Understanding the Performance of Concurrent Error Detecting Superscalar Microarchitectures

Jared C. Smolens, Jangwoo Kim, James C. Hoe, and Babak Falsafi
Computer Architecture Laboratory (CALCM)
Carnegie Mellon University, Pittsburgh, PA 15213
{jsmolens, jangwook, jhoe, babak}@ece.cmu.edu
http://www.ece.cmu.edu/~truss

(Invited Paper)

Abstract**—***Superscalar out-of-order microarchitectures can be modified to support redundant execution of a program as two concurrent threads for soft-error detection. However, the extra workload from redundant execution incurs a performance penalty due to increased contention for resources throughout the datapath. We present four key parameters that affect performance of these designs, namely 1) issue and functional unit bandwidth, 2) issue queue and reorder buffer capacity, 3) decode and retirement bandwidth, and 4) coupling between redundant threads' instantaneous resource requirements. We then survey existing work in concurrent error detecting superscalar microarchitectures and evaluate these proposals with respect to the four factors.*

Keywords—*Microarchitecture reliability, performance, soft errors*

1. INTRODUCTION

A soft error occurs when the voltage level of a digital signal is temporarily disturbed by an unintended mechanism such as radiation, thermal, or electrical noise. Previously, soft errors have been a phenomenon associated with memory circuits, however the diminishing capacitance and noise margins anticipated for future deep-submicron VLSI has raised concerns regarding the susceptibility of logic and flip-flop circuits. Recent studies have speculated that by the next decade, logic and flip-flop circuits could become as susceptible to soft errors as today's unprotected memory circuits [5,8,18]. In light of increasing soft-error concerns, recent research has studied the soft-error vulnerability of the microarchitecture as a whole [11].

Existing designs for soft error detection in the processor use compare execution across replicated lock stepped pipelines within a single core [19], multiple cores in chip multiprocessors [7,10], or multiple processor dies [2,20]. Alternatively, many researchers advocate solutions within the superscalar out-of-order pipelines of modern microprocessors by exploiting idle or underutilized pipeline resources [9,12,13,14,15,16,21,22]. A minimally modified superscalar datapath can support concurrent error detection by executing an application as two independent, redundant threads. This protection requires 1) instructions to be executed redundantly and independently and 2) the redundant outcomes to be compared for correctness. A drawback of this approach is that the doubled workload from redundant execution incurs a significant performance penalty—on the order of 30%—using superscalar datapaths tuned for single threaded execution. Understandably, an application that executes a high number of instructions per cycle (IPC) as a single thread should lose performance when running as two redundant threads on the same datapath. What is less understood is why applications with low IPC as a single thread can still perform poorly as two redundant threads, despite the apparently ample execution resources to support both threads.

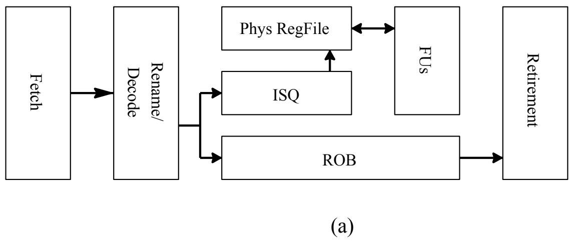
In this paper, we first explain the performance bottlenecks of prior designs in a unified framework. We use 2-k factorial analysis to study four superscalar datapath design factors that affect the performance of redundant execution. The first three factors represent the different resources that support high performance execution: 1) issue and functional unit bandwidth, 2) issue queue (ISQ) and reorder buffer (ROB) capacity, and 3) decode and retirement bandwidth. The fourth factor is the design choice of whether to allow the progress of two redundant threads to stagger elastically. We then apply the four performance factors to a survey of six representative microarchitectures within the concurrent error detecting superscalar design space. We classify each microarchitecture according to the four performance factors and summarize the hardware costs.

2. BACKGROUND

In this section, we first define our superscalar microarchitecture model assumptions for the baseline of our study. Next, we explain the extensions to superscalar microarchitectures to execute a program redundantly as two concurrent threads. Finally, we establish the baseline performance penalty of redundant execution.

This invited paper is based upon *Efficient Resource Scheduling in Concurrent Error Detecting Superscalar Microarchitectures*, previously published in the *37th Annual IEEE/ACM International Symposium on Microarchitecture.*

This work was supported by NSF CAREER award CCF-0347568, NSF award ACI-0325802, and by Intel Corporation.



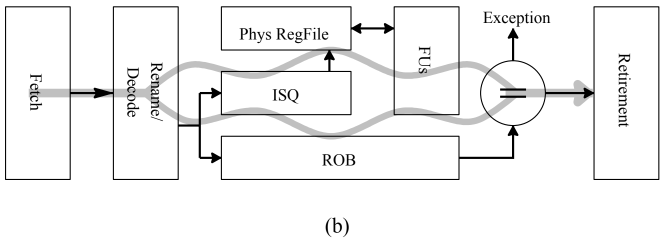


Fig. 1. The (a) SS1 baseline superscalar and (b) SS2 redundant execution microarchitectures.

Table 1. Baseline SS1 parameters.

OoO Core	128-entry ISQ, 512-entry ROB, 64-entry LSQ, 8-way decode, issue, and retirement
Memory System	64K 2-way L1 I/D caches, 64-byte line size, 3-cycle hit, unified 2M 4-way L2, 12-cycle hit, 200-cycle memory access, 32 8-target MSHRs, 4 memory ports
Functional Units (latency)	8 IALU (1), 2 IMUL/IDIV (3/19), 2 FALU (2), 2 FMUL/FDIV (4/12), all pipelined except IDIV and FDIV
Branch Predictor	Combining predictor with 64K gshare, 16K/64K 1st/2nd level PAs, 64K meta table, 2K 4-way BTB, 7 cycle branch misprediction recovery latency

2.1. *SS1: Baseline Superscalar Microarchitecture*

The starting point of all soft-error tolerant designs in this paper is a balanced superscalar speculative out-of- order microarchitecture. We assume a conventional wide-issue design where

1. architectural register names are renamed onto a large physical register file
2. an issue queue provides an out-of-order window for dataflow-ordered instruction scheduling over a mix of functional units (FUs)
3. a reorder buffer tracks ordering and execution status for all in-flight instructions
4. a load/store queue (LSQ) holds speculative memory updates and manages out-of-order memory operations

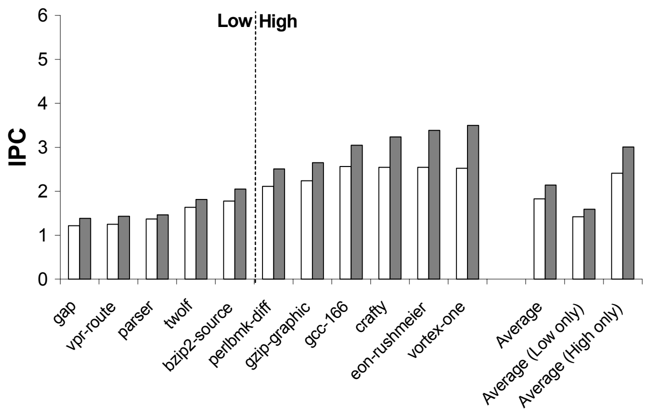
Figure 1(a) shows the baseline microarchitecture with the key datapath elements and their interconnections. Table 1 lists the salient parameters assumed for this paper. The settings in Table 1 are extrapolated from the Alpha EV8 [6]. We refer to this microarchitecture as SS1.

Throughout this paper, we report performance results based on weighted SimPoint regions [17] for 11 integer and 14 floating-point SPEC2K benchmarks running their first reference input (we exclude the integer benchmark *mcf* from our study because it is insensitive to all design parameters in this paper). Average IPCs over multiple benchmarks are computed as harmonic means. The performance simulators used in this paper are derived from the sim-outorder/Alpha performance simulator in Simplescalar 3.0 [4]. For this study, we modify sim-outorder's stock RUU-based model to support the issue queue-based SS1 datapath, MSHRs and memory bus contention.

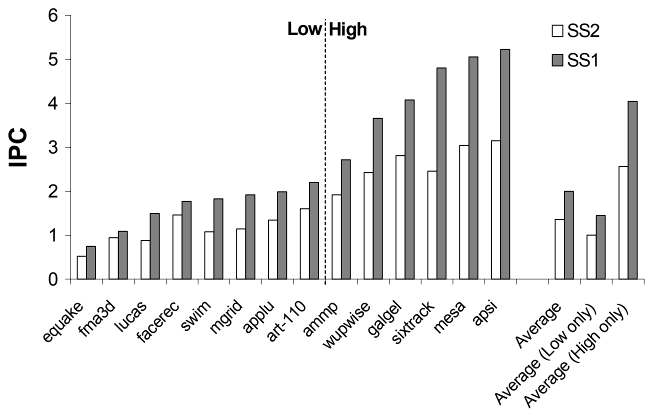
2.2. *SS2: Symmetric Redundant Execution*

Register renaming and dataflow-driven execution in an out-of-order pipeline make possible fine-grain simultaneous execution of independent instruction streams. Figure 1(b) illustrates SS2, a simple approach to execute a program redundantly over the same out-of-order support logic with two data-independent threads [14]. The instruction fetch stream is replicated into an M-thread (main thread) and an R-thread (redundant thread) at the dispatch stage. Register renaming allows the M and R-threads to execute independently in the out-of-order pipeline. Load and store addresses are calculated redundantly, while the memory access required by a load instruction is only performed by the M-thread. A load-value queue [15] retains the returned load value for later consumption by the corresponding load instruction in the R-thread. A single store is retired to the data cache. The redundantly-computed results from each thread are checked against each other, in program order, in the commit stage. If an error is detected, program execution can be restarted at the faulty instruction by a triggering a soft-exception. It is important to note that SS2 remains an 'I'-way issue pipeline, i.e., the ISQ can only issue 'I' instructions per cycle, whether from the M-thread or the R-thread. Furthermore, SS2 holds redundant instruction instances in dedicated ISQ and ROB entries, as on SS1.

We assume storage arrays, such as register files, caches and TLBs, are protected against soft-errors by error correcting codes (ECC). We assume the in-order instruction fetch stages can be deeply pipelined to permit their construction on slower, but electrically more robust, circuits. In addition, we assume the less performance-critical finite state machine sequencers (e.g., for state recovery or TLB replacement) can be protected via circuit-level techniques that trade speed for soft-error



(a) Integer



(b) Floating-point

Fig. 2. Performance impact of redundant execution on SS2.

tolerance. The primary remaining concern is, therefore, in protecting the out-of-order execution pipeline that comprises the functional units, ISQ, LSQ, ROB, etc. Neither information-level nor circuit-level techniques mentioned above are applicable to the out-of-order pipeline due to the pipeline's extreme performance sensitivity and structural complexity.

2.3. SS2 Performance Penalty

Graphs (a) and (b) in Figure 2 establish the baseline SS2 performance penalty, as compared with SS1, on integer and floating-point benchmarks, respectively. In each graph, the benchmarks are sorted according to their SS1 IPC. The benchmarks are further divided into high-IPC versus low-IPC subsets. Overall, the SS2 IPC is 15% lower than SS1 for integer benchmarks and 32% lower for floating-point benchmarks. As expected, the difference in IPC is more significant for high-IPC benchmarks, however, a clear performance loss, 10% for integer and 31% for floating-point, is also experienced by SS2, even for benchmarks with an SS1 IPC well below four (half the pipeline's issue width).

3. FACTORIAL DESIGN ANALYSIS

The analysis in this section establishes the importance of individual bottlenecks in SS2, as well as their interactions. The performance difference between SS2 and SS1 cannot be explained any single bottleneck. The contributing bottlenecks vary from benchmark to benchmark, and even between different phases of the same benchmark. We reduce the factors contributing to performance bottlenecks in SS2 into three resource-related factors and one mechanism-related factor which can be independently applied to the SS2 design:

- X: issue width and functional unit bandwidth
- C: ISQ and ROB capacity
- B: decode and retirement bandwidth
- S: staggering redundant threads (mechanism related)

X-factor: The X-factor represents the issue and functional unit bandwidth, a limit on raw/peak instruction execution throughput of a superscalar pipeline. We combine issue and functional unit bandwidth into a single factor because they must be scaled together in a balanced design. A fundamental property of the X-factor is that if an application on average consumes more than one-half of the issue and functional unit bandwidth in SS1, the same application cannot achieve the equal performance in SS2. In designs which directly address the X-factor, the functional unit count and issue bandwidth double, with corresponding increases in die area from more functional units, increased bypass paths, and additional issue logic complexity.

C-factor: The C-factor represents the ISQ and ROB capacity. The C-factor governs a pipeline's ability to exploit instruction-level parallelism (ILP) by executing instructions out-of-order. Again, we combine the ISQ and ROB capacity into a single factor because they are scaled together in commercial designs. Sharing the ISQ and ROB between two threads has the same effect as halving the out-of-order window size. ISQ and ROB capacity are important to the low-IPC benchmarks because these benchmarks may require the full capacity of the ISQ and ROB to extract maximally the available instruction level parallelism. Although it would be expensive to achieve "SS2+C" by physically doubling the capacity of the ISQ and ROB, we discuss microarchitectural techniques to achieve the same effect in the context of redundant execution.

B-factor: The B-factor represents the various bandwidth limits at the interface to the ISQ and ROB. These include the decode bandwidth, the completion bandwidth and the retirement bandwidth. In SS2, these bandwidth limits impose a bottleneck because they must be shared between the redundant M- and R-threads. We expect the B-factor to have the least observed impact on SS2 performance, because in a balanced design the X and C factors are generally tuned to become limiting factors before saturating the other bandwidth limits. In designs that relax the B-factor, the effective decode and retirement bandwidth available to each thread is doubled.

S-factor: Besides the three resource-related factors, we introduce a fourth factor, stagger (S-factor), that improves the performance of SS2 but is not applicable to SS1. Several previous proposals for redundant execution on SMT advocate maintaining a small stagger (128~256 instructions) between the progress of the leading M-thread and the trailing R-thread [15,16]. The stagger serves to hide cache-miss latencies from the R-thread and to allow the M-thread to provide branch prediction information to the trailing thread. Some elasticity between the progress of the two threads also permits better resource scheduling. A benchmark with a low average IPC can include short program phases with high IPC; SS2 bottlenecks during these instantaneous high-IPC regions rob the application of performance which cannot be made up for during later low-IPC regions. Staggering the instantaneous high-IPC regions of the two redundant threads avoids this bottleneck. The reported benefit of allowing stagger is typically around a 10% increase in the IPC of redundant execution [15].

3.1. 2-k Factorial Analysis

To examine the SS2 design space systematically, we apply 2-k factorial analysis to the CPI[1] (cycles per instruction) of the sixteen configurations. The 2-k factorial analysis disentangles the performance impact of each factor and identifies the interactions between multiple factors. (Refer to [3] for a detailed explanation of 2-k factorial analysis.) The result of the 2-k factorial analysis is a quantitative measure of the average effect of changing each of the four factors individually. In addition, factorial analysis gives the effect due to interactions when changing any two factors, three factors, or all four factors together. Table 2 summarizes the significant effects (> 3%) for high and low-IPC integer and floating-point benchmarks. For rows showing an individual factor, the value under the "Effect" column shows the average percentage CPI decrease (performance increase) between all configurations with the factor enabled versus all configurations with the factor disabled.

The significant factors to increase performance on high-IPC floating-point benchmarks are X, C and S. In other words, their effects sum nearly linearly when more than one factor is enabled. In another example, the two most significant factors for high-IPC integer benchmarks are X and S. In this case, X and S have a significant and negative interaction. In other words, the effect of implementing both X and S is that the added performance improvement will be less than the simple sum of the two effects and some of the performance gain from X could instead be obtained by adding S. The interactions between the three factors (not shown) are insignificant.

1. This is because CPI is additive when normalized with respect to the instruction count (a constant across the microarchitectures), while IPC is not.

Table 2. The main factors and interactions that increase performance in redundant execution.

	Factor	Effect (% CPI change)
Integer: High	X	17.1
	S	7.1
	X+S	-5.2
Integer: Low	X	5.2
	C	4.3
	S	3.1
Floating-point: High	X	33.5
	C	9.9
	S	6.4
Floating-point: Low	C	27.0
	S	6.2
	X	4.6
	S+C	-3.9

4. CONCURRENT ERROR DETECTION DESIGN SPACE

In this section, we survey prior representative concurrent error detecting superscalar microarchitecture proposals and classify their performance with respect to a baseline non-redundant baseline microarchitecture using the four performance factors defined in Section 3. Table 3 summarizes the microarchitectures in terms of the performance factors and presents the performance of an SS2 model that includes those factors, relative to the baseline SS2. We refrain from estimating the performance of DIE [12], because it only partially addresses the X-factor.

4.1. SS2: Dual-Use

A straightforward design for concurrent error detection is the Dual-Use microarchitecture (this corresponds to the SS2 design point discussed earlier) [14]. This design replicating instructions at fetch, decode, rename, and executing the instructions as two symmetric data-independent threads, and comparing each replicated instruction before retirement, as described in Section 2.2. This design splits the fetch, decode and retirement bandwidth between two threads. Furthermore, the two threads must split the queue capacities and share issue bandwidth. Since both threads are considered equally for out-of-order issue, the microarchitecture creates no incentives for stagger between the two threads. The Dual-Use microarchitecture's key benefits are that it can be implemented with little overhead as a new mode on existing non-redundant microarchitectures and that it supports soft

Table 3. Summary of surveyed microarchitectures and the performance of the corresponding 2-k factorial model as a percentage over baseline SS2 performance for low and high IPC benchmarks.

Name	Designation	Added Cost	Int Performance (%)		FP Performance (%)	
			Low	High	Low	High
Dual-Use	SS2	None	0	0	0	0
SRT	SS2+S	Result buffer, branch queue	6	13	12	9
DIVA	SS2+X+C+B	In-order checker w/FUs	14	30	44	61
DIE	SS2+X	Instruction reuse buffer	N/A	N/A	N/A	N/A
O3RS	SS2+C+B	Result Buffer	6	4	35	11
SHREC	SS2+C+B+S	Result buffer	11	16	40	16

error recovery through the existing exception rollback hardware.

4.2. *SS2+S: SRT*

The Simultaneous and Redundantly Threaded (SRT) proposal relaxes the lockstep fetch, decode, and retirement requirements of SS2 in a simultaneous multithreaded (SMT) processor [15]. SRT fetches redundant threads with a modified ICount fetch policy designed to encourage a stagger between the two threads. The main execution retires instructions and frees out-of-order execution resources ahead of the redundant thread, therefore the stagger is limited by number of values which can be stored for comparison between the two threads. The second execution is assisted by transferring all branch outcomes and load values from the first thread. The two threads split decode and retirement bandwidth, ROB occupancy, and the issue bandwidth. As a result, the primary performance improvement over SS2 is characterized by the stagger between the threads. In addition, the second execution requires less issue bandwidth, because the pre-computed branch outcomes allow it to execute non-speculatively.

4.3. *SS2+X+C+B: DIVA*

The Dynamic Implementation Verification Architecture (DIVA) proposal adds a separate, dedicated in-order checking unit with the retirement phase of a main out-of-order pipeline [1]. Instructions, with their operands from the first execution, re-execute in the checker. Since instructions are neither replicated within the main pipeline nor decoded a second time, DIVA effectively gives full queue capacity and decode bandwidth to both threads. The checker independently issues instructions to a set of dedicated functional units, which also provides full issue bandwidth to each thread. As a result, DIVA incurs a small performance penalty from the checker latency, but adds the hardware cost of a checker unit with its associated functional units, to the microarchitecture.

4.4. *SS2+X: DIE*

The Dual Instruction Execution (DIE) proposal [12] reduces the X-factor losses by eliminating re-execution for some instructions with common opcodes and operands that have already been executed and checked. This elimination is done through an instruction reuse buffer, an added structure that caches the results of recently executed and checked instructions with their operands. For instructions that hit in the reuse buffer during the main thread execution, DIE dissolves the need for redundant issue. For instructions that miss, DIE issues the instruction redundantly, as in SS2. DIE trades issue bandwidth and functional units for the area and complexity overheads associated with integrating an instruction reuse buffer into the core.

4.5. *SS2+C+B: O3RS*

Out-of-order Reliable Superscalar (O3RS) addresses the capacity and bandwidth bottlenecks in SS2 [9]. The microarchitecture replicates the decode mechanism for both threads, but only inserts a single entry for each instruction into an ECC-protected ROB. Thus, it provides the same capacity as SS1. Each instruction in the issue queue and ROB issues twice to the functional units and when both copies have completed, the instruction becomes eligible for retirement. Since only one instruction entry retires, the full per-thread retirement bandwidth is also preserved. However, since both instruction instances must issue before being removed from the ISQ, O3RS cannot support a stagger between the threads without directly increasing the ISQ occupancy.

4.6. *SS2+C+B+S: SHREC*

SHared REsource Checker (SHREC) [21] processes main thread instructions in the out-of-order core and checks the execution in-order checking over a common set of functional units. SHREC fetches and decodes instructions once with slower, pipelined soft error-tolerant circuits. However, it executes instructions once in the main out-of-order pipeline, as done in a baseline superscalar microarchitecture. When instructions have completed main execution, they are reissued, in-order, by a special checker unit during idle main thread issue slots. As a result, the redundant thread maintains a variable stagger behind the main execution when there is instantaneous contention and issues when resources are free. Furthermore, the threads do not compete for out-of-order queue capacity, because they share a single ROB entry but the R-thread does not use the out-of-order issue queue. When the instruction's second execution completes, the results are compared with the main execution and instructions retire in-order, as usual.

5. CONCLUSION

In a balanced single-thread superscalar design, supporting a second redundant thread incurs a performance loss due to contention for bandwidth and capacity in the datapath, (e.g., issue bandwidth and the number of functional units, issue queue, reorder buffer size, and decode/retirement bandwidth). Relaxing any single resource bottleneck cannot fully restore the lost performance. On the other hand, it is possible to reclaim significant performance without requiring additional issue and functional unit bandwidth. In this paper, we explored the performance factors in the design space of concurrent error detecting superscalar processors using a 2-k factorial analysis. We surveyed several prior proposals in this design space and classified them over these axes. Overall, an effective design should address all factors in an economical and balanced fashion.

REFERENCES

[1] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd International Symposium on Microarchitecture*, November 1999.

[2] Wendy Bartlett and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January-March 2004.

[3] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. John Wiley and Sons, Inc., 1978.

[4] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997.

[5] Neil Cohen, T.S. Sriram, Norm Leland, David Moyer, Steve Butler, and Robert Flatley. Soft error considerations for deep-submicron CMOS circuit applications. In *IEEE International Electron Devices Meeting: Technical Digest*, pages 315–318, December 1999.

[6] Joel S. Emer. EV8: the post-ultimate alpha. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001. Keynote address.

[7] Mohamed Gomaa, Chad Scarbrough, T.N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[8] T. Juhnke and H. Klar. Calculation of the soft error rate of submicron CMOS logic circuits. *IEEE Journal of Solid State Circuits*, 30(7):830–834, July 1995.

[9] Avi Mendelson and Neeraj Suri. Designing high-performance and reliable superscalar architectures: The Out of Order Reliable Superscalar O3RS approach. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2000.

[10] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 99–110, May 2002.

[11] Shubhendu S. Mukherjee, Christopher T. Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, Dec 2003.

[12] Angshuman Parashar, Sudhanva Gurumurthi, and Anand Sivasubramaniam. A complexity-effective approach to alu bandwidth enhancement for instruction-level temporal redundancy. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[13] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 33rd International Symposium on Microarchitecture*, December 2000.

[14] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[15] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[16] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, June 1999.

[17] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[18] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–398, June 2002.

[19] T.J. Slegel, R.M. Averill III, M.A. Check, B.C. Giamei, B.W. Krumm, C.A. Krygowski, W.H. Li, J.S. Liptay, J.D. MacDougall, T.J. McPherson, J.A. Navarro, E.M. Schwarz, K. Shum, and C.F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, March/April 1999.

[20] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, Boston, Massachusetts, October 2004.

[21] Jared C. Smolens, Jangwoo Kim, James C. Hoe, and Babak Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*, December 2004.

[22] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.