Speculative Multithreaded Processors

Gurindar S. Sohi and Amir Roth

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St. Madison, WI 53706
sohi@cs.wisc.edu

Abstract. Architects of future generation processors will have hundreds of millions of transistors with which to build computing chips. At the same time, it is becoming clear that naive scaling of conventional (superscalar) designs will increase complexity and cost while not meeting performance goals. Consequently, many computer architects are advocating a shift in focus from high-performance to high-throughput with a corresponding shift to multithreaded architectures. Multithreaded architectures provide new opportunities for extracting parallelism from a single program via *thread level speculation*. We expect to see two major forms of thread-level speculation: *control-driven* and *data-driven*. We believe that future processors will not only be multithreaded, but will also support thread-level speculation, giving them the flexibility to operate in either multiple-program/high-throughput or single-program/high-performance capacities. Deployment of such processors will require innovations in means to convey multithreading information from software to hardware, algorithms for thread selection and management, as well as hardware structures to support the simultaneous execution of collections of speculative and non-speculative threads.

1 Introduction

The driving forces behind the tremendous improvement in processing speed have been semiconductor technology and innovative architectures and microarchitectures. Semiconductor technology has provided the "bricks and mortar" — increasingly greater numbers of increasingly faster on-chip devices. Innovations in computer architecture and microarchitecture (and accompanying software) have provided techniques to make good use of these building materials to yield high-performance computing systems. Computer designs are constantly changing as architects search for (and often find) innovations to match technology advances and important shifts in technology parameters; this is likely to continue well into the next decade.

The process of deciding how available semiconductor resources will be used can be decomposed into two. First, the architect must decide on the desired *functionality*: the techniques used to expose, extract and enhance performance. Then comes the problem of *implementation*: the techniques must translated to structures and signals which must themselves be designed, built and verified.

Though described separately, these issues are, in practice, very tightly coupled in the overall design process. In the 1990s, novel functionality played the dominant role in microprocessor design. With a "reasonable" limit on the overall size of a design (e.g., fewer than tens of millions of transistors), the transistor budget could be divided by high-level performance metrics only. Verification was (relatively) simple and many of the problems encountered during implementation were manageable: wire delays were not significant as compared to logic delays, and power requirements were not exorbitant. In the future, however, implementation issues are likely to dominate even basic functionality. Monolithic designs occupying many tens or hundreds of millions of transistors will be very difficult to design, debug, and verify, and increasing wire delays will make intra-chip communication and clock distribution costly. These technology trends suggest designs that are made of replicated components, where each component may be as much as a complete processing element. Distributed, replicated organizations can "divide and conquer" the complexities of design, debug and verification, and can exploit localities of communication to deal with wire delays.

Fortunately, the twin goals of increasing single-program performance and easing implementation are not in conflict. In fact, with the right model for parallelism they can be synergistic. *Speculative multithreading* is such a model, making it a leading candidate for implementation in future-generation processors. In speculative multithreading, a processor is (logically) comprised of replicated processing elements that cooperate on the parallel execution of a conventional sequential program (also referred to as a conventional program thread) that has been divided into chunks called speculative threads. Speculation is a key element. Without speculation, programs can only be divided conservatively into threads whose mutual independence must be guaranteed. Speculation allows these guarantees to be bypassed, producing much more aggressive divisions into threads that are parallel with high probability.

2 Rationale for Speculative Multithreading

The motivation for using speculative multithreading comes from two directions. On one hand, the potential for further increasing single-program performance using known parallelism extraction techniques is diminishing. On the other, technology trends suggest processors that can execute multiple threads of code. These circumstances invite us to find those few innovations that will enable such multithreaded processors to support the parallel execution of a single program.

2.1 Limitations of Existing Techniques to Extract Parallelism

We begin by briefly reviewing the functionality and high-level operation of the incumbent model for achieving high single-program performance — the *superscalar* model. Imperative programs — programs written in imperative languages like Fortran, C, and Java — are defined by a static control flow in which individual instructions read and write named storage locations. At runtime, a superscalar processor unrolls the static control flow to produce a dynamic instruction

stream. The positions of reader and writer instructions in this stream defines the way data flows from one operation to another, i.e., the algorithm itself. A superscalar processor creates a dynamic *instruction window* (an unrolled contiguous segment of the dynamic instruction stream), repeatedly searches this window for un-executed, independent instructions, and attempts to execute these instructions in parallel. Sustained high-performance demands that any given window contain a sufficient number of independent instructions, i.e., a sufficient level of *instruction-level parallelism (ILP)*.

Unfortunately, the way in which imperative programs are written makes consistently high ILP a rarity. In order to preserve their sanity, programmers structure programs in certain ways, a basic technique being the static (and hence dynamic) grouping of dependent instructions. The spatial proximity of related statements helps programmers reason about programs in a hierarchical fashion but limits the amount of independent work that would available in a given window of dynamic instructions. Optimizing compilers attempt to improve the situation by transparently re-ordering instructions, mixing instructions from nearby program regions to improve the overall levels of window ILP. However, while very sophisticated, compiler scheduling is fundamentally limited by compilers' inability to perfectly determine the original intent of the programmer and their commitment to preserve the high-level structure of the original program.

The amount of parallel work being what it is, one option is to build a superscalar processor with an instruction window large enough to simultaneously contain code from different program regions (i.e., different functions or loop iterations). However, even if such a machine could be built — and there are many engineering obstacles to doing so — there is fundamental problem in keeping a large, contiguous instruction window full of *useful* instructions. 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 allows parallelism from different program regions to be exploited in a reasonably independent (i.e., non-contiguous, non-serial) manner. *Speculative multithreading* is such a model. In speculative multithreading, each program region is considered to be a *speculative thread*, i.e., a small program. By executing multiple speculative threads in parallel, additional parallelism can be extracted (especially if each thread is mostly sequential). The threads are subsequently merged to recreate the original program. Speculative multithreading allows a large instruction window to be created as an ensemble of smaller instruction windows, thereby facilitating implementation. In addition, a proper thread division can logically isolate branches in one thread from those in another [27], relieving the fundamental problem of diminishing instruction utility.

2.2 The Emergence of Multithreaded Architectures

Multithreaded processors — processors that support the concurrent execution of multiple threads on a single chip — are beginning to look as if they will dominate the landscape of the next decade. Two multithreaded processor models

are currently being explored. *Simultaneous multithreading (SMT)* [5, 7, 14, 32, 33] uses a monolithic design with most resources shared amongst the threads. *Chip multiprocessing (CMP)* [12] proposes a distributed design (a collection of independent processing elements) with less resource sharing. The SMT model is motivated by the observation that support for multiple threads can be provided on top of a conventional ILP (i.e., superscalar) processor with little additional cost. The CMP model is more conventionally motivated by design simplicity and replication arguments. Both models target independent threads (multithreaded a multiprogrammed workloads) and use multithreading to improve processing *throughput*.

As technology changes, the distinction between the SMT and CMP microarchitectures is likely to blur. Increasing wire delays will require decentralization of most critical processor functionality, while flexible resource allocation policies will enhance the appearance of (perhaps asymmetric) resource sharing. Regardless of the specific implementation, multithreaded processors will logically appear to be collections of processing elements. The interesting question is whether this organization can be exploited to improve not only throughput but also the execution time of a single program. *Thread-level speculation* is the key to enabling this synergy. In addition to executing conventional parallel threads, the logical processors could execute *single programs that are divided into speculative threads*. Speculative multithreaded processors will provide not only high throughput but also high single-program performance when needed.

3 Dividing Programs into Multiple Threads

There are several ways in which to divide programs into threads. We categorize these divisions as *control-driven* and *data-driven* depending on whether threads are divided primarily along control-flow or data-flow boundaries. Each division strategy can be further sub-categorized as either *non-speculative* — the threads are completely independent from the point of view of the processor and any dependence is explicitly enforced using architectural synchronization constructs, or *speculative* — the threads may not be perfectly independent, or synchronized, and it is up to the hardware to detect and potentially recover from violations of the independence assumptions.

The threads obtained from a division of a program are expected to execute on different (logical) processing units. To achieve concurrency, *proximal* threads (i.e., threads that will simultaneously co-exist in the machine) need to be highly data-independent. If data-independence can be achieved, concurrency (and hence performance) can scale almost linearly with the number of threads even for small per-thread window sizes, and efficiency can be kept constant as bandwidth and (hopefully) performance are increased. We expect that speculation can allow data-independence criteria to be achieved more easily, giving speculative solutions distinct performance and applicability advantages over their more conventional non-speculative counterparts.

3.1 Control-Driven Threads

Although the object of multithreading a program is to divide it into data-independent (parallel) threads, the most natural division of an imperative program is along control-flow boundaries into control-driven threads. The architectural semantics of imperative programs are control-driven: instructions are totally ordered and architectural state is precisely defined only at instruction boundaries. Control-flow is explicit while data-flow is implicit in the total order. In control-driven multithreading, the dynamic instruction stream is divided into contiguous segments that can subsequently be "sewn" together end-to-end to reconstruct the sequential execution. The challenge of control-driven multithreading is finding division points that minimize inter-thread data dependences.

We should note here that control-driven multithreading is not the same as parallel programming. Parallel programs do execute multiple concurrent control-driven threads, but these threads exchange data in arbitrary ways. The semantics of a parallel program is rarely the semantics of the individual threads run in series. In contrast, control-driven multithreading is a way of imposing parallel execution on what is in essence a sequential program. Data flows between control-driven threads in one direction only, from sequentially "older" threads to "younger" ones.

Non-Speculative Control-Driven Threads. Without support for detecting and recovering from data-dependence violations or to abort unnecessary threads and discard their effects, non-speculative control-driven multithreading requires strict guarantees about the *execution-certainty* and *data-integrity* of threads. Execution-certainty requirements spawn from the fact that thread execution cannot be *undone*, and mean that non-speculative control-driven threads can only be forked if their execution is known to be needed. In order to maximize concurrency, execution certainty is usually achieved by forking a thread at a previous control-equivalent point, e.g., forking of a loop iteration at the beginning of the previous iteration. Data-integrity refers to the requirement that access to thread shared data must occur (or appear to occur) in sequential order. When we speak of data-integrity, we are mainly concerned with memory-integrity. Support for direct inter-thread register communication is typically not available. We assume that if it is provided then appropriate synchronization is provided along with it. In contrast, inter-thread memory communication is naturally available, meaning that access to any memory location that could potentially be shared with other threads must be explicitly synchronized. Of course, data-sharing/synchronization should be kept to a minimum to allow for adequate concurrency among threads.

With such strict safety requirements, the division of a program into non-speculative threads has traditionally fallen into the realm of the programmer and compiler. The programmer has the deepest knowledge of the parallel dimensions of his algorithm and the potential for data-sharing among different divisions. However, performing thread division by hand is tedious, and manual

attempts to minimize synchronization often lead to errors. In light of these difficulties, much effort has been placed into using the compiler to automatically multithread (parallelize) programs. Although (debugged) compilers don't make errors, and compiler tedium is less of an issue than programmer tedium, compiler multithreading has had success only in very limited domains.

Speculative Control-Driven Threads. Non-speculative 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 this independence is unprovable, then conservative synchronization must be used to guard against the unlikely (but remotely possible) case of a re-ordered communication. Where synchronization is needlessly applied, concurrency and performance are unnecessarily lost.

Speculation can alleviate these problems. In speculative control-driven multithreading, memory does not need to be explicitly synchronized at all. The correct total order of memory operations can be reconstructed from the (explicit or implied) order of the threads. This ordering can be used as the basis for hardware support to detect and potentially recover from inter-thread memory-ordering violations [10, 11]. With such support, access to thread shared data can proceed optimistically, with penalties incurred only in those cases when data is actually shared by proximal threads *and* the accesses occur in non-sequential order. Furthermore, since ordering violation scenarios are typically predictable, slight modifications to the basic mechanism allow it to learn to recognize these scenarios early and artificially synchronize the offending store/load pairs [4, 17].

The execution-certainty constraints can be lifted using similar mechanisms. The ability to recover from inter-thread memory-ordering violations implies the presence of hardware that can buffer or undo changes to architected thread state. This support can be used to undo an entire thread, allowing threads to be spawned at points at which their final usefulness cannot be absolutely guaranteed, but where usefulness likelihood is high and the data-independence (parallelism) characteristics are more favorable.

Speculative control-driven multithreading has been the subject of academic research in the 1990's [1, 6, 9, 13, 16, 27, 29, 34] and is slowly finding its way into commercial products. Sun's MAJC architecture [31] supports such threads, via its Space Time Computing (STC) model. More recently, NEC's Merlot chip [18] uses speculative control-driven multithreading to parallelize the execution of code that can't be parallelized by other known means. 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.

3.2 Data-Driven Threads

Where control-driven multithreading divides programs along control-flow boundaries, data-driven multithreading uses data-flow boundaries as the major divi-

sion criteria. Such a division naturally achieves the desired inter-thread data-independence and resulting parallelism [2, 15, 19, 25, 26].

Data-driven threads are almost ideal from a performance and efficiency standpoint. In its pure form, data-driven multithreading occurs at the granularity of a single instruction [2, 19]. Data-driven instruction sequencing (i.e., fetch) is triggered by the availability of one of its input operands. Instructions enter the machine as soon as they may be able to execute but no sooner. This arrangement maximizes the amount of work that may be used to overlap with long latency instructions, while not wasting resources on instructions that are not ready to use them.

Instruction-level data-driven sequencing is not the only option. Data-driven sequencing may be used on a thread granularity with conventional, control-driven sequencing used at the instruction level [15, 26, 30]. In this organization, instructions from one or several related computations are packed into totally-ordered threads that implicitly specify data-flow relationships. Individual threads are assigned to processing elements and sequenced and executed in a control-driven manner. However, the data-flow relationships *between* threads are represented explicitly and thread creation is triggered in a data-driven manner (i.e., 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 are likely to be of this form.

Non-Speculative Data-Driven Threads. Non-speculative data-driven multithreading is difficult to implement for imperative languages. The main barrier is the incongruity of the requirement of an explicit data-flow program representation and the reality that for imperative programs, data-flow information is often impossible to explicitly specify *a priori* even as it applies to a few well defined boundaries. A data-forwarding error, either of omission or false commission, changes the meaning of the program. The automatic conversion of imperative code to data-flow explicit form has been the subject of some research, but in general, data-driven program representations can only be constructed for code written in functional (data-driven) languages.

Speculative Data-Driven Threads. Non-speculative data-driven multithreading suffers from two major problems. First, programs can generally not 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 the correlation between the executing program and the source code from which it was derived. Sequential semantics (or at least their appearance) is very important for program development, debugging, and the interaction with non-data-driven system components and tasks. The loss of sequential semantics is more serious than simply being a disturbance to the programmer.

Again, speculation is likely to be the key to solving these problems. However, a shift in approach regarding the role of multithreading may be needed first. Two complementary observations guide this new approach. First, program development and debugging will probably require the presence of a "main" or

"architectural" thread whose execution will implement the sequential, control-driven semantics of the program. Second, programs inherently contain sufficient levels of ILP, but this ILP is hindered by long-latency microarchitectural events like cache misses and branch mis-predictions. The parallelism in the program can be extracted if these latencies — which are likely to get relatively longer — can be tolerated. These observations suggest a different role for multithreading, one which does not require dividing the program *per se*. Instead, the program is augmented with "helper" threads that run ahead and *pre-execute* or "solve" problem instructions before they have a chance to cause stalls in the "main" program thread. We believe that it is in this capacity, as high-powered "helper" threads, that speculative data-driven threads can best be used in an imperative context [3, 8, 20, 21, 23, 24, 28, 35].

In the "helper" model, selected computations are copied from the program and packed into data-driven threads [8, 23, 24]. Now, the program is executed as a single control-driven thread, as usual. However, at certain points in the main program, data-driven threads are spawned in order to pre-execute the computation of some future problem instruction. When the main program thread catches up to the data-driven thread, it has the option of picking up the result directly [8, 21–23] or simply repeating the work (albeit with a reduced latency) [20].

The role of speculation is intermingled with the reduced "helper" status of data-driven threads. The fact that the control-driven thread is present and ultimately responsible for the architectural interface, immediately relieves data-driven threads from any correctness obligations. Without these obligations, data-driven threads can be constructed using whatever data-flow information is available. In addition, they need not comprise a complete partitioning of the program; their use may be reserved only for those situations in which their parallelism-enhancing characteristics are most needed.

4 Practical Aspects

Whether future processors will also include support for speculative threads — either control-driven, data-driven or both — depends on the discovery of acceptable solutions to several practical problems. These problems range from the low-level (i.e., how threads should be implemented) to the high-level (i.e., how threads should be used) and cover all levels in between. We briefly touch upon some of these issues in this section.

4.1 System Architecture

The broadest decision that needs to be made and the one that will have the most impact on other decisions is the division of labor and responsibilities between the programmer, compiler, operating system and processor. It is obvious that the processor will execute the threads. However, the answer to the question of what entity should be responsible for other thread-related tasks — from selecting the

threads themselves to spawning, scheduling, resource allocation and communication — is not clear. Placing all of the responsibility on the processor is one attractive option. With near-future processors having nearly one billion transistors, a few million can be dedicated to multithreading-specific management tasks (perhaps as a separate co-processor). A processor-only implementation has no forward or backward compatibility problems, it preserves the current system interface, while enhancing the performance of legacy software. Its drawbacks are added design complexity and the mandate rigidity and simplicity of the thread selection and management algorithms.

Since thread-selection is such an important and delicate problem, it seems logical to push at least that function to software or perhaps even the programmer. Thread-selection algorithms implemented in software can be more sophisticated and may produce better thread divisions. Thread divisions chosen by the programmer — who understands the program at its highest, algorithmic levels — and subsequently communicated to the compiler, may be better still. 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 are typically met with some resistance, especially if they have architectural semantics that need to be implemented.

Our expectation is that speculative thread information is likely to be conveyed from software to hardware, but in an *advisory* form. An example of advisory information are prefetch instructions that are found in many recent architectures. The understanding is that the hardware may act upon this information either fully, partially or selectively, or even ignore it altogether, all without impacting correctness. The option to enhance or refine this information dynamically is left to the processor as well. Restricting speculative thread information to an advisory role relieves the architect from many functionality guarantees that would hamper future generation implementations.

4.2 Specific Hardware Support

For the full power of speculative multithreading to be realized, hardware support is required. Specifically, threads need to be made "lightweight" with mechanisms for fast thread startup and inter-thread communication and synchronization. Hardware support for speculation includes buffering for speculative actions and facilities for fast correct-speculation state commit and, likewise, mis-speculation recovery. The precise support required for control-driven and data-driven threads is somewhat different. An additional challenge is to provide this support, as well as support for conventional parallel threads, using a uniform set of simple mechanisms.

One apparent requirement for the implementation of lightweight threads (speculative and otherwise) is a mechanism for passing values from one thread to another via registers. Memory communication and synchronization is likely to be reasonably fast on a speculatively multithreaded processor, since the bulk of it will occur through the highest level of shared on-chip cache. However, a register path for communication and synchronization is likely to be faster still.

Inter-thread register communication will also allow thread register contexts to be initialized quickly, accelerating thread start-up.

We assume that the register-communication mechanism will implement inter-thread register synchronization. Another requirement is a mechanism for enforcing correct ordering of memory operations from different threads. At a high level, such a mechanism would buffer loads from young threads and compare them with colliding stores from older threads. Designs for inter-thread memory ordering mechanisms are known in both centralized [10] and distributed forms [11]. The distributed form uses a modified cache-coherence protocol that blends naturally with the protocol that implements general data-sharing for parallel threads. We expect this form to find widespread use in future processors.

5 Summary

Future processors will be comprised of a collection of logical processing elements that will collectively execute multiple program threads. To overcome the limitations in dividing a single program into multiple threads that can execute on these multiple logical processing elements, speculation will be used. A sequential program will be "speculatively parallelized" and divided into speculative threads. Speculative threads are not only a good match for the microarchitectures that are likely to result as technology advances, they have the potential to overcome the limitations of currently-known methods to extract instruction-level parallelism.

There are two main types of speculative threads that we expect to be used: control-driven and data-driven threads. Speculative control-driven threads have already begun to appear in commercial products (e.g., Sun's MAJC and NEC's Merlot), while speculative data-driven threads are still in the research phase.

Several technologies will have to be developed before speculative multithreading is commonplace in mainstream processors. These 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 non-speculative threads. Consequently we expect the next decade of processor development to be at least as exciting as previous decades.

Acknowledgements

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

References

1. H. Akkary and M.A. Driscoll. A Dynamic Multithreading Processor. In *Proc. 31st International Symposium on Microarchitecture*, pages 226–236, Nov. 1998.

2. Arvind and R.S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, Mar. 1990.
3. R.S. Chappell, J. Stark, S.P. Kim, S.K. Reinhardt, and Y.N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proc. 26th International Symposium on Computer Architecture*, May 1999.
4. G.Z. Chrysos and J.S. Emer. Memory Dependence Prediction using Store Sets. In *Proc. 25th International Symposium on Computer Architecture*, pages 142–153, Jun. 1998.
5. G.E. Daddis and H.C. Torng. The concurrent execution of multiple instruction streams on superscalar processors. In *Proc. International Conference on Parallel Processing*, pages 76–83, May 1991.
6. P.K. Dubey, K. O'brien, K.A. O'brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading. In *Proc. 1995 Conference on Parallel Architectures and Compilation Techniques*, pages 109–121, Jun. 1995.
7. J. Emer. Simultaneous Multithreading: Multiplying Alpha's Performance. Microprocessor Forum, Oct. 1999.
8. A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. 31st International Symposium on Microarchitecture*, pages 59–68, Dec. 1998.
9. M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, Madison, WI 53706, Nov. 1993.
10. M. Franklin and G.S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, May 1996.
11. S. Gopal, T.N. Vijaykumar, J.E. Smith, and G.S. Sohi. Speculative Versioning Cache. In *Proc. 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, Feb. 1998.
12. L. Hammond, B.A. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, Sep. 1997.
13. L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, Oct. 1998.
14. H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In *Proc. 19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.
15. R.A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. 15 International Symposium on Computer Architecture*, pages 131–140, May 1988.
16. Z. Li, J.-Y. Tsai, X. Wang, P.-C. Yew, and B. Zheng. Compiler Techniques for Concurrent Multithreading with Hardware Speculation Support. In *Proc. 9th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1996.
17. A. Moshovos, S.E. Breach, T.N. Vijaykumar, and G.S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proc. 24th International Symposium on Computer Architecture*, pages 181–193, Jun. 1997.
18. N. Nishi, T. Inoue, M. Nomura, S. Matsushita, S. Toru, A. Shibayama, J. Sakai, T. Oshawa, Y. Nakamura, S. Shimada, Y. Ito, M. Edahiro, M. Mizuno, K. Minami, O. Matsuo, H. Inoue, T. Manabe, T. Yamazaki, Y. Nakazawa, Y. Hirota, and Y. Yamada. A 1 GIPS 1 W Single-Chip Tightly-Coupled Four-Way Multiprocessor with Architecture Support for Multiple Control-Flow Execution. In *Proc. 47th International IEEE Solid-State Circuits Conference*, Feb. 2000.

19. G. Papadopoulos and D. Culler. Monsoon: An Explict Token-Store Architecture. In *Proc. 17th International Symposium on Computer Architecture*, pages 82–91, Jul. 1990.
20. A. Roth, A. Moshovos, and G.S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.
21. A. Roth, A. Moshovos, and G.S. Sohi. Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation. In *Proc. 1999 Internation Conference on Supercomputing*, pages 356–364, Jun. 1999.
22. A. Roth and G.S. Sohi. Register Integration: A Simple and Efficent Implementation of Squash Re-Use. In *Proc. 33rd Annual International Symposium on Microarchitecture*, Dec. 2000.
23. A. Roth and G.S. Sohi. Speculative Data-Driven Multithreading. Technical Report CS-TR-00-1414, University of Wisconsin, Madison, Mar. 2000.
24. A. Roth, C.B. Zilles, and G.S. Sohi. Speculative Miss/Execute Decoupling. In *Proc. Workshop on Memory Access Decoupling in Superscalar and Multithreaded Architectures*, Oct. 2000.
25. S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. 16th Annual International Symposium on Computer Architecture*, pages 46–53, May 1989.
26. M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura. Thread-based Programming for the EM-4 Hybrid Dataflow Machine. In *Proc. 19th Annual International Symposium on Computer Architecture*, pages 146–155, May 1992.
27. G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proc. 22nd International Symposium on Computer Architecture*, pages 414–425, Jun. 1995.
28. Y.H. Song and M. Dubois. Assisted Execution. Technical Report #CENG 98-25, Department of EE-Systems, University of Southern California, Oct. 1998.
29. J.G. Steffan and T.C. Mowry. The Potential for Using Thread Level Data-Speculation to Facilitate Automatic Parallelization. In *Proc. 4th International Symposium on High Performance Computer Architecture*, Feb. 1998.
30. M. Takesue. A Unified Resource Management and Execution Control Mechanism for Data Flow Machines. In *Proc. 14th Annual International Symposium on Computer Architecture*, pages 90–97, Jun. 1987.
31. M. Tremblay. MAJC: An Architecture for the New Millenium. In *Proc. Hot Chips 11*, pages 275–288, Aug. 1999. http://www.sun.com/microelectronics/MAJC/documentation/docs/HC99sm.pdf.
32. D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. 23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.
33. W. Yamamoto and M. Nemirovsky. Increasing Superscalar Performance Through Multistreaming. In *Proc. 1995 Conference on Parallel Architectures and Compilation Techniques*, Jun. 1995.
34. Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proc. 4th International Symposium on High-Performance Computer Architecture*, Feb. 1998.
35. C.B. Zilles and G.S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proc. 27th International Symposium on Computer Architecture*, pages 172–181, Jun. 2000.