Multithreading Comes of Age

Multithreaded Processors Can Boost Throughput on Servers, Media Processors

by Peter Song

In their relentless quest for more performance, many microprocessor designers are wondering, “What’s after superscalar?” With out-of-order and speculative execution seeming to have reached the point of diminishing returns, many are looking for more ways to improve performance without breaking binary compatibility. As faster processors exacerbate memory-latency problems and new IC processes allow more than 10 million transistors on a chip, these designers are giving multithreading another look. Multithreading can significantly improve throughput without huge investments in die size or design complexity. It can simplify multimedia programming by hiding latency problems from software. And unlike VLIW, multithreading maintains full binary compatibility.

Like most advances in processor design, multithreading is an idea that dates back to the 1950s. In the late 1970s, the now defunct Denelcor’s HEP (heterogeneous element processor) used processing elements that could support 16 instruction streams, or threads. By rotating among the 16 threads every cycle and by issuing and maintaining at most one instruction from each thread in the eight-stage pipeline, the processing elements were able to execute instructions correctly without any hardware interlocks.

The HEP used multithreading to simplify pipeline design and improve system throughput at the expense of a single thread’s execution speed. Recent studies, however, claim that multithreaded processors can more than double system throughput with little impact on single-thread performance. They also claim a dramatic gain in system throughput can come from multithreaded processors that are only incrementally larger and more complex than today’s high-performance superscalar processors.

The commercial success of multithreaded processors clearly depends on the availability of operating systems that support multithreading. Fortunately, Windows NT and most Unix operating systems are designed to run on multiprocessor systems and support multithreaded applications. Running these operating systems on multithreaded processors should require little or no software effort.

Multithreaded applications also need to be developed to enable successful deployment of multithreaded processors. The first mass-market multithreaded software is likely to emerge from multimedia applications. The technical needs for processing multiple and continuous media streams, coupled with the market needs for delivering it at consumer-price points, make multithreading a suitable design style for multimedia processors.

Multithreading Reduces Idle Cycles

Multithreading can squeeze more performance from a processor by utilizing otherwise idle cycles. Due to a thread’s limited degree of parallelism and frequent dependencies on long-latency operations, such as cache misses, a processor can incur many idle cycles. Instead of wasting these idle cycles, a multithreaded processor can use them to execute instructions from other threads.

Figure 1 shows hypothetical threads A and B running on conventional and multithreaded processors. By switching to thread B_1 while A_1 is stalled, the multithreaded processor is able to better use the idle cycles between A_1 and A_2. The benefit is increased processor utilization and consequentially higher throughput. Overall, the two threads complete in substantially less time on the multithreaded processor.

Figure 1 also illustrates several key performance issues of multithreading (see sidebar, next page). First is thread-switch overhead, which should be kept to a minimum, since it accomplishes useless work. The amount of this overhead determines which long-latency operations can profitably trigger thread switching. Second is the number of threads the processor can support. Note that the idle cycles between B_1 and A_2 could be better used by a third thread, since A_2 is still stalled. Finally, multithreading can lengthen execution time of individual threads. Note that B_2 resumed many cycles later than it could have because thread A_2 was running.

A multithreaded design can be classified into one of three categories: coarse, fine, and simultaneous. A coarse multithreaded design supports only one active thread—by limiting instructions from only one thread in its execution pipe. A fine multithreaded design supports multiple active threads, but issues instructions from only one thread in a cycle. The HEP falls into this category. A simultaneous multithreaded design issues and executes instructions from multiple threads each cycle. As we will see, all three can be derived from existing designs.

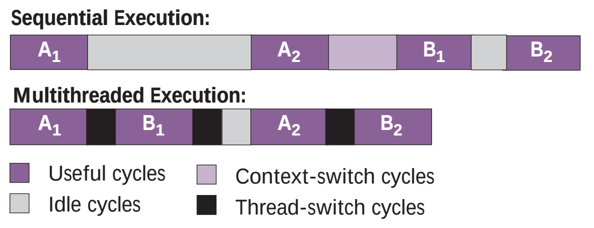


Figure 1. When threads A and B are executed sequentially on a conventional microprocessor, the idle cycles incurred while running threads A and B are wasted. A multithreaded processor can detect these idle cycles and switch to a thread that may have instructions ready to execute.

Utilization Explained

A simple analytical model can explain the increase in utilization through multithreading. Assume that a processor spends its cycles doing one of three things: productive work (P); idling due to a long-latency operation, when the cycles can be made available to other threads (L); idling, when the cycles are wasted (W). With only one thread to execute, the average processor utilization is

$$U = \frac{P}{(P + W + L)}$$

When multiple threads are present, the time spent waiting for a long-latency operation (L) can be used to run the other threads. Assuming that additional threads have no effect on W or L, the average processor utilization is

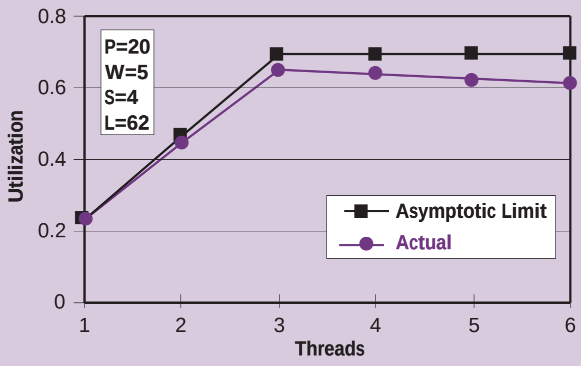
$$U_{linear} = \frac{T \times P}{(P + W + L)}$$

provided that $L \geq T \times (P + W + S)$ where T represents the number of threads and S is the thread-switch overhead. Notice that thread-switch overhead (S) is subsumed by L in the utilization equation. The utilization is in the linear region when adding more threads increases it linearly.

As more threads are added, the sum of P, W, and S for running all threads becomes longer than L. This is known as the saturation region, and adding more threads no longer improves the utilization. The saturation region is expressed as

$$U_{sat} = \frac{P}{(P + W + S)}$$

The upper limit of the utilization is set by the idle cycles (W) and the thread-switch overhead (S). In real systems, additional threads do affect W and L (causing a higher cache-miss rate, for instance), and the utilization decreases with more threads in the saturation region. The figure below shows the linear region up to three threads and the saturation region beyond the three threads.



Multiple Register Sets Speed Thread Switch

Conventional microprocessors provide only one set of architectural, or software-visible, registers. These are registers that collectively represent a thread's state when its execution is interrupted. Architectural registers typically include the register file (most RISC processors have 32 general-purpose registers in it) and a few special-purpose registers, such as the program-counter, control registers, and status registers.

These single-threaded processors are used to support multiple threads today. But before another thread can begin, the current thread's state must be saved in memory so it can properly resume later. Depending on the number of registers involved and cache misses incurred, a thread-switch operation involving saving and restoring registers can take hundreds of cycles. Consequentially, it is unprofitable to support thread switching on operations that take less than a hundred or so cycles.

One way to reduce thread-switch overhead is to provide each thread with its own set of architectural registers. This eliminates the need to save (and restore) a thread's state to run another thread, making a thread-switch on a long-latency operation profitable. It does not, however, eliminate the thread switching needed to execute operating-system service calls or to run more threads than hardware supports.

Replicating a register file might not proportionally increase its area. Because each bit in a register file needs a data line and a word-select line for each port, beyond a certain number of ports its size is set by the width and height needed to route the metal lines, not by the area needed to build the SRAM cell. Using only 20% more area than that of a single-window register file, Sun's UltraSparc designers were able to pack eight SRAM cells, one for each of eight SPARC register windows, under the metal grid of a bit cell (see MPR 10/3/94, p. 7). By having the threads share the ports, a coarse multithreaded processor supporting up to eight threads could use a similar technique.

Coarse Multithreading Switches on Cache Miss

If the events that cause threads to switch are limited to long-latency operations, such as a miss in the level-two (L2) cache, the complexity of adding coarse multithreading to an existing pipeline can be kept to a minimum. An easy implementation is to wait for the instructions prior to the miss to complete, then flush the pipeline and select the next active thread. With a nonblocking cache design, the miss can be serviced independently of the thread switch. When the thread is resumed later, the second try is likely to result in a cache hit. In effect, the miss spawns a prefetch.

A cache miss is a good candidate for causing a thread switch because the instruction that incurs it can easily be restarted. Other long-latency operations, such as divide, may not be as good. It requires a sophisticated pipeline design to have the divide operation continue while the rest of the pipeline executes instructions from a new thread. Conditions such as another divide instruction or an exception from the new thread complicate the pipeline-control and exception-handling logic design.

Supporting only one active thread isolates multithread-related design changes to a few places, keeping them manageable. A context ID (CID) register could be used to select the active set for every register access. The L2 cache-miss signal could be used by the instruction-completion-and-exception logic to interrupt the thread and flush the pipeline. Most high-performance microprocessors already use nonblocking cache designs to support multiple misses and prefetches. Thread-selection logic could update the CID and cause instructions to be fetched from the newly updated PC.

Multiple Threads Affect Large Caches Little

Some critics discount multithreading because, even in efficient multithreaded designs that use only a thread's idle cycles, the additional threads significantly degrade single-thread performance due to their detrimental effect on caches. Eickemeyer[1] recently reported that, although additional threads do increase the miss rate of small data caches (32K or smaller), one or two additional threads have negligible effects on reasonably sized caches (512K or larger).

The reported simulation results are based on TPC-C traces running on a coarse multithreaded processor that switched threads only on an L2 cache miss. For the results presented in Figure 2, the model assumed 8 cycles for an L1 instruction or data miss, 60 cycles for an L2 miss, and 4 cycles for a thread switch. The model stalled the processor on an L1 miss.

For this OLTP (on-line transaction processing) application, additional threads had very little effect on the L1 instruction-cache miss rate. The second thread increased the miss rate of a 32K direct-mapped cache by only 0.9%. In some cases, the miss rates actually decreased, due to code sharing among the threads. As set associativity increased, the increase in miss rates declined, as the conflict misses between threads were reduced.

Additional threads had a more significant impact on the L1 data-cache miss rate. The second thread increased the miss rate by 10% on a 32K direct-mapped cache. With five more threads, the miss rate increased by 31% for the same cache. As expected, additional threads affected larger caches less. Furthermore, the miss rate per thread was lower for a 32K cache with two threads than for a 16K cache with only one thread, indicating that a larger shared cache was better utilized than correspondingly smaller split caches.

For the larger L2 cache, the effect of multithreading was even smaller. Adding the second thread to a direct-mapped 512K cache increased the miss rate by only 5%. For two- and four-way set-associative caches, the second thread caused a negligible change in miss rates. As L1 and L2 cache sizes continue to grow, effects of multithreading will get smaller.

Multithreading's small effect on L2 cache-miss rates implies that it should provide a significant performance gain by hiding miss latency. With 32K L1 and 1M L2 caches, adding the second thread increases throughput by 24%.

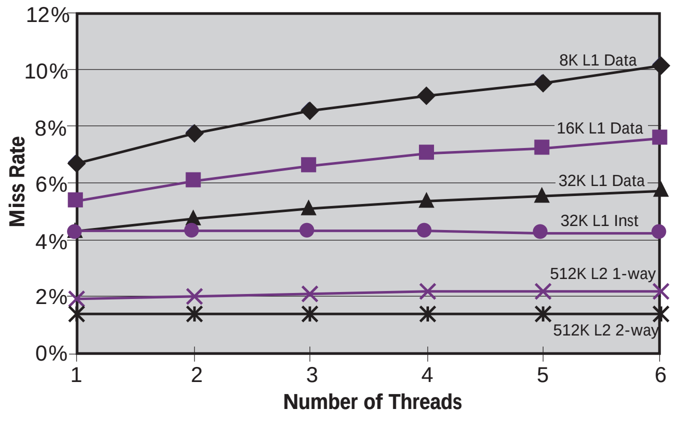


Figure 2. Multithreading effects on cache-miss rates for TPC-C benchmark are shown. In general, multithreading affects larger and set-associative caches less. (Source: *23rd Annual International Symposium on Computer Architecture*, pp. 203–212, May 1996)

With 8K L1 and 256K L2 caches, the increase in throughput, at 41%, is even greater.

As L2 miss latency is increased from 50 to 70 cycles, throughput decreases, but the relative gain in throughput due to multithreading increases. Since multithreading has little effect on a reasonably sized L2 cache and tolerates longer latency well, it will scale well as CPU speeds outstrip memory speeds, increasing apparent latency. The design must provide enough bandwidth from the L2 to L1 caches to account for the increased L1 cache-miss traffic, however.

This data is encouraging: a throughput gain of 30% to 50% on OLTP applications is possible with only two to three threads. Transaction processing, however, may be an ideal environment for multithreading, because a large number of threads routinely execute the same programs. Furthermore, a gain in throughput is more important than a slight loss in single-thread's performance in high-workload server environments.

Fine Multithreading Improves CPU Utilization

As CPUs reach higher frequencies, not only do memory accesses take more cycles, but the longer pipeline causes other common operations to take more cycles. For instance, the 500-MHz Alpha 21264 needs 12–14 CPU cycles for an off-chip L2 cache access, 8–16 cycles for an integer multiply, and an average of 11 cycles to recover from a mispredicted branch (see MPR 10/28/96, p. 11). These modest-latency operations will cause short but frequent pipeline stalls, keeping a single-threaded or even coarse multithreaded processor's efficiency low. Fine multithreading attempts to boost efficiency by using even these short and frequent idle cycles to execute instructions from other threads.

The simplest scheme is to allocate one issue cycle to each thread in rotation, as used by HEP and by MicroUnity's ill-fated MediaProcessor (see MPR 10/23/95, p. 11). The Media-Processor rotates among five threads to issue one instruction into its superlong 1-GHz pipeline. This five-way

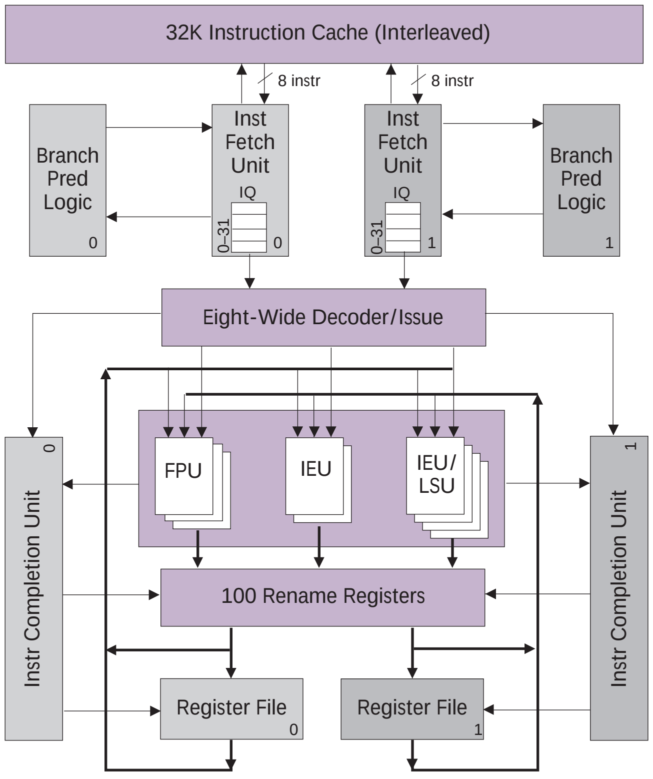


Figure 3. A hypothetical two-way simultaneous multithreaded processor has two register files, two instruction-completion units, and two instruction-fetch mechanisms but shares the other resources. Instructions from both threads can be issued together in the same cycle.

multithreading cuts the effective latency (as seen by an individual thread) of a five-stage add to one, a ten-stage load to two, and a fifteen-stage multiply-add to three. Its pipeline control for dependency and forwarding is no more complicated than in a single-threaded design, since each stage knows which thread it is processing using a simple modulo-five counter. The design does require up to five times more latches and perhaps a scheme to selectively clock latches.

A somewhat more intelligent issue scheme is used in Tera Computer's MTA (Multithreaded Architecture). It can support up to 128 active threads at once. On every cycle, the thread-selection logic issues one instruction from a ready thread. A thread is ready when its next instruction has no dependency on instructions that are still in execution. For register-to-register instructions that have fixed latencies, a cycle-counting scheme makes the thread unavailable. The benefit is that, since the stalled threads are not considered for issue, the threads that are making progress are given more issue bandwidth. With enough active threads to mask average pipeline latency, the pipeline can be fully utilized at the expense of stalling some ready threads for longer than necessary.

Interlocks Maintain Single-Thread's Performance

Laudon[2] proposed adding hardware interlocks to HEP-style fine multithreading to minimize the loss in single thread's performance. Hardware interlocks allow instructions from one thread to be issued every cycle, thereby achieving the same performance as a single-threaded processor. When multiple threads are present, instruction issue rotates among the ready threads, reducing the effective latency seen by each.

To hide latency longer than can be masked by thread-rotating, a thread can use a BACKOFF instruction to specify the number of cycles in which it will not be ready for issue. By preventing the next instruction from prematurely entering the pipeline, the BACKOFF instruction can reduce pipeline blockage. The instruction is most useful for stalling on a fixed-latency operation, such as a divide. On a variable-latency operation, such as a load that may hit or miss the cache, it poses the classical scheduling problem. An optimistic scheduling could block the pipeline for other threads, and a pessimistic one could unnecessarily stall the thread. For checking a thread's readiness after a variable-latency operation, register scoreboarding is more effective than the cycle-counting scheme or the BACKOFF instruction.

Tullsen[3] described a simultaneous multithreading (SMT) scheme that claims to achieve 5.4 instructions per cycle (IPC) on an eight-issue design. Figure 3 depicts their hypothetical SMT processor, without the 32K L1 data cache, 256K L2 cache, and 2M L3 cache. Their simulation studies show that, without SMT, even aggressive superscalar techniques cannot take full advantage of an eight-issue design. Although it is difficult to generalize from their simulation studies to real systems, SMT is likely to raise IPC by issuing more instructions from multiple threads each cycle. In effect, it takes advantage of the higher aggregate instruction-level parallelism present in multiple threads.

Tullsen also reported two interesting findings. First, simultaneous multithreading raises IPC significantly with a minimal reduction in a single thread's performance. The SMT processor increased IPC by more than 30% with two threads, from 2.1 to 2.8, and by 84% with six threads, to 3.9. SMT support incurs a 2% drop in single-thread IPC, compared with a superscalar processor without it. The drop comes from adding an extra pipeline stage for register reads and another for writes to compensate for the larger register file. In the study, each thread ran 300 million instructions for each of eight benchmarks (seven from SPEC92 and *Tex*), arranged so no two threads ran the same benchmark at the same time.

Second, fetching from the thread with the fewest instructions in the instruction queues each cycle gives the most gain in throughput—23% with eight threads. It maximizes the throughput of fixed-size queues with the instructions that spend the least time in them. Two other policies, fetching from the thread with the fewest branches in the instruction queues and fetching from the thread with the fewest outstanding data-cache misses, also perform better than the round-robin scheme.

Second Thread Adds Little Complexity

A wide-issue superscalar design can be modified to support simultaneous multithreading. A larger register file is needed to support multiple register sets, which will lengthen its access time, but this may not be the critical speed path. The register file needs the same number of ports and rename registers as in a single-threaded design with similar execution width.

Simultaneous multithreading also requires, for each thread, mechanisms to retire instructions and to selectively flush them when handling mispredicted branches and exceptions. A selective flush mechanism is easy to devise (e.g., internally tag each instruction and register with the thread ID), but it complicates efficient and fast-access queue design because it randomly leaves invalid entries.

As in other high-performance processor designs, the most performance-critical design issues are in the instruction-fetch logic. Most, if not all, of the fetch-acceleration mechanisms need to be duplicated to take advantage of the wide-issue core. These include features like branch-prediction logic and related tables, branch-target caches, and the return-address stack. The instruction cache needs to be nonblocking to handle multiple outstanding misses.

For just two threads, however, the added complexity of simultaneous multithreading can be manageable. A nonblocking instruction cache is easier to design than the nonblocking data caches that today's high-performance processors already support. Isolating thread-specific logic into separate modules can ease replication. In fact, the bulk of design and functional verification work can be done with a design that supports only one thread. The modules needed by the second thread can be added later, when the interactions between the threads need to be tested or when physical design and verification call for them.

Multiple Threads Raise Few Architectural Issues

One way to introduce multiple threads into an existing architecture is to make each thread appear as if it were a complete processor. Doing so makes the existing system and software solutions for symmetric multiprocessor (SMP) designs readily applicable to multithreaded systems. In effect, this is one way to build an SMP on a chip (see MPR 10/2/95, p. 16). Its advantage over building an SMP by replicating entire CPUs is that, with simultaneous multithreading, it can provide flexible partitioning of CPU and memory bandwidth, as needed by the threads. Traditional SMP uses permanently fixed partitioning and relies on software to balance the CPU bandwidth for each processor.

This symmetric multithreaded architecture duplicates both user- and supervisor-mode states, including trap vectors that specify the trap-handler addresses. This allows simultaneous exceptions from multiple threads to occur and simplifies the system programming model. The trap-handler code can be—and is likely to be—shared by all threads. Providing a trap-vector-base register for each thread and defining each trap vector as an offset from this base register could make code sharing easier.

Making all threads behave identically simplifies processor design and verification and makes the architecture scalable. Asynchronous interrupts and reset can be treated in the same manner. Instead of differentiating only one thread to service asynchronous interrupts, the thread-selection logic can interrupt the lowest-priority or an arbitrary thread.

Providing a separate reset vector for each thread maintains identical behavior for all threads and, at the same time, provides the differentiation an operating system may need. For instance, it is easier to have only one thread go through the system-boot sequence. Today's SMPs solve this problem by keeping each processor's boot code in its local memory.

To avoid using a busy-waiting loop for an idle thread, an instruction to SLEEP and a mechanism to WAKE a suspended thread are desirable. The Tera MTA provides the STREAM_CREATE instruction for a thread to activate another thread. This instruction passes the address of an instruction stream, trap-vector-base address, mode and mask bits, and up to three words of data. A thread executes the STREAM_QUIT instruction to return to the idle state.

Multithreading Cheaply Boosts Server Throughput

If multithreading is so good and inexpensive, why hasn't anyone built a commercial success out of it? For one thing, there hasn't been a big demand for inexpensive multiprocessor servers. And for the small market that wanted multiprocessors for throughput, the immense memory and resilient disk storage required were far more expensive than the CPUs used in these machines.

Due to the affordable Pentium/Windows NT combination and the explosive growth of the Internet, the demand for inexpensive servers is growing rapidly. Computer Intelligence estimates 227,900 units of Pentium/Windows NT servers were shipped in 1996, up sharply from 96,200 units a year earlier. The nonworkstation server market for Windows NT and Unix combined is projected to grow to 2.3 million units by 2001. Furthermore, due to the dramatic drop in both DRAM and disk-storage costs, the cost of CPUs has become a significant portion of these inexpensive servers. A multithreaded microprocessor, with its additional threads, can boost these servers' throughput at an incremental CPU cost.

Only now are designers being given semiconductor technology that gives the sense of "excess transistors." Their first urge may be to put these extra transistors in caches. Undoubtedly, larger caches improve performance, but the rate of improvement diminishes beyond a reasonable size; the forthcoming PA-8500, for example, will contain 1.5M of cache (see MPR 3/10/97, p. 4). Additional threads offer more performance gain than do enormous caches. With an increasingly larger fraction of chip area being devoted to cache, the die-area cost of additional threads is small and will get smaller over time.

The shorter CPU cycle time and the greater speed difference between logic and memory devices have made the latency problem worse. For the 1-GHz designs in progress, DRAM access will take upwards of a hundred cycles. Other common operations will take tens of cycles. The host of latency-hiding techniques already being used in high-end processors, some with enough hardware to execute up to 80 instructions out of order, may also have reached the point of diminishing returns. A more general latency-tolerating technique, such as multithreading, is necessary as designers look to deliver more performance.

PC Multimedia Can Provide Additional Threads

Multithreading is even better suited for multimedia PCs, where modem, video, audio, and even 3D graphics data streams may need to be processed simultaneously. The lack of locality and the huge size of multimedia data streams make caching less effective and latency a bigger problem. Yet these data streams need to be processed within fixed time limits to provide seamless and vivid multimedia experiences.

Media processors like Mpact, TriMedia, and the ill-fated MSP use on-chip data memory instead of cache, along with a host of prefetch, write-back, and DMA functions. Software pipelining—prefetching (the next block of a data stream) and storing (results of the previous block of the data stream) while processing the current block of a data stream—is their way of hiding latency. For the sake of simple hardware designs, these processors turn the latency problem into difficult and unnatural programming.

A multithreaded approach, in which a data stream is processed by one or more threads, simplifies programming. The difficult task of writing software-pipelined programs becomes easier when hardware can effectively hide latency behind instructions that process other data streams. In fact, latency ceases to be an application's problem when each thread is given enough CPU and memory bandwidth to process its data stream in the allotted time—provided that hardware supports multiple active threads or low-overhead thread switching. For a thread processing a modem or audio data stream, however, getting enough bandwidth doesn't necessarily guarantee that it will meet its real-time deadlines.

By no means do media processors have exclusive rights to multithreaded multimedia processing. They simply have more media-specific instructions (and integrated functions) and fewer general-purpose instructions. Most existing architectures, with their multimedia extensions, have instruction sets that are adequate for designing competitive multithreaded media processing.

A Multithreaded CPU to Appear by 2000

Multithreading is a latency-hiding technique that works well with existing and new architectures. It can improve a server's throughput performance at an incremental CPU cost. In OLTP environments that value throughput as well as response time, a significant gain in throughput is worth more than a slight loss in response time. Multithreading can also increase throughput as well as simplify writing multimedia application programs.

Multithreading offers a range of implementation choices. A coarse multithreaded processor can be a derivative of an existing high-end design with little added die area and design complexity. A fine or simultaneous multithreaded processor can take advantage of short but increasingly frequent idle cycles. It will have less impact on single-thread performance than does a coarse multithreaded design. As more transistors become available on a single die, the die area spent on supporting additional threads will get smaller.

Merced, the first implementation of IA-64, is expected to tape out early next year. Although we don't anticipate it to be a multithreaded design, we do expect future implementations of IA-64 to use this technique. As IA-64 is initially aimed at workstations and servers, a multithreaded IA-64 design could provide multiprocessor performance without additional CPUs.

AMD, Cyrix, and other x86 vendors may choose to counter Merced with multithreaded designs that offer increased system and multimedia performance without a new instruction set. As superscalar design techniques reach a point of diminishing returns, multithreading can provide yet another dimension to improving x86 performance. Due to the x86's stack-based floating-point instruction set, however, even a multithreaded x86 processor would fall short of Merced's floating-point performance.

The first multithreaded microprocessor is likely to be a media processor. The need to process multiple data streams, many with wide data-level parallelism but little locality, calls for a wide execution core that can hide latency. In addition, the absence of existing software and established development tools calls for the simple programming model that a multithreaded design can provide. A multithreaded vector processor has a future in multimedia processing and could appear by the end of this decade. M

References and Bibliography

[1] Eickemeyer, R.; R. Johnson; et al. "Evaluation of Multithreaded Uniprocessors for Commercial Application Environments." In *23rd Annual International Symposium on Computer Architecture*, pp. 203–212, May 1996.

[2] Laudon, J.; A. Gupta; et al. "Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors." In *Multithreaded Computer Architectures: A Summary of the State of the Art*, edited by Guang Gao, et al, pp. 167–200. Kluwer Academic Publishers, Norwell, Mass., 1994.

[3] Tullsen, D.; S. Eggers; et al. "Exploiting Choices: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor." In *23rd Annual International Symposium on Computer Architecture*, pp. 191–202, May 1996.

[*] Moore, S. *Multithreaded Processor Design*. Kluwer Academic Publishers, Norwell, Mass., 1996.

[*] Tera Computer Company, *http://www.tera.com*.