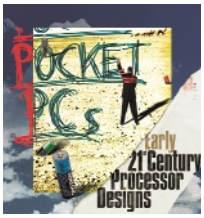
Speculative Multithreaded Processors



Speculation will overcome the limitations in dividing a single program into multiple threads that can execute on the multiple logical processing elements needed to enhance performance through parallelization.

Gurindar S. Sohi
Amir Roth
University of Wisconsin-Madison

Semiconductor technologies—along with innovative computer architectures—have provided the bricks and mortar for building phenomenal improvements in processing speed during the past decade, culminating ultimately in the hundreds of millions of transistors used to build increasingly fast on-chip devices. Innovations in computer microarchitecture and accompanying compilers have enabled us to make good use of these building materials to provide high-performance computing systems.

Typically, we decide how to use available semiconductor resources in two steps. First, we choose the desired functionality—the techniques for extracting and enhancing performance. In the implementation phase, we translate those techniques into structures and signals that we must then design, build, and verify. Although often described separately, in practice these two phases are tightly coupled.

During the 1990s, novel functionality played the dominant role in processor design. Given a reasonable limit on overall design size—for example, fewer than tens of millions of transistors—we could divide up the transistor budget simply by using high-level performance metrics. Doing so made verification relatively simple, and designs did not have to explicitly account for wire delays, which were not significant compared to logic delays.

In the future, implementation issues will likely dominate even basic functionality. We have begun to realize that scaling conventional superscalar designs increases complexity and cost with no guarantee that such designs will meet performance goals. Monolithic designs that use hundreds of millions of transistors will be very difficult to design, debug, and verify, and increasing wire delays will make intrachip communication and clock distribution costly. Consequently, some computer architects advocate a shift from high-performance to high-throughput processing, using distributed components that divide and conquer design process complexity and exploit communication locality to overcome wire delays. With this trend comes a renewed and increasing interest in multithreaded architectures. Such architectures can extract parallelism from a sequential program via thread-level speculation—be it control-driven or data-driven—giving them the flexibility to operate in both multiple-program, high-throughput and single-program, high-performance environments.

RATIONALE FOR SPECULATIVE MULTITHREADING

Fortunately, the twin goals of increasing single-program performance and decreasing implementation difficulty don't necessarily conflict. The motivation for using speculative multithreading comes from two directions: On the one hand, we are already witnessing the diminishing potential of current techniques to extract parallelism from single programs and thus increase their performance; on the other, technology trends suggest the onset of commercial processors that can simultaneously execute multiple independent threads.[1] Thus, we are almost compelled to find innovations that will enable multithreaded processors to support the parallel execution of a single program.

A speculative multithreaded processor consists logically of replicated processing elements that coopera-

0018-9162/01/$10.00 © 2001 IEEE

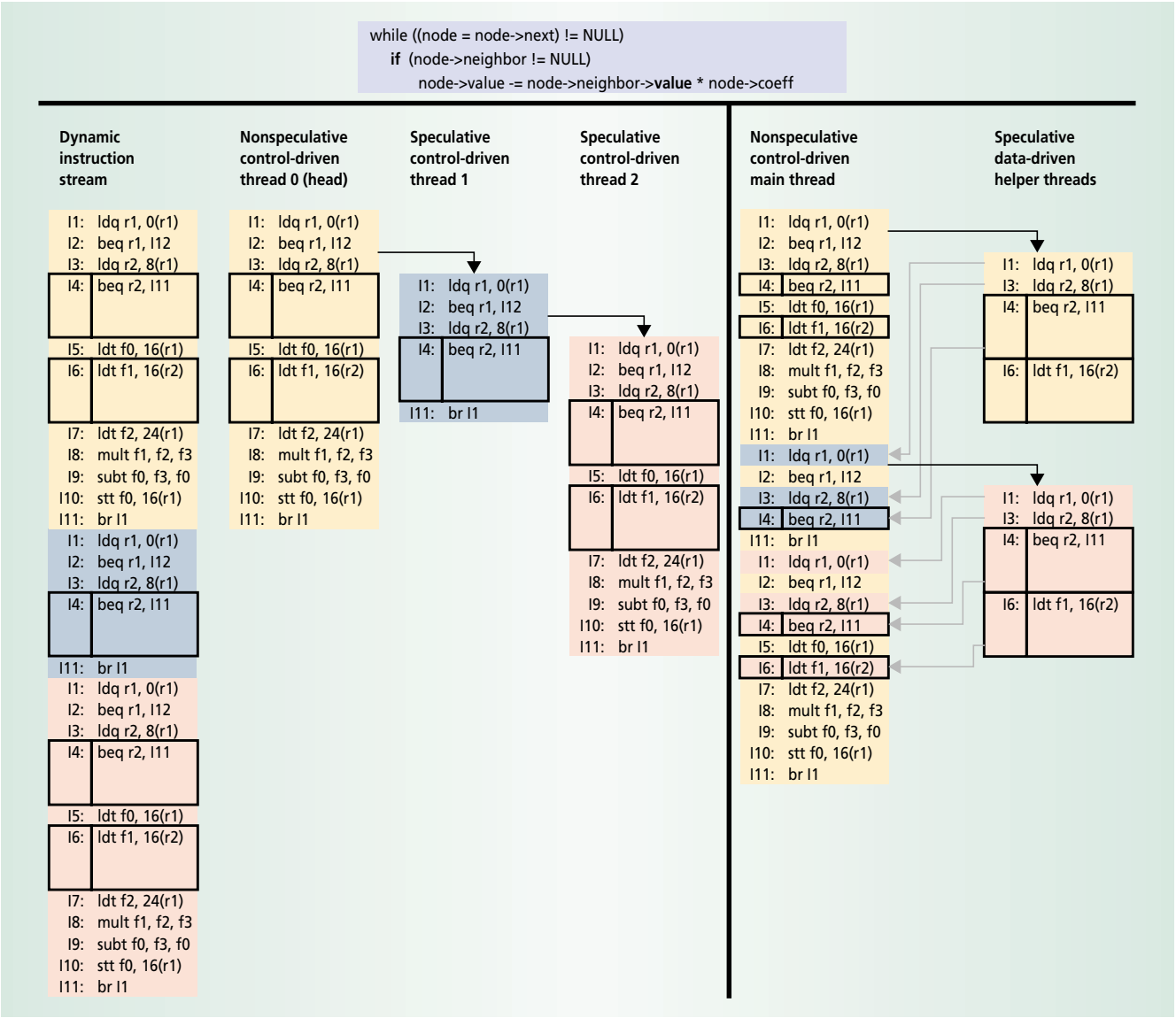


Figure 1. A dynamic instruction stream. A superscalar processor operates on the group of instructions at the frontier of the dynamic instruction stream as the processor gradually unrolls the stream.

tively perform the parallel execution of a conventional sequential program—also called a *program thread*—divided into chunks called speculative threads. Speculation is key: Without speculation, we can only divide programs conservatively into nonspeculative threads whose mutual independence and execution is guaranteed. Speculation enables much more aggressive divisions that can exploit threads whose independence and execution may not be guaranteed but are parallel, and likely to be executed, with high probability.

Current parallelism-extraction method limitations

Today, the superscalar model holds the incumbent position for achieving high single-program performance. Imperative programs—written in languages like Fortran, C, and Java—are defined by a static control flow. As Figure 1 shows, at runtime, the processor unrolls the static control flow to produce a dynamic instruction stream. A superscalar processor operates on the group of instructions at the frontier of the dynamic instruction stream as the processor gradually unrolls the stream. The processor repeatedly searches this dynamic instruction window for unexecuted, independent instructions and attempts to execute these instructions in parallel. To attain sustained high performance, each window should contain enough independent instructions to support effective instruction-level parallelism (ILP).

Unfortunately, imperative programming conventions make consistently high ILP rare. Programmers tend to group dependent statements together into static structures—helping them reason about their programs but limiting the independent work available in any given dynamic instruction window. Optimizing compilers attempt to improve window ILP by trans-

parently reordering instructions from nearby program regions, but even sophisticated compiler scheduling is fundamentally limited by the compiler's inability to perfectly determine the programmer's intent and its commitment to preserve the program's high-level structure and semantics.

Given the amount of parallel work being done, we could conceivably build a superscalar processor with an instruction window large enough to simultaneously contain code from different program regions—specifically, different functions or loop iterations. However, over and above the many engineering obstacles, maintaining a large, contiguous window full of useful instructions poses a fundamental problem. Specifically, the decreasing accuracy of a series of branch predictions leads to an exponentially decreasing likelihood that instructions at the tail of the window will be useful.

Overcoming this problem requires a model that lets parallelism from different program regions be exploited in a reasonably independent—that is, noncontiguous and nonserial—manner. The speculative multithreading model considers each program region to be a speculative thread or small program. By executing multiple speculative threads in parallel, high degrees of concurrency can be achieved in an aggregate fashion, especially if each thread is mostly sequential. The model subsequently merges the threads to recreate the original program. Speculative multithreading lets us fashion a large instruction window

Piranha: Exploiting Single-Chip Multiprocessing

Luiz André Barroso, Kourosh Gharachorloo, Tom Heynemann, Dan Joyce, David Lowell, Harland Maxwell, Joel McCormack, Ravishankar Mosur, Jeff Sprouse, Robert Stets, and Scott Smith
Compaq Computer Corp.

Today's microprocessor industry struggles with escalating development and design costs, which arise from exceedingly complex processors that push the limits of instruction-level parallelism. Meanwhile, such designs yield diminishing returns and are ill-suited for commercial applications such as database and Web workloads, which constitute the high-performance servers' most important market. These server applications typically suffer from large memory stall times, exhibit little instruction-level parallelism, and have no use for high-performance floating-point or multimedia functionality.

Fortunately, increasing chip densities provide architects with many options for tackling design complexities while addressing commercial applications' needs. Integrating all system-level components onto the processor die—as the upcoming Alpha 21364 does—enables a more efficient memory hierarchy without further increasing design complexity. Beyond that, exploiting the abundance of thread-level parallelism in commercial workloads through simultaneous multithreading or chip multiprocessing seems promising. The chip multiprocessing approach is particularly useful for addressing design complexity because it enables using simpler cores.

The Piranha project's[1] primary goals are to

- build a system that achieves superior performance on commercial workloads, and
- effectively address design cost and complexity issues.

Our research prototype aggressively exploits chip multiprocessing by integrating eight simple Alpha processor cores along with a two-level cache hierarchy, memory controllers, coherence protocol engines, and an interconnect router onto a single chip. Combining simple, single-issue in-order processor cores with an industry-standard ASIC design methodology should let us complete our design with a shorter schedule and smaller team and budget than a commercial microprocessor requires.

Although each Piranha processor core is substantially slower than a conventional next-generation processor because of its simpler design and the constraints of an ASIC process, integrating eight cores onto a single chip provides Piranha with a two-fold to threefold performance margin on important commercial workloads. This advantage can approach a factor of five using full-custom instead of ASIC logic.

Most of Piranha's architectural innovations lie in the memory and interconnect subsystems, including a shared eight-way banked, eight-way associative noninclusive L2 cache; highly specialized microprogrammed coherence protocol engines; and an aggressive two-level coherence protocol. Piranha is particularly efficient for applications with little instruction-level parallelism because it can exploit thread-level parallelism to issue multiple independent misses and better utilize its aggressive memory. In addition, significant constructive cache interference among threads for applications such as databases lets a relatively small, 1-Mbyte L2 cache effectively handle the eight processor cores.

While the exceedingly complex general-purpose architecture of most current processors is not optimal for any given application domain, Piranha's focused design targets an important market segment at the possible expense of other workloads, resulting in superior performance and improved time to market.

Reference

1. L.A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 2000), ACM Press, New York, June 2000, pp. 282-293.

The authors are affiliated with Compaq Computer Corp. Contact ***Luiz André Barroso*** *at Luiz.Barroso@compaq.com.*

out of an ensemble of smaller instruction windows, thereby facilitating implementation. In addition, proper thread division can logically isolate branches in one thread from those in another,[2] relieving the fundamental problem of diminishing instruction utility.

Multithreaded architectures

Multithreaded processors that support concurrent execution of multiple threads on a single chip look to dominate the next decade, with two models currently being explored. Simultaneous multithreading (SMT) uses a monolithic design with most resources shared among the threads.[1,3] Chip multiprocessing (CMP) proposes a distributed design that uses a collection of independent processing elements with less resource sharing.[4]

SMT strives to provide low-cost multithreading support atop conventional ILP superscalar processors, whereas CMP offers design simplicity and replication arguments. Both models target independent threads and use multithreading to improve throughput. If these threads are derived from a single program, the increased throughput also results in higher single-program performance. The "Piranha: Exploiting Single-Chip Multiprocessing" sidebar outlines a CMP implementation, while the "Cray MTA: Multithreading for Latency Tolerance" sidebar describes a multithreaded processor implementation.

As technology advances, the distinction between the SMT and CMP microarchitectures will likely blur. Regardless of the specific implementation, multithreaded processors will logically appear to be collections of processing elements. Whether we can exploit this organization to improve both the throughput and execution time of a single program remains uncertain. With thread-level speculation, the logical processors can execute conventional parallel threads as well as single programs divided into speculative threads.

DIVIDING PROGRAMS INTO MULTIPLE THREADS

There are several ways to divide programs into threads. We categorize these divisions as control-driven or data-driven, depending on whether threads divide primarily along control-flow or dataflow boundaries. We can further subcategorize these divi-

Cray MTA: Multithreading for Latency Tolerance

Burton Smith, Cray

In the uniform shared-memory programming model, computer system performance does not depend significantly on data placement in memory. The Cray MTA implements this model using a very high bandwidth interconnection network and the parallelism generated by fine-grained multithreading to tolerate memory latency.

The MTA system accommodates up to 256 custom multithreaded processors. Instead of data caches, each processor switches context every cycle among as many as 128 instruction streams, or hardware threads, choosing on each cycle only from those streams ready to execute their next instruction. The processors thereby tolerate up to 128 cycles of memory latency while achieving high utilization. Further, each stream can issue as many as eight memory references without waiting for earlier ones to finish, augmenting the processors' memory latency tolerance to a maximum of 1,024 cycles.

The instructions are 64 bits wide and can contain a memory reference operation, an arithmetic or logical operation, and a branch or simple arithmetic or logical operation. By issuing an instruction on nearly every cycle, each processor achieves a sustainable two operations per cycle, irrespective of data placement in memory.

The MTA's thread-based programming model permits mixing implicit and explicit parallelism. The virtual machine has an unbounded number of processors with uniform access to all memory locations. The programmer can specify an unbounded number of threads that interact via shared data structures. The runtime system acquires physical resources from the operating system and uses these resources to implement the virtual machine.

The runtime environment normally multiplexes hardware streams among an unbounded number of threads. If the runtime environment encounters insufficient parallelism to fully occupy its processor resources, it surrenders some protection domains to the operating system for allocation to other tasks. Conversely, the runtime demands additional processor resources as parallelism increases.

In addition to providing a high level of single-thread optimization, MTA compilers perform automatic parallelization of Fortran, C, and C++ source code. The compilers restructure loop nests to enhance parallelism. They perform whole-program analysis and optimization, including in-line expansion and parallelization of loops containing function calls and input-output operations. Linear recurrences are automatically parallelized, as are reductions even in the presence of unknown dependences—as in histogramming, for example.

The programmer can insert pragmas and directives to help the compiler perform automatic parallelization. The parallelism that the compiler discovers and exploits supplements whatever the user explicitly generates. Inner-loop parallelism is easy for the compilers to discover and manage, whereas outer-loop parallelism may be evident only to the programmer.

The tremendous software support that the MTA provides for parallel programming depends crucially on its uniform shared-memory architecture, which multithreading alone makes possible.

***Burton Smith** is chief scientist at Cray in Seattle. Contact him at burton@cray.com.*

In speculative, control-driven multithreading, memory need not be explicitly synchronized at all.

sions as nonspeculative or speculative. From the processor's point of view, nonspeculative threads are completely independent, with any dependence being explicitly enforced using architectural synchronization constructs. Speculative threads, on the other hand, need not be perfectly independent or synchronized, with developers leaving it up to the hardware to detect and recover from violations of the independence assumptions.

Threads obtained from program division are expected to execute on different logical processing units. To achieve concurrency, proximal threads—threads that will simultaneously coexist in the machine—must be highly data-independent. If we achieve such data-independence, concurrency, and, hence, performance, can scale almost linearly with the number of threads even for small per-thread window sizes. Efficiency remains constant as bandwidth and, hopefully, performance increase. We believe that speculation can allow data-independence criteria to be achieved more easily, giving speculative solutions distinct performance advantages over their nonspeculative counterparts.

Control-driven threads

Multithreading seeks to divide programs into data-independent parallel threads. For imperative programs, the most natural division occurs along control-flow boundaries because of the boundaries' control-driven architectural semantics: Instructions are totally ordered, and architectural state is precisely defined only at instruction boundaries, an approach that provides explicit control flow and implicit data flow. Control-driven multithreading divides the dynamic instruction stream into contiguous segments that the system can subsequently "sew" together end-to-end to reconstruct the sequential execution. Control-driven multithreading presents the challenge of finding division points that minimize interthread data dependencies.

Control-driven multithreading differs from parallel programming. Whereas parallel programs execute multiple concurrent control-driven threads, these threads exchange data in arbitrary ways and their semantics rarely match the semantics of individual threads run in series. In contrast, control-driven multithreading lets us impose parallel execution on what is essentially a sequential program. The data flows between control-driven threads in one direction only—from sequentially older threads to younger ones.

Nonspeculative control-driven threads. Without support for detecting and recovering from data-dependence violations or for aborting unnecessary threads and discarding their effects, nonspeculative control-driven multithreading requires strict guarantees about thread execution certainty and data integrity. Because thread execution cannot be undone, execution-certainty requirements dictate that we only fork nonspeculative control-driven threads if we know their execution is needed. To maximize concurrency, we usually achieve execution certainty by forking a thread at a previous control-equivalent point—for example, forking a loop at the beginning of the previous iteration.

Data integrity requires that access to thread-shared data occurs or appears to occur in sequential order. When we speak of data integrity, we refer mainly to memory integrity. Support for direct interthread register communication is typically not available, but, when it is, we assume that appropriate synchronization is provided. In contrast, interthread memory communication is naturally available, meaning that access to any memory location potentially shared with other threads must be explicitly synchronized. Data sharing and synchronization should be kept to a minimum in these cases to allow for adequate concurrency among threads.

With such strict safety requirements, dividing a program into nonspeculative threads has traditionally fallen to the programmer and compiler. The programmer has the deepest knowledge of the algorithm's parallel dimensions, as well as the potential for data sharing among different divisions. However, the tedium of manual thread division often leads to errors. Because debugged compilers don't make errors, computer scientists have expended considerable effort to have compilers automate program multithreading and parallelism, but have had success only in very limited domains.

Speculative control-driven threads. Nonspeculative control-driven multithreading suffers from two major problems. First, execution-certainty requirements limit thread division to control-independent program points, which may not satisfy the primary data-independence criteria. Second, even when proximal threads are data-independent, if we cannot prove this independence, conservative synchronization must be used to guard against the unlikely but remotely possible case of reordered accesses. Wherever developers apply synchronization needlessly, they unnecessarily lose concurrency and performance.

Speculation can alleviate these problems. In speculative, control-driven multithreading, memory need not be explicitly synchronized at all. The correct total order of memory operations can be reconstructed from the explicit or implied order of threads. This ordering can be used as the basis for hardware support to detect and potentially recover from interthread, memory-ordering violations.[5,6] With such support, access to thread-shared data can proceed optimistically, with penalties incurred only in those cases where proximal threads actually share data and the accesses occur in

nonsequential order. Further, since ordering-violation scenarios are typically predictable, slight modifications to the basic mechanism let it learn to recognize these scenarios early and artificially synchronize the offending store-and-load pairs.[6]

Execution-certainty constraints can be lifted using similar mechanisms. We can recover from interthread memory-ordering violations by using hardware that buffers or undoes changes to the architected thread state. Such hardware support allows threads to be spawned at points where their final usefulness cannot be absolutely guaranteed, but where they show a high likelihood of usefulness and more favorable data-independence characteristics.

Speculative control-driven multithreading has been the subject of academic research since the 1990s and is slowly finding its way into commercial products. Sun's MAJC architecture[7] supports such threads via its space time computing model. A prototype chip from NEC—Merlot—uses speculative control-driven multithreading to parallelize the execution of code that can't be parallelized by other known means.[8] We expect that more processors will make use of speculative control-driven threads in the coming decade, as this technology moves from the research phase into commercial implementations.

Data-driven threads

Where control-driven multithreading divides programs along control-flow boundaries, data-driven multithreading uses dataflow boundaries as the major division criterion. Such a division naturally achieves the desired interthread data-independence and resulting parallelism.[9] Data-driven threads provide almost ideal performance and efficiency optimization. In its pure form, data-driven multithreading occurs at the granularity of a single instruction.[9] Data-driven sequencing—or fetch—of an instruction is triggered by the availability of one of its input operands. Instructions enter the machine as soon as they can execute, but no sooner. This arrangement maximizes the amount of work that can overlap long latency instructions, while not wasting resources on instructions unready to use them.

Options besides instruction-level, data-driven sequencing exist. Data-driven sequencing can also be used on a thread granularity with conventional, control-driven sequencing at the instruction level.[10] In this organization, instructions from one or several related computations are packed into totally ordered threads that implicitly specify dataflow relationships. Individual threads, assigned to processing elements, sequence and execute in a control-driven manner.

However, the dataflow relationships between threads are represented explicitly, and thread creation is triggered in a data-driven manner—by the availability of its data inputs from the outcome of a previous thread. The data-driven threads we expect to see in future processors will likely take this form.

Nonspeculative data-driven threads. Imperative languages cannot implement nonspeculative data-driven multithreading easily. The main barrier arises from an imperative program's inability to specify a priori an explicit dataflow program representation because dataflow information only applies to a few well-defined boundaries. A data-forwarding error—either of omission or false commission—changes the program's meaning. The automatic conversion of imperative code to dataflow-explicit form has been the subject of some research, but data-driven program representations can generally be constructed only for code written in functional, data-driven languages.

Speculative data-driven threads. Nonspeculative data-driven multithreading suffers from two major problems. First, programs generally cannot be divided into data-driven threads. Second, even in cases where a division is possible, the resulting representation breaks the sequential semantics created by the programmer and destroys the correlation between the executing program and the source code from which it was derived. Sequential semantics, or at least its appearance, is very important for program development, debugging, and the interaction with non-data-driven system components and tasks. The loss of sequential semantics causes more serious consequences than simply disturbing the programmer.

Again, speculation might solve these problems if we shift our approach to multithreading. We begin with the idea that a sequential semantics requires the presence of a *main* or *architectural* thread that executes the program in full. Such a main thread means that data-driven threads can only perform ancillary work and that their presence will inevitably exact some overhead on the total system. However, the main-thread concept also means that dividing a program into disjoint threads is not strictly necessary and that speculative data-driven threads can concentrate on performance-enhancing tasks without correctness obligations. It is likely that data-driven threads will play the role of *helper* threads that run ahead of a main thread and absorb microarchitectural latencies on its behalf.

We believe that programs inherently contain sufficient levels of ILP, obscured by long-latency microarchitectural events like cache misses and branch mispredictions. High single-thread performance can be achieved if these latencies, which will likely become relatively longer, can somehow be removed or hidden from the main thread. This task can be accomplished by augmenting the program with data-driven helper

We expect that more processors will make use of speculative control-driven threads in the coming decade.

We expect that thread selection will be implemented in software and conveyed to hardware, but in an advisory form.

threads that pre-execute the computations of problem instructions before they cause stalls in the main program thread.

The helper model copies selected computations from the program and packs them into data-driven threads.[11,12] While the program is still executed as a single control-driven thread, data-driven threads spawn at certain points in the main program that compute some future instruction. When the main program thread catches up to the data-driven thread, it has the option of using the result or repeating the computation, albeit with a reduced latency.[12]

Speculation intertwines with the reduced helper status of data-driven threads. The main thread's ultimate responsibility for the architectural interface immediately relieves data-driven threads from any correctness obligations. Without these obligations, data-driven threads can be constructed using available dataflow information. In addition, data-driven threads need not comprise a complete partitioning of the program—their use can be reserved for only those situations that most need their parallelism-enhancing characteristics.

SYSTEM ARCHITECTURE

Future support for speculative threads depends on the discovery of acceptable solutions to several problems, ranging from low-level thread-implementation to high-level thread-usage strategies.

The most important decision involves the division of labor between the programmer, compiler, operating system, and processor. The processor will execute the threads, but what entity should be responsible for other thread-related tasks such as thread selection, spawning, scheduling, resource allocation, and communication? Placing all responsibility on the processor presents an attractive option. Given that future processors will have nearly a billion transistors, a few million could be dedicated easily to multithreading-specific management tasks.

A processor-only implementation has no forward- or backward-compatibility problems; it preserves the current system interface while enhancing the performance of legacy software. This implementation has drawbacks, however, including added design complexity and the mandated rigidity and simplicity of thread-selection and management algorithms.

Because thread selection presents an important and delicate problem, we could assign this function to software or even the programmer, thereby probably producing much better thread divisions. However, any path in which multithreading information flows from or through software to the hardware requires a change in the software-hardware interface. Such changes typically meet with resistance, especially if they have architectural semantics that must be implemented. Successful approaches will likely be those that perturb an existing architecture only slightly, and preferably not at all.

We expect that thread selection will be implemented in software and conveyed to hardware, but in an advisory form. The prefetch instructions found in recent architectures provide one example of advisory information. In this model, hardware can act upon the information fully, partially, or selectively, or even ignore it altogether, all without impacting correctness. The processor retains the option of enhancing or refining this information dynamically as well. Restricting speculative thread information to an advisory role relieves the architect of many functionality guarantees that would hamper future-generation implementations.

We must develop several technologies before speculative multithreading becomes commonplace in mainstream processors. These technologies include means for conveying thread information from software to hardware, algorithms for thread selection and management, and hardware and software to support the simultaneous execution of a collection of speculative and nonspeculative threads. Our research group is investigating several aspects of speculative multithreaded processors. This work includes identifying problem scenarios that could benefit from multithreading, identifying and selecting appropriate threads, determining methods of conveying thread information from software to hardware, and investigating the microarchitectural aspects of simultaneously executing an ensemble of speculative and nonspeculative threads in a profitable manner. The main challenge, which encompasses all aspects of this work, involves finding appropriate threads for relevant problem situations so that an adequate benefit can be achieved when they are executed on an underlying microarchitecture.

Acknowledgments

This work was supported in part by National Science Foundation grants MIP-9505853, CCR-9900584 and 0071924, donations from Intel and Sun Microsystems, the University of Wisconsin Graduate School, and by an Intel Foundation Graduate Fellowship.

References

1. J. Emer, "Simultaneous Multithreading: Multiplying Alpha's Performance," presentation, Microprocessor Forum, Oct. 1999.
2. G.S. Sohi, S. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Int'l Symp. Computer Architecture,* ACM Press, New York, 1991, pp. 414-425.
3. D.M. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multi-

threading Processor," *Proc. 23rd Int'l Symp. Computer Architecture,* ACM Press, New York, 1996, pp. 191-202.
4. L. Hammond, B.A. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *Computer*, Sept. 1997, pp. 79-85.
5. M. Franklin and G.S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Trans. Computers,* May 1996, pp. 552-571.
6. A. Moshovos et al., "Dynamic Speculation and Synchronization of Data Dependence," *Proc. 24th Int'l Symp. Computer Architecture,* ACM Press, New York, 1997, pp. 181-193.
7. M. Tremblay, "MAJC: Architecture: A Synthesis of Parallelism and Scalability, *IEEE Micro,* Nov./Dec., 2000, pp. 12-25.
8. N. Nishi et al., "A 1-GIPS 1-W Single-Chip Tightly Coupled Four-Way Multiprocessor with Architecture Support for Multiple Control-Flow Execution," *Proc. 47th Int'l IEEE Solid-State Circuits Conf.,* IEEE Press, Piscataway, N.J., 2000, pp. 418-475.
9. Arvind and R.S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Trans. Computers,* Mar. 1990, pp. 300-318.
10. R.A. Iannucci, "Toward a Dataflow/von Neumann Hybrid Architecture," *Proc. 15th Int'l Symp. Computer Architecture,* ACM Press, New York, 1988, pp. 131-140.
11. R.S. Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," *Proc. 26th Int'l Symp. Computer Architecture,* ACM Press, New York, 1999, pp. 186-195.
12. A. Roth and G.S. Sohi, "Speculative Data-Driven Multithreading," *Proc. 7th Int'l Symp. High-Performance Computer Architecture* (HPCA-7), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 37-48.

Gurindar S. Sohi *is a professor in the Computer Sciences and Electrical and Computer Engineering Departments at the University of Wisconsin-Madison. His research interests focus on architectural and microarchitectural techniques for high-performance microprocessors, including instruction-level parallelism, out-of-order execution with precise exceptions, nonblocking caches, speculative multithreading, and memory-dependence speculation. He received a PhD in electrical and computer engineering from the University of Illinois, Urbana-Champaign. He is a member of the IEEE and the ACM. Contact him at sohi@cs.wisc.edu.*

Amir Roth *is a PhD candidate in the Department of Computer Sciences at the University of Wisconsin-Madison. His research interests are in the area of computer architecture, primarily the design of future high-performance microprocessors. He is also interested in emerging applications and their performance needs, compiler technology, and opportunities for software-hardware cooperation. He received an MS in computer science from the University of Wisconsin-Madison. He is a member of the ACM. Contact him at amir@cs.wisc.edu.*