Reducing the Soft-Error Rate of a High-Performance Microprocessor

Unlike traditional approaches, which focus on detecting and recovering from faults, the techniques introduced here reduce the probability that a fault will cause a declared error. The first approach reduces the time instructions sit in vulnerable storage structures. The second avoids declaring errors on benign faults. Applying these techniques to a microprocessor instruction queue significantly reduces its error rate with only minor performance degradation.

Christopher T. Weaver
Joel Emer
Shubhendu S. Mukherjee
Intel

Steven K. Reinhardt
University of Michigan
Intel

Single-bit upsets from transient faults have emerged as a key challenge in microprocessor design. These faults arise from energetic particles—such as neutrons from cosmic rays and alpha particles from packaging material—generating electron-hole pairs as they pass through a semiconductor device. Transistor source and diffusion nodes can collect these charges. A sufficient amount of accumulated charge can invert the state of a logic device, such as an SRAM cell, a latch, or a gate, thereby introducing a logical fault into the circuit's operation.[1] Because this type of fault does not reflect a permanent device failure, we call it soft or transient.

Soft errors will be an increasing burden for microprocessor designers as the number of on-chip transistors continues to grow exponentially. The raw error rate per latch or SRAM bit is projected to remain roughly constant or decrease slightly for the next several technology generations.[2,3] Thus, unless we add error protection mechanisms or use a more robust technology (such as silicon-on-insulator), a microprocessor's error rate will grow in direct proportion to the number of devices we add to it in each succeeding generation.

A direct approach to reducing error rates involves adding error correction or recovery mechanisms to a design. Unfortunately, these mechanisms come at a significant cost in power, performance, and area. Furthermore, these techniques might be overkill for most of the microprocessor market, which requires good reliability but not bulletproof operation. This article focuses on an alternative approach to reducing a microprocessor's soft-error rate: reducing the likelihood that a transient fault will cause the processor to declare an error

0272-1732/04/$20.00 © 2004 IEEE

condition. We propose two variations on this theme, with specific application to a microprocessor instruction queue.

Effects of soft errors

Figure 1 illustrates the possible outcomes of a single-bit fault. (In this article, we consider only single-bit faults; multiple-bit faults are orders of magnitude less common.) Outcomes labeled 1, 2, and 3 indicate nonerror conditions. Outcome 4 represents the most insidious form of error, silent data corruption (SDC), in which a fault induces the system to generate erroneous outputs. To avoid SDC, designers often employ basic error detection mechanisms, such as parity. These mechanisms provide fail-stop behavior but no error correction, leading to detected unrecoverable errors (DUEs).

We subdivide DUE events according to whether the detected fault would affect the final outcome of the execution. We call benign detected faults false DUE events (outcome 5 in Figure 1) and others true DUE events (outcome 6). In most situations, it is impossible for a processor to determine at the time a fault is detected whether it is benign. The conservative approach is to signal all detected faults as errors (for example, via a machine-check exception).

Interestingly, protecting a structure with an error detection mechanism increases the structure's overall error rate contribution. Faults that would have been benign will be signaled, generating false DUE events. These faults are in addition to errors that would have been SDC events and are now true DUE events.

Microprocessor vendors typically set target maximum SDC and DUE rates for their processors. We can calculate SDC and DUE rates for a particular processor structure as the product of the raw error rate of the structure's storage cells (that is, the single-bit-fault rate) and the probability that a single-bit fault will cause an SDC event (outcome 4 in Figure 1) or a DUE event (outcomes 5 or 6). We call the latter probabilities the device's architectural vulnerability factors (AVFs). The SDC and DUE rates for the entire processor are simply the sums of the rates of all the structures in the processor.

Mukherjee et al. introduced the concept of architecturally correct execution (ACE) to compute the AVFs of microprocessor structures.[4] Architecturally correct execution is any execution that generates results consistent with the system's correct operation as observed by a user. Individual instructions can generate incorrect results without violating ACE if those results are never observed outside the system (for example, they are dead values). We call a bit an ACE bit if it contains information that will affect the program's final outcome; otherwise, it is an un-ACE bit.

An unprotected storage cell's SDC AVF is the fraction of cycles in which it contains an ACE bit. For a storage cell covered by an error detection scheme with no recovery, the fraction of cycles in which it contains an ACE bit corresponds to its true DUE AVF (that is, the probability of a true DUE event given a single-bit fault). The false DUE AVF reflects the probability that an unrecoverable error is signaled for an un-ACE bit.

The total DUE AVF is the sum of the true and false DUE AVFs. Our two techniques for reducing soft error rates target two different components of overall AVF: true errors that contribute to SDC and/or the true DUE AVF, and false errors that contribute to the false DUE AVF.

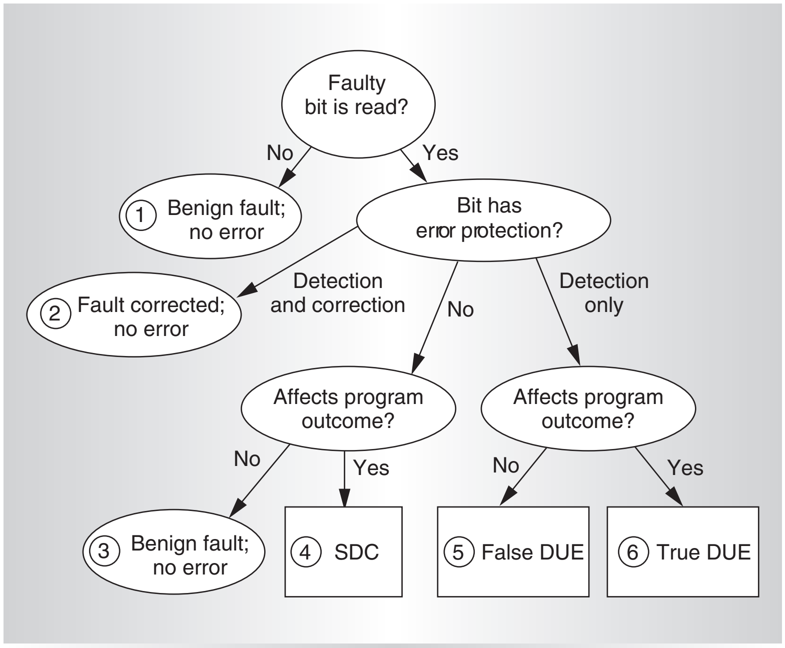

Figure 1. Classification of possible outcomes of a faulty bit in a microprocessor. (SDC: silent data corruption; DUE: detected unrecoverable error)

Reducing exposure to radiation

Our first technique reduces AVF by reducing the exposure of ACE objects to neutron and alpha radiation. The basic idea is to keep ACE objects in protected memory and fetch them to vulnerable storage only when needed. If the vulnerable storage has no protection, its error rate will contribute to the processor's SDC rate. If the vulnerable storage has error detection but no recovery, its error rate will contribute to the processor's DUE rate.

For example, microprocessors often aggressively fetch instructions from protected memory, such as main memory protected by error-correcting codes or a read-only instruction cache protected by parity (but recoverable because instructions can be refetched from main memory on a parity error). However, pipeline hazards can cause these instructions to stall in the instruction queue. The longer such instructions reside in the instruction queue, the greater the likelihood that a neutron or alpha particle will strike them.

We propose that processors squash (or remove) instructions from the instruction queue after events resulting in pipeline stalls and refetch the instructions when the pipeline resumes execution. We specifically examine squashing after data cache load misses. (Tullsen and Brown proposed a similar policy to improve performance in multithreaded machines.[5]) Because we examine an in-order machine, squashing all instructions after a load miss should have minimal performance impact. At the same time, it should lower AVF by reducing instructions' exposure to neutron and alpha strikes.

Quantifying performance-reliability tradeoffs

Traditionally, the fault tolerance community has used the term mean time to failure (MTTF) to reason about error rates in processors and systems. Although MTTF provides a metric for error rates, it does not allow us to reason about the tradeoff between error rates and a processor's performance. We introduce the concept of mean instructions to failure as one approach to reason about this tradeoff. MITF tells us how many instructions a processor commits, on average, between two errors. MITF is related to MTTF as follows:

$$\text{MITF} = \frac{\text{number of committed instructions}}{\text{number of errors encountered}} = \frac{\text{number of committed instructions}}{\dfrac{\text{total execution time in cycles}}{\text{frequency} \times \text{MTTF}}} = \text{IPC} \times \text{frequency} \times \text{MTTF}$$

where IPC is the number of instructions per cycle.

As with SDC and DUE rates, there are corresponding SDC and DUE MTTFs and MITFs. Hence, for example, a processor running at 2 GHz with an average IPC of 2 and a DUE MTTF of 10 years would have a DUE MITF of 1.3×10^{18} instructions.[6]

A higher MITF implies a greater amount of work done between errors. Assuming that increasing MITF is desirable within certain bounds, we can use MITF to reason about the tradeoff between performance and reliability. Because MTTF = 1/(raw error rate × AVF), we have

$$\text{MITF} = \frac{\text{IPC} \times \text{frequency}}{\text{raw error rate} \times \text{AVF}} = \frac{\text{frequency}}{\text{raw error rate}} \times \frac{\text{IPC}}{\text{AVF}}$$

Thus, at a fixed frequency and raw error rate, MITF is proportional to the ratio of IPC to AVF. More specifically, SDC MITF is proportional to IPC/SDC AVF and DUE MITF is proportional to IPC/DUE AVF. Mechanisms that reduce both AVF and IPC, such as the one proposed in the previous section, might be worthwhile only if they increase MITF—that is, if they increase the IPC-to-AVF ratio by reducing AVF relative to the base case to a greater degree than they reduce IPC.

Although we can use MITF to reason about performance versus AVF for incremental changes, we must be cautious not to misapply it. For example, it could be argued that doubling processor performance while reducing MTTF by 50 percent is a reasonable tradeoff because MITF would remain constant. However, this explanation might not be adequate for customers who see their equipment fail twice as often.

Reducing false DUE by tracking

Our second approach to reducing the soft error rate addresses false DUE AVF. False DUE events arise from detected unrecoverable errors that would not have affected the system's final output in the absence of error detection. For example, a fault in a wrong-path instruction in the instruction queue would not affect any user-visible state. However, the processor is unlikely to know in the issue stage whether or not an instruction is on the correct path, and thus the processor might be forced to raise a machine check exception on detecting any instruction queue parity error. Figure 1 relates false DUE events to other possible fault outcomes.

Our earlier classification[4] identifies three sources of false errors for the instruction queue:

- *Instructions whose results the microarchitecture will never commit.* Examples are wrong-path instructions and predicated-false instructions.
- *Instruction types that are neutral to errors.* No-ops, prefetches, and branch prediction hint instructions, for example, don't affect correctness. Consequently, faults in bits other than the opcode bits will not affect a program's final outcome.
- *Dynamically dead instructions.* These instructions generate values that ultimately don't affect the result. We classify dynamically dead instructions as first-level or transitive. First-level dynamically dead (FDD) instructions are those whose results are not read by any other instruction. Transitive dynamically dead (TDD) instructions are those whose results are used only by FDD instructions or other TDD instructions.

The key to addressing false DUE events is to propagate error information from the point where an error is detected to a later point where the processor can determine whether the error will affect the system's output. We introduce a new bit for this purpose called the π bit, or the pi (possibly incorrect) bit.

A π bit is logically associated with each instruction as it flows down the pipeline. We initially clear the π bit to indicate the absence of any error. On detecting an error, a pipeline stage sets the affected instruction's π bit instead of raising a machine check exception. Subsequently, the instruction issues and flows down the pipeline. When the instruction reaches the commit point, we can determine whether the instruction was on the wrong path. If so, we ignore the π bit, avoiding a false DUE event if the bit was set. If not, we have the option to raise the machine check error at the instruction's commit point. A strike on the π bit itself could result in a false DUE event.

Fujitsu's recent Sparc processor propagates parity errors along the pipeline in a similar fashion.[7] However, instead of tracking false DUE events, Fujitsu's Sparc uses the parity bit to restart its pipeline from the instruction that received the parity error.

We can avoid false DUE events resulting from dynamically dead instructions in the register file by propagating π bits to registers. Instead of raising an error if an instruction's π bit is set, we can transfer the instruction's π bit to its destination register. If the instruction is dead (FDD), no subsequent instruction will read this register, and the register's π bit will not be examined. We thus avoid raising a false error. If a subsequent instruction reads a register with the π bit set, we can signal an error. This mechanism is similar to previous mechanisms for tracking exceptions on speculatively executed instructions, such as Rogers and Li's poison bit,[8] Mahlke et al.'s speculative modifier bit,[9] and the Itanium architecture's NaT (not a thing) bit. The proposed, but eventually canceled, Alpha 21464 microarchitecture had a similar mechanism to replay instructions dependent on a load miss.

We can suppress false errors on some TDD instructions by propagating the π bit through instructions instead of raising an error whenever a source register's π bit is set. Specifically, an instruction reading a register could OR the π bits of all its source registers with its own π bit and carry it along the pipeline. This approach propagates the π bit along dependence chains. We can signal an error when a set π bit propagates to an instruction that attempts to write to memory or an I/O device.

If we further propagate the π bit from instructions and registers to memory values, we can track false DUE events in memory structures, such as store buffers and caches, as well as additional TDD instructions. For example, we can attach a π bit to each cache block and transfer each store instruction's π bit to the cache block. Subsequently, when a load reads the cache block, it transfers the π bit to its destination register. In this situation, an error is signaled if data with a set π bit propagates beyond the cache—for example, because of a cache writeback or an I/O operation.

We don't expect all hardware structures in a processor or an entire system to be populated with π bits. When data or instructions covered by a π bit affect architectural state in a structure not covered by a π bit, the system can no longer track the possible error. At this point, the implementation should signal an error if the π bit is set.

Suppressing false errors on neutral instruction types

The system need not raise an error on non-opcode bits of instructions such as no-ops, prefetches, or branch prediction hints. However, to identify such instructions, the hardware must decode the instruction at every place it wants to avoid a false error. Instead, we propose using another bit called the anti-π bit, which is associated with each instruction when we decode it. We set the anti-π bit for neutral instruction types and clear it for others. Then, when the instruction queue gets a parity error on an entry's non-opcode bits, it identifies neutral instructions using the anti-π bit and does not set the π bit on that instruction. In other words, the anti-π bit neutralizes the π bit for those entries. Alternatively, the instruction queue could set the π bit but carry both the anti-π bit and the π bit to the retire unit and follow the appropriate decision there. The anti-π bit can be generalized to other activities that do not affect a program's correctness. For example, we could attach an anti-π bit to each memory request generated by a hardware data prefetcher.

Postcommit error-tracking buffer

One shortcoming of using π-bit propagation to identify dead instructions is that the system can no longer determine which instruction originally caused the error. This lack of information can complicate some recovery schemes. We devised an alternate mechanism, the postcommit error-tracking (PET) buffer, which also avoids signaling errors on a subset of FDD instructions and can precisely determine the offending instruction that could have encountered a true error. An *n*-entry PET buffer contains an entry for the *n* most recently retired instructions along with their π bits. When the PET buffer is full, the system examines the π bit of the oldest instruction about to be evicted.

If its π bit is set, the hardware scans the instructions in the PET buffer to determine if its result was overwritten by a subsequent instruction without an intervening read. If so, the instruction is FDD and the error can be ignored. However, if the PET buffer cannot verify that the instruction was an FDD instruction, it must signal an error. Obviously, the PET buffer's coverage of false errors on FDD instructions depends on the PET buffer's size. The PET buffer is similar to but much simpler than the history buffer described by Smith and Plezkun.[10] The hardware must scan the PET buffer only when a π bit is set for an instruction being evicted. Such scans should not affect performance because errors in an individual system occur infrequently (on the order of days).

Evaluation

We evaluated our proposals through detailed modeling of an Itanium2-like IA64 processor[11] scaled to current technology, using the Asim framework.[12] We drove this model with carefully selected samples of each SPEC CPU2000 benchmark.

Table 1 shows how the average IPC and AVFs change when we squash all instructions in the 64-entry instruction queue after a load miss in the L1 or L0 caches. (The processor we use for our evaluation has three levels of caches: L0, L1, and L2. L0 cache misses have a latency of 10 cycles, and L1 cache misses have a latency of about 25 cycles.) When we squash instructions on load misses in the L1 cache, the IPC decreases by only 1.7 percent, for a corresponding reduction of 24 percent in SDC and 18 percent in DUE AVFs. However, when we squash instructions on L0 misses, the IPC decreases by 10 percent, for a corresponding reduction of only 35 percent

Table 1. Effect of squashing on IPC and instruction queue's SDC and DUE AVFs. Numbers are averaged across all benchmarks.

Design point	IPC	SDC AVF (%)	DUE AVF (%)	IPC/SDC AVF	IPC/DUE AVF
No squashing	1.21	29	62	4.1	2.0
Squash on L1 load misses	1.19	22	51	5.6	2.3
Squash on L0 load misses	1.09	19	48	5.7	2.3

in SDC and 23 percent in DUE AVFs. Squashing on L1 misses appears more profitable because the SDC MITF (proportional to IPC/SDC AVF) and DUE MITF (proportional to IPC/DUE AVF) go up 37 and 15 percent, respectively. Squashing on L0 misses adds very little MITF gain over squashing on L1 misses alone.

Figure 2 quantifies how our π-bit techniques avoid false errors and thus lower the instruction queue's false DUE AVF. Propagating the π bit to the commit point allows us to avoid false errors on wrong-path and falsely predicated instructions. On average, this reduces the instruction queue's false DUE AVF by 18 percent. However, as the figure shows, the impact is greater for integer benchmarks, which have a higher fraction of such instructions. In contrast, the anti-π bit has a larger impact on floating-point benchmarks than integer benchmarks (60 percent versus 35 percent reduction in false DUE AVF) because of the larger impact of no-ops and prefetches. Overall, the anti-π bit reduces the false DUE AVF by 49 percent.

Figure 2 also shows the effects of a 512-entry PET buffer. This relatively small PET buffer works because instructions that overwrite a register without an intervening read often occur within a few hundred committed instructions. On average, the 512-entry PET buffer accounts for about 32 percent of the FDD instructions, reducing the false DUE AVF by another 3 percent.

The remaining techniques shown in Figure 2 (π bit on the register file, π bit until store commit, and π bit until I/O commit) improve coverage further but don't allow precise determination of the corrupted instruction. Adding the π bit to the register file and declaring an error on a register read with a set π bit tracks all FDD instructions, reducing the instruction queue's false DUE AVF an additional 11 percent beyond the PET buffer's reduction. If we carry the π-bit information through the pipeline and examine the π bit only when a

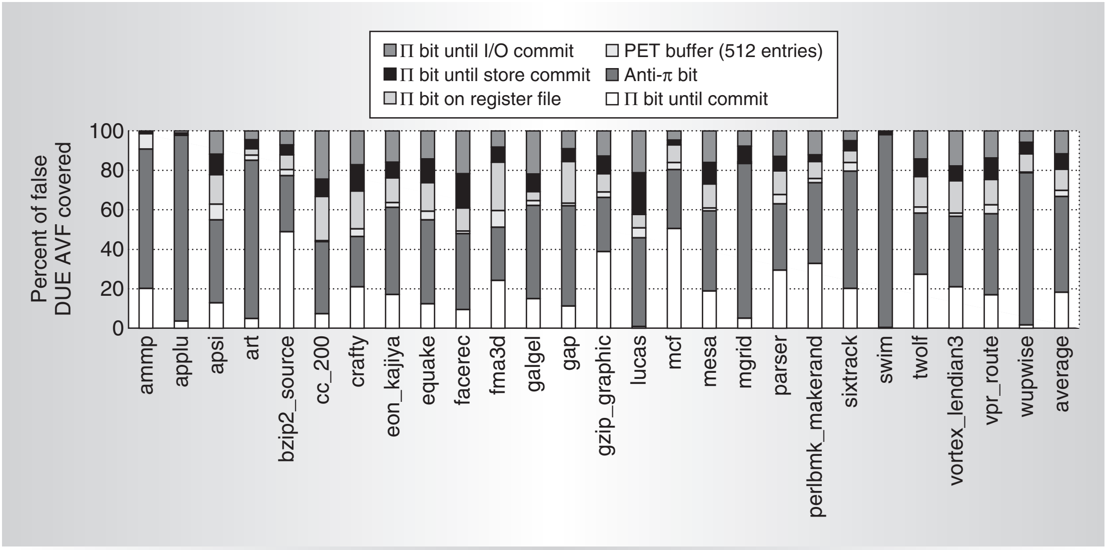

Figure 2. Coverage of the instruction queue's false DUE AVF using various tracking techniques.

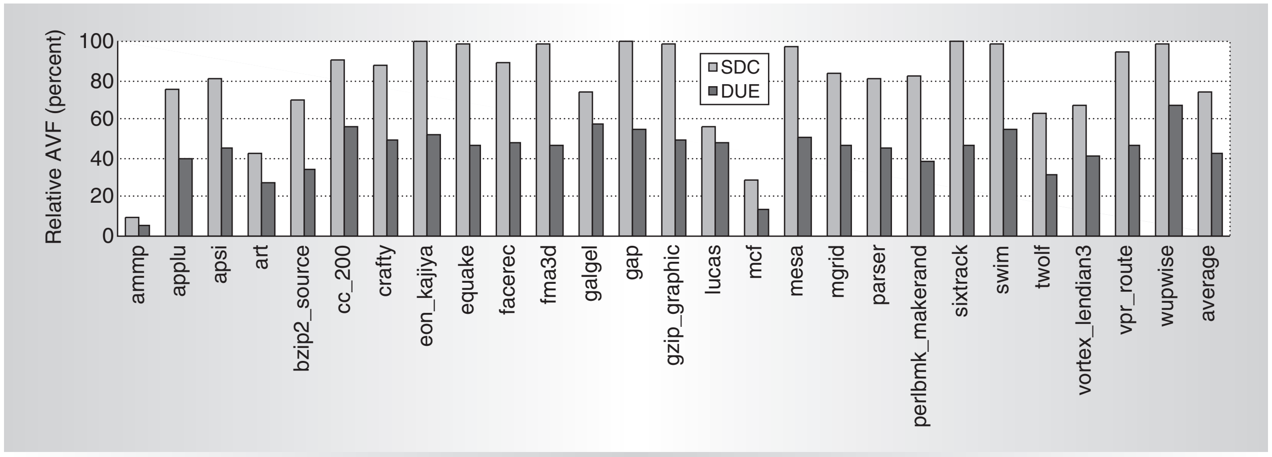

Figure 3. Effect of exposure and false DUE reduction techniques on an instruction queue's SDC and DUE AVFs. (Relative SDC AVF = SDC AVF with optimizations/SDC AVF with no optimizations. Relative DUE AVF is defined similarly.)

store commits its data or a load attempts to get its data from the store buffer, we can avoid false errors resulting from TDD instructions tracked via registers. On average, this extension reduces the instruction queue's false DUE AVF by another 8 percent. Finally, if we carry the π bit through the entire processor and memory system, we must signal an error only when the processor interacts with an I/O device. This would remove the remaining false errors by avoiding signaling errors on FDD and TDD instructions tracked via memory, further reducing the instruction queue's false DUE AVF by 12 percent.

Figure 3 shows the result of combining our two techniques to reduce the overall SDC AVF of an unprotected instruction queue and the DUE AVF of a parity-protected instruction queue. We combine instruction squashing on an L1 cache miss with the π-bit implementation that carries the π bit until the store commit point. The latter technique applies to the parity-protected queue only, and does not impact IPC. We get an average improvement of 26 percent in SDC AVF on the unprotected queue from squashing alone. Combining the squashing and π-bit tracking mechanisms on a parity-protected queue gives a 57 percent reduction in DUE AVF. In both cases, squashing results in an average IPC reduction of 2 percent.

The techniques we've described here complement traditional approaches to fault detection and fault tolerance. Reducing the exposure of ACE objects in vulnerable structures may allow designers to avoid the cost of fault detection mechanisms, particularly in systems with moderate reliability requirements. Avoiding false unrecoverable errors may allow simpler fault detection mechanisms to suffice in place of more expensive fault recovery schemes. We've illustrated these techniques by applying them to an instruction queue, but they can also be used to reduce the AVF of other processor structures such as the register file. MICRO

References

1. J.F. Ziegler et al., "IBM Experiments in Soft Fails in Computer Electronics (1978–1994)," *IBM J. Research and Development*, vol. 40, no. 1, Jan. 1996, pp. 3-18.
2. S. Hareland et al., "Impact of CMOS Scaling and SOI on Soft Error Rates of Logic Processes," *Proc. 2001 Symp. VLSI Technology*, IEEE Press, 2001, pp. 73-74.
3. T. Karnik et al., "Scaling Trends of Cosmic Rays Induced Soft Errors in Static Latches beyond 0.18 μ," *Proc. 2001 Symp. VLSI Circuits*, IEEE Press, 2001, pp. 61-62.
4. S.S. Mukherjee et al., "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," *Proc. 36th Ann. Int'l Symp. Microarchitecture* (Micro-36), 2003, pp. 29-40.
5. D. Tullsen and J.A. Brown, "Handling Long-Latency Loads in a Simultaneous Multi-

threaded Processor," *Proc. 34th Ann. Int'l Symp. Microarchitecture* (Micro-34), 2001, pp. 318-327.
6. D. Bossen, "CMOS Soft Errors and Server Design," *2002 IEEE Int'l Reliability Physics Symp. Tutorial Notes—Reliability Fundamentals*, IEEE Press, 2002, pp. 121_07.1–121_07.6.
7. H. Ando et al., "A 13 GHz Fifth Generation SPARC64 Microprocessor," *Proc. Int'l Solid-State Circuits Conf.* (ISSCC 03), vol. 1, 2003, pp. 246-491.
8. A. Rogers and K. Li, "Software Support for Speculative Loads," *Proc. 5th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 92), ACM Press, 1992, pp. 38-50.
9. S.A. Mahlke et al., "Sentinel Scheduling for VLIW and Superscalar Processors," *Proc. 5th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 92), ACM Press, 1992, pp. 238-247.
10. J.E. Smith and A.R. Plezkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Trans. Computers*, vol. 37, no. 5, May 1988, pp. 562-573.
11. K. Krewell, "Intel's McKinley Comes into View," *Microprocessor Report*, vol. 15, no. 10, Oct. 2001, p. 1.
12. J. Emer et al., "Asim: A Performance Model Framework," *Computer*, vol. 35, no. 2, Feb. 2002, pp. 68-76.

Christopher T. Weaver is a hardware engineer in the VSSAD labs at Intel's Massachusetts Microprocessor Design Center, where he works in the Fault-Aware Computing Technology Project. His research interests include reliable computing, performance modeling, semicustom and ASIC design, and power and fault modeling. Weaver has BS degrees in electrical engineering, computer engineering, and multidisciplinary studies from North Carolina State University and an MS in computer science and engineering from the University of Michigan.

Joel Emer is an Intel fellow at the Massachusetts Microprocessor Design Center, where he leads the VSSAD Group. His research interests include high-performance microarchitecture, multithreaded processors, processor pipeline organization, processor reliability, and performance-modeling frameworks. Emer has a PhD in electrical engineering from the University of Illinois. He is a Fellow of the IEEE and the ACM.

Shubhendu S. Mukherjee is the manager and technical leader of Intel's Fault-Aware Computing Technology (FACT) Group. His research interests include processor reliability and interconnection networks. Mukherjee has a BTech in computer science and engineering from the Indian Institute of Technology, Kanpur, and an MS and a PhD, both in computer science, from the University of Wisconsin-Madison. He is a member of the ACM and a senior member of the IEEE.

Steven K. Reinhardt is an associate professor of electrical engineering and computer science at the University of Michigan in Ann Arbor and a consultant to Intel. His research interests include processor architecture, memory systems, and computer system simulation. Reinhardt has a BS from Case Western Reserve University and an MS from Stanford University, both in electrical engineering, and a PhD in computer science from the University of Wisconsin-Madison. He is a member of the IEEE Computer Society and the ACM.

Direct questions and comments about this article to Steven K. Reinhardt, EECS Dept., University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109; stever@eecs.umich.edu.