Theme Feature

Trace Processors: Moving to Fourth-Generation Microarchitectures

Trace processors rely on hierarchy, replication, and prediction to dramatically increase the execution speed of ordinary sequential programs. The authors describe some of the ways these processors will meet future technology demands.

James E. Smith
University of Wisconsin-Madison

Sriram Vajapeyam
Indian Institute of Science

Fundamentally new generations of microarchitectures have been occurring approximately every two decades since the 1940s. Each generation has been driven by advances in underlying hardware technologies, and by attempts to extract and realize higher degrees of instruction-level parallelism. Given this pattern and the continued push for higher performance, we are midway through the third generation and are currently laying the groundwork for the fourth.

Technology trends are clear. By the end of the next decade a single IC chip will contain several hundred million, if not a billion, transistors. The Semiconductor Industry Association's road map[1] projects processors with 350 million transistors in 2007 and with 800 million by 2010. These large numbers of transistors result from greatly reduced feature sizes and lead to higher wiring densities. Thus, a major challenge is to use these transistors effectively and to accommodate the dramatic shifts in design constraints that will result from these changes.

As the sidebar "Why Large Uniprocessors?" describes, there are primarily three ways to respond to this challenge: build a multiprocessor on chip, integrate more of the computer system on a chip, or build a large uniprocessor, which would realize the fourth generation of microarchitectures. We have chosen to explore building large uniprocessors, specifically *trace processors*. A trace processor can execute ordinary serial programs written in standard languages at much higher speeds than are currently possible. It replicates superscalar pipelines (characteristic of the current microarchitecture generation) to form a set of connected processing elements. To this is added a level of hierarchy for control and data. A high-level control unit partitions the instruction stream into segments, or *traces*. A specially organized cache holds traces, and the processor fetches and executes traces as a unit. Another important feature is the heavy use of prediction for both control and data, which increases the exploitable parallelism in ordinary programs.

Although we describe features of the trace processor's architecture, our goal in this article is to focus on trace processors as a vehicle for describing the requirements of a fourth-generation microarchitecture. In this we include the technology trends that drive those requirements and the underlying features to support the microarchitecture.

REQUIREMENTS OF A NEW GENERATION

Figure 1 diagrams the four generations of microarchitectures. The first generation (top), serial processors, began in the 1940s with the first electronic digital computers and ended in the early 1960s. Serial processors fetch and execute each instruction before going to the next. The second generation was distinguished by pipelining and similar methods for overlapping instruction execution. IBM Stretch was a precursor of this generation, and the CDC 6600 was probably the first to achieve commercial success. The 6600 was followed shortly by pipelined processors in IBM mainframes. Second-generation microarchitectures using pipelining were the norm for high-performance processing until the late 1980s.

The third and current generation is characterized by superscalar processors, which first appeared in commercially available processors in the late 1980s. Out-of-order instruction execution also became widespread, though it was pioneered by a second-generation machine, the IBM 360/91. Both superscalar and out-of-order execution processors are still widely used.

For higher performance, there will likely be a transition to a fourth generation during the next decade. As Figure 1 shows, high-performance processors of the next generation will be composed of multiple superscalar pipelines, with some higher level control that dispatches groups of instructions to the individual superscalar pipes. This organization will solve the communication scalability problem of very wide issue superscalar processors while retaining a high clock rate and exploiting higher instruction-level parallelism.

Technology drivers

Beyond simple transistor count, we see processor technology being driven by high wire delays, and chip design and test cost.

Wire delays. Wire delays will soon dominate gate delays.[2] Basically, short wire delays have historically gone down quadratically as wire lengths shrink; however, physical limitations in metal wiring will reduce this to a linear improvement. Long wires, where remote communication delays dominate, will see little or no overall improvement in delays.

To compensate for high relative wire delays, a design must be partitioned in a way that maximizes the local communication of data values. Designers can probably best accomplish this by dividing the microarchitecture into multiple processing elements, each no larger than today's superscalar processors. Coordinating these processing elements to act as a single, unified processor will require an additional level of microarchitecture hierarchy for both control (distribution of instructions) and data (for communicating values among the processing elements).

Chip design and test. If trends continue, the cost and time to design and test a chip will dominate time to market and overall manufacturing time and cost. A new chip technology is developed every two to three years. To be competitive, microprocessor companies must have a design completed and verified when a new chip technology is ready to go into production.

To keep design and test costs manageable, designers are likely to create subsystems similar in complexity to today's superscalar processors and to combine replicated subsystems into a full processor. RAM-based devices for enhancing performance apply replication at a lower level, for example, in caches and prediction tables.

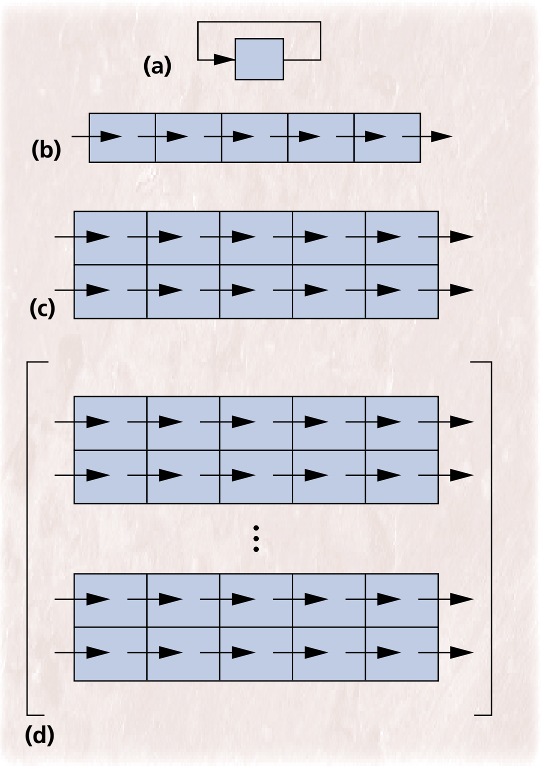

Figure 1. Four processor generations: (a) serial, (b) pipelined, (c) superscalar, and (d) fourth generation. The fourth-generation processor is likely to use hierarchy and replication to greatly increase the execution speed of sequential programs.

Instruction-level parallelism

As we mentioned, each new generation of microarchitectures strives to increase performance through

Why Large Uniprocessors?

There are primarily three approaches to dealing with very large numbers of transistors on a chip:

- Put a multiprocessor on a single chip.
- Integrate more of the computer on a chip: the processor, DRAM, and parts of the I/O subsystem.
- Build a large uniprocessor.

We favor the third approach, although the other two are important for some classes of applications and should continue to be explored. Both follow the well-established trend toward higher levels of system integration, and each has its pros and cons.

A tightly integrated single-chip multiprocessor, for example, will have low interprocessor communication delays. However, it will not solve the decades-old problem of how to develop parallel general-purpose applications.

Manufacturers are already announcing products that take the second approach, and for some applications—especially embedded applications—this approach clearly has significant cost advantages. For future high-performance systems, however, even a billion transistors will not be enough to hold the main memory required, and a significant number of the transistors will still have to be directed toward higher processor performance.

Thus, for high-performance computing, we favor building large uniprocessors. This also follows a well-established trend: that of continually improving processor organization to enhance ordinary single program performance—a trend that the computer industry has come to depend on. As argued in 1967 by Gene Amdahl (in his famous "Amdahl's Law" talk), a large, powerful uniprocessor provides speedup on virtually every program. Furthermore, multiple uniprocessors of the type we envision could eventually be put on a chip or be combined with DRAM.

support for a higher degree of instruction-level parallelism. A typical processor today, as shown in Figure 2, can issue four to six instructions per clock cycle. Parallelism in the instruction stream is exploited by performing different phases of instruction processing in an overlapped fashion (pipelining), and by issuing and executing multiple instructions in parallel (superscalar execution).

As designers seek higher degrees of instruction-level parallelism, the logic and wiring complexity for several parts of the superscalar pipeline increase, making delay paths longer. A recent study[3] looked at the delays of key pipelined components as superscalar processors achieve higher parallelism and as technology moves to smaller features. The study shows that for a given feature size, instruction-issue logic moderately increases delays for higher degrees of instruction-level parallelism. Reducing feature size decreases the delays, but they are likely to remain an important design consideration because of faster clock cycles. What really stands out are bypass delays: the complexity of bypass paths (long wires) grows quadratically with the number of functional units required to support higher instruction-level parallelism. This is significant because as we described earlier, wire delays are not likely to scale well for smaller features, so bypass delays will become critical.

WHY TRACE PROCESSORS?

These requirements for a fourth generation translate into three major design goals:

- *Make parallel instructions more visible.* The hardware can exploit parallelism only by simultaneously issuing instructions from the set of instructions visible to the issue hardware, the instruction window in Figure 2. A processor that attains a high degree of instruction-level parallelism must have a large instruction window, and the instructions in it must be useful—not on a mispredicted branch path, for example.
- *Dynamically partition hierarchical parallelism.* If processors are to be built around replication and hierarchy, control hardware must be able to allocate parts of a program to the replicated units in a way that enhances parallelism. The hardware should also be able to do this dynamically in a way that is transparent to the software. The processor should be able to quickly scan the program in large steps, allocating many instructions in each step to a replicated unit.
- *Incorporate speculation for both control and data.* Control flow speculation in the form of branch prediction is already an important feature of superscalar processors and will doubtless remain so. In addition, dealing with data dependences is likely to become an important part of future processor microarchitectures. Other forms of speculation are also likely to become important; for example, speculation that deals with memory addressing hazards.

We believe the trace processing paradigm meets all these goals and will become a widely used fourth-generation microarchitecture. Figure 3 depicts the major components of a trace processor. It incorporates parallel processing elements with hierarchical control and communication. Instruction fetch hardware unwinds programs into traces, each of which may have eight to 32 instructions as well as embedded, predicted conditional branches. The traces are placed in a trace cache,[4] and a trace fetch unit subsequently reads traces from the trace cache and parcels them out to the parallel processing elements. Thus, the trace becomes the basic execution unit throughout the

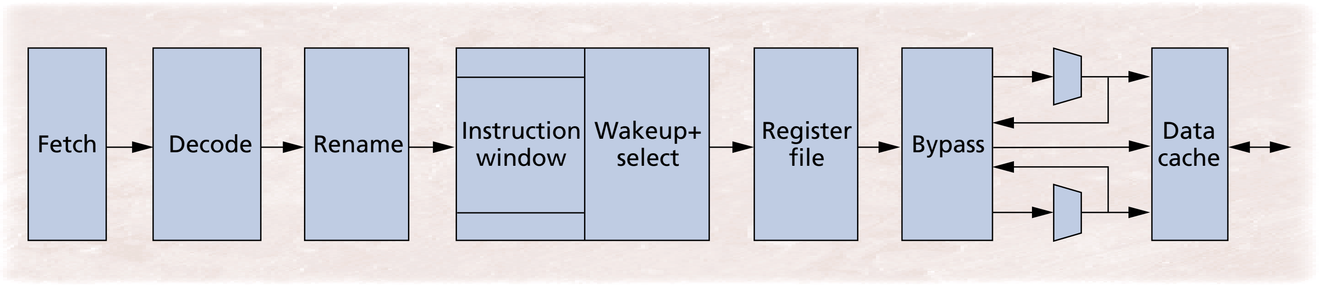

Figure 2. Instruction flow through a typical superscalar pipeline. The fetch unit reads multiple instructions every cycle from the instruction cache (not shown). The next pipe stage decodes instructions and renames their registers so that only true register data dependences remain. Dispatch logic routes renamed instructions into a set of buffers, collectively referred to as the instruction window, where they wait for their source operands and the appropriate functional unit to become available. Wakeup logic associated with the instruction window detects when an instruction's source operands are ready; select logic chooses from among the ready instructions and issues instructions to the appropriate functional units. Instructions fetch source operands from the register file, or they are bypassed from earlier instructions in the pipeline.

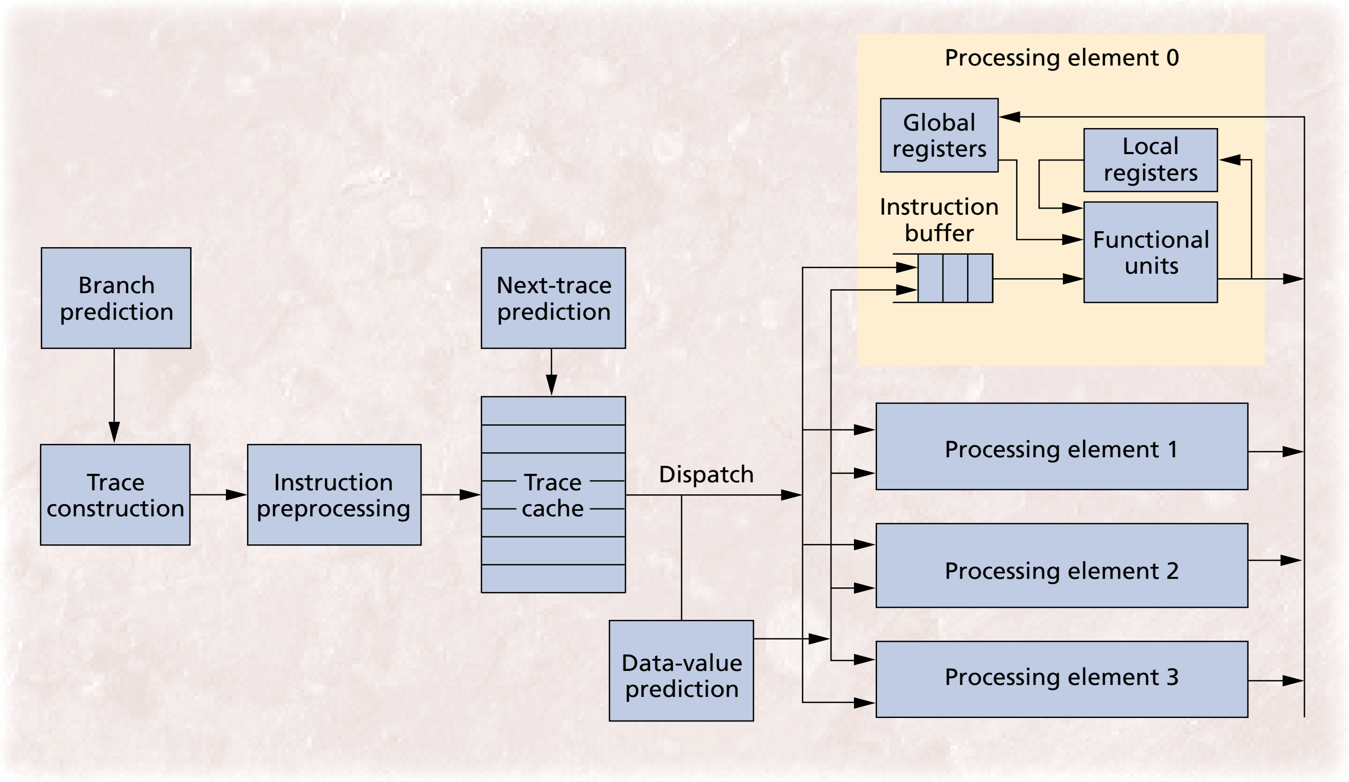

Figure 3. Architecture of the trace processor. If each processing element issues four instructions per cycle, the four-element system shown can achieve an aggregate performance of 16 instructions per cycle.

processor. By exploiting the characteristics of traces and frequently reusing them, we can build mechanisms that support a peak throughput rate of one trace per clock cycle and simultaneously maintain a fast clock cycle.

The trace processor is similar to the multiscalar paradigm developed by Guri Sohi, which he describes in the sidebar "Multiscalar: Another Fourth-Generation Processor." Both paradigms use multiple superscalar pipelines and a level of control hierarchy. The trace processor has several key differences, however. The most significant is its use of hardware-generated dynamic traces rather than compiler-generated static tasks.

Instruction preprocessing

Instruction preprocessing is an important step in lifting processing from individual instructions to traces. A trace cache miss causes a trace to be built through conventional instruction fetching with branch prediction. Blocks of instructions are preprocessed before being put in the trace cache, which greatly simplifies processing after they are fetched. Preprocessing can include capturing data dependence relationships, combining and reordering instructions, or determining instruction resource requirements[5]—all of which can be reused. To support precise interrupts, information about the original instruction order must also be saved with the trace.

Trace caches

A single entry in the trace cache holds an entire trace. The trace cache is indexed using the next-address field of the previously fetched trace combined with prediction information returned by the trace predictor. Thus, an entire trace consisting of multiple basic blocks is fetched in one clock cycle, without the need for multiple cache lookups; multiported caches; or complicated and time-consuming mask, alignment, and concatenation operations on multiple cache blocks. Such logic is moved off the critical path to the trace construction hardware.

Next-trace prediction

Because traces are the basic unit for fetching and execution, control flow prediction is moved up to the trace level. Next-trace prediction must be able to predict multiple branches per cycle, either explicitly or implicitly—we favor the implicit approach. Work with multiscalar machines[6] shows that traces can be predicted accurately on the basis of the immediately preceding trace sequence.

Instruction dispatch

During the dispatch phase, instructions move from the trace cache to the instruction buffers in the processing elements. Only intertrace dependence checking and register renaming are required. The dispatch

Multiscalar: Another Fourth-Generation Processor

Guri Sohi, University of Wisconsin-Madison

The multiscalar paradigm first appeared about 1990 as a paradigm for a circa 2000 processor. The name came from its structure: the computing engine is a collection of sequential (or scalar) processors that cooperate in executing a sequential program.[1]

A conventional superscalar processor sequences through a static program to create a single dynamic window of instructions, schedules instructions for execution from this window, and communicates instruction results to other waiting instructions. The multiscalar processor does all of this but in a more powerful, decentralized manner.

Interoperation communication can be carried out more efficiently if the total instruction window is broken into subwindows, with (frequent) intrawindow and (less frequent) interwindow communication. Likewise, instruction scheduling becomes more efficient if the overall schedule is treated as an ensemble of (several) smaller schedules, where the smaller schedule is the schedule in a subwindow.

This “splitting” is the reason for multiscalar’s earlier moniker: the Expandable Split Window paradigm. The remaining issue is how to improve the sequencing process. Here multiscalar uses multiple sequencers to sequence through a sequential program—each sequencer (speculatively) sequences through (a portion of) the program that results in a portion (subwindow) of the instruction window; collectively the multiple sequencers capture a portion of the dynamic instruction stream.

Figure A1 shows a static program represented as a task flow graph, in which each task is a collection of instructions, for example, part of a (large) basic block, a basic block, a collection of basic blocks, a loop iteration, an entire loop, and so on. A task sequencer (speculatively) sequences through the program a task at a time, assigning the task to a processing element (PE). The processing element unravels the task to determine the dynamic instructions to be executed, and executes them.

Figure A2 shows the dynamic instruction stream divided into task-sized steps. The four tasks A, B, D, and E are assigned to processing elements 0 through 3.

To execute an instruction, the processor must establish dependence relationships. The interesting case is dependences on instructions in (predecessor) tasks that are currently executing. For register data, a *create mask* is used. Bits in the create mask correspond to each of the logical registers; a bit is set to one if the register is potentially written by the task. The accumulation of the predecessor tasks’ create masks provides the necessary register dependence information—the tasks that generate these values will eventually send them to later tasks. For memory operations, the situation is more involved. When a processing element is ready to execute a load, the processing element does not even know if previous tasks have stores, let alone stores to a given memory location. Here multiscalar resorts to *data dependence speculation*—speculating that a load does not depend on instructions executing in predecessor tasks. An address resolution buffer also checks that the speculation was correct, squashing instructions if it is not. Thus the multiscalar paradigm has at least two forms of speculation: *control* speculation, which the task sequencer uses, and *data dependence* speculation, which each processor performs. It could also use other forms of speculation, such as data value speculation, to alleviate intertask dependences.

Information that allows a sequential program to run in a (speculatively) parallel fashion (such as task flow information and register masks) can be implemented in a variety of ways: it could reside within the instructions of the static program, or could be computed dynamically and reside off to the side to allow a higher degree of compatibility with existing instruction sets and binaries.

Reference

1. G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, “Multiscalar Processors,” *Proc. Int’l Symp. Computer Architecture*, ACM Press, New York, 1995, pp. 414-425.

***Guri Sohi** is a professor of computer science at the University of Wisconsin-Madison. Contact him at sohi@cs.wisc.edu.*

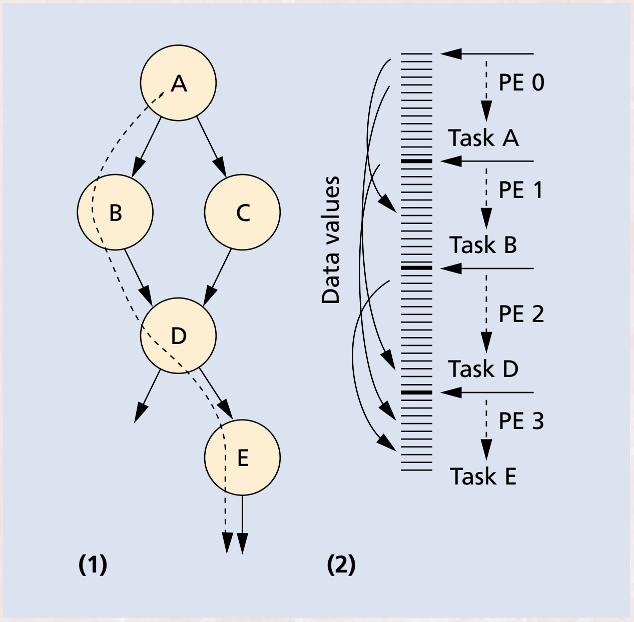

Figure A. Multiscalar task processing. The multiscalar paradigm divides a program, represented by (1) its task flow graph into (2) a dynamic instruction stream, sequential tasks executed in parallel by four processing elements (PE 0, 1, 2, and 3). Data values are for both memory and registers. The dashed arrow in (1) represents the execution path.

logic may also predict a trace's input data values; the trace's execution is based on data speculation. Data prediction at the trace level tends to decouple the traces and enhance instruction-level parallelism.

Hierarchical registers

Even though the trace processor uses a conventional set of logical registers, physical registers are divided into local and global sets.[7] The hierarchical organization of registers allows small register files with fast access times and fewer ports per file. The trace dispatcher remaps the trace's source and destination registers to the global registers without the need for intratrace dependence checking. The dispatcher maps local registers with reusable mappings based on the intratrace dependences detected during instruction preprocessing. Typically dispatch logic can remap a 16-instruction trace line using register rename logic as complex as that used by a conventional four-way superscalar processor.

Data value prediction

Successfully predicting a trace's input data values makes the trace data independent, allowing the trace to execute immediately and in parallel with any other trace. The development of data value prediction is just beginning. Initial work[8] focused on constant value predictions—many instructions produce the same value repeatedly. We are currently attempting to characterize the potential accuracy of data value prediction. Early results, not yet published, suggest that sophisticated methods can predict data values with up to 80 percent accuracy for fairly irregular integer programs such as the gcc compiler. Data value prediction is a good example of using RAM, in the form of prediction tables, to enhance processor performance.

Instruction issuing

Each conventional superscalar processing element issues a small number of instructions from its trace every clock cycle, thus distributing issue operations and avoiding a centralized control bottleneck. The expected parallelism within a single trace is suitable for execution in a modest superscalar unit. As multiple processing elements issue instructions in parallel, both intratrace and intertrace parallelism are exploited, which leads to large increases in overall instruction-level parallelism.

Result forwarding, register file access

Because the processing elements and register files are distributed, so is the communication of register data. The relatively simple bypass paths within a unit allow local result forwarding in a single cycle—typical of today's superscalar processors. Global paths are used for communicating global register results between processing units. The global bypass paths are likely to require multiple clock cycles. Bypass paths with this structure are already starting to appear in advanced superscalar processors.[9]

Memory systems

Memory systems for fourth-generation processors are an important topic unto themselves. Clearly, memory systems for the trace processor will have to provide very high bandwidth. Distributed, multiported caches will have to supply data to the processor's multiple processing elements. This cache system will probably have some characteristics in common with today's small multiprocessor systems. In particular, designers will have to maintain coherence among distributed caches. A large, interleaved cache system is also possible, although designers will have to deal with the additional latency in such systems.

Another very important issue is the parallel resolution of memory addressing hazards. Each processing element in the trace processor generates a stream of load and store requests to memory. Moreover, these address streams are generated speculatively and out-of-order. The hardware to sort out the address streams and make sure that all memory locations are accessed in correct order will have to be fairly sophisticated. The address resolution buffer is a proposed solution for multiscalar processors.[10] The ARB may also be useful for other fourth-generation processors. As initially proposed, it is a centralized device, separate from data caches. However, it is likely to evolve to distributed mechanisms that merge address resolution and data caching.

Because new generations of microarchitecture are so widely separated in time, it is difficult to look ahead to the next generation early in the lifetime of the current one. However, as a generation becomes mature and future hardware technologies are better defined, the next generation starts to become visible. We are at that stage now. The last time we were in a similar position was almost 20 years ago—when superscalar processors began to be discussed. Once again, it is time to lay the groundwork for many more years of high-performance processor development.

We believe that the next-generation microarchitecture will be able to execute 16 or more instructions per cycle while processing ordinary binary programs. But to achieve this level of parallelism without negatively affecting the clock cycle, the processor microarchitecture must rely heavily on replication and hierarchy. We have proposed one such architecture. Regardless of the details of the fourth-generation microarchitecture, many supporting microarchitectural technologies need to be developed—some of which will also be useful for

The trace processing paradigm will become a widely used fourth-generation architecture.

near-term implementations of superscalar processors. And whatever the outcome, the next decade of processor development, driven by the need for high performance and enabled by huge chip transistor budgets, promises to be an exciting time. ❖

Acknowledgments

We thank Guri Sohi for many enlightening and stimulating discussions on advanced high-ILP processors. We also thank all those—especially Eric Rotenberg, Quinn Jacobson, Tulika Mitra, Yiannakis Sazeides, Sridhar Gopal, Subbarao Palacharla, T. N. Vijaykumar, Scott Breach, and Andreas Moshovos—who have done initial research and given considerable thought to the direction of future-generation processors.

This work was supported in part by NSF grant MIP-9505853 and by the US Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order D346. The views and conclusions contained herein are those of the authors and do not necessarily represent the official policies or endorsements, either expressed or implied, of the US Army Intelligence Center and Fort Huachuca, or the US government.

References

1. Semiconductor Industry Assoc., *The National Technology Roadmap for Semiconductors*, San Jose, Calif., 1994.
2. N.C. Wilhelm, *Why Wire Delays Will No Longer Scale for VLSI Chips*, Sun Microsystems Lab. Report SMLI TR-95-44, Mountain View, Calif., 1995.
3. S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity Effective Superscalar Processors," *Proc. Int'l Symp. Computer Architecture*, ACM Press, New York, 1997, pp. 206-218.
4. E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache: A Low-Latency Approach to High-Bandwidth Instruction Fetch," *Proc. Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 24-34.
5. S.W. Melvin, M.C. Shebanow, and Y.N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proc. Workshop Microprogramming and Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1988, pp. 60-63.
6. Q. Jacobson et al., "Control Flow Speculation in Multiscalar Processors," *Proc. Third Ann. Symp. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 218-229.
7. S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences," *Proc. Int'l Symp. Computer Architecture*, ACM Press, New York, 1997, pp. 1-12.
8. M. Lipasti and J. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proc. Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 226-237.
9. L. Gwennap, "Digital 21264 Sets New Standard," *Microprocessor Report*, Oct. 28, 1996, pp. 11-16.
10. M. Franklin and G.S. Sohi, "ARB: A Hardware Mechanism for Dynamic Memory Disambiguation," *IEEE Trans. Computers*, Feb. 1996, pp. 552-571.

***James E. Smith** is a professor of electrical and computer engineering at the University of Wisconsin-Madison. His current research interests are new paradigms for exploiting instruction-level parallelism. Smith received a PhD in computer science from the University of Illinois. He is a member of the IEEE Computer Society and ACM.*

***Sriram Vajapeyam** is an assistant professor of computer science at the Indian Institute of Science. His research interests include processor architectures and memory subsystems. Vajapeyam received a BTech in electrical engineering from the Indian Institute of Technology, Madras, and a PhD in computer science from the University of Wisconsin-Madison.*

Contact Smith at ECE Dept., Univ. of Wisconsin-Madison, 1415 Johnson Dr., Madison, WI 53706; jes@ece.wisc; or Vajapeyam at the Indian Inst. of Science, Bangalore, India 560012; sriram@csa.iisc.ernet.in.