Microprocessors — 10 Years Back, 10 Years Ahead

Gurindar S. Sohi

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

1 Introduction

Continuing improvements in semiconductor technology — as characterized by Moore's law — have provided computer architects with an increasing number of faster transistors with which to build microprocessors. In the past decade, architects have seized these opportunities to build microprocessors that bear little resemblance to the microprocessors of the 1970s and 1980s. With Moore's law projected to hold beyond the next decade, microprocessor architects will have even larger transistor budgets with which to build innovative microprocessors; the microprocessor circa 2010 is likely to bear little resemblance to today's microprocessor.

The driving force behind the innovation in microprocessors for the past two decades has been the quest for higher performance. Arguably the best way to achieve this goal is an integrated approach that combines innovation in all aspects of the problem: algorithms, software, and hardware. Computer architects typically work with a given algorithm, and concern themselves with hardware and software. Within this context, the best technical solution is perhaps an integrated software/hardware solution, where a compiler works in concert with the architecture and implementation, possibly with changes to the hardware/software interface (*i.e.,* the instruction set architecture). However, this approach has not proven to be viable in the long term for a variety of reasons, both technical and non-technical. Perhaps the most compelling (non-technical) reason is that it is not practical to change the instruction set frequently. Radical changes to instruction sets for general-purpose processors happen infrequently, perhaps only once every few decades (*e.g.,* DEC's change from VAX to Alpha), with incremental changes/additions (*e.g.,* Intel's MMX enhancements to IA-32) being more common. This changes the model for achieving the desired goal of higher performance: both software and hardware are forced to work within the constraints of a fixed (or nearly fixed) software/hardware interface. Consequently, many of the innovations in microprocessors have been in the microarchitecture, the building blocks of a microprocessor. A circa 1990 microprocessor (*e.g.,* the Intel 486) and a circa 2000 microprocessor (*e.g.,* the Intel Pentium IV) of the same family have essentially the same instruction set architecture, but radically different microarchitectures.

To understand the innovations in microarchitecture, let us start out with the CPU performance equation: $Time = N \times CPI \times T$, where $Time$ is the time taken to execute a program, N is the number of instructions executed dynamically, CPI is the number of cycles per instruction, and T is the clock cycle time. Improving execution time means reducing the three terms on the right-hand side of the equation. The microprocessor architect typically has little influence over the first term — that is the realm of instruction set designers and compiler writers — so the emphasis is on decreasing the second and third terms.

Semiconductor technology allows faster transistors, and these translate into a faster clock cycle. The clock cycle can be made even faster by pipelining the logic into more stages. Since not all technologies improve at the same rate (*e.g.,* logic speeds have increased much faster than memory speeds), a faster clock cycle results in increasing (relative) latencies of operations, which translates into an increase in CPI. The role of the microprocessor architect is then to develop microarchitectural features that not only prevent an increase in CPI as the clock cycle is reduced, but even decrease it. The additional transistors provide ample resources for implementing new features that aim to achieve this goal.

2 10 Years Back: The Emergence of Speculative Execution Microarchitectures

To decrease CPI, parallelism is used to overlap the processing of instructions. For a microprocessor architect, this has meant *fine-grain*, or *instruction-level parallelism (ILP)*, leaving more coarse-grain parallelism to be exploited by multiprocessors. Exploiting ILP is a cost-effective way of making use of chip real estate and improving performance. Many of the techniques in the microprocessor architect's toolbox increase the exploitation of ILP in a program's execution.

Techniques to exploit ILP can either be static — those used in EPIC [8], or they can be dynamic — those used by superscalar processors such as Compaq's Alpha 21264, Intel's Pentium II, and others. We will limit our discussion to dynamically-scheduled superscalar processors since these dominated the 1990s, whereas general-purpose microprocessors using statically-scheduled ILP techniques were only announced towards the end of the 1990s.

The processing of an instruction requires many steps, and we want to overlap as many steps as possible to increase throughput; more overlap requires more ILP. To understand the increasing demand for ILP, consider the increase in the number of instructions that can be "in flight" at a given time in a microprocessor. In the 1970s, microprocessors executed one instruction at a time, taking many clock cycles to execute that instruction — there was only one instruction in flight. The 1980s were the decade of pipelining; a typical pipeline had 5 stages, processing a single instruction per clock cycle, resulting in up to 5 instructions in flight. The 1990s were characterized by deeper pipelines, and wider instruction issue (processing multiple instructions per cycle): a 2-issue, 5-stage pipeline Pentium processor, a 3-issue, 10-stage pipeline Pentium II, and a 3-issue, 20-stage

pipeline Pentium IV, have about 10, 30, and 60 instructions (more accurately, operations) in flight, respectively.

A strategy that relies on keeping many instructions in flight can only succeed if many "useful" instruction can be kept in flight, and this requires the processor to identify the path that will be taken through the program. Since a branch instruction occurs every 5 to 6 instructions in typical programs, techniques to prevent branches from stalling instruction fetching had to be developed: the flow of instructions cannot be stopped when a branch is encountered. (With 50-60 instructions in flight there are likely to be about 10 branch instructions in flight, so clearly branches cannot be processed sequentially.) Early machines proposed *predicting* the direction of a branch and fetching instructions from the predicted path [1]. But fetching alone is not enough, since the number of pipeline stages devoted to instruction fetching constitute only a fraction of the pipeline. This then bring up the notion of *speculative execution*: instructions from a predicted path of a branch must also be executed so that more instructions can be kept in flight, and overlap in processing increased.

The 1990s were the decade of speculative execution, *i.e.,* the decade where speculative execution processors entered the mainstream of processor design. Today it is more apt to call the above *control speculation*, since the speculation is on the outcome of a control (branch) instruction. Other forms of speculation have since appeared. With control speculation, the outcome of a branch instruction is predicted, and instructions from the predicted path executed in a speculatively. If the prediction turns out to be incorrect, the speculative instructions are *squashed* (or *aborted*). Control speculative execution necessitates microarchitectural mechanisms to support it [17, 24]. These include mechanisms to guide the speculation: branch predictors, and mechanisms to recover from a misspeculation: physical registers, register renaming, and precise exceptions.

The branch predictor is a key component, since the accuracy of the predictor directly determines the utility of the instructions being executed speculatively. The 1990s saw a lot of research in the design of branch predictors. Early processors used simple 2-bit predictors [22], which quickly gave way to 2-level adaptive predictors [32], as transistor resources increased. Processors of the late 1990s even employ multiple branch predictors since experiments suggested that different prediction automaton work better on different types of branches [14].

The first major component of speculative execution hardware is *storage* where results of (speculatively-executed) instructions can reside — typically only non-speculative values can reside in the logical (or architectural) register file and the memory system. Storage is needed to hold values until they are consumed, or *committed* to architectural state. This storage can be provided in several ways, including reservations stations, reorder buffers, and physical register files.

The second major component is a *precise exception mechanism*. The program which the processor executes is written with the assumption of sequential execution — instructions are executed one at a time. Thus while a processor may overlap the execution of instructions in a pipeline, execute instructions out of program order, and even execute instructions speculatively, it must appear to an

outside observer (*e.g.,* the creator of the program) that the instructions executed sequentially. In particular, it must be possible to recover the precise state of the program at any given time. A precise state at an arbitrary point in a program's execution corresponds to the state that would result if all instructions prior to the point of interest have completed execution (and updated machine state), and no subsequent instructions have affected machine state. There are several mechanism to recover a precise machine state. These include *reorder buffers, history buffers* (or *checkpoints*), and *future files* [21].

The third major component is a *register renaming mechanism*. With many in-flight instructions, there may be many distinct values associated with a logical register; register renaming helps find the correct value. With a renaming mechanism values that would reside in a logical register at different points in time during the execution of a program (*e.g.,* results of two instructions separated by a branch) can reside in different physical storage elements, thereby allowing the execution of the two instructions to be overlapped. Without different storage elements to hold the results, and an accompanying register renaming mechanism, only one of the instructions could be processed at a time.

Different dynamically-scheduled superscalar processors of the 1990s used different combinations of mechanisms to support speculative execution (*e.g.,* physical register files with history buffers for precise exceptions, or reservation stations and reorder buffer) — no two processors implement the same functionality in the same way. Some processors even used different mechanisms for recovering precise versions of different state (*e.g.,* reorder buffers for registers, and history buffers for register rename mappings). Collectively, the mechanisms allow instructions to be scheduled dynamically to maximize overlap in instruction processing, yet retain the appearance of sequential execution. Many microprocessor architects have viewed the constraint of maintaining the appearance of sequential execution as an asset rather than a liability: sequential execution provides a precise definition of a total order in which events have to occur, and this facilitates debugging and verification of hardware. Without a total order, it can be difficult to verify and debug hardware and software, since the sequence of events that creates a problem cannot be repeated. Thus, even though out-of-order execution and register renaming was proposed in the 1960s (albeit, without speculative execution) [28], the lack of precise state recovery mechanisms, and the consequent difficulties of debugging, had cast a shadow on these techniques.

The increased processing rate brought on by microarchitectural innovations has placed increasing demands on the memory system. Already handicapped by a widening gap between logic and memory speeds, more innovation was needed to deal with the increasing latency, as well as with increasing bandwidth demands. Multi-level caches arose to plug the latency gap; most high-performance microprocessors today have two levels of cache on the chip, and many have an additional level of cache off chip. To meet the bandwidth demands, the upper levels of the cache hierarchy (the ones closer to the CPU) became *non-blocking*, allowing requests to be overlapped, *i.e.,* allowing misses to be overlapped with hits, and with other misses [12, 25]. In addition, to service multiple hits per cy-

cle, caches have recently become multi-ported [25]. The impact of the increasing bandwidth demands has stretched all the way to the main memory, with high-bandwidth Rambus DRAMs (RDRAMs) replacing traditional DRAMs for high performance applications.

3 Current: The Blossoming of Speculative Execution Microarchitectures

While there was resistance to speculative execution and dynamic scheduling in the late 1980s and early 1990s, this resistance was overcome, and today high-end processors from most companies support both. Having implemented microarchitectural mechanisms to support these techniques within existing transistor budgets, microarchitects asked: (i) what else could these microarchitectural mechanisms be used for, (ii) how to use the additional transistors made available by Moore's law? These lead to an "obvious" question: could we use speculation to overcome other constraints? The basic mechanisms to support one form of speculation could possibly support other forms of speculation as well, and additional resources can be used both to increase the amount of speculative execution, as well as to improve the accuracy of different forms of speculation.

Processors that have been announced circa 2000 extend speculation beyond the basic control speculation model of the 1990s. In control speculation, a speculation was made on the outcome (taken or not taken) of a branch instruction. Modern microprocessors extend the notion of speculation to include *data speculation*, where the data values, or instruction relationships that are based upon data values, are speculated. Data speculation can be used to overcome arbitrary dependence constraints, including ambiguous- and true-dependence constraints. In its most general form, speculation could be applied to predict the value of arbitrary data items, *e.g.,* the result of an address calculation operation [2], or the value loaded from memory [13]. There has been a significant amount of research on this subject recently, but the prediction accuracies for the general form of data value speculation are currently not sufficient to allow this technique to achieve performance improvements; in many cases performance actually degrades. However, data speculation can be used profitably in other forms. Two forms used in circa 2000 microprocessors are described next.

Ambiguous dependences constrain the scheduling of load instructions. Without speculation, a load instruction cannot be (dynamically) scheduled to execute before a prior store instruction, because they might access the same memory location. This restriction, which unnecessarily constrains parallelism, can be overcome with *data dependence speculation* — speculating that the ambiguous dependence is actually not a dependence, *i.e.,* the load is independent of prior stores (whose store addresses are unknown). When a dependence exists, however, the speculation is incorrect; speculation accuracy can be improved by speculating when the ambiguous dependence is likely to resolve to no dependence, and not speculating otherwise. One way to improve the speculation is to predict the addresses which the store (and load if need be) will access, and use these pre-

dictions to assess if a true dependence is likely to be violated. This technique is cumbersome, due to the need for predicting different store addresses, and not very accurate. An alternative is to use *data dependence prediction*, where the dependence relationships (between stores and loads) are predicted. Recent work has shown that these dependence relationships are very stable, and can be predicted with very high accuracy [4, 15]. Dependence prediction and speculation is being used in several processors that are being designed circa 2000.

Another form of data speculation is used in Intel's Pentium IV processor [7]. Traditionally a cache operation is carried out atomically: data is accessed from the cache data array and the cache tags are checked to see if the correct data is being accessed. Typically these two sub-operations take different amounts of time. An atomic cache access means waiting for the slower operation (typically tag matching) to complete, increasing the latency of the overall operation. Speculation can be used to reduce the expected latency of a cache access as follows: the access is divided into its two constituent operations, data access and tag matching. Data is read from the data arrays, a speculation is made that it is a cache hit, and the data returned to the processor immediately. Later when the tags are checked, the speculation is verified. If the speculation was incorrect, *i.e.,* the reference was a cache miss, the offending instruction, and instructions dependent on it, are replayed. (Another way of looking at this speculation is as a load value speculation, with the cache serving as the "value predictor," and the tag matching logic providing the verification.)

In addition to the two new forms of speculation described above, other forms of speculation are being researched, and are likely being considered for processors that are being designed. These new forms of speculation are also resulting in refinements of techniques to recover from misspeculations. For example, for control speculation, machines typically squashed all instructions following a misspeculated branch, since these instructions were unlikely to contain instructions that were control- and data-independent of the misspeculated branch. However, for data speculation, a brute-force squashing is likely to squash useful instructions (instructions that are independent of the offending instruction). Accordingly, *selective squashing* or *selective recovery/replay* mechanisms have been invented.

4 Near Future: The Emergence of Clustered and Multithreaded Microarchitectures

The coming decade will bring even more challenges, as well as opportunities, for microprocessor architects. The challenges will include *ease of design and verification*, the growing importance of *wire delays*, and the increase in *power consumption*. 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 constraints 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. The impact of power consumption on microarchitecture is still being investigated, but some researchers believe that distributed, replicated microarchitectures are likely to have better power/performance characteristics that centralized microarchitectures. Meanwhile, opportunities will be provided by even more transistor resources, and by the emergence of multithreaded workloads. Important workloads, such as server workloads, are being written as multithreaded applications, inviting microprocessor architects to use multithreading to improve the overall processing effectiveness.

The challenges and opportunities of the next decade are likely to lead to microprocessors with clustered microarchitectures that are capable of running multiple threads of code simultaneously. Several multithreaded processor models are currently being explored. *Simultaneous multithreading (SMT)* [5, 11, 30, 31] extends a "traditional" dynamically-scheduled superscalar processor to support the simultaneous execution of multiple programs. *Chip multiprocessing (CMP)* [9], as the name implies, proposes a distributed design, along the lines of a more traditional multiprocessor, on a single chip. The distinction between the SMT and CMP microarchitectures is likely to blur over time. Increasing wire delays will require decentralization of most critical processor functionality, while flexible resource allocation policies will enhance the appearance of resource sharing. In either case, multithreaded processors will logically appear to be collections of processing elements with support for speculative execution. In this context, microprocessors are expected to employ *thread-level speculation* to overcome barriers to traditional methods of parallelizing a single program. Thus, 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.

5 10 Years Ahead: The Blossoming of Speculative Multithreaded Microarchitectures

With support for both speculation and multithreading, novel techniques for using speculative threads are likely to be discovered. Research into some of these techniques is already in progress, and the expectation is that some of these research discoveries will be implemented in circa 2010 microprocessors. We briefly review some of this ongoing research below.

Threads can broadly be classified into *control-driven* and *data-driven* threads, depending on whether threads are divided primarily along control-flow or data-flow boundaries. Each category 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.

Despite extensive research, compiler generated non-speculative threads (*e.g.,* those generated by parallelizing compilers) have not held much promise beyond numeric programs because of the difficulties of statically dividing a program into such threads. The analysis required to create threads statically has too many unknowns (*e.g.,* ambiguous dependences), thwarting parallelization efforts. Again, speculation can be used to overcome constraints imposed by unknown information, and a program *dynamically parallelized* into speculative threads. Thus a program will appear to be sequential, statically, but speculatively execute in parallel, dynamically. Speculation is likely to be applied to both control- and data-driven threads.

Speculative control-driven multithreading has been the subject of academic research in the 1990's [10, 23, 27] and is slowly finding its way into commercial products. Sun's MAJC architecture [29] supports such threads, via its Space Time Computing (STC) model. More recently, NEC's Merlot chip [16] uses speculative control-driven multithreading to parallelize the execution of code that cannot 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.

Speculative data-driven threads are likely to be employed as "helper" threads which assist the "main" program thread. These helper threads that run ahead and *pre-execute* or "solve" performance-degrading problem instructions before they have a chance to cause stalls in the main program thread. There has been a fair amount of research into this issue recently [3, 6, 18–20, 26, 33], with commercial adoptions likely over the course of the next decade.

6 Summary

The microarchitecture of microprocessors has seen a dramatic change in the past decade; the same is expected for the next decade. The most significant transition of the past decade is that simple in-order processing microarchitectures have given way to dynamic-scheduling, out-of-order execution, and speculative execution. Speculative execution, initially applied to overcome control dependences, is now being used in a variety of ways, to overcome ambiguous- and even true-dependence constraints. The coming decade is expected to result in even more innovation in microprocessor microarchitectures, as microprocessors begin to support multithreaded execution, and as even more novel uses of speculation are found. A promising model for next decade microprocessors is thread-level speculation, where speculation is applied to parallelize the execution of programs that defy traditional methods of parallelization.

Acknowledgements

The author would like to thank various organizations that have supported his research over the years, including the National Science Foundation (NSF), the Defense Advanced Projects Agency (DARPA), companies such as Intel and Sun

Microsystems, and the University of Wisconsin Graduate School. The contributions of various graduate students is also gratefully acknowledged.

References

1. D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development*, pages 8–24, Jan. 1967.
2. Todd M. Austin, Dionisios N. Pnevmatikatos, and Gurindar S. Sohi. Streamlining data cache access with fast address calculation. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 369–381, June 1995.
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. J. Emer. Simultaneous Multithreading: Multiplying Alpha's Performance. Microprocessor Forum, Oct. 1999.
6. 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.
7. P. Glaskowsky. Pentium 4 (Partially) Previewed. *Microprocessor Report*, 14(8), Aug. 2000.
8. Linley Gwennap. Intel, HP Make EPIC Disclosure. *Microprocessor Report*, 11(14), Oct. 1997.
9. L. Hammond, B.A. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, Sep. 1997.
10. 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.
11. 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.
12. D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proc. 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.
13. M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value Locality and Load Value Prediction. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct. 1996.
14. Scott McFarling. Combining Branch Predictors. Technical Report WRL Technical Note, TN-36, Digital Equipment Corporation, Jun. 1993.
15. 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.
16. 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.
17. Y. N. Patt, S. W. Melvin, W. W. Hwu, and M. Shebanow. Critical issues regarding hps, a high performance microarchitecture. In *Proc. 18th Annual Workshop on Microprogramming*, pages 109–116, Dec. 1985.
18. 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.
19. 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.
20. A. Roth and G.S. Sohi. Speculative Data-Driven Multithreading. In *Proc. 7th International Symposium on High-Performance Computer Architecture*, Jan. 2001.
21. J. E. Smith and A. R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proc. 12th Annual International Symposium on Computer Architecture*, Jun. 1985.
22. J.E. Smith. A Study of Branch Prediction Strategies. In *Proc. 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.
23. G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proc. 22nd International Symposium on Computer Architecture*, pages 414–425, Jun. 1995.
24. G.S. Sohi and S. Vajapeyam. Instruction Issue Logic for High-Performance Interruptable Pipelined Processors. In *Proc. 14th International Symposium on Computer Architecture*, pages 27–34, May 1987.
25. Gurindar S. Sohi and Manoj Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proc. 4th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 52–62, Apr. 1991.
26. Y.H. Song and M. Dubois. Assisted Execution. Technical Report #CENG 98-25, Department of EE-Systems, University of Southern California, Oct. 1998.
27. 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.
28. R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, pages 25–33, Jan. 1967.
29. 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.
30. 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.
31. W. Yamamoto and M. Nemirovsky. Increasing Superscalar Performance Through Multistreaming. In *Proc. 1995 Conference on Parallel Architectures and Compilation Techniques*, Jun. 1995.
32. T-Y. Yeh and Y.N. Patt. Two-level Adaptive Training Branch Prediction. In *Proc. 24th International Symposium on Microarchitecture*, pages 51–61, Nov. 1991.
33. 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.