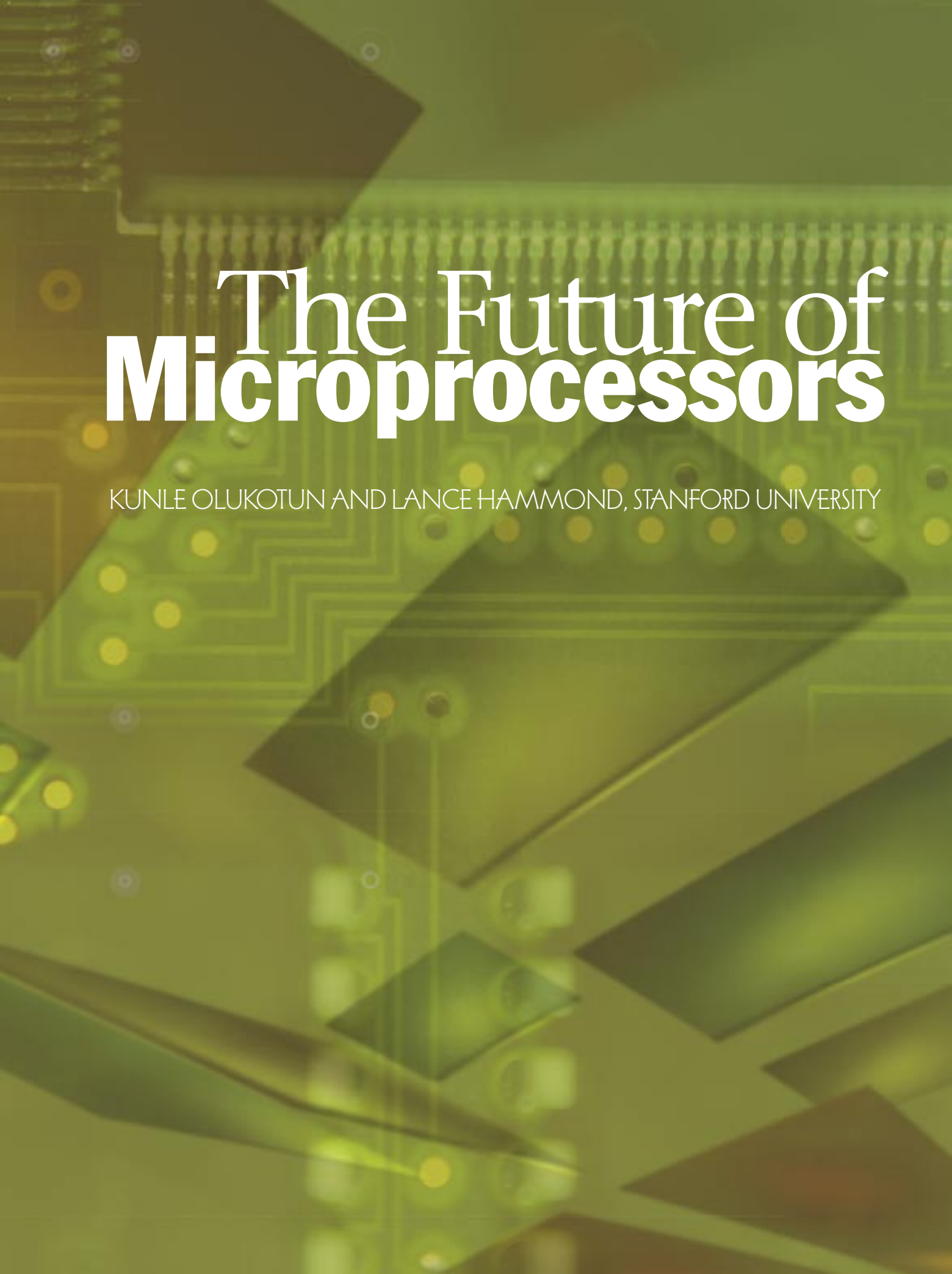
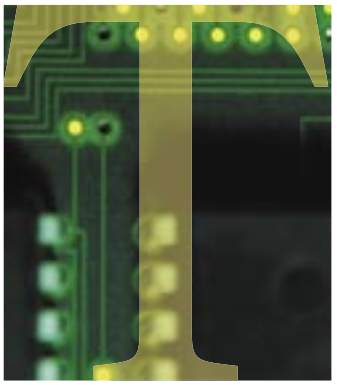


The Future of Microprocessors

KUNLE OLUKOTUN AND LANCE HAMMOND, STANFORD UNIVERSITY

Chip multiprocessors' promise of huge performance gains is now a reality.



The performance of microprocessors that power modern computers has continued to increase exponentially over the years for two main reasons. First, the transistors that are the heart of the circuits in all processors and memory chips have simply become faster over time on a course described by Moore's law,[1] and this directly affects the performance of processors built with those transistors. Moreover, actual processor performance has increased *faster* than Moore's law would predict,[2] because processor designers have been able to harness the increasing numbers of transistors available on modern chips to extract more *parallelism* from software. This is depicted in figure 1 for Intel's processors.

An interesting aspect of this continual quest for more parallelism is that it has been pursued in a way that has been virtually invisible to software programmers. Since they were invented in the 1970s, microprocessors have continued to implement the conventional von Neumann computational model, with very few exceptions or modifications. To a programmer, each computer consists of a single processor executing a stream of sequential instructions and connected to a monolithic "memory" that holds all of the program's data. Because the economic benefits of backward compatibility with earlier generations of processors are so strong, hardware designers have essentially been limited to enhancements that have maintained this abstraction for decades. On the memory side, this has resulted in processors with larger cache memories, to keep frequently accessed portions of the conceptual "memory" in small, fast memories that are physically closer to the processor, and large register files to hold more active data values in an

The Future of Microprocessors

extremely small, fast, and compiler-managed region of "memory."

Within processors, this has resulted in a variety of modifications designed to achieve one of two goals: increasing the number of instructions from the processor's instruction sequence that can be issued on every cycle, or increasing the clock frequency of the processor faster than Moore's law would normally allow. Pipelining of individual instruction execution into a sequence of stages has allowed designers to increase clock rates as instructions have been sliced into larger numbers of increasingly small steps, which are designed to reduce the amount of logic that needs to switch during every clock cycle. Instructions that once took a few cycles to execute in the 1980s now often take 20 or more in today's leading-edge processors, allowing a nearly proportional increase in the possible clock rate.

Meanwhile, superscalar processors were developed to execute multiple instructions from a single, conventional instruction stream on each cycle. These function by dynamically examining sets of instructions from the instruction stream to find ones capable of parallel execution on each cycle, and then executing them, often out of order with respect to the original program.

Both techniques have flourished because they allow instructions to execute more quickly while maintaining the key illusion for programmers that all instructions are actually being executed sequentially and in order, instead of overlapped and out of order. Of course, this illusion is not absolute. Performance can often be improved if programmers or compilers adjust their instruction scheduling and data layout to map more efficiently to the underlying pipelined or parallel architecture and cache memories, but the important point is that old or untuned code will still execute correctly on the architecture, albeit at less-than-peak speeds.

Unfortunately, it is becoming increasingly difficult for processor designers to continue using these techniques to enhance the speed of modern processors. Typical instruction streams have only a limited amount of usable parallelism among instructions,[3] so superscalar processors that can issue more than about four instructions per cycle achieve very little additional benefit on most applications. Figure 2 shows how effective real Intel processors have been at extracting instruction parallelism over time. There is a flat region before instruction-level parallelism was pursued intensely, then a steep rise as parallelism was utilized usefully, followed by a tapering off in recent years as the available parallelism has become fully exploited.

Complicating matters further, building superscalar processor cores that can exploit more than a few instructions per cycle becomes very expensive, because the complexity of all the additional logic required to find parallel instructions dynamically is approximately proportional to the square of the number of instructions that can be issued simultaneously. Similarly, pipelining past about 10-20 stages is difficult because each pipeline stage becomes too short to perform even a minimal amount of

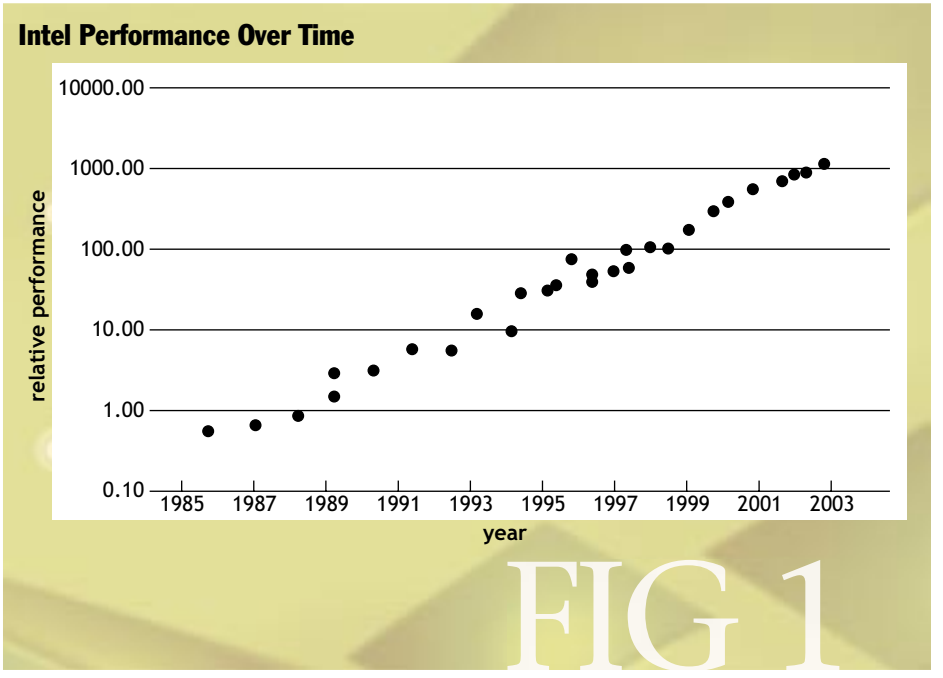


FIG 1

logic, such as adding two integers together, beyond which the design of the pipeline is significantly more complex. In addition, the circuitry overhead from adding pipeline registers and bypass path multiplexers to the existing logic combines with performance losses from events that cause pipeline state to be flushed, primarily branches. This overwhelms any potential performance gain from deeper pipelining after about 30 stages.

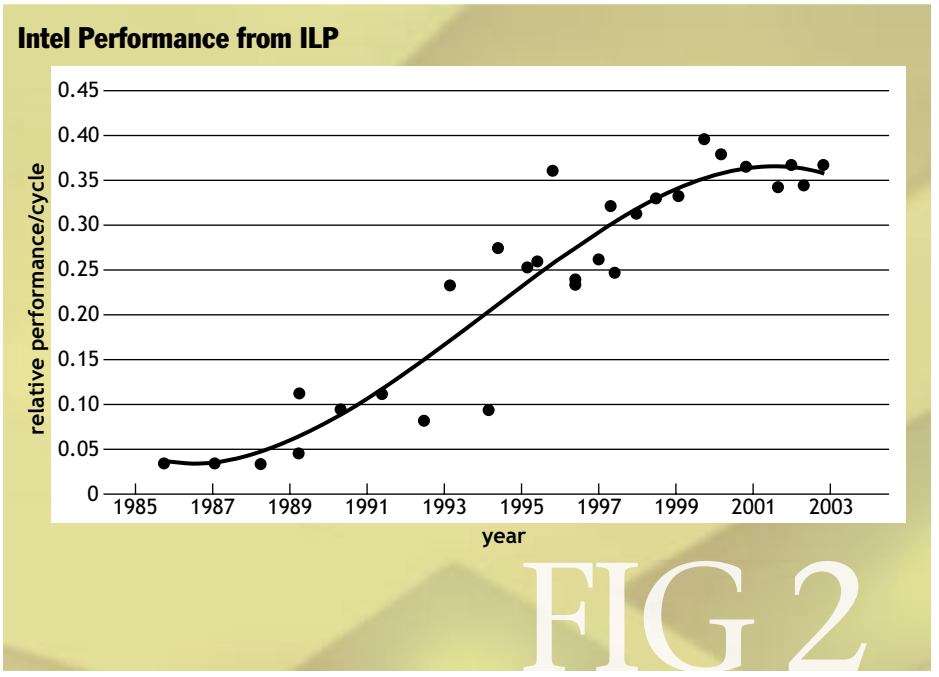
Further advances in both superscalar issue and pipelining are also limited by the fact that they require ever-larger numbers of transistors to be integrated into the high-speed central logic within each processor core—so many, in fact, that few companies can afford to hire enough engineers to design and verify these processor cores in reasonable amounts of time. These trends have slowed the advance in processor performance somewhat and have forced many smaller vendors to forsake the high-end processor business, as they could no longer afford to compete effectively.

Today, however, all progress in conventional processor core development has essentially stopped because of a simple physical limit: power. As processors were pipelined and made increasingly superscalar over the course of the past two decades, typical high-end microprocessor power went from less than a watt to over 100 watts. Even though each silicon process generation promised a reduction in power, as the ever-smaller transistors required less power to switch, this was true in practice only when existing designs were simply "shrunk" to use the new process technology. Processor designers, however, kept using more transistors in their cores to add pipelining and superscalar issue, and switching them at higher and higher frequencies. The overall effect was that exponentially more power was required by each subsequent processor generation (as illustrated in figure 3).

Unfortunately, cooling technology does not scale exponentially nearly as easily. As a result, processors went from needing no heat sinks in the 1980s, to moderate-size heat sinks in the 1990s, to today's monstrous heat sinks, often with one or more dedicated fans to increase airflow over the processor. If these trends were to continue, the next generation of microprocessors would require very exotic cooling solutions, such as dedicated water cooling, that are economically impractical in all but the most expensive systems.

The combination of limited instruction parallelism suitable for superscalar issue, practical limits to pipelining, and a "power ceiling" limited by practical cooling limitations has limited future speed increases within conventional processor cores to the basic Moore's law improvement rate of the underlying transistors. This limitation is already causing major processor manufacturers such as Intel and AMD to adjust their marketing focus away from simple core clock rate.

Although larger cache memories will continue to improve performance somewhat, by speeding access to the single "memory" in the conventional model, the simple fact is that without more radical changes in processor design, microprocessor performance increases will slow dramatically in the future. Processor designers must find new ways to *effectively* utilize the increasing transistor budgets in high-end silicon chips to improve performance in ways that minimize both additional power usage *and* design complexity. The market for microprocessors has become stratified into areas with different performance requirements, so it is useful to examine the problem from the point of view of these different performance requirements.



The Future of Microprocessors

THROUGHPUT PERFORMANCE IMPROVEMENT

With the rise of the Internet, the need for servers capable of handling a multitude of independent requests arriving rapidly over the network has increased dramatically. Since individual network requests are typically completely independent tasks, whether those requests are for Web pages, database access, or file service, they are typically spread across many separate computers built using high-performance conventional microprocessors (figure 4a), a technique that has been used at places like Google for years to match the overall computation throughput to the input request rate.[4]

As the number of requests increased over time, more servers were added to the collection. It has also been possible to replace some or all of the separate servers with multiprocessors. Most existing multiprocessors consist of two or more separate processors connected using a common bus, switch hub, or network to shared memory and I/O devices. The overall system can usually be physically smaller and use less power than an equivalent set of uniprocessor systems because physically large components such as memory, hard drives, and power supplies can be shared by some or all of the processors.

Pressure has increased over time to achieve more performance per unit volume of data-center space and per watt, since data centers have finite room for servers and their electric bills can be staggering. In response, the server manufacturers have tried to save space by adopting denser server packaging solutions, such as blade servers and switching to multiprocessors that can share components. Some power reduction has also occurred through the sharing of more power-hungry components in these systems. These short-term solutions are reaching their practical limits, however, as systems are reaching the maximum component density that can still be effectively air-cooled. As a result, the next stage of development for these systems involves a new step: the CMP (chip multiprocessor).[5]

The first CMPs targeted toward the server market implement two or more conventional superscalar processors together on a single die.[6,7,8,9] The primary motivation for this is reduced volume—multiple processors can now fit in the space where formerly only one could, so overall performance per unit volume can be increased. Some savings in power also occurs because all of the processors on a single die can share a single connection to the rest of the system, reducing the amount of high-speed communication infrastructure required, in addition to the sharing possible with a conventional multiprocessor. Some CMPs, such as the first ones announced from AMD and Intel, share only the system interface between processor cores (illustrated in figure 4b), but others share one or more levels of on-chip cache (figure 4c), which allows interprocessor communication between the CMP cores without off-chip accesses.

Further savings in power can be achieved by taking advantage of the fact that while server workloads require high throughput, the *latency* of each request is generally

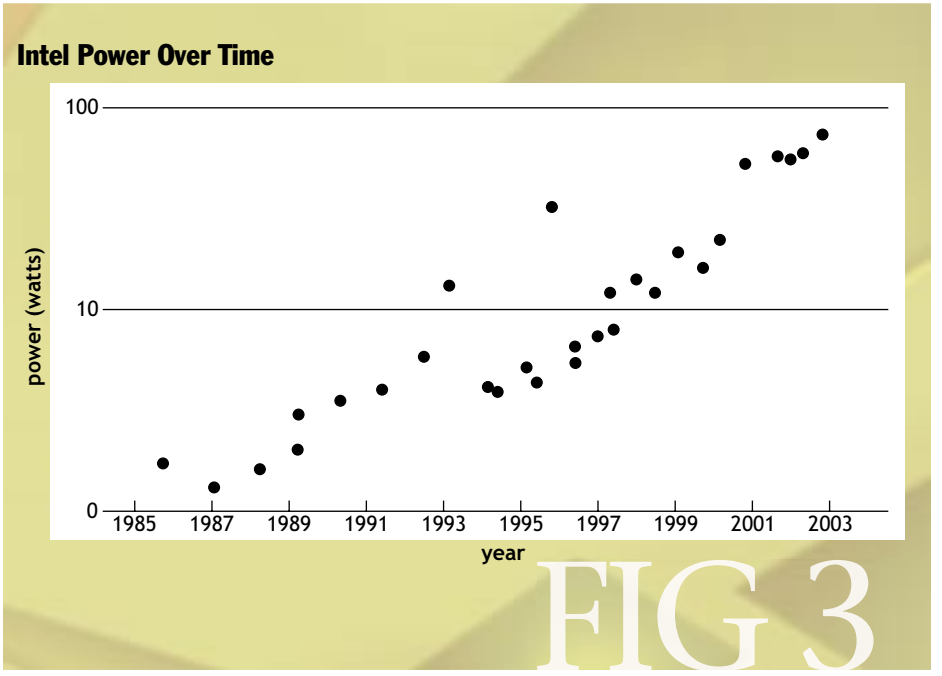


FIG 3

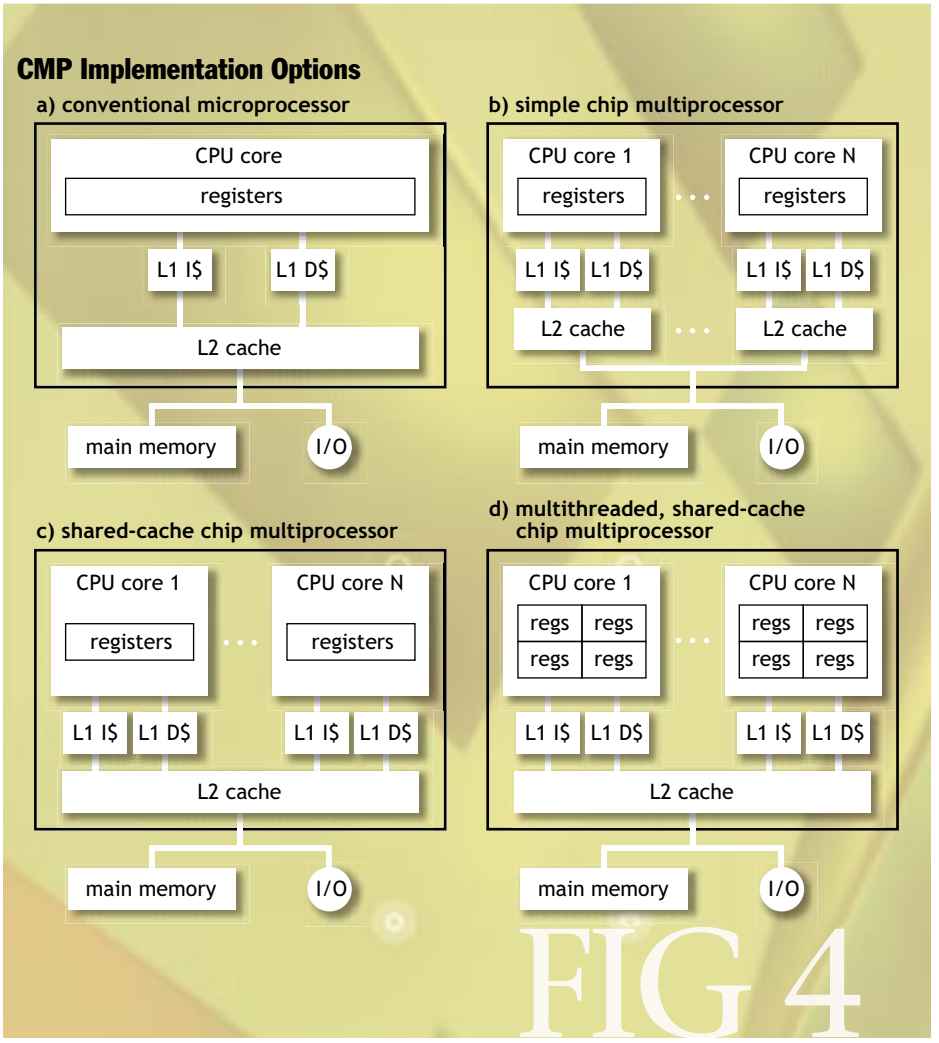
not as critical.[10] Most users will not be bothered if their Web pages take a fraction of a second longer to load, but they will complain if the Web site drops page requests because it does not have enough throughput capacity. A CMP-based system can be designed to take advantage of this situation.

When a two-way CMP replaces a uniprocessor, it is possible to achieve essentially the same or better throughput on server-oriented workloads with just *half* of the original clock speed. Each request may take up to twice as long to process because of the reduced clock rate. With many of these applications, however, the slowdown will be much less, because request processing time is more often limited by memory or disk performance than by processor performance. Since two requests can now be processed simultaneously, however, the overall throughput will now be the same or better, unless there is serious contention for the same memory or disk resources.

Overall, even though performance is the same or only a little better, this adjustment is still advantageous at the system level. The lower clock rate allows us to design the system with a significantly lower power supply voltage, often a nearly linear reduction. Since power is proportional to the *square* of the voltage, however, the power required to obtain the original performance is much lower—usually about half (half of the voltage squared = a quarter of the power, per processor, so the power required for both processors together is about half), although the potential savings could be limited by static power dissipation and any minimum voltage levels required by the underlying transistors.

For throughput-oriented workloads, even more power/performance and performance/chip area can be achieved by taking the "latency is unimportant" idea to its extreme and building the CMP with many small cores instead of a few large ones. Because typical server workloads have very low amounts of instruction-level parallelism and many memory stalls, most of the hardware associated with superscalar instruction issue is essentially wasted for these applications. A typical server will have tens or hundreds of requests in flight at once, however, so there is enough work available to keep many processors busy simultaneously.

Therefore, replacing each large, superscalar processor in a CMP with several small ones, as has been demonstrated successfully with the Sun Niagara,[11] is a winning policy. Each small processor will process its request more slowly than a larger, superscalar processor, but this latency slowdown is more than compensated for by the fact that the same chip area can be occupied by a much larger number of processors—about four times as many, in the case



The Future of Microprocessors

of Niagara, which has eight single-issue SPARC processor cores in a technology that can hold only a pair of superscalar UltraSPARC cores.

Taking this idea one step further, still more latency can be traded for higher throughput with the inclusion of multithreading logic within each of the cores.[12,13,14] Because each core tends to spend a fair amount of time waiting for memory requests to be satisfied, it makes sense to assign each core several threads by including multiple register files, one per thread, within each core (figure 4d). While some of the threads are waiting for memory to respond, the processor may still execute instructions from the others.

Larger numbers of threads can also allow each processor to send more requests off to memory in parallel, increasing the utilization of the highly pipelined memory systems on today's processors. Overall, threads will typically have a slightly longer latency, because there are times when all are active and competing for the use of the processor core. The gain from performing computation during memory stalls and the ability to launch numerous memory accesses simultaneously more than compensates for this longer latency on systems such as Niagara, which has four threads per processor or 32 for the entire chip, and Pentium chips with Intel's Hyperthreading, which allows two threads to share a Pentium 4 core.

LATENCY PERFORMANCE IMPROVEMENT

The performance of many important applications is measured in terms of the execution latency of individual tasks instead of high overall throughput of many essentially unrelated tasks. Most desktop processor applications still fall in this category, as users are generally more concerned with their computers responding to their commands as quickly as possible than they are with their computers' ability to handle many commands simultaneously, although this situation is changing slowly over time as more applications are written to include many "background" tasks. Users of many other computation-bound applications, such as most simulations and compilations, are typically also more interested in how long the programs take to execute than in executing many in parallel.

Multiprocessors can speed up these types of applications, but it requires effort on the part of programmers to break up each long-latency thread of execution into a large number of smaller threads that can be executed on many processors in parallel, since automatic parallelization technology has typically functioned only on Fortran programs describing dense-matrix numerical computations. Historically, communication between processors was generally slow in relation to the speed of individual processors, so it was critical for programmers to ensure that threads running on separate processors required only minimal communication with each other.

Because communication reduction is often difficult, only a small minority of users bothered to invest the time and effort required to parallelize their programs in a way that could achieve speedup, so these techniques were taught only in advanced, graduate-level computer science courses. Instead, in most cases programmers found that it was just easier to wait for the next generation of uniprocessors to appear and speed up their applications for "free" instead of investing the effort required to parallelize their programs. As a result, multiprocessors had a hard time competing against uniprocessors except in very large systems, where the target performance simply exceeded the power of the fastest uniprocessors available.

With the exhaustion of essentially all performance gains that can be achieved for "free" with technologies such as superscalar dispatch and pipelining, we are now entering an era where programmers *must* switch to more parallel programming models in order to exploit multiprocessors effectively, if they desire improved single-program performance. This is because there are only three real "dimensions" to processor performance increases beyond Moore's law: clock frequency, superscalar instruction issue, and multiprocessing. We have pushed the first two to their logical limits and must now embrace multiprocessing, even if it means that programmers will be forced to change to a parallel programming model to achieve the highest possible performance.

Conveniently, the transition from multiple-chip systems to chip multiprocessors greatly simplifies the problems traditionally associated with parallel programming. Previously it was necessary to minimize communication between independent threads to an extremely low level, because each communication could require hundreds or even thousands of processor cycles. Within any CMP with a shared on-chip cache memory, however, each communication event typically takes just a handful

of processor cycles. With latencies like these, communication delays have a much smaller impact on overall system performance. Programmers must still divide their work into parallel threads, but do not need to worry nearly as much about ensuring that these threads are highly independent, since communication is relatively cheap. This is not a complete panacea, however, because programmers must still structure their inter-thread synchronization correctly, or the program may generate incorrect results or deadlock, but at least the performance impact of communication delays is minimized.

Parallel threads can also be much smaller and still be effective—threads that are only hundreds or a few thousand cycles long can often be used to extract parallelism with these systems, instead of the millions of cycles long threads typically necessary with conventional parallel machines. Researchers have shown that parallelization of applications can be made even easier with several schemes involving the addition of *transactional* hardware to a CMP.[15,16,17,18,19] These systems add buffering logic that lets threads attempt to execute in parallel, and then dynamically determines whether they are actually parallel at runtime. If no inter-thread dependencies are detected at runtime, then the threads complete normally. If dependencies exist, then the buffers of some threads are cleared and those threads are restarted, dynamically serializing the threads in the process.

Such hardware, which is only practical on tightly coupled parallel machines such as CMPs, eliminates the need for programmers to determine whether threads are parallel as they parallelize their programs—they need only choose *potentially* parallel threads. Overall, the shift from conventional processors to CMPs should be less traumatic for programmers than the shift from conventional processors to multichip multiprocessors, because of the short CMP communication latencies and enhancements such as transactional memory, which should be commercially available within the next few years. As a result, this paradigm shift should be within the range of what is feasible for "typical" programmers, instead of being limited to graduate-level computer science topics.

HARDWARE ADVANTAGES

In addition to the software advantages now and in the future, CMPs have major advantages over conventional uniprocessors for hardware designers. CMPs require only a fairly modest engineering effort for each generation of processors. Each member of a family of processors just requires the stamping down of additional copies of the core processor and then making some modifications to relatively slow logic connecting the processors together to accommodate the additional processors in each generation—and *not* a complete redesign of the high-speed processor core logic. Moreover, the system board design typically needs only minor tweaks from generation to generation, since externally a CMP looks essentially the same from generation to generation, even as the number of processors within it increases.

The only real difference is that the board will need to deal with higher I/O bandwidth requirements as the CMPs scale. Over several silicon process generations, the savings in engineering costs can be significant, because it is relatively easy to stamp down a few more cores each time. Also, the same engineering effort can be amortized across a large family of related processors. Simply varying the numbers *and* clock frequencies of processors can allow essentially the same hardware to function at many different price/performance points.

AN INEVITABLE TRANSITION

As a result of these trends, we are at a point where chip multiprocessors are making significant inroads into the marketplace. Throughput computing is the first and most pressing area where CMPs are having an impact. This is because they can improve power/performance results right out of the box, without any software changes, thanks to the large numbers of independent threads that are available in these already multithreaded applications. In the near future, CMPs should also have an impact in the more common area of latency-critical computations. Although it is necessary to parallelize most latency-critical software into multiple parallel threads of execution to really take advantage of a chip multiprocessor, CMPs make this process easier than with conventional multiprocessors, because of their short interprocessor communication latencies.

Viewed another way, the transition to CMPs is inevitable because past efforts to speed up processor architectures with techniques that do not modify the basic von Neumann computing model, such as pipelining and superscalar issue, are encountering hard limits. As a result, the microprocessor industry is leading the way to multicore architectures; however, the full benefit of these architectures will not be harnessed until the software industry fully embraces parallel programming. The art of multiprocessor programming, currently mastered by only a small minority of programmers, is more complex than programming uniprocessor machines and requires an understanding of new computational principles, algorithms, and programming tools. Q

The Future of Microprocessors

REFERENCES

1. Moore, G. E. 1965. Cramming more components onto integrated circuits. *Electronics* (April): 114–117.
2. Hennessy, J. L., and Patterson, D. A. 2003. *Computer Architecture: A Quantitative Approach, 3rd Edition*, San Francisco, CA: Morgan Kaufmann Publishers.
3. Wall, D. W. 1993. *Limits of Instruction-Level Parallelism*, WRL Research Report 93/6, Digital Western Research Laboratory, Palo Alto, CA.
4. Barroso, L., Dean, J., and Hoezle, U. 2003. Web search for a planet: the architecture of the Google cluster. *IEEE Micro* 23 (2): 22–28.
5. Olukotun, K., Nayfeh, B. A., Hammond, L. Wilson, K. and Chang, K. 1996. The case for a single chip multiprocessor. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII):* 2–11.
6. Kapil, S. 2003. UltraSPARC Gemini: Dual CPU Processor. In *Hot Chips 15* (August), Stanford, CA; http://www.hotchips.org/archives/.
7. Maruyama, T. 2003. SPARC64 VI: Fujitsu's next generation processor. In *Microprocessor Forum* (October), San Jose, CA.
8. McNairy, C., and Bhatia, R. 2004. Montecito: the next product in the Itanium processor family. In *Hot Chips 16* (August), Stanford, CA; http://www.hotchips.org/archives/.
9. Moore, C. 2000. POWER4 system microarchitecture. In *Microprocessor Forum* (October), San Jose, CA.
10. Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B. 2000. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture* (June): 282–293.
11. Kongetira, P., Aingaran, K., and Olukotun, K. 2005. Niagara: a 32-way multithreaded SPARC processor. *IEEE Micro* 25 (2): 21–29.
12. Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., and Smith, B. 1990. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing* (June): 1–6.
13. Laudon, J., Gupta, A., and Horowitz, M. 1994. Interleaving: a multithreading technique targeting multiprocessors and workstations. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems:* 308–316.
14. Tullsen, D. M., Eggers, S. J., and Levy, H. M. 1995. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture* (June): 392–403.
15. Hammond, L., Carlstrom, B. D., Wong, V., Chen, M., Kozyrakis, C., and Olukotun, K. 2004. Transactional coherence and consistency: simplifying parallel hardware and software. *IEEE Micro* 24 (6): 92–103.
16. Hammond, L., Hubbert, B., Siu, M., Prabhu, M., Chen, M., and Olukotun, K. 2000. The Stanford Hydra CMP. *IEEE Micro* 20 (2): 71–84.
17. Krishnan, V., and Torrellas, J. 1999. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers* 48 (9): 866–880.
18. Sohi, G., Breach, S., and Vijaykumar, T. 1995. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture* (June): 414–425.
19. Steffan, J. G., and Mowry, T. 1998. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture* (February): 2–13.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

KUNLE OLUKOTUN is an associate professor of electrical engineering and computer science at Stanford University, where he led the Stanford Hydra single-chip multiprocessor research project, which pioneered multiple processors on a single silicon chip. He founded Afara Websystems to develop commercial server systems with chip multiprocessor technology. Afara was acquired by Sun Microsystems, and the Afara microprocessor technology is now called Niagara. Olukotun is involved in research in computer architecture, parallel programming environments, and scalable parallel systems.

LANCE HAMMOND is a postdoctoral fellow at Stanford University. As a Ph.D. student, Hammond was the lead architect and implementer of the Hydra chip multiprocessor. The goal of Hammond's recent work on transactional coherence and consistency is to make parallel programming accessible to the average programmer.