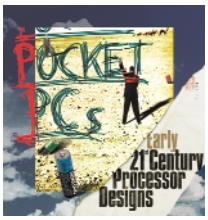
Instruction-Level Distributed Processing



Shifts in hardware and software technology will soon force designers to look at microarchitectures that process instruction streams in a highly distributed fashion.

James E. Smith
University of Wisconsin-Madison

For nearly 20 years, microarchitecture research has emphasized instruction-level parallelism, which improves performance by increasing the number of instructions per cycle. In striving for such parallelism, researchers have taken microarchitectures from pipelining to superscalar processing, pushing toward increasingly parallel processors. They have concentrated on wider instruction fetch, higher instruction issue rates, larger instruction windows, and increasing use of prediction and speculation. In short, researchers have exploited advances in chip technology to develop complex, hardware-intensive processors.

Benefiting from ever-increasing transistor budgets and taking a highly optimized, "big-compile" view of software, microarchitecture researchers made significant progress through the mid-1990s. More recently, though, researchers have seemingly reduced the problem to finding ways of consuming transistors, resulting in hardware-intensive, complex processors. The complexity is not just in critical path lengths and transistor counts: There is high intellectual complexity in the increasingly intricate schemes for squeezing performance out of second- and third-order effects.

Substantial shifts in hardware technology and software applications will lead to general-purpose microarchitectures composed of small, simple, interconnected processing elements, running at very high clock frequencies. A hidden layer of implementation-specific software—co-designed with the hardware—will help manage the distributed hardware resources to use power efficiently and to dynamically optimize executing threads based on observed instruction dependencies and communication.

In short, the current focus on instruction-level parallelism will shift to instruction-level distributed processing (ILDP), emphasizing interinstruction communication with dynamic optimization and a tight interaction between hardware and low-level software.

TECHNOLOGY SHIFTS

During the next two or three technology generations, processor architects will face several major challenges. On-chip wire delays are becoming critical, and power considerations will temper the availability of billions of transistors. Many important applications will be object-oriented and multithreaded and will consist of numerous separately compiled and dynamically linked parts.

Wire delays

Both short (local) and long (global) wires cause problems for microprocessor designers. With local wires, the problem is congestion: For many complex, dense structures, transistor size does not determine area requirements—wiring does. Global wire delays will not scale as well as transistor delays largely because shrinking wire sizes and limits on wire aspect ratios will cause resistance per unit length to increase at a faster rate than wiring distances shrink. Hierarchical wiring, with thicker long-distance wires will certainly help, but it is unlikely that the number of wiring levels will increase fast enough to stay ahead of the problem.[1] For example, in future 35-nanometer technology, the projected number of static RAM cells reachable in one clock cycle will be about half of that today.[2]

Of course, reachability is a problem with general logic, too. As logical structures become more complex and consume relatively more area, global delays will increase simply because structures are farther apart.

0018-9162/01/$10.00 © 2001 IEEE

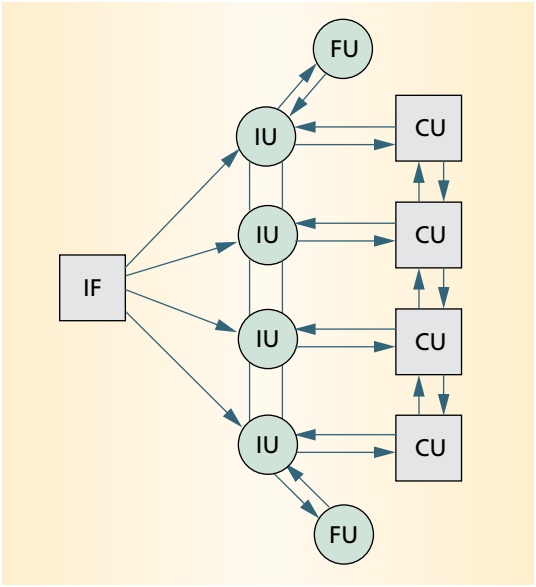


Figure 1. An example of instruction-level distributed processing, microarchitecture, consisting of an instruction fetch (IF) unit, integer units (IU), floating point units (FU), and cache units (CU). The units communicate via point-to-point interconnections that will likely consume one or more clock cycles each.

Put another way, simple logic will likely improve performance directly by reducing critical paths, but also indirectly by reducing area and overall wire lengths. This is not a new idea: All of Seymour Cray's designs benefited from this principle.

Power consumption

Dynamic power is proportional to the clock frequency, the transistor switching activity, and the supply voltage squared, so higher clock frequencies and transistor counts have made dynamic power an important design consideration today. Dependence on voltage level squared forces a countering trend toward lower voltages.

In the future, the power problem is likely to get much worse. To maintain high switching speeds with reduced voltage levels, developers must continue to lower transistor threshold voltages. Doing so makes transistors increasingly leaky: Current passes from source to drain even when the transistor is not switching. The resulting static power consumption will likely become dominant in the next two or three chip-technology generations.

There are few solutions to the static power problem. The design could selectively gate off the power supply to unused parts of the processor, but doing so is a relatively slow process that can create difficult-to-manage transient currents. Using fewer transistors or at least fewer leaky ones is about the only other option.

Software issues

General-purpose computing has shifted emphasis to integer-oriented commercial applications, where irregular data is common and data movement is often more important than computation. Highly structured data remains important for multimedia applications that tend to use integers and low-precision floating-point data. Often library-oriented, however, special processors or instruction-set extensions support these applications.

Microarchitecture researchers typically assume a high level of compiler optimization, sometimes using profile-driven feedback, but in practice many application binaries are not highly optimized. Further, global compile-time optimization is incompatible with object-oriented programming and dynamic linking. Consequently, microarchitects face the challenge of providing high performance for irregular, difficult-to-predict applications, many of which have not been compiler-optimized.

On-chip multithreading

Multithreading, which brings together microarchitecture and applications, has a lengthy tradition, primarily in very-high-end systems that have not enjoyed a broad software base. For example, multiprocessing became an integral component of large IBM mainframes and Cray supercomputers in the early 1980s. However, the widespread use of multithreading has been a chicken-and-egg problem that now appears nearly solved.

To continue the exponential complementary metal-oxide semiconductor (CMOS) performance curve, many chips now support multiple threads and others soon will. Multiple processors[3] or wide superscalar hardware capable of supporting simultaneous multithreading (SMT) provide this support.[4,5] Software that has many parallel, independent transactions—such as many Web servers—will take advantage of hardware-supported on-chip multithreading, as will many general-purpose applications.

INSTRUCTION-LEVEL DISTRIBUTED PROCESSING

The exploitation of technology advances and the accommodation of technology shifts have both fostered computer architecture innovation. For example, the cache memory innovation helped avoid tremendous slowdowns by adapting to the widening gap in performance between processor and dynamic RAM (DRAM) technologies.

At this point, shifts in both technology and applications are driving microarchitecture innovation. Innovators will strive to maintain long-term performance improvement in the face of increasing on-chip wire delays, increasing power consumption, and irregular throughput-oriented applications that are incompatible with big, static, highly optimized compilations.

ILDP microarchitecture

The ILDP microarchitecture paradigm deals effectively with technology and application trends. A processor following this paradigm consists of several

distributed functional units, each fairly simple and with a very high-frequency clock, as Figure 1 shows.

Relatively long wire delays imply processor microarchitectures that explicitly account for interinstruction and intrainstruction communication. As much as possible, the microarchitecture should localize communication to small units while organizing the overall structure for communication. Communication among units will be point-to-point with delays measured in clock cycles. A significant part of the microarchitecture design effort will involve partitioning the processor to accommodate these delays. To keep the transistor counts low and the clock frequency high, the microarchitecture core will keep low-level speculation to a relative minimum. Determinism is inherently simpler than prediction and recovery.

With high-intraprocessor-communication delays, the number of instructions the processor executes per cycle may level off or decrease, but developers can increase overall performance by running smaller distributed processing units at a much higher clock rate. The processor's structure and clock speeds have implications for global clock distribution. The processor will likely contain multiple clock domains, possibly asynchronous from one another.

Clock speed

Developers show growing awareness that clock speed holds the key to increased performance. For the past several years, they have pushed instruction-level parallelism, making some gains, but now, with results diminishing, they are turning to higher clock speeds. It's not a new approach: reduced-instruction-set computer (RISC) proponents have long debated the role of clock speed, Cray's approach used a very fast clock, and the Intel processor evolution follows a trend toward faster clocks by using successively deeper pipelines. The original Pentium procssor had a relatively short pipeline of five stages. This grew to 12 stages in the highly successful Pentium Pro/II/III series of processors, and the recent Pentium 4 uses 20 stages. In contrast to Intel processors, in which the pipelines have become extremely deep, the challenge will be to emphasize simplicity to keep pipelines shallow and efficient, even with a very fast clock.

Using a very fast clock in an ILDP computer tends to increase dynamic power consumption, but the processors' very modular, distributed nature permits better microarchitecture-level-power management. With replication of most units, clock gating can manage resource usage and dynamic power consumption. In particular, control logic can monitor computation resources and use or not use subsets of replicated units, depending on computation requirements.[6]

Regarding static power, a high-frequency clock uses fast leaky transistors more effectively. If transistors consume power even when idle, keeping them busy with active work is preferable—which a fast clock does. In addition, the replicated distributed units make implementing selective power gating easier. Also, some units may be just as effective if built with slower transistors, especially if multiple parallel copies of the unit are available to provide throughput.

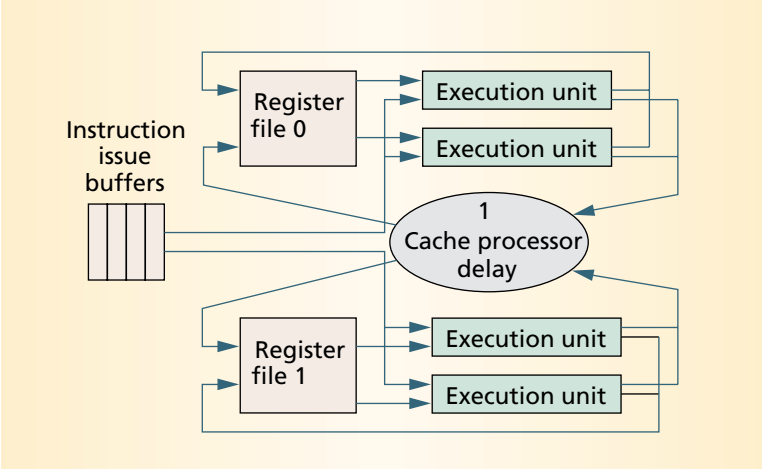


Figure 2. Alpha 21264 clustered microarchitecture. Instructions are steered to one of two processing clusters that have duplicated register files. Communicating the results produced in one cluster to the other cluster takes a full clock cycle.

For on-chip multithreading, ILDP provides the enticing possibility of combining a chip multiprocessor with simultaneous multithreading, particularly by partitioning computation elements among threads. With simple replicated units, different subsets of units can be assigned to individual threads. As in SMT, the units share the processor as a whole, but as in a multiprocessor, any unit services only one thread at a time. The challenge for developers will be managing the threads and resources of such a fine-grained distributed system.

Dependence-based microarchitecture

An important ILDP processor class, clustered dependence-based architecture,[7] organizes processing units into clusters and steers dependent instructions to the same cluster for processing, as in Compaq's Alpha 21264 microarchtitecture. This processor has two clusters and routes different instructions to each cluster at issue time, as Figure 2 shows. One cluster's results require an additional clock cycle to route to the other. The 21264's data dependencies tend to steer communicating instructions to the same cluster. A faster clock cycle compensates for an additional intercluster delay, leading to higher overall performance.

Generally, developers can divide a dependence-based design into several clusters; within a cluster there can be further division of instruction process-

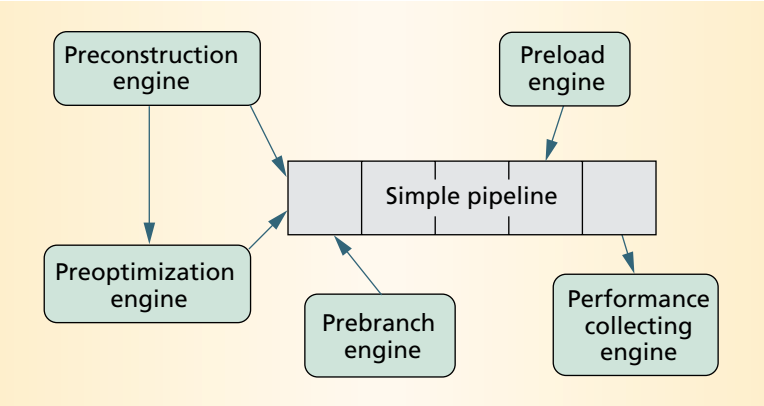


Figure 3. A heterogeneous instruction-level distributed processing chip architecture. A simple processor core is surrounded by helper engines that perform speculative tasks off the critical path and enhance overall performance.

ing, cache processing, integer processing, floating-point processing, and so on. The compiler can form dependent instructions or the processor can do it at various stages in the pipeline. If instruction decode logic can steer dependent instructions to the same cluster prior to issue, it simplifies instruction control logic within a cluster because instructions can be issued in first-in, first-out (FIFO) order. Meanwhile, instructions in different clusters can be issued out-of-order. Waiting until the issue stage, as in the 21264, slightly reduces interunit communication, but at the expense of more complex issue logic.

Heterogeneous ILDP

A heterogeneous ILDP model has outlying helper or service engines surrounding a simple core pipeline, as Figure 3 shows. Because these helper engines lie outside the critical processing path, their communication delays with respect to the main pipeline are noncritical; thus, they can even use slower transistors to reduce static power consumption.

Amir Roth and Guri Sohi have proposed a helper engine that preloads data by performing speculative pointer chasing.[8] Glenn Reinman's branch engine preexecutes instruction sequences to arrive at branch outcomes early.[9] An even more advanced helper engine is Yuan Chou and John Shen's instruction coprocessor that executes its own instruction stream to optimize instruction sequences.[10]

CO-DESIGNED VIRTUAL MACHINES

An ILDP computer clearly needs some type of higher-level management of the distributed instruction execution resources. This management includes routing instructions and data among the processor's units and allocating distributed resources for multiple simultaneous threads and power management.

One option uses compiler-level software to analyze instruction interactions. At compile time, the software determines or predicts interinstruction dependencies and communication and encodes this information into machine-level instructions. At runtime, the hardware uses this information to steer instruction control and data information through the distributed processing elements.

Another option uses hardware to determine important instruction interactions, with hardware tables collecting dynamic history information as programs execute. Later, instructions can acquire steering control and data information by accessing this history information.

Overall management of processor resources is another important consideration. For example, an ILDP will likely need resource load balancing for good performance—at both the instruction level and thread level. For power efficiency, gating off unused or unneeded resources requires usage analysis and coordination, especially if power gating is widespread across a chip.

The ILDP optimization and management functions can be done either by hardware or software. Although viable, software approaches based on conventional operating systems and compilers require recompilation and OS changes to fit each ILDP hardware platform—as well as intimate knowledge of the hardware implementation. The disadvantages of a hardware-based approach include the need for additional complex, power-consuming hardware and a limited scope for observing and collecting information related to the executing instruction stream.

A more radical solution uses currently evolving dynamic optimizing software and virtual machine technologies. A co-designed virtual machine combines hardware and software to implement a virtual architecture. This combination provides hardware implementors with a software layer in a hidden portion of DRAM main memory, allowing relatively complex dynamic program analysis and implementation-dependent optimization. The Transmeta Crusoe processor[11] and the IBM Daisy research project[12] use this base technology primarily to support whole-system binary translation to very long-instruction word (VLIW) sets. The IBM 390 processor uses a similar technology—millicode—to support execution of complex instructions.[13]

Figure 4 shows the virtual machine's overall structure. The physical main memory space is divided into conventional architected memory and hidden memory that a virtual machine monitor uses. The system places VMM code and data in hidden memory at boot time. The VMM manages ILDP resources and modifies instructions by adding information to guide instruction and data routing. The ILDP hardware can perform dynamic instruction scheduling, so the system does not require large-scale code optimization and software scheduling, unlike the VLIW-targeted implementations.

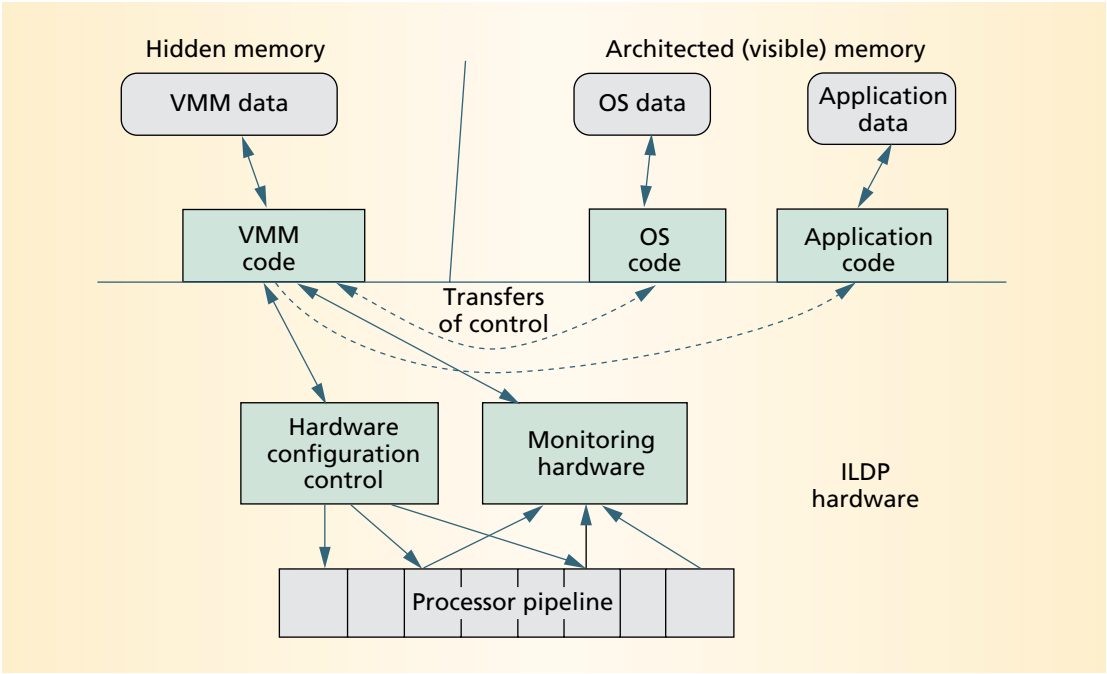


Figure 4. Supporting instruction-level distributed processing (ILDP) with a codesigned virtual machine. The virtual machine monitor (VMM) resides in hidden memory and uses monitoring and configuration hardware to manage ILDP resources. Control can be transferred to the VMM code at any time. Dashed lines show the control transfers.

At certain times during normal program execution, hardware can save the current program counter value and load it with a pointer to the VMM. Developers can design the hardware to invoke the VMM in this manner for selected instruction types, program traps, or system calls and returns, and a VMM-controlled timer can initiate periodic traps to the VMM so that monitoring hardware can read the information.

After it takes control, the VMM saves the architected state and uses the processor in the normal fashion, fetching instructions from hidden memory and accessing data from its space in hidden memory. It can use this information to maintain a history of resource usage and reconfigure hardware or modify instruction and data flow. When finished, the VMM returns control of the interrupted program, and hardware resumes normal program execution. Meanwhile, the system uses a conventional off-the-shelf operating system and application software.

INSTRUCTION SETS

Historically, designers have made major changes to instruction set architectures in discrete steps. After the original mainframes, industry standard architectures (ISAs) more or less stabilized until the early 1970s, when minicomputers appeared. These machines used relatively inexpensive packaging and interconnections, giving developers an opportunity to rethink ISAs. Based on lessons learned from relatively irregular mainframe instruction sets, developers aimed toward regularity and orthogonality. To incorporate these properties, they developed minicomputer ISAs—such as the PDP-11 and VAX-11 instruction sets—that supported relatively powerful, variable-length instructions.

As microprocessors evolved toward general-purpose computing platforms, developers again rethought ISAs, this time with hardware simplicity as a goal. The resulting RISC instruction sets allowed a fully pipelined processor implementation to fit on a single chip. Because transistor densities have increased, hardware can now dynamically translate older, more complex microprocessor ISAs into RISC-like operations.

In retrospect, it seems that each change in packaging and technology—mainframes to minicomputers to microprocessors—has initiated instruction-set innovation. Perhaps we should take another serious look at ISAs—this time motivated by on-chip communication delays, high-speed clocks, and advances in translation and virtual machine software.

We can and should optimize ISAs for ILDP, focusing on communication and dependence. To reduce delays, we should make sure the ISA easily expresses interinstruction communication in a natural way. Architects should optimize ISAs for very fast execution, with emphasis on small, fast memory structures, including caches and registers. In contrast, most recent ISAs, including RISC and VLIW sets, have emphasized computation and independence—for example, by grouping independent instructions in close proximity.

Although maintaining compatibility with legacy program binaries tends to inhibit the development of

Table 1. Basic instruction types for an accumulator-based ISA.

Instruction	Size (bytes)
R <- A	1
A <- R	1
A <- A op R	2
A <- A op imm	4
A <- M(R op imm)	4
M(R op imm) <- A	4
R <- M(A op imm)	4

new ISAs, virtual machine technology and binary translation enable new implementation-level ISAs. Here, the VMM can translate and cache binaries in hidden memory—basically the Transmeta Crusoe and IBM Daisy paradigm, but applied to ILDP rather than a VLIW set. In this case, translations can be kept relatively simple with hardware and software sharing the burden of instruction scheduling and data routing.

To make ILDP instruction set concepts less abstract, consider an ISA that incorporates a register file hierarchy by using

- a small fast register file for local communication within a cluster of processing elements, and
- a larger global register file for intercluster communication.

This ISA uses variable-length instructions to provide smaller instruction footprints and smaller caches. As an extreme example, consider an ISA with 64 general-purpose registers and a single accumulator for performing operations. All operations must involve the accumulator, so both dependent operations and local-value communication are explicitly apparent. Such an ISA needs only one general-purpose register field per instruction, so it can be quite compact, with instructions 1, 2, or 4 bytes in length. In the basic instruction types shown in Table 1, the first two instructions copy data to and from a register; accumulator A is the single accumulator and accumulator R is a general-purpose register. The next two instructions depict operations on data held in the accumulator and general register file. The last three instructions indicate loads and stores.

With an ISA of this type, dependent instructions naturally chain together via the accumulator and are contiguous in the instruction stream. With a clustered ILDP implementation, the instruction fetch/decode hardware can steer all instructions in a dependent chain to the same cluster, steering the next dependent chain to another cluster. If the hardware renames the accumulator within each cluster, the processor can exploit parallelism among dependence chains, with the general registers handling global communication. The instruction queues in each cluster simply issue instructions in FIFO order, and fast local data communication is through the accumulator.

The single-accumulator ISA implicitly specifies communication and dependence information, although whether it is better to create a new ISA or simply append hint bits containing similar information to an existing one remains unclear.

Processor design, as with most complex engineering problems, is an ongoing process of reinventing, borrowing, and adapting—with a little innovation thrown in from time to time. New applications and evolving hardware technology are constantly changing the mix of techniques that lead to optimal engineering solutions.

Performance through simplicity to achieve a fast clock rate is a reinvention of the Cray designs. But how should this be done with modern very high-density CMOS technologies and nonnumeric applications? Developers can borrow distributed systems methods and apply them at the processor level to solve load balance, resource allocation, and communication problems. But how can they effectively apply these methods when the processing units are as small as individual instructions? Virtual machine methods have been around in one form or another for at least 30 years and can now be adapted to form hidden "microoperating systems" for managing the small distributed systems that processors are becoming. But how should developers divide functions (and complexity) between hardware and VMM software to optimize performance or power saving? Will combining new instruction sets with dynamic translation provide significant benefits?

These and many other important and interesting research issues remain to be explored as the processor architecture evolves toward instruction-level distributed processing. ✸

Acknowledgments

I thank the National Science Foundation (grant CCR-9900610), IBM, Sun Microsystems, and Intel for supporting this work.

References

1. T.N. Theis, "The Future of Interconnection Technology," *IBM J. Research and Development*, vol. 44, no. 3, 2000, pp. 379-390.
2. V. Agarwal et al., "Clock Rate versus IPC: The End of the Road for Conventional Microprocessors," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* ACM Press, New York, 2000, pp. 248-259.

3. L. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Int'l Symp. Computer Architecture*, ACM Press, New York, 2000, pp. 282-293.
4. R. Eichemeyer et al., "Evaluation of Multithreaded Uniprocessors for Commercial Application Environments," *Proc. 23rd Ann. Int'l Symp. Computer Architecture*, IEEE Press, Piscataway, New Jersey, 1996, pp. 203-212.
5. K. Diefendorff, "Compaq Chooses SMT for Alpha," *Microprocessor Report*, 6 Dec. 1999, pp. 1, 6-11.
6. D.H. Albonesi, "The Inherent Energy Efficiency of Complexity-Adaptive Processors," *Proc. 1998 Power-Driven Microarchitecture Workshop*, IEEE Press, Piscataway, N.J., 1998, pp. 107-112.
7. S. Palacharla, N. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. 24th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1997, pp. 206-218.
8. A. Roth and G. Sohi, "Effective Jump-Pointer Prefetching for Linked Data Structures," *Proc. 26th Ann. Int'l Symp. Computer Architecture,* IEEE Press, Piscataway, N.J., 1999, pp. 111-121.
9. G. Reinman, T. Austin, and B. Calder, "A Scalable Front-End Architecture for Fast Instruction Delivery," *Proc. 26th Ann. Int'l Symp. Computer Architecture,* IEEE Press, Piscataway, N.J., 1999, pp. 234-245.
10. Y. Chou and J.P. Shen, "Instruction Path Coprocessors," *Proc. 27th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 2000, pp. 270-281.
11. A. Klaiber, "The Technology Behind Crusoe Processors," tech. brief, Transmeta, Santa Clara, Calif., 2000; http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf.
12. K. Ebcioglu and E.R. Altman, "DAISY: Dynamic Compilation for 100% Architecture Compatibility," *Proc. 24th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1997, pp. 26-37.
13. C.F. Webb and J.S. Liptay, "A High-Frequency Custom CMOS S/390 Microprocessor," *IBM J. Research and Development,* July 1997, pp. 463-474.

James E. Smith *is a professor in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His research interests include high-performance processors and virtual machine architectures. He is currently on sabbatical at the IBM T.J. Watson Research Center in Yorktown Heights, New York. Contact him at jes@ece.wisc.edu*