A Taxonomy to Enable Error Recovery and Correction in Software

Vilas Sridharan
ECE Department
Northeastern University
360 Huntington Ave.
Boston, MA 02115
vilas@ece.neu.edu

Dean A. Liberty
Advanced Micro Devices
90 Central St.
Boxborough, MA 01719
dean.liberty@amd.com

David R. Kaeli
ECE Department
Northeastern University
360 Huntington Ave.
Boston, MA 02115
kaeli@ece.neu.edu

Abstract

Over the past several years, reliability research has largely used the following taxonomy of errors: Undetected Errors (sometimes referred to as Silent Data Corruption, or SDC), Detected Uncorrectable Errors (DUE), and Corrected Errors (CE). While this taxonomy is suitable to characterize hardware error detection and correction (EDAC) techniques, it does not provide enough granularity to assess system-wide recovery techniques. Specifically, it does not provide the ability for architects to assess DUE severity, an ability which is crucial to evaluating many error tolerance schemes.

In this work, we present a complete error taxonomy for recovery which classifies the severity of all possible outcomes of an error, including categories for software-recoverable and software-correctable errors. We describe the concepts of error localization and temporal containment which are necessary for hardware to deem an error software-correctable. Finally, we present an evaluation of data poisoning, an error tolerance scheme whose primary benefit cannot be quantified without the ability to distinguish the severity of detected errors.

1. Introduction

Over the past several years, a standard taxonomy for error detection has been established and adopted by most prior error tolerance research [1]. This taxonomy classifies errors as undetected, which can lead to Silent Data Corruption (SDC), detected but uncorrectable (DUE), or corrected (CE). The adoption of this taxonomy has enabled a wide variety of research in the field of error detection and correction (EDAC) by allowing researchers to communicate with one another using a common set of definitions and to make apples-to-apples comparisons of various error tolerance techniques.

This taxonomy, while suitable to assess some hardware-only EDAC techniques (e.g., Error-Correcting Codes), does not possess enough granularity to properly assess many system recovery techniques that span both hardware and software. A standard taxonomy for system recovery will enable researchers to discuss the effectiveness of a broader class of techniques geared towards system recovery. These types of techniques are under-represented in the literature despite their significant benefits to system reliability. This is partly due to the nature of the field of computer architecture, in which researchers tend to focus on either hardware or software, but is also due to the lack of a common architectural interface for system recovery.

The major objective of this work is to provide a new level of standardization in the field of recovery by proposing a common framework to reason about the severity of detected errors. We anticipate that this will enable several positive outcomes. First, standardization will allow architects and designers to properly assess the *system-wide* impact of various error tolerance schemes. Second, by allowing distinctions between the severity of errors, priorities for future error recovery can be highlighted. Finally, by explicitly establishing a category of software-correctable errors, a standardized framework can encourage research and development into software correction techniques for these types of errors, an area that has seen little research to date.

1.1. Motivation

Several conditions motivate the idea that error recovery should be architected and that a full taxonomy of error outcomes should be established. First, while there are a variety of mechanisms that cause errors in modern systems, virtually all errors can be modeled as an unexpected change in the state of an architectural resource. A standard taxonomy can exploit this architectural abstraction by allowing a common recovery framework to handle many classes of error. Second, a combination of Moore's Law scaling and an increased amount of error detection per core suggests that the rate of detected errors will increase in future processors. Techniques such as ECC with interleaving can reduce the rate of uncorrectable errors but cannot eliminate them. Handling hardware-uncorrected errors will therefore be an essential aspect of future system reliability. Finally, it is often desirable to partition error tolerance techniques between hardware and software. This requires a well-defined interface for communication.

1.2. Contributions

In this work, we propose a new error recovery architecture that meets all of these requirements. We propose three categories of detected errors: *Globally Uncorrectable Errors*, *Locally Uncorrectable Errors*, and *Locally Correctable Errors*. We specify the requirements that hardware must meet in order to implement this taxonomy, and evaluate data poisoning, a well-known error tolerance technique whose primary benefit is not quantifiable without this taxonomy.

The major contributions of this work are:

- a full taxonomy of error outcomes for recovery that allows quantification of the severity of detected errors; and
- the introduction of *temporal containment* as a way of providing a class of software-correctable errors.

The rest of this paper is organized as follows: Section 2 discusses background and previous work in reliability techniques; Section 3 introduces the concepts of localization and temporal containment and presents our full error taxonomy; Section 4 presents the results of our evaluation using this taxonomy on data poisoning; and Section 5 discusses future work in this area.

2. Background

Once a full taxonomy has been established, hardware and software can be developed independently to recover from each class of error; where a particular error falls within this taxonomy is specific to each system's implementation. The metric of interest when evaluating system reliability depends on the system requirements. *Availability* is most crucial for some applications, while others may be more concered with *reliability* or the fraction of jobs that complete correctly. Any taxonomy should be general enough to evaluate a technique's impact on both metrics, but specific enough to provide guidelines on how to achieve each possible outcome.

In this work, we define error *recovery* as the ability of a system to continue execution beyond the point of an error. *Full recovery* is the repair of the error condition such that all running programs are unaffected. *Partial recovery* involves the failure of the affected programs while the rest of the system continues unaffected. We define error *correction* as the ability to repair the faulty data in question. System recovery may or may not include correction of the underlying error. Error correction has most often been considered as an attribute of hardware (for instance, Error-Correcting Codes in caches and memory), but correction can take place in either hardware or software.

2.1. Related Work

There has been much previous work on reliability and recovery techniques, starting with mainframes [2] and gradually encompassing lower-end systems. We briefly present a survey of the work in this section. Lockstepped processors (or cores) as implemented on the HP NonStop platform [3] and the IBM G5 processor [4] remain the state of the art in terms of fault coverage. A less robust but significantly less expensive option for fault tolerance is hardware Redundant Multi-Threading (RMT) [5] [6] [7]. A variety of similar software-only or hybrid hardware-software techniques have also been proposed [8] [9] [10]. Lastly, there has been research into hardware containment schemes to limit the propagation of a soft error [11] [12] [13] and checkpointing schemes that allow more extensive rollback recovery than would otherwise be possible [14].

3. Classifying Error Outcomes

For a system to attempt recovery in software, hardware must provide a guarantee that further software execution will not result in data corruption. This section describes the varying levels of confidence that hardware can provide in order to enable further software execution.

3.1. Spatial Containment

If the underlying error is uncorrectable by hardware, any system recovery must be performed in software.

This leads to a dangerous situation: software executing on potentially faulty hardware. Hardware must ensure that further software execution will not exacerbate the error condition. Specifically, the hardware must prevent any breaches of data integrity (silent data corruption) by guaranteeing that the system will halt prior to a breach occurring. A *local* error is an error for which hardware can make this guarantee; other errors are *global* errors. Upon detection of a global error, the only safe course of action for hardware is to halt the system. From an availability perspective, a local error is less severe than a global error.

When an error is presented to the system, the hardware can often associate the data or instruction in error with a scope. We define an error's *spatial scope* as the set of all resources (e.g., an architectural register) that can potentially consume the erroneous data. The error's scope may or may not include the data's producer. In general, identifying the producer is not a well-bounded problem and does not help to guarantee correct operation. The first step in localizing an error is for hardware to uniquely *identify* the scope of the error. Once an error has been associated with a scope, the hardware must further ensure that the error remains *contained* to that scope under all circumstances. Containment implies that the error cannot propagate outside its associated scope. If containment is lost at any point, the hardware must halt the system to prevent data corruption.

If hardware can identify an error's scope and contain the error to that scope, the error is said to be *localized*, and the integrity of data and instructions outside of that scope can be guaranteed. Therefore, the system can attempt recovery in software to limit the damage from the error and avoid system downtime.

3.2. Temporal Containment

Localizing an error confines an error to a set of software resources but does not guarantee the state of those resources at the time of the error; the resource's state must be treated as indeterminate, likely requiring termination of any process using the resource.

To avoid process termination, hardware must provide enough information for the resource to be corrected. This requires hardware to contain the error to a particular set of operations on the resource. An operation that falls within this set is potentially corrupt and its results should not be used. The entire set of potentially corrupt operations is an error's *temporal scope*; the youngest and oldest operations in this scope are the Forward Error Barrier (FEB) and Backward Error Barrier (BEB), respectively. If hardware can guarantee the integrity of operations outside the set [BEB, FEB], correction of the underlying error can allow operations to resume from the Backward Error Barrier, avoiding the need for process termination.

A common method of identifying an error's temporal scope is to provide a precise exception when an error is consumed; this guarantees that the committed system state is not corrupt and that the system can resume operation from this state once the underlying error is corrected. Most current systems do not provide a precise exception on a hardware error since the costs of doing so outweigh the benefits when the DUE rate is low. However, the cost in system downtime can be substantial as the DUE rate increases, and future systems must consider temporal containment to maintain high reliability.

3.3. Error Attribution

An error's spatial and temporal scopes include only architected (software-visible) operations, but errors are often detected on micro-architectural transitions. To achieve error containment, hardware must *attribute* the micro-architectural transition which detected the error to its initiating architectural operation. This can necessitate significant information transfer, as a micro-architectural transition may be physically and temporally distant from its associated architectural event. For example, a read to a memory location on one processor can cause a remote cache copyback; an error during this copyback must be associated with the initiating read. Mechanisms for this communication must be provided by hardware.

Some micro-architectural transitions are not associated with an initiating architectural event. An error detected on an *architecturally invisible* transition may not be localizable or temporally containable (for example, errors detected on cache writebacks). On these transitions, hardware must either halt the system or provide a method to inhibit the error while maintaining data integrity. Data poisoning is one such method [15].

Other micro-architectural transitions can be associated with more than one architectural event. For example, a cache miss can be associated with multiple load operations. In these cases, hardware must associate the micro-architectural transition with the first architectural event (as defined by program order) *regardless of which event actually initiated the transition*. If hardware cannot guarantee this behavior, then the error will not be temporally contained.

3.4. Putting It All Together: A Full Error Taxonomy

A full taxonomy of errors is presented in Figure 1. The first step in the error taxonomy is error detection. Undetected errors can lead to Silent Data Corruption (SDC), which is the most severe outcome for systems valuing reliability. Error detection, though often regarded as a feature of hardware, can take place in either hardware or software. Enabling software-detected errors to reuse the same taxonomy and recovery features as hardware-detected errors can increase error coverage and improve reliability.

Detected errors must be classified according to outcome. If hardware cannot localize an error, it must halt the system to preserve data integrity. This is a Globally Uncorrectable Error (GUE). If hardware can localize but not temporally contain the error, the system must terminate any processes that might consume a resource within the error's spatial scope. This is a Locally Uncorrectable Error (LUE). If hardware can temporally contain an error, software may be able to correct the error and continue execution; this is a Locally Correctable Error (LCE). Finally, an error that can be corrected in hardware is a Corrected Error (CE); these errors have no correctness impact on software.

Although this taxonomy must be architected, the choice of how to handle a specific error is an implementation-specific decision based on factors unique to a particular system such as failure rate targets or cost constraints. For instance, one hardware design might classify a single-bit error in a register as an LCE, while another may correct a single-bit error (CE) and report only multi-bit errors as LCE. Despite the underlying differences, both LCEs can be handled by the same recovery software. This is the power of a standard error taxonomy: it enables system architects to reason about general-purpose recovery and correction algorithms without worrying about the specific details of an error (e.g., the number of bits in error).

4. A Detailed Example: Data Poisoning

As an example of the availability benefits of localization, we perform an analysis of an error tolerance technique to determine its effect on system availability when accounting for localization. We choose a well-known technique, data poisoning, that has been implemented on high-end servers and mainframes over the past several years [15], and quantify its localization benefits on a processor cache. We do not claim our results to be definitive of all implementations of data poisoning; the benefits will be heavily dependent on the particulars of the poisoning implementation, the system configuration, and the workload examined. Rather, we use this as an example to demonstrate the benefits of assessing error severity when analyzing error tolerance techniques.

Parameter	Value
Issue Width	8 instructions
Commit Width	8 instructions
Physical Registers	256 integer / 256 Floating Point
Issue Queue	64 entries
Re-Order Buffer	192 entries
Load-Store Queue	32 loads / 32 stores
Frequency	2 GHz
L1 D-Cache	64 kB, 2 cycle access, 2-way s.a. write-back, allocate-on-miss
L1 I-Cache	32 kB, 2 cycle access, 2-way s.a.
L2 Cache	2 MB, 10 cycle access, 8-way s.a. write-back, allocate-on-miss
Memory Latency	100 ns

Table 1. Simulated Machine Configuration

4.1. Experimental Setup

For our study, we use the detailed CPU model provided in the M5 Simulator System [16] with the parameters shown in Table 1. This CPU models an Alpha 21264-like micro-architecture [17], and the parameters are chosen to be representative of an aggressive, high-performance microprocessor. We have extended the CPU model with a framework to measure AVF using ACE Analysis [1].

Our workload consists of the SPEC CPU2000 integer and floating-point benchmarks, using the single early simulation points given by Simpoint analysis [18]. We execute for 100M instructions to warm the caches, simulate for 100M instructions, and then cool down for 200M cycles. *Cooldown* was introduced in [19] and reduces the number of entries whose ACEness is unknown at the end of the primary simulation phase by continuing execution until their ACEness is determined. This alleviates the edge effects introduced by terminating a workload prior to completion. Our poisoning experiments require tracking addresses within main memory, where data lifetimes are much longer than in caches. Therefore, in addition to the standard cooldown period, we examine the address stream for a further 2B cycles to reduce the amount of Unknown AVF due to addresses in memory that have not been re-referenced during the simulation and cooldown period.

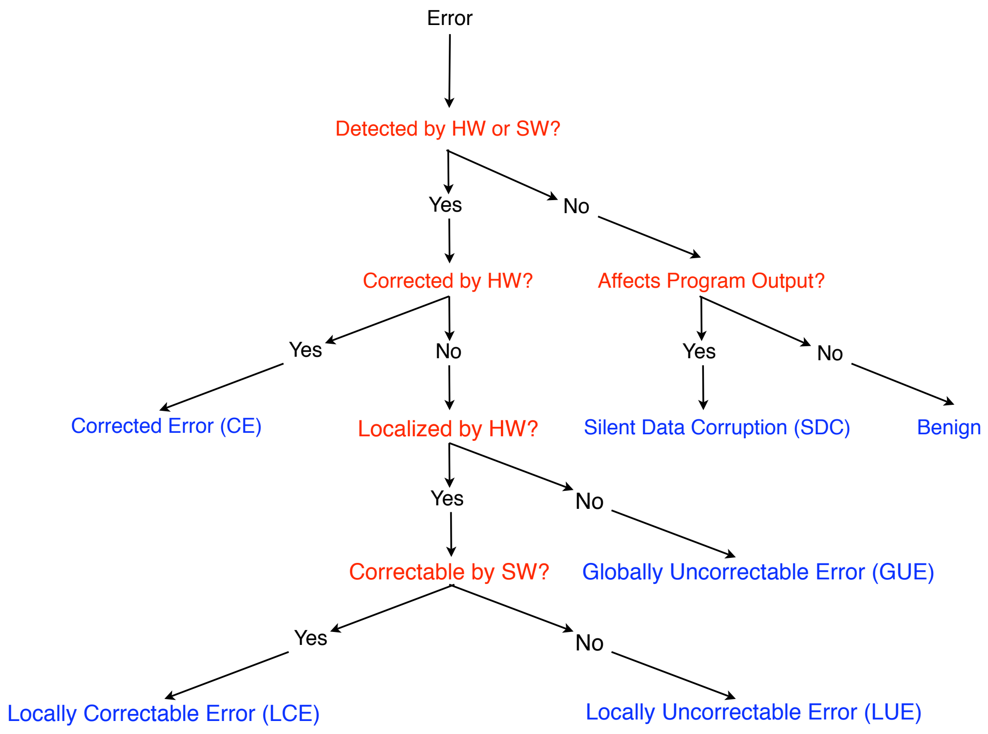

Figure 1. Taxonomy of Error Outcomes

4.2. Cache AVF

Just as a cache line may spend a fraction of its lifetime as ACE and the rest as unACE, a cache line's ACE time can also be split between LUE and GUE. When an error is detected on a read to a clean line, enough information exists to localize the error since the consumer thread is known. If an error is detected when a dirty line is written back, the consumer is not known and the error cannot be localized. Therefore, hardware must halt the system to avoid losing containment of the error. In a multi-processor (or multi-core) environment with inter-cache communication, reads from the local CPU still contain enough information to localize an error in the data. Read requests from a remote cache, however, cause the preceeding ACE time to be GUE if the local cache has no way of communicating to the consumer that the line is in error.

Figure 2 shows the AVF of our processor's L2 cache across several of the SPEC CPU2000 benchmarks. The AVF has four components: *GUE*, *ACE (GUE or LUE)*, *LUE*, and *Unknown*. As before, the GUE and LUE components are the fraction of the AVF resulting from GUE and LUE cycles. The ACE (GUE or LUE) component is time that is known to be ACE but cannot

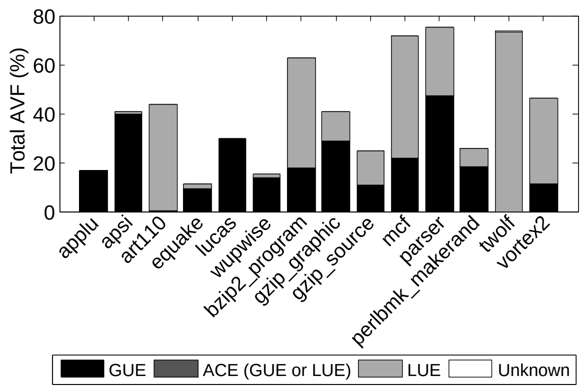

Figure 2. L2 Cache AVF with localization.

be categorized as either GUE or LUE. The Unknown component is time that may be GUE, LUE, or unACE. The latter two categories are artifacts of simulation sampling; if a program is simulated to completion, the AVF would consist only of GUE and LUE. These unknowns can be reduced, but not necessarily eliminated, by using a large cooldown window [19]. With our simulation parameters, only *twolf* has a small Unknown AVF component.

For many of the benchmarks, the GUE AVF is a substantial fraction of the overall AVF; in fact, for several of the SPECfp benchmarks, the majority of all L2 Cache AVF is GUE. Since GUE AVF will result in a system halt, designers should seek to reduce GUE AVF even at the expense of an increase in LUE AVF.

4.3. Poisoning Implementation

Data poisoning can be defined as the ability to include error information on data transferred between structures. Without poisoning, a cache cannot communicate error information to the receiver and must not allow faulty data to propagate beyond its boundaries. Thus, an exception must be raised when a request is received for data that contain an error. With poisoning, however, a structure can avoid raising an exception by *poisoning* the data and fulfilling the request. The receiver sees the poison indicator and can mark the data as bad within its local cache. An exception will be raised only if the data eventually leave the scope of the poisoning infrastructure or are consumed.

In our simulated system, we poison data at a granularity of 4 bytes. Each 4 byte chunk has a poison indicator that can be set and propagated with the data, allowing the system to delay an exception until the data are consumed. We raise an exception only if the poisoned data are read from the physical register file by a correct-path instruction. Thus, poisoned data that are never consumed will not cause an exception and do not contribute to the AVF of the cache.

4.4. Results

Figure 3 shows a breakdown of our results. Poisoning effectively converts all GUE to either LUE or unACE by localizing the error to its consumers. (This assumes the tag and other associated metadata are not damaged. Since a single strike is unlikely to affect both tag and data bits and the rate of temporal double errors is small, this is a reasonable assumption [20].) Using the language of the taxonomy, poisoning only the bad data provides localization and providing a precise interrupt to the consumer provides temporal containment. The reduction in GUE AVF is accompanied by an increase in both the LUE and Unknown (LUE or unACE) AVF. The Unknown AVF is larger with poisoning due to the long lifetimes of main memory.

Another recognized benefit of data poisoning is its ability to reduce the *false DUE* rate in a system by deferring errors until they are actually consumed by a processor. This converts DUE AVF to unACE and would appear in our simulations as a reduction in the sum of the known-ACE AVF components (GUE, GUE or LUE, and LUE). Comparing Figure 3 to Figure 2, we can see that for many of the benchmarks (*applu*, *art*, *equake*, *lucas*, *wupwise*, *bzip2_program*, *parser*, *perlbmk*, and *twolf*) the reduction in false DUE is negligible. For the other benchmarks, there is a potential reduction in false DUE (if some of the Unknown AVF is unACE), but this is a small effect relative to the conversion from GUE to LUE.

We would like to stress that we do not conclude that poisoning does not reduce the false DUE rate, but only that the effect is small for our workloads. However, our results show that even on benchmarks with potential false DUE reduction, a more significant effect of data poisoning is to convert GUE to LUE. Thus, although poisoning can reduce DUE AVF and increase the reliability of a single thread, its more substantial benefit is to increase system availability by reducing GUE AVF.

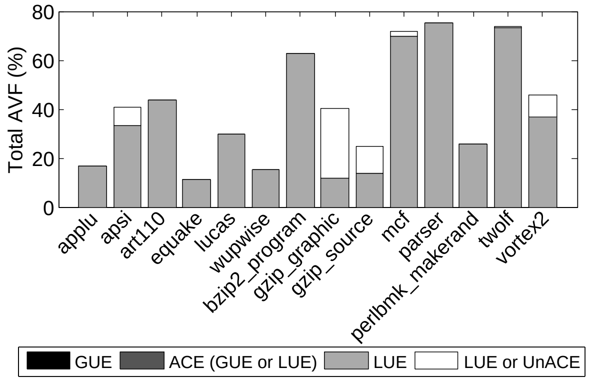

Figure 3. The effect of data poisoning on AVF. With poisoning, the GUE AVF drops to zero with a corresponding increase in the LUE AVF.

5. Conclusion and Future Work

This work has focused on system recovery and motivated the need for recovery techniques that extend beyond simple hardware correction. To that end, we have introduced a taxonomy capable of classifying the severity of uncorrectable errors. Error localization can allow a software error handler to provide partial recovery and avoid system downtime. Temporal containment can allow an error handler to correct the error and provide full recovery. Our taxonomy can quantify these effects; we provide an example by analyzing data poisoning, which has benefits that are impossible to measure without this ability.

The benefits of providing temporal containment of errors, while currently modest, will increase as the detected error rate of systems scales with Moore's

Law. Enabling systems to classify errors as software-correctable can lead to new software error correction techniques that can be used to maintain robust system reliability despite a high rate of uncorrectable errors.

References

[1] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 29.

[2] M. Y. Hsiao, W. C. Carter, J. W. Thomas, and W. R. Stringfellow, "Reliability, availability, and serviceability in ibm computer systems: A quarter century of progress," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 453–468, 1981.

[3] A. Wood, R. Jardine, and W. Bartlett, "Data integrity in hp nonstop servers," in *SELSE '06: The Second Workshop on System Effects of Logic Soft Errors*, Austin, TX, April 2006.

[4] T. Slegel, R. Averill, M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, "Ibm's s/390 g5 microprocessor design," *Micro, IEEE*, vol. 19, no. 2, pp. 12–23, Mar/Apr 1999.

[5] E. Rotenberg, "Ar-smt: A microarchitectural approach to fault tolerance in microprocessors," in *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, 1999, p. 84.

[6] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2000, pp. 25–36.

[7] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 87–98.

[8] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *CGO '05: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 243–254.

[9] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 148–159.

[10] C. Wang, H. seop Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 244–258.

[11] B. T. Gold, J. C. Smolens, B. Falsafi, and J. C. Hoe, "The granularity of soft-error containment in shared-memory multiprocessors," in *SELSE '06: The Second Workshop on System Effects of Logic Soft Errors*, Austin, TX, April 2006.

[12] B. T. Gold, J. Kim, J. C. Smolens, E. S. Chung, V. Liaskovitis, E. Nurvitadhi, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Truss: A reliable, scalable server architecture," *IEEE Micro*, vol. 25, no. 6, pp. 51–59, 2005.

[13] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2007, pp. 470–481.

[14] M. Prvulovic, Z. Zhang, and J. Torrellas, "Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 111–122.

[15] N. Quach, "High availability and reliability in the itanium processor," *IEEE Micro*, vol. 20, no. 5, pp. 61–69, 2000.

[16] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, July-Aug. 2006.

[17] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[18] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2003, p. 244.

[19] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing architectural vulnerability factors for address-based structures," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 532–543.

[20] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt, "Cache scrubbing in microprocessors: Myth or necessity?" in *PRDC '04: Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 37–42.