that code has been loaded into the processor. Another way to think of this would be to say that the OS runs programs, and the processor executes them.

The other thing that I should clarify before proceeding is that the way that I divide up the processor in this and other articles differs from the way that Intel's literature divides it. Intel will describe its processors as having an "in-order front end" and an "out-of-order execution engine." This is because for Intel, the front-end consists mainly of the instruction fetcher and decoder, while all of the register rename logic, out-of-order scheduling logic, and so on is considered to be part of the "back end" or "execution core." The way that I and many others draw the line between front-end and back-end places all of the out-of-order and register rename logic in the front end, with the "back end"/"execution core" containing only the execution units themselves and the retire logic. So in this article, the front end is the place where instructions are fetched, decoded, and re-ordered, and the execution core is where they're actually executed and retired.

## Preemptive multitasking vs. Cooperative multitasking

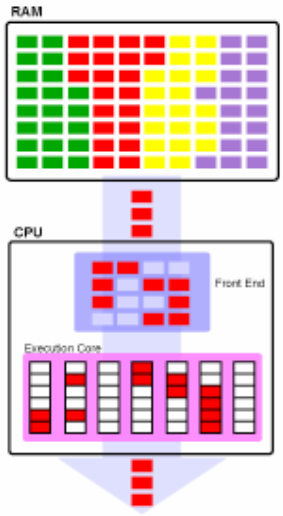
While I'm on this topic, I'll go ahead and take a brief moment to explain preemptive multitasking versus cooperative multitasking. Back in the bad old days, which wasn't so long ago for Mac users, the OS relied on each program to give up voluntarily the CPU after its time slice was up. This scheme was called "cooperative multitasking" because it

relied on the running programs to cooperate with each other and with the OS in order to share the CPU among themselves in a fair and equitable manner. Sure, there was a designated time slice in which each program was supposed to execute, and but the rules weren't strictly enforced by the OS. In the end, we all know what happens when you rely on people and industries to regulate themselves--you wind up with a small number of ill-behaved parties who don't play by the rules and who make things miserable for everyone else. In cooperative multitasking systems, some programs would monopolize the CPU and not let it go, with the result that the whole system would grind to a halt.

Preemptive multi-tasking, in contrast, strictly enforces the rules and kicks each program off the CPU once its time slice is up. Coupled with preemptive multi-tasking is **memory protection**, which means that the OS also makes sure that each program uses the memory space allocated to it and it alone. In a modern, preemptively multi-tasked and protected memory OS each program is walled off from the others so that it believes it's the only program on the system.

# Each program has a mind of its own

The OS and system hardware not only cooperate to fool the user about the true mechanics of multi-tasking, but they cooperate to fool each running program as well. While the user thinks that all of the currently running programs are being executed simultaneously, each of those programs thinks that it has a monopoly on the CPU and memory. As far as a running program is concerned, it's the only program loaded in RAM and the only program executing on the CPU. The program believes that it has complete use of the machine's entire memory address space and that the CPU is executing it continuously and without interruption. Of course, none of this is true. The program actually shares RAM with all of the other currently running programs, and it has to wait its turn for a slice of CPU time in order to execute, just like all of the other programs on the system.




**Single-threaded CPU**

In the above diagram, the different colored boxes in RAM represent instructions for four different running programs. As you can see, only the instructions for the red program are actually being executed right now, while the rest patiently wait their turn in memory until the CPU can briefly turn its attention to them.

Also, be sure and notice those empty white boxes in the pipelines of each of the execution core's functional units. Those empty pipeline stages, or **pipeline bubbles**, represent missed opportunities for useful work; they're execution slots where, for whatever reason, the CPU couldn't schedule any useful code to run, so they propagate down the pipeline empty.

Related to the empty white boxes are the blank spots in above CPU's front end. This CPU can issue up to four instructions per clock cycle to the execution core, but as you can see it never actually reaches this four-instruction limit. On most cycles it issues two instructions, and on one cycle it issues three.

## A few terms: process, context, and thread

Before continuing our discussion of multiprocessing, let's take a moment to unpack the term "program" a bit more. In most modern operating systems, what users normally call a program would be more technically termed a **process**. Associated with each process is a **context**, "context" being just a catch-all term that encompasses all the information that completely describes the process's current state of execution (e.g. the contents of the CPU registers, the program counter, the flags, etc.).

Processes are made up of **threads**, and each process consists of at least one thread: the main thread of execution. Processes can be made up of multiple threads, and each of these threads can have its own local context in addition to the process's context, which is shared by all the threads in a process. In reality, a thread is just a specific type of stripped-down process, a "lightweight process," and because of this throughout the rest of this article I'll use the terms "process" and "thread" pretty much interchangeably.

Even though threads are bundled together into processes, they still have a certain amount of independence. This independence, when combined with their lightweight nature, gives them both speed and flexibility. In an SMP system like the ones we'll discuss in a moment, not only can different processes run on different processors, but different threads from the same process can run on different processors. This is why applications that make use of multiple threads see performance gains on SMP systems that single-threaded applications don't.

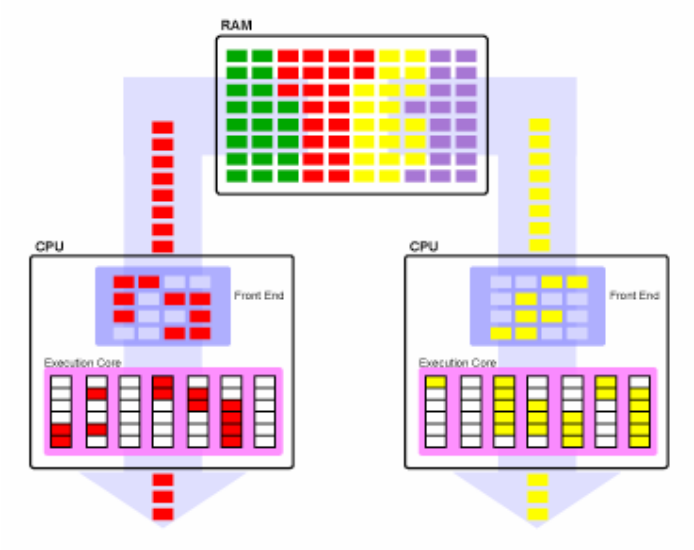
## Fooling the processes: context switches

It takes a decent amount of work to fool a process into thinking that it's the only game going. First and foremost, you have to ensure that when the currently executing process's time slice is up, its context is saved to memory so that when the process's time slice comes around again it can be restored to the exact same state that it was in when its execution was halted and it was flushed from the CPU to make room for the next process. When the process begins executing again and its context has been restored exactly as it was when it left off last, it has no idea that it ever left the CPU.

This business of saving the currently executing process's context, flushing the CPU, and loading the next process's context, is called a **context switch**. A context switch for a full-fledged, multithreaded process will obviously take a lot longer than a context switch for an individual thread within a process. So depending on the amount of hardware support for context switching and the type of context switch (i.e. a process switch or a thread switch), a context switch can take a decent amount of time, thereby wasting a number of CPU cycles. Cutting back on context switches improves execution efficiency and reduces waste, as does the extensive use of multithreading since thread switches are usually faster than full-sized process switches.

# SMP to the rescue?

One way to not only cut down on the number of context switches but also to provide more CPU execution time to each process is to build a system that can actually execute more than one process at the same time. The conventional way of doing this on the PC is to add a second CPU. In an SMP system, the OS can schedule two processes for execution at the exact same time, with each process executing on a different CPU. Of course, no process is allowed to monopolize either CPU (in most desktop operating systems) so what winds up happening is that each running process still has to wait its turn for a time slice. But since there are now two CPUs serving up time slices the process doesn't have to wait nearly as long for its chance to execute. The end result is that there is more total execution time available to the system so that within a given time interval each running process spends more time actually executing and less time waiting around in memory for a time slice to open up.




**Single-threaded SMP**

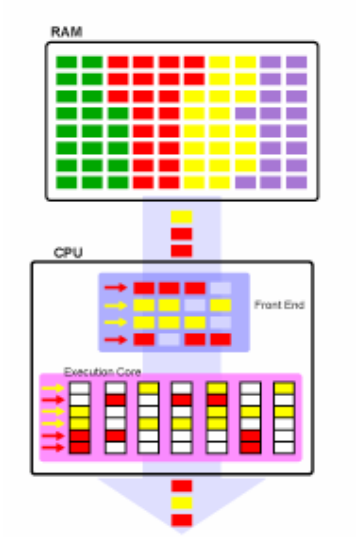
In the above diagram, the red program and the yellow process both happen to be executing simultaneously, one on each processor. Once their respective time slices are up, their contexts will be saved, their code and data will be flushed from the CPU, and two new processes will be prepared for execution.

One other thing that you might notice about the preceding diagram is that not only is the number of processes that can simultaneously execute doubled, but the number of empty execution slots (the white boxes) is doubled as well. So in an SMP system, there's twice as much execution time available to the running programs, but since SMP doesn't do anything to make those individual programs more efficient in the way that they use their time slice there's about twice as much wasted execution time, as well.

So while SMP can improve performance by throwing transistors at the problem of execution time, the overall lack of increase in the execution efficiency of the whole system means that SMP can be quite wasteful.

# Superthreading with a multithreaded processor

One of the ways that ultra-high-performance computers eliminate the waste associated with the kind of single-threaded SMP described above is to use a technique called **time-slice multithreading**, or **superthreading**. A processor that uses this technique is called a **multithreaded processor**, and such processors are capable of executing more than one thread at a time. If you've followed the discussion so far, then this diagram should give you a quick and easy idea of how superthreading works:




**Superthreaded CPU**

You'll notice that there are fewer wasted execution slots because the processor is executing instructions from both threads simultaneously. I've added in those small arrows on the left to show you that the processor is limited in how it can mix the instructions from the two threads. In a multithreaded CPU, each processor pipeline stage can contain instructions for one and only one thread, so that the instructions from each thread move in lockstep through the CPU.

To visualize how this works, take a look at the front end of the CPU in the preceding diagram. In this diagram, the front end can issue four instructions per clock to any four of the seven functional unit pipelines that make up the execution core. However, *all four instructions must come from the same thread*. In effect, then, each executing thread is still confined to a single "time slice," but that time slice is now one CPU clock cycle. So instead of system memory containing multiple running threads that the OS swaps in and out of the CPU each time slice, the CPU's front end now contains multiple executing threads and its issuing logic switches back and forth between them on each clock cycle as it sends instructions into the execution core.

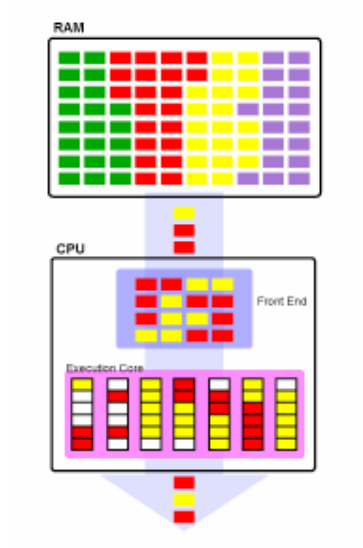
Multithreaded processors can help alleviate some of the latency problems brought on by DRAM memory's slowness relative to the CPU. For instance, consider the case of a multithreaded processor executing two threads, red and yellow. If the red thread requests data from main memory and this data isn't present in the cache, then this thread could stall for many CPU cycles while waiting for the data to arrive. In the meantime, however, the processor could execute the yellow thread while the red one is

stalled, thereby keeping the pipeline full and getting useful work out of what would otherwise be dead cycles.

While superthreading can help immensely in hiding memory access latencies, it does not, however, address the waste associated with poor instruction-level parallelism within individual threads. If the scheduler can find only two instructions in the red thread to issue in parallel to the execution unit on a given cycle, then the other two issue slots will simply go unused.

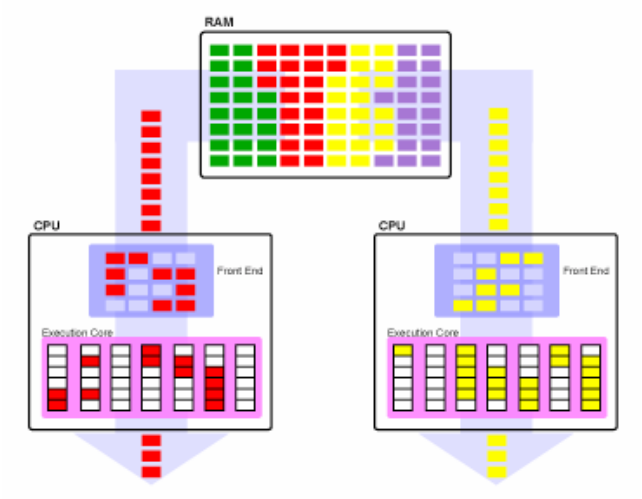
# Hyper-threading: the next step

**Simultaneous multithreading** (SMT), a.k.a. **hyper-threading**, takes superthreading to the next level. Hyper-threading is simply superthreading without the restriction that all the instructions issued by the front end on each clock be from the same thread. The following diagram will illustrate the point:




**Hyper-threaded CPU**

Now, to really get a feel for what's happening here, let's go back and look at the single-threaded SMP diagram.



**Single-threaded SMP**

If you look closely, you can see what I've done in the hyper-threading diagram is to take the execution patterns for both the red and the yellow threads in the SMP diagram and combine them so that they fit together on the single hyper-threaded processor like pieces from a puzzle. I rigged the two threads' execution patterns so that they complemented each other perfectly (real life isn't so neat) in order to make this point: the hyper-threaded processor, in effect, acts like two CPUs in one.

From an OS and user perspective, a simultaneously multithreaded processor is split into two or more **logical processors**, and threads can be scheduled to execute on any of the logical processors just as they would on either processor of an SMP system. We'll talk more about logical processors in a moment, though, when we discuss hyper-threading's implementation issues.

Hyper-threading's strength is that it allows the scheduling logic maximum flexibility to fill execution slots, thereby making more efficient use of available execution resources by keeping the execution core busier. If you compare the SMP diagram with the hyper-threading diagram, you can see that the same amount of work gets done in both systems, but the hyper-threaded system uses a fraction of the resources and has a fraction of the waste of the SMP system; note the scarcity of empty execution slots in the hyper-threaded machine versus the SMP machine.

To get a better idea of how hyper-threading actually looks in practice, consider the following example: Let's say that the OOE logic in our diagram above has extracted all of the instruction-level parallelism (ILP) it can from the red thread, with the result that it will be able to issue two instructions in parallel from that thread in an upcoming cycle. Note that this is an exceedingly common scenario, since research has shown the average ILP that can be extracted from most code to be about 2.5 instructions per cycle. (Incidentally, this is why the Pentium 4, like many other processors, is equipped to issue at most 3 instructions per cycle to the execution core.) Since the OOE logic in our example processor knows that it can theoretically issue up to four instructions per cycle to the execution core, it would like to find two more instructions to fill those two empty slots so that none of the issue bandwidth is wasted. In either a single-threaded or multithreaded processor design, the two leftover slots would just have to go unused for the reasons outlined above. But in the hyper-threaded design, those two slots can be filled with instructions from another thread. Hyper-threading, then, removes the issue bottleneck that has plagued previous processor designs.

# Implementing hyper-threading

Although hyper-threading might seem like a pretty large departure from the kind of conventional, process-switching multithreading done on a single-threaded CPU, it actually doesn't add too much complexity to the hardware. Intel reports that adding hyper-threading to their Xeon processor added only %5 to its die area. To understand just how hyper-threading affects the Pentium 4 Xeon's microarchitecture and performance, let's briefly look in a bit more detail at the Xeon's SMT implementation.

Intel's Xeon is capable of executing at most two threads in parallel on two logical processors. In order to present two logical processors to both the OS and the user, the Xeon must be able to maintain information for two distinct and independent thread contexts. This is done by dividing up the processor's microarchitectural resources into three types: replicated, partitioned, and shared. Let's take a look at which resources fall into which categories:

| | |
|---|---|
| **Replicated** | • Register renaming logic<br>• Instruction Pointer<br>• ITLB<br>• Return stack predictor<br>• Various other architectural registers |
| **Partitioned** | • Re-order buffers (ROBs)<br>• Load/Store buffers<br>• Various queues, like the scheduling queues, uop queue, etc. |
| **Shared** | • Caches: trace cache, L1, L2, L3<br>• Microarchitectural registers<br>• Execution Units |

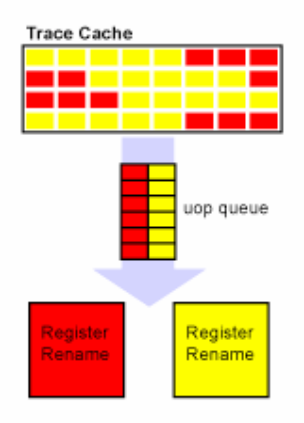
## Replicated resources

There are some resources that you just can't get around replicating if you want to maintain two fully independent contexts on each logical processor. The most obvious of these is the instruction pointer (IP), which is the pointer that helps the processor keep track of its place in the instruction stream by pointing to the next instruction to be fetched. In order to run more than one process on the CPU, you need as many IPs as there are instruction streams keep track of. Or, equivalently, you could say that you need one IP for each logical processor. In the Xeon's case, the maximum number of instruction streams (or logical processors) that it will ever have to worry about is 2, so it has 2 IPs.

Similarly, the Xeon has two register allocation tables (RATs), each of which handles the mapping of one logical processor's eight architectural integer registers and eight architectural floating-point registers onto a shared pool of 128 GPRs (general purpose registers) and 128 FPRs (floating-point registers). So the RAT is a replicated resource that manages a shared resource (the microarchitectural register file).

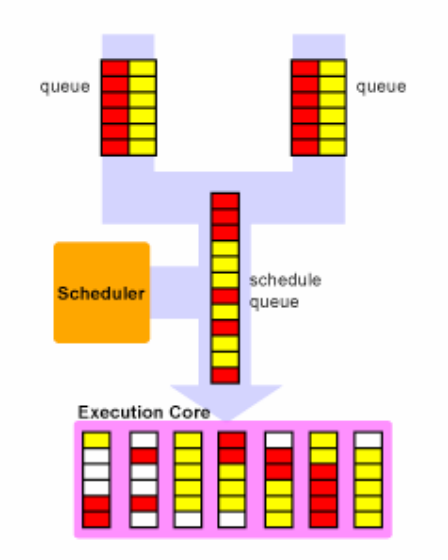
## Partitioned resources

The Xeon's partitioned resources are mostly to be found in the form of queues that decouple the major stages of the pipeline from one another. These queues are of a type that I would call "statically partitioned." By this, I mean that each queue is split in half, with half of its entries designated for the sole use of one logical processor and the other half designated for the sole use of the other. These statically partitioned queues look as follows:




**Statically Partitioned Queue**

The Xeon's fscheduling queue is partitioned in a way that I would call "dynamically partitioned." In a scheduling queue with 12 entries, instead of assigning entries 0 through 5 to logical processor 0 and entries 6 through 11 to logical processor 1, the queue allows any logical processor to use any entry but it places a limit on the number of entries that any one logical processor can use. So in the case of a 12-entry scheduling queue, each logical processor can use no more than six of the entries.




**Dynamically Partitioned Queue**

Be aware that the above diagram shows only one of the Xeon's three scheduling queues.

From the point of view of each logical processor and thread, this kind of dynamic partitioning has the same effect as fixed partitioning: it confines each LP to half of queue. However, from the point of view of the physical processor, there's a crucial difference between the two types of partitioning. See, the scheduling logic, like the register file and the execution units, is a shared resource, a part of the Xeon's microarchitecture that is SMT-unaware. The scheduler has no idea that it's scheduling code from multiple threads. It simply looks at each instruction in the scheduling queue on a case-by-case basis, evaluates the instruction's dependencies, compares the instruction's needs to the physical processor's currently available execution resources,

and then schedules the instruction for execution. To return to the example from our hyper-threading diagram, the scheduler may issue one red instruction and two yellow to the execution core on one cycle, and then three red and one yellow on the next cycle. So while the scheduling queue is itself aware of the differences between instructions from one thread and the other, the scheduler in pulling instructions from the queue sees the entire queue as holding a single instruction stream.

The Xeon's scheduling queues are dynamically partitioned in order to keep one logical processor from monopolizing them. If each scheduling queue didn't enforce a limit on the number of entries that each logical processor can use, then instructions from one logical processor might fill up the queue to the point where instructions from the other logical processor would go unscheduled and unexecuted.

One final bit of information that should be included in a discussion of partitioned resources is the fact that when the Xeon is executing only one thread, all of its partitioned resources can be combined so that the single thread can use them for maximum performance. When the Xeon is operating in single-threaded mode, the dynamically partitioned queues stop enforcing any limits on the number of entries that can belong to one thread, and the statically partitioned queues stop enforcing their boundaries as well.

Shared resources are at the heart of hyper-threading; they're what makes the technique worthwhile. The more resources that can be shared between logical processors, the more efficient hyper-threading can be at squeezing the maximum amount of computing power out of the minimum amount of die space. One primary class of shared resources consists of the execution units: the integer units, floating-point units, and load-store unit. These units are not SMT-aware, meaning that when they execute instructions they don't know the difference between one thread and the next. An instruction is just an instruction to the execution units, regardless of which thread/logical processor it belongs to.

The same can be said for the register file, another crucial shared resource. The Xeon's 128 microarchitectural general purpose registers (GPRs) and 128 microarchitectural floating-point registers (FPRs) have no idea that the data they're holding belongs to more than one thread--it's all just data to them, and they, like the execution units, remain unchanged from previous iterations of the Xeon core.

Hyper-threading's greatest strength--shared resources--also turns out to be its greatest weakness, as well. Problems arise when one thread monopolizes a crucial resource, like the floating-point unit, and in doing so starves the other thread and causes it to stall. The problem here is the exact same problem that we discussed with cooperative multi-tasking: one resource hog can ruin things for everyone else. Like a cooperative multitasking OS, the Xeon for the most part depends on each thread to play nicely and to refrain from monopolizing any of its shared resources.

For example, if two floating-point intensive threads are trying to execute a long series of complex, multi-cycle floating-point instructions on the same physical processor, then depending on the activity of the scheduler and the composition of the scheduling queue one of the threads could potentially tie up the floating-point unit while the other thread stalls until one of its instructions can make it out of the scheduling queue. On a non-SMT processor, each thread would get only its fair share of execution time because at the end of its time-slice it would be swapped off the CPU and the other thread would be swapped onto it. Similarly, with a time-slice multithreaded CPU no one thread can tie up an execution unit for multiple consecutive pipeline stages. The SMT processor, on the other hand, would see a significant decline in performance as each thread contends for valuable but limited execution resources. In such cases, an SMP solution would be

far superior, and in the worst of such cases a non-SMT solution would even give better performance.

The shared resource for which these kinds of contention problems can have the most serious impact on performance is the caching subsystem.

## Caching and SMT

For a simultaneously multithreaded processor, the cache coherency problems associated with SMP all but disappear. Both logical processors on an SMT system share the same caches as well as the data in those caches. So if a thread from logical processor 0 wants to read some data that's cached by logical processor 1, it can grab that data directly from the cache without having to snoop another cache located some distance away in order to ensure that it has the most current copy.

However, since both logical processors share the same cache, the prospect of cache conflicts increase. This potential increase in cache conflicts has the potential to degrade performance seriously.

## Cache conflicts

You might think since the Xeon's two logical processors share a single cache, this means that the cache size is effectively halved for each logical processor. If you thought this, though, you'd be wrong: it's both much better and much worse. Let me explain.

Each of the Xeon's caches--the trace cache, L1, L2, and L3--is SMT-unaware, and each treats all loads and stores the same regardless of which logical processor issued the request. So none of the caches know the difference between one logical processor and another, or between code from one thread or another. This means that one executing thread can monopolize virtually the entire cache if it wants to, and the cache, unlike the processor's scheduling queue, has no way of forcing that thread to cooperate intelligently with the other executing thread. The processor itself will continue trying to run both threads, though, issuing fetches from each one. This means that, in a worst-case scenario where the two running threads have two completely different memory reference patterns (i.e. they're accessing two completely different areas of memory and sharing no data at all) the cache will begin thrashing as data for each thread is alternately swapped in and out and bus and cache bandwidth are maxed out.

It's my suspicion that this kind of cache contention is behind the recent round of benchmarks which show that for some applications SMT performs significantly worse than either SMP or non-SMT implementations within the same processor family. For instance, these benchmarks show the SMT Xeon at a significant disadvantage in the memory-intensive portion of the reviewer's benchmarking suite, which according to our discussion above is to be expected if the benchmarks weren't written explicitly with SMT in mind.

In sum, resource contention is definitely one of the major pitfalls of SMT, and it's the reason why only certain types of applications and certain mixes of applications truly benefit from the technique. With the wrong mix of code, hyper-threading decreases performance, just like it can increase performance with the right mix of code.

# Conclusions

Now that you understand the basic theory behind hyper-threading, in a future article on Prescott we'll be able to delve deeper into the specific modifications that Intel made to the Pentium 4's architecture in order to accommodate this new technique. In the meantime, I'll be watching the launch and the subsequent round of benchmarking very closely to see just how much real-world performance hyper-threading is able to bring to the PC. As with SMP, this will ultimately depend on the applications themselves, since multithreaded apps will benefit more from hyper-threading than single-threaded ones. Of course, unlike with SMP there will be an added twist in that real-world performance won't just depend on the applications but on the specific mix of applications being used. This makes it especially hard to predict performance from just looking at the microarchitecture.

The fact that Intel until now has made use of hyper-threading only in its SMP Xeon line is telling. With hyper-threading's pitfalls, it's perhaps better seen as a compliment to SMP than as a replacement for it. An SMT-aware OS running on an SMP system knows how to schedule processes at least semi-intelligently between both processors so that resource contention is minimized. In such a system SMT functions to alleviate some of the waste of a single-threaded SMP solution by improving the overall execution efficiency of both processors. In the end, I expect SMT to shine mostly in SMP configurations, while those who use it in a single-CPU system will see very mixed, very application-specific results.

# Bibliography


- Susan Eggers, Hank Levy, Steve Gribble. Simultaneous Multithreading Project. University of Washington
- Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen. "Simultaneous Multithreading: A Platform for Next-generation Processors." *IEEE Micro*, September/October 1997, pages 12-18.
- Jack Lo, Susan Eggers, Joel Emer, Henry Levy, Rebecca Stamm, and Dean Tullsen. "Converting Thread-Level Parallelism Into Instruction-Level Parallelism via Simultaneous Multithreading." *ACM Transactions on Computer Systems*, August 1997, pages 322-354.
- "Hyper-Threading Technology." Intel.
- Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, Michael Upton. "Hyper-Threading Technology Architecture and Microarchitecture." Intel.