HYPERTHREADING TECHNOLOGY IN THE NETBURST MICROARCHITECTURE

BY USING EXISTING PROCESSOR RESOURCES MORE EFFICIENTLY, HYPERTHREADING TECHNOLOGY IMPROVES PERFORMANCE AT LITTLE COST AND INCREASES CHIP SIZE BY LESS THAN 5 PERCENT.

David Koufaty
Deborah T. Marr
Intel

Hyperthreading technology, which brings the concept of simultaneous multithreading to the Intel architecture, was first introduced on the Intel Xeon processor in early 2002 for the server market. In November 2002, Intel launched the technology on the Intel Pentium 4 at clock frequencies of 3.06 GHz and higher, making the technology widely available to the consumer market. This technology signals a new direction in microarchitecture development and fundamentally changes the cost-benefit tradeoffs of microarchitecture design choices.

This article describes how the technology works, that is, how we make a single physical processor appear as multiple logical processors to operating systems and software. We highlight the additional structures and die area needed to implement the technology and discuss the fundamental ideas behind the technology and why we can get a 25-percent boost in performance from a technology that costs less than 5 percent in added die area. We illustrate the importance of choosing the right sharing policy for each shared resource by describing, examining, and comparing three different sharing policies: partitioned resources, threshold sharing, and full sharing. The choice of policy depends on the traffic pattern, complexity and size of the resource, potential deadlock/livelock scenarios, and other considerations. Finally, we show how this technology significantly improves performance on several relevant workloads.

The die photos and descriptions in this article illustrate the technology's first implementations on Intel's Xeon and Pentium 4 processor families. These first implementations emphasized cost containment. Future implementations should provide even greater performance benefits.

Background: Processor microarchitecture

Traditional approaches to processor design have focused on higher clock speeds, instruction-level parallelism, and cache hierarchies. An orthogonal set of techniques leverages the thread-level parallelism to further improve processor performance.

Higher clock speed

Techniques to achieve higher clock speeds involve pipelining the microarchitecture to finer granularities, also called superpipelining. Higher clock frequencies can greatly improve performance by increasing the number of instructions executed each second. Because a superpipelined microarchitecture has far more instructions in flight, handling events that disrupt the pipeline—for example, cache miss-

Published by the IEEE Computer Society

es, interrupts, and branch mispredictions—can be costly.

Instruction-level parallelism

Instruction-level parallelism refers to techniques to increase the number of instructions executed each clock cycle. For example, a superscalar processor has multiple parallel execution units that can process instructions simultaneously, so that several instructions can execute each clock cycle. However, with simple in-order execution, just having multiple execution units isn't enough; the challenge is to find enough instructions to execute. One technique is out-of-order execution, whereby the processor evaluates a large window of instructions simultaneously and sends them to execution units on the basis of instruction dependencies rather than program order. With out-of-order execution, however, instruction dependencies and cache misses limit the number of instructions that can be executed simultaneously.

Cache hierarchies

Cache hierarchies have traditionally served to reduce the number of cycles processors spend waiting for data from memory. Having frequently used data on the processor caches reduces the frequency of accesses to the slower memory. Current microprocessors use multiple levels of caches, with smaller and faster caches located closer to the processor. Cache hierarchies, however, are limited by cache latency and die area, because larger caches have higher latencies and require a larger fraction of the processor area.

Thread-level parallelism

Today's software developers want to execute an increasing number of different tasks simultaneously. These workloads can take the form of multithreaded applications; for example, online transaction processing and Web services have an abundance of software threads that can execute simultaneously. It can also entail multiple applications, with users Web browsing, listening to music, and encoding/decoding video streams all at the same time. Intel architects have been trying to leverage this so-called thread-level parallelism to improve performance while controlling transistor count and power consumption.

In recent years, researchers have discussed other techniques to further exploit thread-level parallelism. One of these techniques is chip multiprocessing (CMP), whereby two processors, each with a full set of execution and architectural resources, reside on a single die.[1,2] The processors might or might not share a large on-chip cache. CMP is largely orthogonal to conventional multiprocessor systems because a multiprocessor configuration can have multiple CMP processors. Hewlett-Packard and IBM recently announced products incorporating two processors on each die.[3,4] However, a CMP chip is significantly larger than a single-core chip and therefore more expensive to manufacture. Moreover, it doesn't address die size and power considerations.

Another approach is to let a single processor execute multiple threads by switching between them. With *time-slice multithreading*, the processor switches between software threads after a fixed time period.[5-7] Although it can result in wasted execution slots, time-slice multithreading can effectively minimize the effects of long latencies to memory. *Switch-on-event* multithreading switches threads on long-latency events such as cache misses.[8] This approach can work well for server applications that have numerous cache misses and where the two threads are executing similar tasks. However, time-slice and switch-on-event multithreading techniques both fail to achieve optimal overlap of many sources of inefficient resource usage, such as branch mispredictions and instruction dependencies.

Finally, there is *simultaneous multithreading*,[9,10] whereby multiple threads execute on a single processor without switching. Simultaneous execution of threads uses processor resources most effectively, maximizing performance relative to transistor count and power consumption.

Hyperthreading technology brings the simultaneous multithreading approach to the Intel architecture. This article discusses the architecture and the first implementation of hyperthreading technology on the Netburst microarchitecture.

Hyperthreading technology architecture

Hyperthreading technology makes a single physical processor appear to be multiple log-

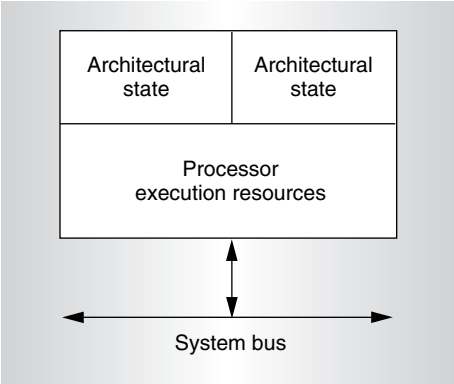

Figure 1. A physical processor capable of hyperthreading technology has two copies of the architectural state and thus appears to have two logical processors.

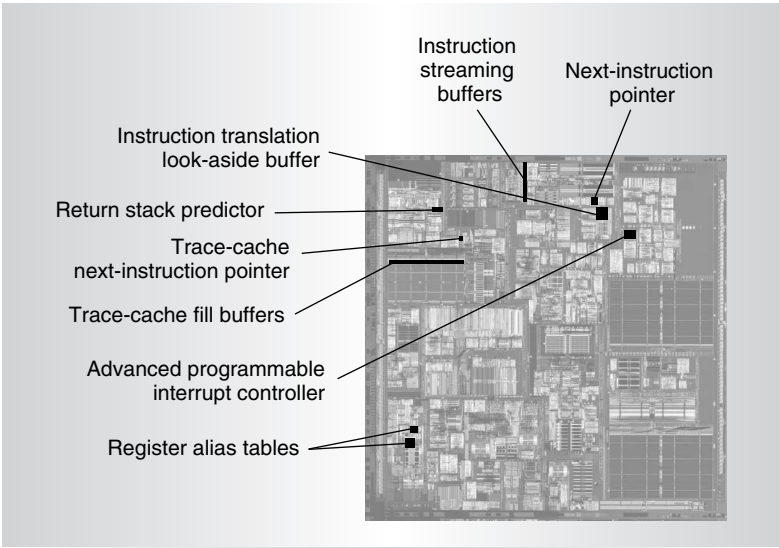

Figure 2. Intel Pentium 4 and the visible processor resources duplicated to support hyperthreading technology. Hyperthreading requires duplication of additional miscellaneous pointers and control logic, but these are too small to point out.

ical processors. There is one copy of the architectural state for each logical processor, and these 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, it means that instructions from logical processors will persist and execute simultaneously on shared execution resources. This can greatly improve processor resource utilization.

The hyperthreading technology implementation on the Netburst microarchitecture has two logical processors on each physical processor. Figure 1 shows a conceptual view of processors with hyperthreading technology capability.

Each logical processor maintains a complete set of the architectural state. The architectural state consists of registers, including general-purpose registers, and those for control, the advanced programmable interrupt controller (APIC), and some for machine state. From a software perspective, duplication of the architectural state makes each physical processor appear to be two processors. Each logical processor has its own interrupt controller, or APIC, which handles just the interrupts sent to its specific logical processor.

Hyperthreading technology is fully compatible with existing software and hardware. However, software optimizations like those described in the *Intel Pentium 4 Processor Optimization Reference Manual*[11] will result in better performance. Newer operating systems, such as Microsoft Windows XP, are already optimized for the best performance.

Die size and complexity

The vast majority of techniques that improve processor performance from one generation to the next are complex and often significantly increase die size and power costs. These techniques increase performance, but not with 100-percent efficiency. Because of limited parallelism in instruction flows, doubling the number of execution units in a processor doesn't double the processor's performance. Similarly, simply doubling the clock rate doesn't double performance, because a certain number of processor cycles are lost to branch mispredictions. Assuming the same process technology, processor die area has grown at a rate three times that of integer performance.[12]

Hyperthreading technology can deliver a large performance improvement at minimal cost because it entails only a small increase in die size. Logical processors share nearly all resources on the physical processor, including caches, execution units, branch predictors, control logic, and buses. The increase in die

size is due to a second architectural state, additional control logic, and replication of a few key processor resources. This limited replication of processor resources, as Figure 2 shows, accounts for most of the die size increase. The transistors required for the extra architectural state and the additional control logic consume an extremely small amount of die space.

The duplicated structures indicated in Figure 2 reduce complexity and improve performance. The *register alias tables* map the architectural registers to physical rename registers. The architectural registers must be tracked independently for each logical processor, requiring a separate table for each logical processor.

Duplicating the *next-instruction pointer* and associated control logic permits independent tracking of program progress for each logical processor. There are two sets of next-instruction pointer logic: one at the trace cache, which serves as the first-level instruction cache and stores decoded instructions; and the other set at the instruction decoder logic for use in the case of a trace-cache miss.

The *return stack predictor* is duplicated for accurate tracking of call/return pairs. This allows for improved call/return prediction. *Instruction streaming buffers* and *trace-cache fill buffers* are front-end buffers duplicated for effective instruction prefetch.

Designers duplicated the *instruction translation look-aside buffer* because there was enough room and its small size made replication simpler than sharing. Duplicating the APIC registers allows interrupts to go to each logical processor independently.

Even though the die area increase was small, the increase in design complexity was substantial. Hyperthreading technology challenged many basic assumptions about single-threaded out-of-order design.

First, designers had to devise many new algorithms to let both logical processors share the logic, and they had to revisit other algorithms to prioritize microoperations, or micro-ops, from different logical processors. For example, algorithms that depend on the age of a micro-op became much more complicated because it was not clear how to determine the age and priority of instructions from two different logical processors. Also, designers paid special attention to addressing potential livelock scenarios in which one logical processor blocks the other. Designers devised algorithms that inherently avoided livelocks, but also added fallback algorithms just in case.

Second, logic complexity was high because of pointer manipulation, additional multiplexers, duplicated state, and new boundary conditions. The x86 architecture was already complex enough, but hyperthreading adds two logical processors that can operate in any combination of x86 operating modes and events.

Finally, hyperthreading technology opens up a whole new space of validation. Increased complexity dramatically increases the validation effort. To validate that two logical processors could operate as if they were two physical processors, engineers had to validate every combination of major operating mode and event. Two logical processors have many more interactions than two physical processors in a conventional multiprocessor system, because they share the same physical resources. On the platform side, designers had to carefully review and optimize chipset, BIOS, operating systems, and applications.

Microarchitecture choices and tradeoffs

The current hyperthreading technology implementation required some microarchitecture choices and tradeoffs. The choice of a resource sharing policy for each shared resource is important because it can dramatically impact performance. In determining how to share resources, we chose among possible sharing schemes that included

- *partition*, dedicating equal resources to each logical processor;
- *threshold*, flexible resource sharing with a limit on maximum resource usage; and
- *full sharing*, flexible resource sharing with no limit on the maximum resource usage.

The choice required us to consider throughput versus fairness and potential livelock scenarios, as well as die size and complexity.

Partition

In a partitioned resource, each logical processor can use only half the entries. Clearly, resource partitioning has the advantage of simplicity and low complexity. It is a good choice

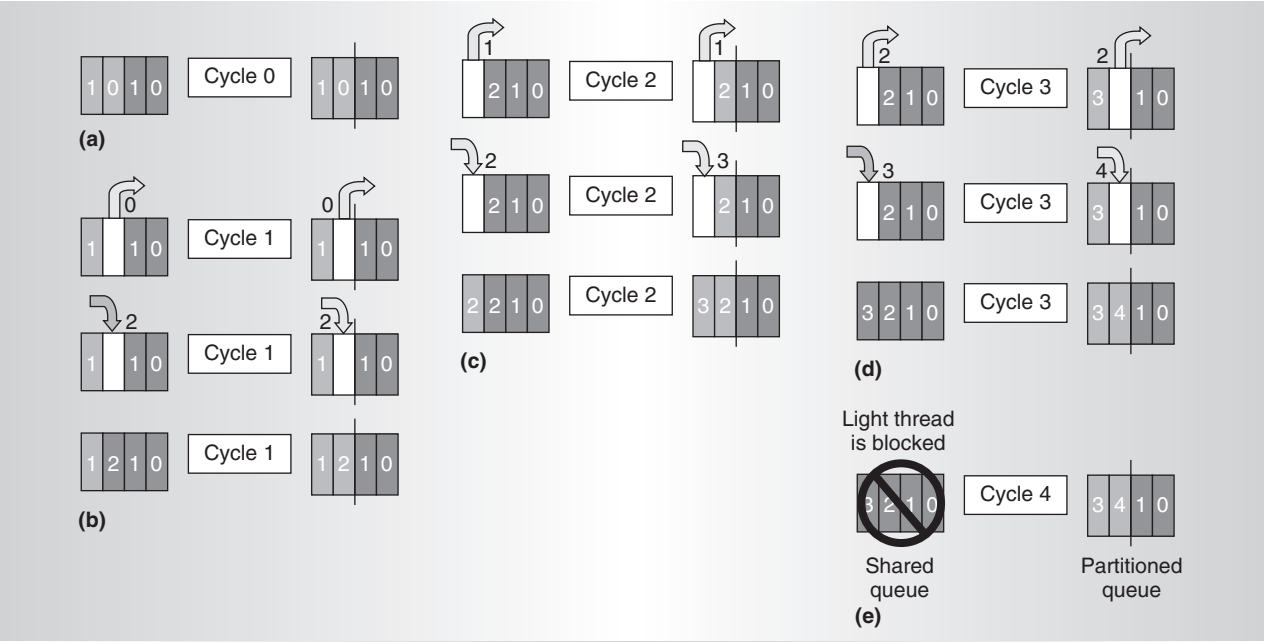

Figure 3. Comparison of a shared and a partitioned queue. The slower (dark) thread has a downstream stall, such as a data-cache miss. In this situation, the queues will not send any slower micro-ops to the next pipeline stage. The figure shows how the queues will progress through cycles 0 (a), 1 (b), 2 (c), and 3 (d). Eventually, in cycle 4 (e) the shared queue lets a slower thread block the progress of the faster (light) thread.

for resources when you expect the structure's utilization to be generally high and somewhat unpredictable. For example, partitioning is a good choice for major pipeline queues, which provide buffering to avoid pipeline stalls and, ideally, remain full most of the time. However, because software thread execution speeds differ at any instant in time, the rate at which the queues fill and empty is unpredictable. By partitioning these queues, we can allow slip between a fast and a slow software thread, preventing a slow thread from blocking or slowing down the faster thread and thereby making the best use of each pipeline stage.

Figure 3 shows how this works. At the start, in Figure 3a, both the shared queue and the partitioned queue have two light-shaded and two dark-shaded micro-ops. Both the light micro-ops and the dark micro-ops are labeled 1 and 0.

In cycle 1, Figure 3b, both the shared and partitioned queues send light micro-op 0 down to the next pipeline stage. In the shared queue, the previous pipeline stage sends dark micro-op 2, but in the partitioned queue, because the dark thread is already occupying its maximum number of entries, the previous pipeline stage sends a light micro-op instead (light micro-op 2). At the end of cycle 1, the shared queue has one light micro-op and three dark micro-ops. The partitioned queue has two micro-ops of each shade.

In cycle 2, Figure 3c, both the shared and partitioned queues send a light micro-op to the next pipeline stage, and the previous pipeline stage delivers a light micro-op in both cases. The shared queue gets a light micro-op in this cycle because in the previous cycle it sent a dark micro-op. In general, in-order pipeline stages will alternate between light and dark micro-ops unless the staging queue after the pipeline stage is full or the previous staging queue has no micro-ops available to work on.

In cycle 3, Figure 3d, both queues again send a light micro-op to the next pipeline stage. The previous pipeline stage sends a dark micro-op in the case of the shared queue and a light micro-op in the case of the partitioned queue. At the end of cycle 3, the shared queue has four dark micro-ops and no light micro-ops, while the partitioned queue still has two of each.

In Figure 3e, the shared queue is now blocked because it has no light micro-ops, and the dark thread has a downstream stall. The partitioned queue can continue to issue light micro-ops.

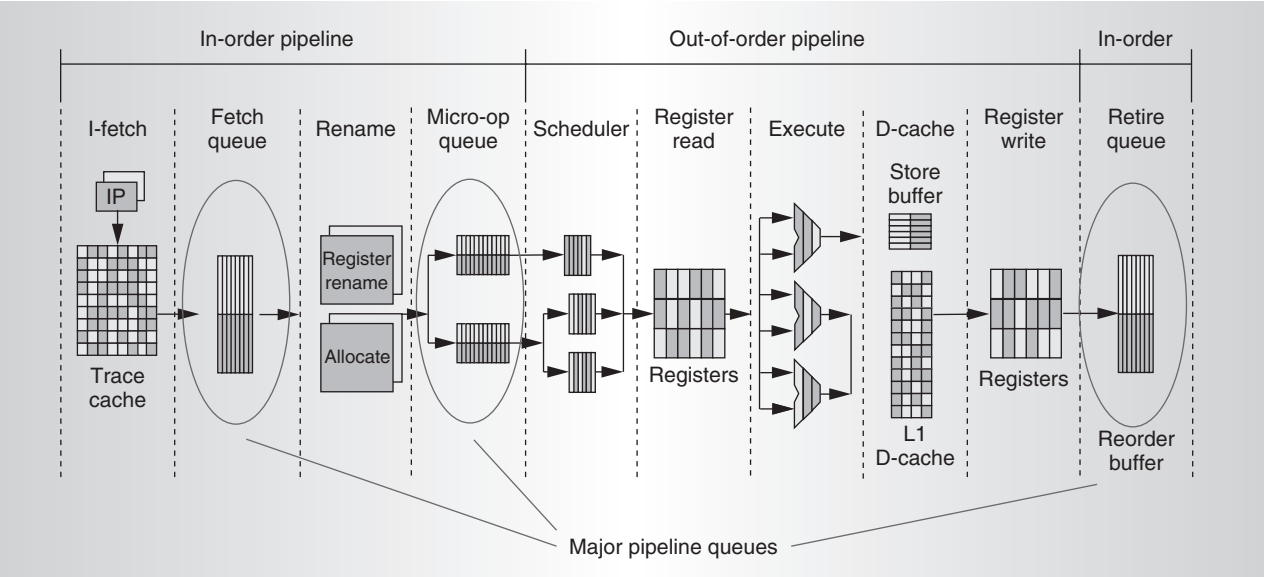

Figure 4. In this view of a Netburst microarchitecture's execution pipeline, the light and dark areas indicate the resource utilization of the two software threads running on the two logical processors.

Figure 4 shows a basic execution pipeline of the Netburst microarchitecture. The microarchitecture, detailed in the literature,[13] works on micro-ops and decodes each x86 instruction into one or more micro-ops. As the figure shows, micro-ops are first fetched from the trace cache; then they're renamed, scheduled, executed, and finally retired. At a very high level, there is an in-order front-end pipeline, an out-of-order execution pipeline, and in-order retirement. In the out-of-order pipeline, instruction dependency chains determine execution resource utilization more than any arbitration schemes.

It's especially important to guarantee fairness and progress for the pipeline's in-order parts. Therefore, a partitioned scheme works best for the major pipeline queues in the in-order pipeline. If there is a front-end stall (say, because of a trace-cache miss), the back end can continue to take micro-ops from the micro-ops queue. If there is a back-end stall (say, because of a data cache miss), the front end can continue to fill the queue. Large queues can keep both the front end and the back end mostly busy when one end is temporarily stalled for one logical processor.

As Figure 3 shows, if the two logical processors fully shared these queues, a slow thread could gain an unfair share of the resources and prevent a fast thread from making progress. Because the slow thread is often stalled, its micro-ops start to pile up in the queues. In time, the slow thread will collect more and more entries, because it competitively shares entries with the fast thread. Eventually, the slow thread will get most, if not all, of the queue, thereby slowing the fast thread's progress. A partitioned queue, however, lets the fast thread always have half of the entries and advance at its own pace.

Partitioning resources is simple, entails little implementation complexity, and ensures fairness and progress for both logical processors.

Threshold

Another way of sharing resources is to limit the maximum resource usage. This scheme, instead of partitioning the resource, puts a threshold on the number of resource entries a logical processor can have.

This approach is ideally suited for small structures where the resource utilization is bursty, and the length of time a micro-op stays in the structure is short, fairly uniform, and predictable. Processor schedulers are a good example of where threshold sharing is the best choice. Scheduler throughput is high because the schedulers assume that load instructions will hit in the cache, so micro-ops don't linger in the schedulers (a separate reissue mechanism would resubmit micro-

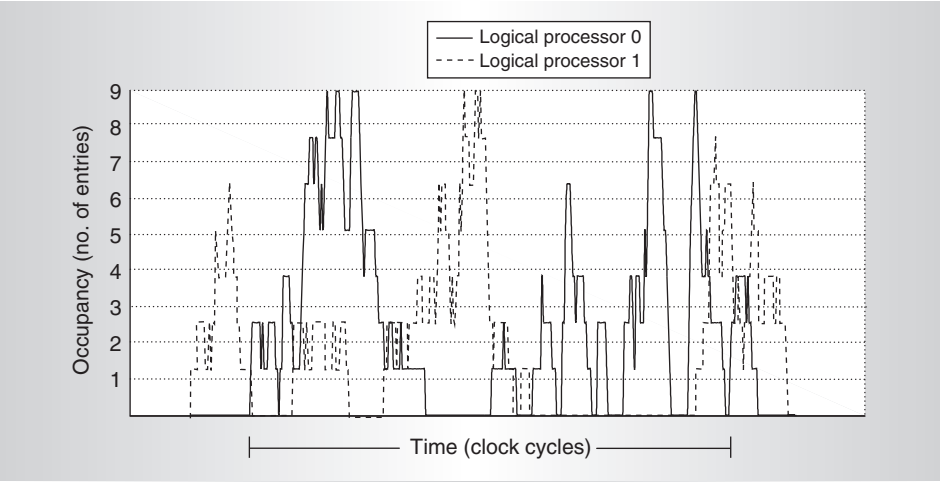

Figure 5. Snapshot of scheduler occupancy on a transaction processing workload over a short period of time. Each data point is the instantaneous scheduler occupancy for its respective logical processor, measured by the number of entries occupied by each thread.

ops to execution units in the event of a cache miss). Also, the schedulers are very small, to enable speed. They run at twice the clock frequency, so a 3-GHz processor has schedulers running at 6 GHz.

The allocation of micro-ops to these schedulers is round-robin until a logical processor reaches its threshold number of entries. At that point, it cannot allocate more micro-ops until it dispatches some of its current entries.

Figure 5 shows scheduler occupancy over a number of processor clock cycles. Although average scheduler utilization is low, the activity can be bursty. A threshold limiting the maximum number of entries each logical processor can use prevents one logical processor from blocking the other's access to the scheduler. The threshold lets the scheduler look for maximum parallelism among micro-ops across both threads, thereby improving execution resource utilization.

Full sharing

Fully shared resources, the most flexible mechanism for resource sharing, do not limit the maximum resource usage for a logical processor. In general, fully shared resources are good for large structures in which working-set sizes are variable, and one logical processor cannot starve the other.

Processor caches are a good example of structures best suited to the full-sharing policy. In the Netburst microarchitecture, all processor caches are shared. First, this allows for better overall performance than with a partitioned or threshold cache because cache interference is usually modest. Second, some applications benefit from a shared cache because they share code and data, minimizing redundant data in the caches. Finally, high set associativity minimizes conflict misses between logical processors. The second- and third-level caches (if present) are eight-way set associative.

Because hyperthreading technology is a new architectural field, we implemented multiple resource management algorithms in some areas of the processor. This includes the cache-sharing policy. This feature lets us experiment with various cache management policies on real systems. Figure 6 shows results for some of those experiments and the advantage of using a shared cache. The figure compares the results of running multiple workloads on two cache configurations: fully shared and partitioned. For each workload, the figure shows the cache hit rate and performance impact of a fully shared cache normalized to those of a partitioned cache. We collected cache miss statistics using the Intel Pentium 4 event-monitoring counters,[14] specifically the second-level cache's load-misses-retired event. The workload consisted of running two copies of the same application. This scenario highlights the modest cache interference in a shared cache.

Performance

Hyperthreading technology improves overall performance in two ways. First, it speeds up applications that are already multithreaded. In this case, each logical processor will run software threads from the same application. Second, it speeds up a workload consisting of multiple applications by multitasking. In this case, each logical processor will likely run threads from different applications.

Figure 7 shows the hyperthreading technology performance boost on current popular software packages. The technology delivers

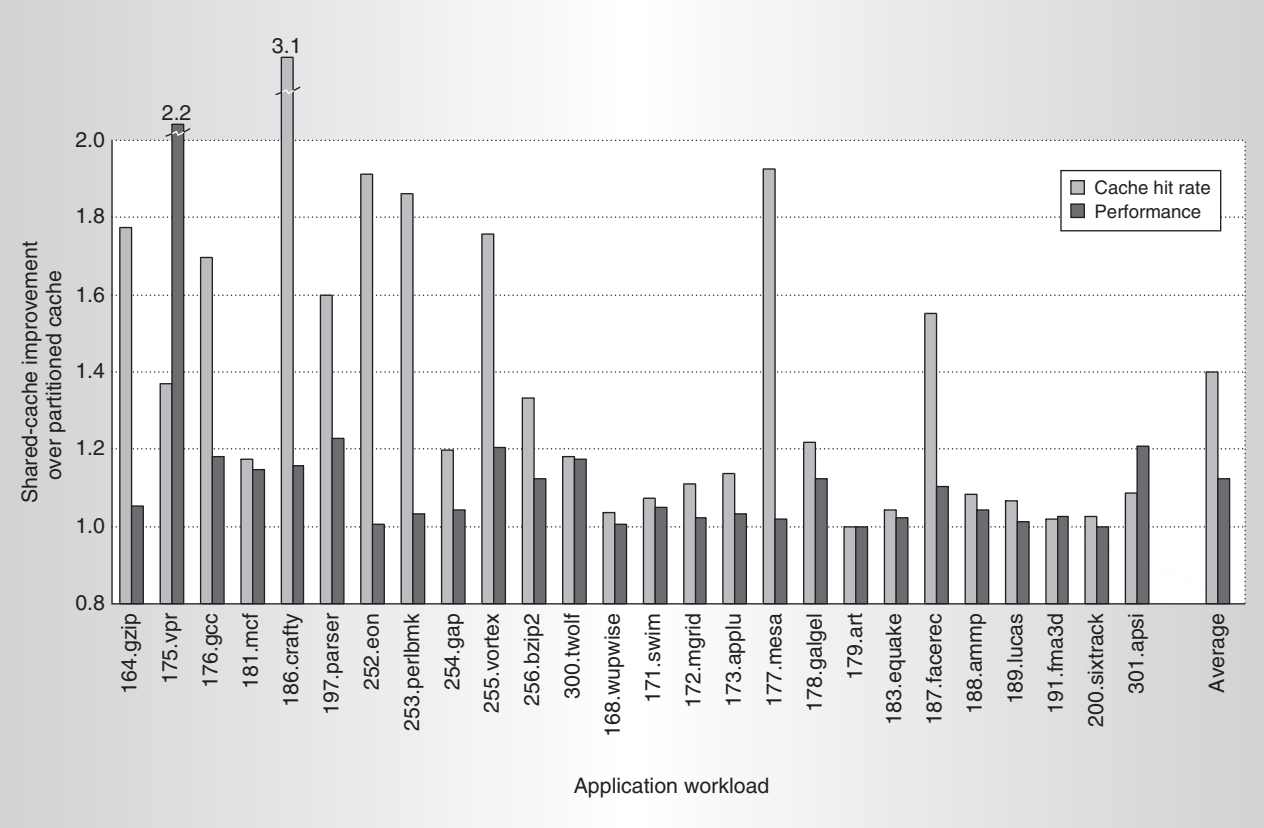

Figure 6. Cache hit rate and overall performance impact for a fully shared cache normalized against values for a partitioned cache. On average, the shared cache had a 40-percent better cache hit rate and 12-percent better performance. Notice that no single application workload lost performance because of the shared cache.

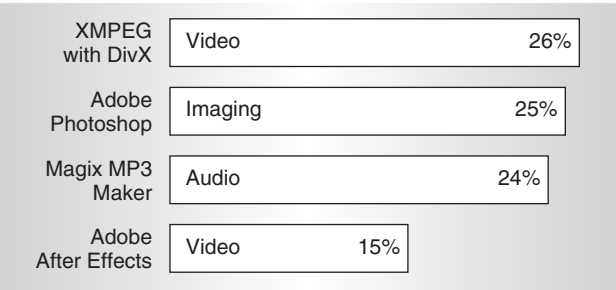

Figure 7. Hyperthreading technology performance gains on several popular multithreaded software packages.

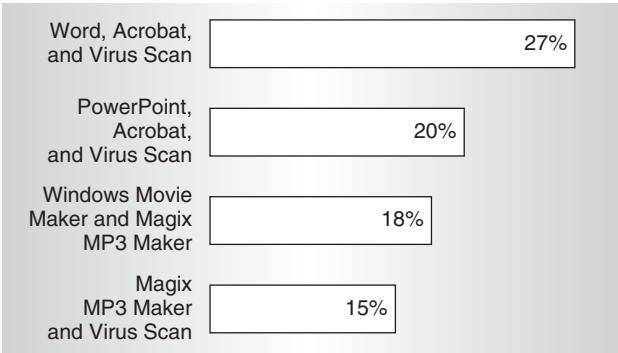

Figure 8. Hyperthreading technology performance boost on multitasking workloads.

a 15 to 26 percent performance boost on these multithreaded applications.

Figure 8 shows the performance benefits that hyperthreading technology delivers for several multitasking workloads. The performance boost, 15 to 27 percent, resembles that of the multithreaded case. The "Test system configuration" sidebar describes the system used for these performance tests.

With a resource sharing policy matched to the traffic and performance requirements of each resource, hyperthreading technology can increase resource utilization and improve performance. Intel is committed to this new and challenging microarchitecture direction. More than two and a half years of

Test system configuration

Hyperthreading technology requires a computer system with an Intel Pentium 4 processor running at 3.06 GHz or higher, a chipset and BIOS that utilize this technology, and an operating system that includes optimizations for the technology.

The system used to obtain the performance increases shown in Figures 7 and 8 consists of a 3.06-GHz Intel Pentium 4 processor with hyperthreading technology (enabled/disabled), an Intel Desktop Board D850EMV2, a 256-Mbyte PC1066 RDRAM, all platform configuration Leadtek WinFast A250 Ultra TD GeForce 4/ nVidia GeForce 4 4x AGP graphics, an nVidia Detonator 4 reference driver 28.32, an Intel Application Accelerator v2.2.2128, the Intel Chipset Software Installation Utility v4.00.1009, an IBM 80-Gbyte 120GXP IC35L080AVVA07-0 ATA-100 hard drive, Intel C and Fortran compilers 5.01 for SPEC, DirectX 8.1, Windows XP (build 2600), and a 100-Mbps Intel Pro/100+ Management PCI LAN card.

Performance tests and ratings reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration might affect actual performance.

system experimentation has provided enormous insight and helped direct future implementation choices. We expect to continuously improve hyperthreading technology for years to come. MICRO

Acknowledgments

Many architects, design engineers, and software engineers made significant contributions to the architecture and microarchitecture design choices. Glenn Hinton first proposed and then championed the idea of adding hyperthreading technology to the Netburst microarchitecture. Key architects who worked with design engineers and helped define microarchitecture algorithms include Darrell Boggs, Doug Carmean, Per Hammarlund, David Hill, Alan Kyker, David Sager, and Mike Upton. David Burns coordinated the enormous validation effort. Bryant Bigbee, Shiv Kaushik, and Jim Crossland made fundamental contributions by providing key insights from the operating system and software perspective.

References

1. L.A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 282-293.
2. L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *Computer*, vol. 30, no. 9, Sept. 1997, pp. 79-85.
3. D.J.C. Johnson, "HP's Mako Processor," *Microprocessor Forum*, Oct. 2001, http://www.cpus.hp.com/technical_references/mpf_2001.pdf.
4. J.M. Tendler, S. Dodson, and S. Fields, "POWER4 System Microarchitecture," tech. white paper, IBM Server Group, Oct. 2001.
5. R. Alverson et al., "The TERA Computer System," *Proc. Int'l Supercomputing Conf.*, IEEE CS Press, 1990, pp. 1-6.
6. M. Fillo et al., "The M-Machine Multicomputer," *Proc. 28th Ann. Int'l Symp. Microarchitecture* (Micro-28), IEEE CS Press, Nov. 1995, pp. 146-156.
7. B.J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Proc. SPIE Real Time Signal Processing IV*, 1981, pp. 241-248.
8. A. Agarwal et al., "APRIL: A Processor Architecture for Multiprocessing," *Proc. 17th Ann. Int'l Symp. Computer Architecture* (ISCA 90), IEEE CS Press, 1990, pp. 104-114.
9. D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Ann. Int'l Symp. Computer Architecture* (ISCA 95), ACM, 1995. pp. 392-403.
10. D. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 23rd Ann. Int'l Symp. Computer Architecture* (ISCA 96), ACM, 1996, pp. 191-202.
11. *Intel Pentium 4 Processor Optimization Reference Manual*, Intel Corp., order no. 248966, http://developer.intel.com/design/pentium4/manuals.
12. D.T. Marr et al., "Hyperthreading Technology Architecture and Microarchitecture," *Intel Technology J.*, vol. 6, no. 1, Feb. 2002, http://www.intel.com/technology/itj/2002/volume06issue01/.
13. G. Hinton et al., "The Microarchitecture of the Pentium 4 Processor," *Intel Technology J.*, 1st quarter 2001, http://www.intel.com/technology/itj/q12001.htm.
14. *IA-32 Intel Architecture Software Developer's Manual, Vol. 3: System Programming Guide*, Intel Corp., 2001, order no. 244472, http://developer.intel.com/design/pentium4/manuals.

David Koufaty is a CPU architect with Intel's Desktop Products Group and is responsible for hyperthreading technology performance.

His main research interests are processor microarchitecture and performance. He has BS and MS degrees from Simón Bolívar University, Venezuela, and a PhD in computer science from the University of Illinois at Urbana-Champaign.

Deborah T. Marr is one of the CPU architects in the Intel Desktop Products Group responsible for hyperthreading technology. Her research interests include high-performance microarchitecture and performance analysis. She has a BS in electrical engineering and computer science from the University of California, Berkeley, and an MS in electrical and computer engineering from Cornell University.

Direct questions and comments about this article to David Koufaty, Intel Corp., JF4-354, 2111 NE 25th Ave., Hillsboro, OR 97124-5961; dkoufaty@ichips.intel.com.