Understanding the Propagation of Hard Errors to Software and its Implications for Resilient System Design *

Man-Lap Li, Pradeep Ramachandran, Sarita Adve, Vikram Adve, Yuanyuan Zhou
Department of Computer Science
University of Illinois at Urbana-Champaign
{manlapli,pramach2,sadve,vadve,yyzhou}@uiuc.edu

Abstract

With continued CMOS scaling, future shipped hardware will be increasingly vulnerable to in-the-field faults. To be broadly deployable, the hardware reliability solution must incur low overheads, precluding use of expensive redundancy. We explore a co-designed hardware-software solution that treats most hardware faults as software bugs and leverages common mechanisms for hardware and software reliability, thereby amortizing some of the overhead. Fundamental to such a solution is a characterization of how hardware faults in different microarchitectural structures of a modern processor propagate through the application and OS. This paper aims to provide such a characterization, identify low-cost detection methods to intercept fault propagation, and to provide guidelines for a complete co-designed reliability solution. We focus on hard faults because they are increasingly important and have different system implications than the much studied transients. We achieve our goals through fault injection experiments with a microarchitecture level full system timing simulator.

Keywords: *Permanent fault, fault-injection, resilient system*

1 Introduction

As we move into the late CMOS era, hardware reliability will be a major obstacle to reaping the benefits of increased integration projected by Moore's law. It is expected that components in shipped chips will fail for many reasons including aging or wear-out, infant mortality due to insufficient burn-in, soft errors due to radiation from external sources and the IC package, design defects, and so on [6]. Such a scenario requires mechanisms to detect, diagnose, recover from, and possibly repair/reconfigure around these failed components so that the system can provide reliable and continuous operation.

The reliability challenge today pervades almost the entire computing market. A reliability solution that can be effectively deployed in the broad market must incur limited area, performance, and power overhead. As an extreme upper bound, the cost of reliable operation cannot exceed the benefits of scaling. In the recent SELSE-2 workshop, an industry panel converged on a 10% area overhead target to handle all sources of chip errors as a guideline for academic researchers [1]. In this context, traditional high-end solutions involving excessive redundancy are no longer viable. For example, the conventional popular solution of dual modular redundancy for fault detection implies at least a 100% overhead in performance throughput and power. Solutions such as redundant multithreading and its various flavors improve on this, but still incur large overheads in performance and/or power [25].

Two high-level observations motivate our work. First, the hardware reliability solution need handle only the device faults that propagate through higher levels of the system and become observable to software. Second, these software manifestations of hardware faults are analogous to manifestations of software errors in many ways. It may therefore be possible to leverage techniques for handling software errors for hardware faults.

These observations motivate using a unified co-designed hardware/software framework for both hardware and software reliability. Such an approach to hardware reliability could potentially be more cost-effective than software-oblivious approaches because (1) much (but not necessarily all) of the overhead incurred may already be paid for software reliability; (2) software techniques may pay lower common-case overhead while incurring greater overhead when an actual error is detected, whereas hardware mechanisms often incur *permanent* overheads in area, performance, and/or power; and (3) a software-aware hardware reliability solution is potentially easier to customize to the target application (e.g., using application-specific recovery and application-specific tuning of fault coverage vs. overhead).

This paper takes a step in exploring the feasibility of providing hardware reliability through such a co-designed hardware-software solution that treats hardware faults as software bugs. Fundamental to such a solution is an understanding of how faults in different hardware structures of a modern superscalar processor propagate through the application and operating system software, and low-cost methods to intercept this propagation within "reasonable" time. The goals of this paper are to (1) identify and quantitatively characterize the relevant aspects of the propagation of hardware faults to software in a modern processor, and (2) to use this

*This work is supported in part by an IBM faculty partnership award, the MARCO/FCRP Gigascale Systems Research Center, the National Science Foundation under Grant No. NSF CCF-0541383, and an equipment donation from AMD.

characterization to derive a set of guidelines for a complete co-designed solution for hardware reliability, including fault detection, diagnosis, and recovery.

This paper focuses on permanent hardware faults (vs. transients) because of the increasing importance of such faults due to phenomena such as wear-out and insufficient burn-in (Section 2), because transients have already been the subject of much recent study, and because permanent faults pose significant challenges different from transients. For example, a permanent fault may possibly manifest to software faster than a transient (because it lasts longer), but for the same reason, it is also less likely to be masked and more likely to corrupt the operating system resulting in an irrecoverable system crash (unless intercepted quickly). Further, after software exposes a permanent fault, the system must diagnose the faulty unit and repair it or reconfigure around it. This is generally expensive, and means that permanent fault detectors cannot afford many false positives (unlike some techniques proposed for transient fault detection [34]). Section 2 further elaborates on these differences and discusses how prior work for transients fits in a complementary way within the framework implied by our results.

We explore the following questions for different microarchitectural structures in a modern superscalar processor:

- How often does a permanent fault in the structure result in an easily observable software misbehavior and what is this misbehavior? The answer to this question determines which microarchitectural structures can be covered through low-cost software-level detection techniques (including the specific techniques and their coverage) and which structures need specialized hardware-level protection.

- For the detectable faults, what is the latency from the time the fault results in a corruption of the architectural state of the application until the time it is detected? This has implications for recovery mechanisms – with a small latency, simple hardware checkpointing and rollback can be used while a large latency may require the use of software checkpointing and recovery (leveraging software reliability methods). (This assumes a strategy for repair and/or reconfiguration around the permanent fault, as discussed later.)

- What is the latency from the time the fault results in a corruption of the architectural state of the *operating system* until the time it is detected? This has profound implications for recovery since software checkpointing and recovery of the OS is far more complex than for the application.

To answer the above questions, we inject a total of 4480 hard faults (stuck-at and bridging faults) in several microarchitectural structures in a modern processor and run SPEC benchmarks on this faulty hardware using a full system microarchitecture-level simulator. Ideally, we would use a lower level simulator for fault injections (at least a Verilog/VHDL level); however, this was not possible due to our requirement of modeling the operating system and following the fault for a very large number of execution cycles (10 million cycles). Our primary findings are as follows:

- We find that for all structures studied except for the floating point unit, most faults that propagate to software eventually either cause fatal traps (crashes) or result in small infinite loops (hangs). Thus, the use of fatal traps as symptoms of hard faults is a promising zero-cost software-level detection technique for a significant class of faults. For hangs, a simple hardware infinite loop detector can be used ([20] and Section 3) – our results show that this loop detector need only be on when the operating system is executing. Overall, the fault detection coverage is over 98% for structures other than the floating point unit.

- We find that the latency from the time the application state is corrupted to the time the fault is detected (through a hang or a crash) is often within 100k instructions (which is in the microseconds range for GHz processors) – this can be handled with efficient hardware checkpointing schemes such as SafetyNet [29] and Revive [21], using simple buffering of persistent state output (and input) to solve the output and input commit problems. In all cases, the latency is within 10M instructions (roughly a few ms for GHz processors). The high latency cases can be handled using software checkpointing, with an application specific tradeoff between buffering persistent outputs/inputs for milliseconds and full application recovery.

- We find that a large fraction of the faults corrupt operating system state. The state corruption to detection latency for the OS is within 100k OS instructions most of the time for all microarchitectural structures other than the register file (and floating point unit). This implies that hardware checkpointing of OS state can be used as a powerful method to recover the OS from a large fraction of faults – this is significant since it is difficult to recover the OS using software only mechanisms.

Our results have significant implications for resilient system design (Section 5). Most notably, they show that software level crashes and hangs are a powerful low-cost detection technique for a large class of permanent hardware faults. Further, the detection latencies are often small enough to allow even operating system recovery through hardware supported checkpointing.

2 Related Work

Software-level detection and fault injection and propagation studies.

There is a large body of literature on detecting hardware faults through monitoring various software behavior; most notably, control flow signatures, crashes, and hangs [12, 20, 22, 24, 27, 32]. There is also a large body of work that performs hardware (and software) fault injections to characterize the fault tolerant behavior of a system [2, 14, 16, 17]. Both of these classes of work perform fault injection experiments and follow the path of fault propagation through software much like our work.

Our work differs from the above work in several ways. First, we take a microarchitectural view since our goal is to understand which hardware structures could be adequately covered by inexpensive software level techniques, and which would require more expensive hardware support because of inadequate software-level coverage. We therefore perform fault injections into explicit microarchitectural structures in modern out-of-order superscalar processors; e.g., the register alias table and the reorder buffer. Our use of a microarchitecture level simulator allows such experiments. Much (but not all) prior work on fault injection is in the context of real systems (or high level simulations), where processor microarchitectural units are not exposed.

Second, most prior work injects transient or intermittent faults, where intermittent faults are usually modeled like transients except that they last a small number of cycles (e.g., up to 4 cycles). We study permanent faults because these are predicted to become increasingly important with growing concern from phenomena such as aging and inadequate burn-in [6, 7, 30, 36]. Permanent faults are significantly different from transients and intermittents that last a few cycles. For example, most transients have been shown to be masked at various levels of the system (device, circuit, microarchitecture, application) [26]. On the other hand, permanents are unlikely to be completely masked. While this means that a permanent fault may possibly manifest to software faster than a transient (because it possibly affects many instructions), a permanent fault is also more likely to corrupt the operating system possibly resulting in an irrecoverable system crash. We therefore perform a detailed analysis of when and for how long the OS state is corrupted. Further, after software exposes a permanent fault, the system must diagnose the faulty unit and repair it or reconfigure around it – this is not an issue with transients, where the system can simply rollback to the previous checkpoint and re-execute. Diagnosis for hard faults based on software level symptoms is generally expected to be expensive, and means that permanent fault detectors cannot afford many false positives. In contrast, previous work for transients has suggested use of branch mispredictions and L2 cache misses as symptoms for transient error detection [34]. Using these as symptoms for hard error detection would be impractical since these events occur frequently enough for the large latencies for diagnosis.

Third, while there have been fault injection studies at the microarchitecture or lower levels (e.g., Wang et al.'s study of soft errors at the Verilog level [34]), our work is distinguished by our study of both the application and operating system through using a full system simulator. Many of the results from this work would not be possible from user-only architecture or lower level simulators. For example, corruptions of the OS state are difficult to recover from – our work models such corruptions and shows that in many cases, the detection latencies are small enough to use efficient hardware checkpointing for recovery.

Fault tolerant systems.

There is a vast amount of literature on fault tolerant architectures. High-end commercial systems often provide fault tolerance through system-level or coarse-grain redundancy (e.g., replicating an entire processor or a major portion of the pipeline) [5, 19]. Unfortunately, this approach incurs significant area, performance, and power overheads. As mentioned in Section 1, our focus is on low-cost reliability for a broader market, where some parts of the market may even be willing to trade off some coverage for cost. There has been substantial microarchitecture level work in this context, where redundancy is exploited at a finer microarchitectural granularity. While much of that work handles soft errors [3, 12, 13, 23, 24, 25, 34] recently, there has been substantial work on handling hard errors. We focus on that work here.

Austin proposed DIVA, a very efficient checker processor that is tightly coupled with the main processor's pipeline to check every committed instruction for errors [3]. While DIVA can be used to provide detection of hard errors, it does not provide mechanisms for diagnosis or repair. It also requires ensuring that the checker itself does not have errors. Bower et al. incorporated hard error diagnosis with DIVA checkers [9]. As an instruction moves through the processor pipeline, it keeps track of the different hardware structures it has visited. On detecting an error in an instruction, counters associated with the structures visited by the instruction are incremented. Depending on a heuristic based on the value of the counters, specific structures are deemed failed. Inherent performance-driven redundancy in a superscalar allows disabling a failed unit and continuing with other available instances of the same unit [9, 31]. This technique uses n DIVA processors for an n-way superscalar.

Weglarz et al. investigated the use of redundant threading to perform online-testing of a processor for hard faults [35]. However, this is a preliminary feasibility study and does not discuss how these tests are generated, coverage, or detection latency. Shyam et al. have recently proposed online testing of certain structures in the microprocessor for hard-faults and recover by disabling them and rolling back to a hardware checkpoint [28]. Since these tests are run only when the structures are idle, the performance loss incurred is rather small. However, the feasibility of such an approach is unclear for unstructured control elements that are generally hard to test.

All of the above schemes that incorporate diagnosis and recovery incur significant overhead that is paid almost all the time, in area, performance, and/or power.

In contrast to the above, the motivation for our work is a reliability solution that pays minimal cost in the common case where there are no errors, and potentially high cost in the uncommon case when an error is detected. Specifically, using application crashes as a detection mechanism has zero detection overhead until there is actually an error. Using hangs requires only slight "always-on" overhead (updates of a branch counter to periodically measure the frequency of branches), which is likely already paid by standard performance counter support for current processors. We also require checkpoint/rollback support; however, analogous support is assumed by previous schemes as well [8, 28]. Additionally, we allow for the possibility of checkpoint support in software and leveraging such support that may be already present for software reliability. Finally, working at the software level, we only detect errors that are not masked by the

hardware or software.

3 Methodology

We perform our fault injection experiments in a simulated microprocessor. The subsequent sub-sections describe the simulation environment, microarchitecture configuration, workloads, fault models, detection techniques, and metrics used.

3.1 Simulation Environment

Ideally, for fault injection experiments, we would like to use a real system or a low-level (e.g., gate level) simulator. However, modern processors do not provide enough observability and control to perform the microarchitecture level fault injections that are of interest to us. We therefore use simulation. Although low-level simulators would provide the ability to use more accurate fault models, they present a tradeoff in speed and the ability to model long running workloads with OS activity. Given our emphasis on understanding the impact of faults on the OS and the need to simulate for long periods, gate level simulation was not feasible. We therefore chose to use a microarchitecture level simulator.

We use a full system simulation environment comprising the Wisconsin GEMS microarchitectural and memory timing simulators [18] in conjunction with the Virutech SIMICS full system simulator [33]. Together, these simulators provide cycle-by-cycle microarchitecture level timing simulation of a real workload running on a real operating system on a modern out-of-order superscalar processor and memory hierarchy.

A full-blown Solaris OS running on SPARC-V9 instruction-set architecture (ISA) is simulated using this infrastructure. The simulation is based on a timing-first approach where the same instruction is executed by both the timing-accurate GEMS timing simulator and the functionally accurate SIMICS functional simulator [18]. The resulting architecture states of GEMS and SIMICS are compared every cycle to detect any mismatches between the states.

The injected fault initially affects only the timing simulator's execution. Once a mismatch between the *architectural state* of the functional and the timing simulator is detected, we flag a *corruption* in the architecture state.[1] Since the retiring instruction can be either privileged or non-privileged, architecture state corruption of both the application and the OS can be detected.

3.2 Fault Model

The focus of this study is on permanent or hard faults, with the goal of modeling increasingly important phenomena such as wear-out or infant mortality due to incomplete burn-in [6, 7, 36]. Precise fault models for wear-out are still a subject of research [15]. In this paper, we use the well established stuck-at-0 and stuck-at-1 fault models as well as bridging faults (bridged to gnd and Vcc). Recent work has suggested that some wear-out faults (on the critical paths of a circuit) may initially manifest as timing violations for a few hours before resulting in hard breakdown [10]. Modeling such faults requires lower level simulation than our current infrastructure, along with its attendant tradeoffs (Section 3.1). We expect that the results shown here will likely extend for other errors that are persistent as well.

[1] GEMS does not implement all the instructions in the SPARC ISA and hence, a mismatch between the architecture states of GEMS and SIMICS may not be the result of an injected fault. However, we flag an architecture state corruption only when the retiring instruction is known to be faulty.

Structure	Fault location
Instruction Decoder	Input latch
Integer ALU	Output latch
FP ALU	Output latch
Register Bus	Output Data bus
Int Reg file	Physical register num 200
Reorder Buffer (ROB)	Destination Register num field
Register Alias Table (RAT)	Logical→Physical mapping

Table 1. Microarchitectual structures in which wear-out faults are modeled. Various bits in array structures and latches are modeled with wear-out due to stuck-at and bridging faults.

Table 1 lists the structures in which faults are injected along with the location of the injected fault in each structure.

For each faulty structure, we perform fault injection experiments with four different bits (one bit per injection). This allows us to factor out the differences between the various bits in the structure.

3.3 Workload

Our workloads comprise of a collection of 4 Floating Point applications (mesa, art, ammp and equake) and 6 Integer applications (parser, mcf, bzip2, gzip, gcc and twolf) from the SPEC CPU 2000 benchmark suite. These applications are simulated inside of a Solaris-9 Operating System running on a UltraSparc-III+ processor. The full-system simulation framework allows us to model all system calls and OS interactions of the applications and help understand the effects of wear-out faults on the Operating System as well.

After the application has completed initialization, four random points are chosen to inject a hardware fault in one of the many structures that are modeled faulty. After a fault is injected, the workload is simulated for 10 million instructions and the the presence of the hardware fault is detected using software-level symptoms.

The workloads that we use are application intensive, performing a majority of their system calls and other OS interactions during the initialization phases. The OS activity in the 10 millions simulated instructions is $<8\%$. We identified phases in these initializations where activity in the OS constituted 15-70% of the execution time and performed fault injection experiments in those phases as well. However, the high-level results and implications derived from those experiments were similar to the experiments that had low OS activity. Hence, the results presented here are only for injections into the former points.

3.4 Fault Detection

An injected fault is said to be activated when the injection results in a corruption of the architectural state (Section 3.1). If the fault is never activated, the fault is said to be *masked.*

Faults that are not masked result in corrupting the architecture state of the application or the OS or both. As a result, the application or OS may perform some illegal operations

Trap type	Description
RED state	Recover Error and Debug state
Data Access Exception	Invalid data access (page invalid or protected)
Division by Zero	A divide by zero integer error
Illegal Instruction	Unable to decode opcode
Memory Misaligned	Access to a misaligned memory address
Watchdog Reset	No instruction retired in the last 2^{16} ticks

Table 2. List of *fatal* traps monitored as software symptoms of hardware failure

that may result in a crash or a hang. We use software-level mechanisms to detect such crashes and detect hangs using small amount of hardware support (see below).

However, not every corruption of architectural state may result in a crash or a hang. The corrupted application, or OS, may perform illegal operations that result only in data corruptions without resulting in a crash or a hang. Such *Silent Data Corruptions (SDC)* currently fall outside our detection mechanism. The fault may also be masked by the application if the corrupted memory value does not affect outputs generated by the application. Since we simulate only about 10 million instructions of the application, this masking of the fault by the application is currently not determined by our framework (our coverage numbers are therefore conservative).

The following discusses how we detect crashes and hangs.

3.4.1 Detecting Crashes

Our detection mechanism looks for software-level symptoms to detect abnormal application behavior. We monitor the traps thrown when instructions are executed with a look-out for *fatal* traps that are typically not thrown during a correct program execution. These *fatal* traps are thrown when the instructions perform some illegal operations like division by zero, accessing an invalid or protected page, etc. Since OS instructions are also simulated on the same processor, the *fatal* trap can be from an instruction in the OS as well. Table 2 lists the Solaris traps that are denoted as *fatal* traps that lead to crashes.

3.4.2 Detecting Hangs

Previous work has proposed hardware support to detect application and OS hangs with high fidelity but some area and power overhead [20]. Several optimizations to that work are possible. First, our results show that most of the hangs occur in the OS. Thus, the hang detector can potentially be customized to OS software; minimally, it need be activated only when execution enters privileged mode (reducing any power overhead). Second, a simple detector based on a simple heuristic can initially be used (e.g., based on the frequency of branches) – if that heuristic is satisfied, then a more complex mechanism involving hardware or software can be invoked.

For the purpose of our simulations, we use a heuristic based hang detector that is based on monitoring all the executed branches. A table of counters, indexed by the PC of the branch instruction, is accessed every time a branch is executed and the corresponding counter is incremented. Once any counter exceeds 100,000 (this corresponds to the corresponding branch constituting *1%* of the total executed instructions), the detector flags a hang. Hangs in the OS are distinguished from hangs in the application as the privileged instructions are distinguishable from non-privileged instructions.[2]

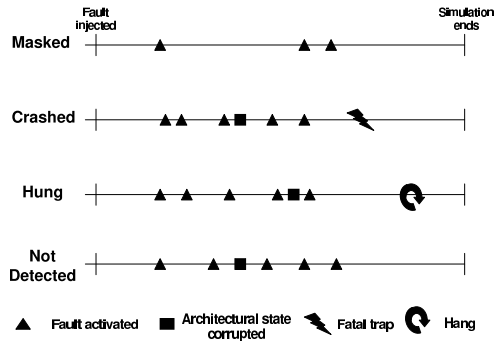

Figure 1. Different outcomes of an injected fault. *Masked:* the fault may be activated but does not cause an architectural state corruption. *Crashed:* the activated fault leads to a fatal trap. *Hung:* the activated fault causes a hang in the execution. *Not Detected:* the fault is activated but cause neither a fatal trap nor a hang.

Figure 1 shows the different outcomes of an injected fault, along with the mechanisms that we use to detect them.

3.5 Metrics

To help understand the feasibility of using software-level methods to detect hardware errors, we report coverage and latency numbers for our detection methods.

Coverage: The coverage of a detection mechanism denotes the percentage of non-masked faults that are detected. Hardware faults that are not masked and result in either crashes or hangs can be detected by our symptom-based detection methods. Thus, the coverage numbers that we report are evaluated as

$$Coverage = \frac{Crashes + Hangs}{Total\ injections - Masked}$$

Detection latency: In addition to high coverage, a good detection mechanism should also detect the fault with a short latency so as to facilitate easier diagnosis and aid the recovery and repair process. In our experiments, the detection latency of a fault that resulted in a crash is reported as the total number of instructions retired after the architecture state corruption of either the OS or the application until the crash. For hangs, this latency is reported as the number of instructions retired from the architecture state corruption to the 100th occurrence of the instruction that was found to be part of an infinite loop.

4 Results

Detection of hardware faults using software-only methods presents a zero to low overhead solution to detect and diagnose the presence of hardware faults. Aided by minimal hardware support, such solutions possibly open new dimensions in recovery and repair from hardware faults. This section presents results to help understand the feasibility of such an approach, based on the fault injection experiments detailed in the previous section.

[2]In our simulations, we actually start the table updates after seeing architecture state corruption and a deviation in the trap behavior of the current run from the fault-free run.

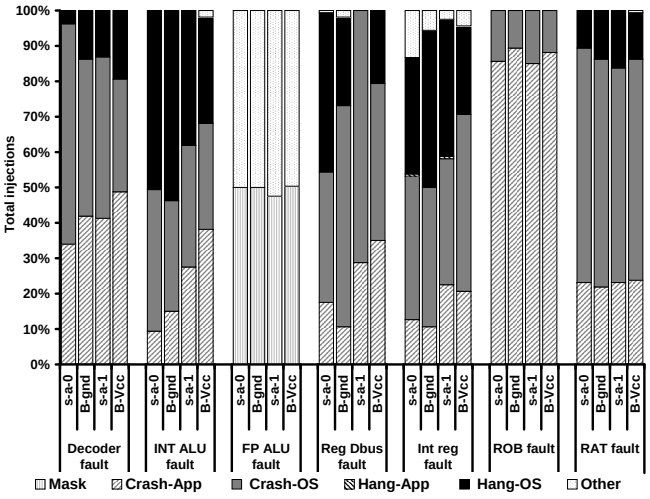

Figure 2. Propagation of hardware faults through software. For each microarchitectural structure and fault model, the corresponding bar shows the percentage of injected faults that are masked, that cause crashes and hangs in the application and OS. The topmost bar shows the percentage of injected faults that are not detected using our method. We see that a large fraction of hardware faults can be detected at the software level using appropriate symptoms.

4.1 How do Permanent Faults Propagate through Software?

We begin by first understanding how these hardware failures propagate to the software level. We quantify the effect of the faults that affect the software, along with the feasibility of detecting them using our software-level approach.

Figure 2 shows how hardware faults propagate through software for a given microarchitectural structure and four fault models. Each bar represents fault injections into the corresponding microarchitectural structure with one of the stuck-at fault models (stuck at 0 and 1) and the bridging fault models (bridged-to-ground and bridged-to-Vcc, shown as B-gnd and B-Vcc, respectively). In each bar, the lowest stack represents the percentage of injections that are architecturally masked. The next two stacks represent the injections that resulted in a crash of the application (Crash-App) and the OS (Crash-OS), respectively. The next two stacks represent the injections that result in hangs in the application (Hang-App) and the OS (Hang-OS), respectively. These stacks represent the injections that can be detected using our symptom based detection. The topmost stack (Other) gives the injections that are not detected. As described in section 3, some part of this bar may be composed of cases that may be masked by the application, or may result in SDC.

Table 3 summarizes the above data to show the total coverage of our symptom-based software-level detection mechanisms for different microarchitectural structures.

From this data, we find that permanent faults in the processor are highly software visible as over 98% of the faults in all structures except FP ALU result in crashes and hangs. This high rate is mainly due to the injected faults affecting the control flows of the software (either the application or the OS or both). These results imply that software-level detection methods based on crashes and hangs are effective in detecting permanent faults in many microarchitectural structures of a modern processor (even though most such faults are not masked like transient faults).

On the other hand, faults in some structures like the FP ALU cannot be detected by this software-level method and warrant other software symptoms or some kind of hardware support for detection (e.g. residue codes [4]). Contrary to faults in other structures, faults in the FP ALU seldom affect the control path of the software. In particular, FP ALU is usually not involved in address computations in the software while other structures are. Thus, corruptions in the FP ALU will at the worst affect the correctness of the data computation but are least likely to result in incorrect memory addresses (the effects of faults in other structures) and eventually lead to crashes or hangs. Hence, when compared with faults in other structures, faults in FP ALU are hard to detect through crashes and hangs.

The faults injected into the microarchitecture structures constituted both stuck-at and bridging faults. However, we find that faults in either fault model propagate in similar fashion to the software, resulting in little variation across the number of crashes and hangs caused. This is because the underlying fault, which is either stuck-at or bridging, is a permanent fault causing highly intrusive behavior in the software's execution.

Figure 2 also shows that the low OS activity ($< 8\%$) in all of these application does not necessarily translate to higher rates of application crashes and hangs. In fact, majority of the crashes detected (54.1%) were in the OS and almost all (99.7%) of the hangs detected were in the OS. This can be explained by the fact that, in general, the OS is more control-intensive than applications and hence the injected faults are more likely to cause the OS to crash/hang. Also, some errors in the application will result in non-fatal traps such as page misses and trap into the OS code. Since the hardware fault is still present, the trap handler is likely to activate the fault and finally causes a crash or a hang in the OS.

Current practices rarely involve checkpointing the OS and hence, the OS cannot be recovered when it is corrupted if special care is not provided to make OS fault-tolerant.

Although less than 18.1% of injected faults in all structures, except the ROB, result in crashes or hangs in the application, over 87% of the faults in the ROB result in crashes in the application. This is because faults in the ROB directly affect the dependence chain of instructions and hence lead to early crashes by causing a violation of this dependence. Since the simulated applications have low OS activity, these faults are activated more by the application, resulting in its crash.

To understand the reason for such a high percentage of crashes and hangs in the OS, we further investigate the software sites that were corrupted as a result of the underlying fault before a crash occurs. Since the variation of crashes and hangs across different fault models is small, results in the following subsections aggregate all the fault models.

Mechanism	Coverage for microarchitectural structures						
	Decoder fault	INT ALU fault	FP ALU fault	Reg Dbus fault	Int Reg fault	ROB fault	RAT fault
Crashes	87.34% (559/640)	56.41% (361/640)	0.00% (0/323)	76.72% (491/640)	57.81% (370/640)	100.00% (640/640)	86.41% (553/640)
Hangs	12.50% (80/640)	43.13% (276/640)	0.00% (0/323)	22.66% (145/640)	35.47% (227/640)	0.00% (0/640)	13.44% (86/640)
Total	99.84% (639/640)	99.53% (637/640)	0.00% (0/323)	99.38% (636/640)	93.28% (597/640)	100.00% (640/640)	99.84% (639/640)

Table 3. Coverage from hangs and crashes for faults that corrupt the architectural state.

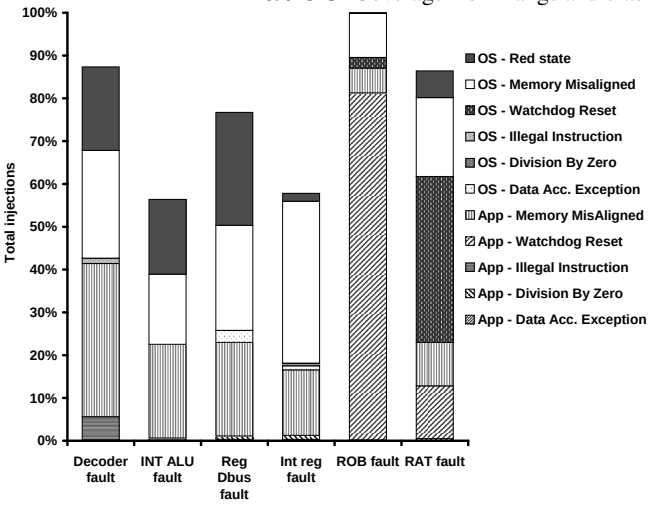

Figure 3. Number of crashes in the application (bottom hatched portions) and OS (top non-hatched portions) and the corresponding fatal trap types. The height of each bar is the percentage of the total number of faults that resulted in crashes in the corresponding microarchitecture structure. A fault in the underlying hardware results in crashes in not just the application, but also the OS.

4.2 Software Sites and Causes for Crashes

Figure 3 shows the number of crashes that occur in the application (bottom hatched portions) and the number that occur in the OS (top non-hatched portions). For each portion, the figure also shows the distribution of the different types of fatal traps that result in crashes. The height of each bar is the percentage of injections in the corresponding structure that results in a crash.

From Figure 3, we see that faults in many hardware structures result in a wide variety of *fatal* traps as they corrupt various software sites in both the application and the OS. Some of the traps are discussed in the rest of the section.

Even though faults are injected in various structures, majority of the faults (over 45% of the injected faults in all structures, except the ROB) lead to accesses to misaligned addresses. As many of the instructions within the software are responsible for accessing the memory, the injected fault is likely to corrupt one of these instructions and eventually result in accessing a misaligned memory location.

SPARC's high-priority Red State Exception, which constitutes 21% of the crashes for faults in the Decoder, INT ALU and Reg Dbus, is a good indication of the presence of a permanent fault. Faults that result in this trap causes nested traps to be thrown in the OS, triggering this reset.[3] While this built-in error flagging mechanism in SPARC is effective in reporting hard faults in the hardware, the system state will already be corrupted when this trap is thrown.

Faults in the ROB and RAT result in mutating the destination register to which an instruction writes its data. This often results in the dependent instructions indefinitely waiting for its source operands, triggering a Watchdog Timer Reset trap.

While one would expect a fault in the decoder to result in a large number of Illegal Instruction traps, it constitutes less than 8% of the fatal traps thrown when a fault is present in the decoder. This is because the majority of the injected faults in the instruction word mutate the instruction by changing either the opcode (to another legal opcode) or the register values.

4.3 Software Components Corrupted by Hardware Faults

As the previous subsection discusses the various types of traps thrown due to the injected faults, the fatal traps only show the final crash sites and in no way do they suggest the correctness of the application and the OS at the point of a crash. To further understand the propagation of the hardware faults, this section focuses on the software components (application or OS) corrupted when a crash or a hang is detected.

When a crash or a hang is detected, if only the application's state was corrupted but the system state remained integral, the application can likely be recovered through application-level checkpointing. However, if in any cases the OS state was corrupted, recovery and repair would require some form of OS checkpointing, which is difficult and so far has been proposed only for a virtual machine approach [11].

For all the crashes and hangs caused due to faults injected into each microarchitecture structure, Figure 4 shows the percentage of injections that resulted in only application state corruption, OS (and possibly application) state corruption and corruption of neither the application nor the OS. The height of each bar gives the percentage of faults injected into the given structure that resulted in crashes or hangs.

The figure shows that over 78.5% of the faults in all structures except ROB corrupt the OS before resulting in a hang or crash. This shows that even though an injected fault may first corrupt the application, it is not very likely to cause an immediate crash or hang of the application. Instead, through other types of non-fatal traps, the fault eventually corrupts the OS before a crash or a hang results. Hangs in the OS is far more severe than application hangs because if the OS hangs, nothing gets to be scheduled, whereas if an application hangs, other applications can still run.

Faults in ROB are both easy to detect and to recover. Contrary to faults in other structures, all of the ROB faults result in crashes. Furthermore, over 81.8% of faults have only the application states corrupted before crashes occur. The high rate of application-only corruption is due mainly to the high intrusiveness of the fault (as discussed in Section 4.2) and the short latency to activate the fault. The former property

[3] The SPARC-V9 architecture throws a Red State Exception when a trap is thrown at (maximum_trap_level - 1) trap level. The UltraSparc-III+ processor simulated has a maximum_trap_level of 5.

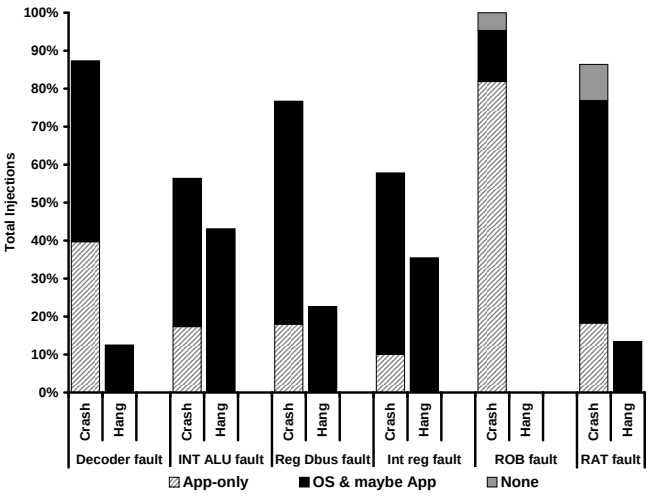

Figure 4. For all the crashes and hangs caused due to faults injected into each microarchitecture structure, the figure shows the percentage of cases when only the application was corrupted, when the OS (and possibly the app) was corrupted and when neither were corrupted. The height of each bar gives the percentage of crashes and hangs from faults injected in that structure. We see that for many cases, the hardware fault corrupts the system state as well.

causes almost an immediate crash and the latter implies that the application first activates the fault most of the time.

When compared with other structures except ROB, faults in Decoder result in a higher number (39.7%) of application-only state corruptions. These faults are highly intrusive because they not only corrupt the data (register operands and immediates) used by the instructions but they also mutate an instruction into another instruction. The latter effect causes corruptions in both the data path and the control path and results in many crashes of the application before other non-fatal traps are thrown.

Faults in the INT ALU, Reg Dbus, Int Reg file and RAT take inherently longer to detect and hence result in substantially corrupting both the application and system state before resulting in a hang or a crash.

There are a small number of cases of faults in the ROB (4.7%) and RAT 9.5% where neither the application nor the OS states are corrupted (the *None* portions in Figure 4) before a crash occurs. In the ROB cases, the fault causes an instruction to write a value into a wrong but unmapped physical register. Since the wrong physical register is part of the architecture state, there is no state corruption visible. However, the execution crashes because a future instruction that depends on the value written by the instruction incurring the fault cannot make progress (its source register is forever seen as busy). This case triggers the watchdog timer interrupt, resulting in a crash without an architecture state violation.

In the RAT cases, similar to the ROB cases described above, the logical-to-physical mapping is corrupted to point to an unmapped physical register. Subsequently, the dependent instruction will wait for its source indefinitely and causes a watchdog timer interrupt, again resulting in a crash without an architecture state violation.

We see that over 79.7% of the injections in faults in the

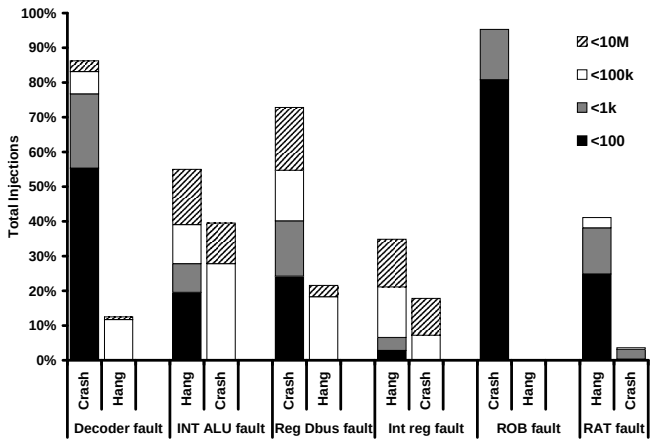

Figure 5. Crash and hang detection latency for application state corruptions in terms of total instructions executed from the application architecture state corruption to a crash, or a hang, for injections in different microarchitectural structures.

INT ALU, Reg Dbus, Int Reg file and RAT result in corrupting the system state. Hence, special hardware or software support to facilitate checkpoint and recovery of the OS becomes important. The feasibility of such an approach is dependent on the latency at which the software detects these failures.

4.4 Detection Latency

Detection latency is a crucial parameter since it affects the checkpointing and recovery mechanisms. Specifically, it affects the checkpointing interval, the amount of states that needs to be preserved for a checkpoint, and the cost of recovery. Small latencies allow the use of frequent but efficient hardware checkpoints and fast and complete recovery for both the application and the OS Large detection latencies potentially require (infrequent) software checkpointing, longer restart on recovery, and dealing with the input and output commit problems which could thwart full recovery.

We study the detection latencies for OS corruptions separately from application corruptions because the two entail different tradeoffs. Software checkpointing of the OS is difficult and so far has only been proposed for a virtual machine approach [11]. Therefore, short detection latencies coupled with hardware support for checkpointing are likely to be more effective for OS recovery.

4.4.1 Latency from Application State Corruptions

For each microarchitectural component, Figure 5 shows a histogram of the number of retired instructions from an application architecture state corruption to the detection of a hang or a crash. While Figure 2 already shows the number of runs that result in crashes and hangs within the 10M-instruction simulation interval, Figure 5 categorizes the runs that have corrupted application states before a hang or a crash according to their detection latencies. This information can help us understand how recovery schemes that tolerate shorter latency than application-level software checkpointing (billions to trillions of instructions) can be effective in recovering from a permanent fault.

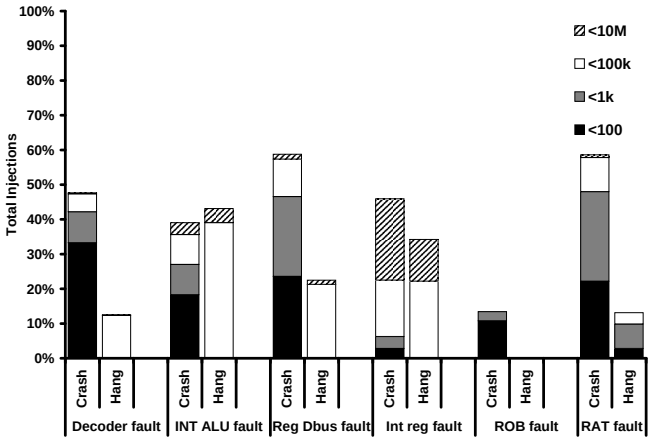

Figure 6. **Crash and hang detection latency for OS state corruptions in terms of OS instructions executed from the OS architecture state corruption to a crash or a hang, for faults in different microarchitectural components.**

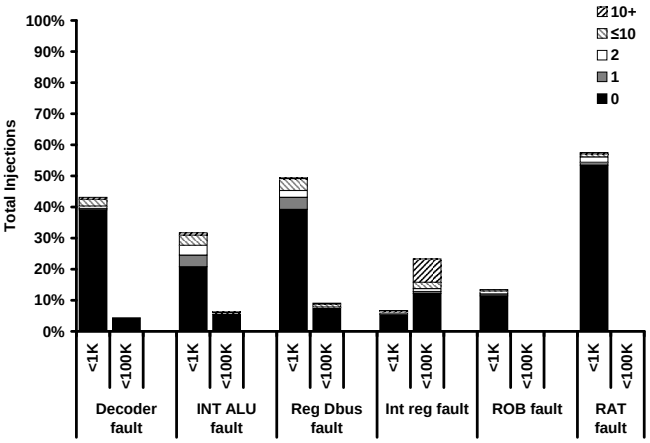

Figure 7. **Number of times the Application-OS boundary is crossed from the OS architecture state corruption to the fatal trap, for different fatal trap latencies.**

From the figure, we see that over 65.2% of the runs in all structures except Int Reg result in a crash or a hang within 100k instructions (microseconds range for GHz processors). Within this interval, the faults can be caught with efficient hardware checkpointing support (e.g., SafetyNet easily supports multiple checkpoints with a checkpoint interval of 100k cycles), and replayed with simple buffering of persistent state output and input to solve the input/output commit problem.

Furthermore, almost all of the faults in ROB and RAT result in hangs and crashes within 1k instructions. For these cases, even hardware checkpoint and recovery techniques with very short checkpoint intervals can be used to recover from a hard fault.

The figure also shows that a large number of the faults in Int Reg cannot be detected within 100k instructions. This is because faults in this structure are less intrusive and have a lower fault activation rate than those in other structures. While hardware checkpointing techniques may be ineffective in recovering from these faults, software checkpointing techniques will be able to recover the application state. However, this case requires considering a tradeoff between complete recovery by buffering persistent state outputs and inputs for up to 10 million instructions (few milliseconds for GHz processors) or risking incomplete recovery while immediately committing external outputs. Milliseconds of delay for many output operations (e.g., disks) do not violate software semantics and so should not pose a problem.

Thus, when the underlying hardware fault corrupts only the application, hardware- or software-level checkpoint and recovery methods can be exploited, depending on the type of coverage vs. overhead tradeoff desired. For example, systems where availability is most important may choose to forgo high coverage for applications.

4.4.2 Latency from OS State Corruptions

From section 4.3, we saw that over 78.5% of the injected faults resulted in corrupting the system state. When the system state is corrupted, application and system recovery is hard as the OS is rarely checkpointed. Special hardware to help with checkpointing the OS can allow for efficient recovery of the entire system.

Although the detection latency in terms of the total number of instructions is within the software checkpointing interval (as seen in the previous section), the latency is too large for hardware checkpointing for full coverage. However, to recover the system, one has to only recover the OS (this at least preserves the integrity of the system though the application is no longer recoverable). Thus, if the number of OS instructions executed after the OS was corrupted is sufficiently small, such buffering in hardware may be feasible.

Figure 6 shows a histogram of the number of OS instructions retired from the first OS architectural state corruption to a crash or a hang across different structures. Each portion of the bar represents the number of faults that are detected within the corresponding latencies.

The figure shows that over 49.7% (60.6% without the Int Reg faults) of the crashes in all structures resulted in crashes or hangs can be seen within 1k OS instructions. Given this short interval, hardware buffering of the OS instructions is effective to recover from majority of the faults. Further, hardware support for checkpoint intervals of up to 100k OS instructions (e.g., as in [21, 29]) implies more than 96% of all but Int Reg faults are recoverable. This implies that a moderate amount of buffering of OS instructions can recover systems that are corrupted by a permanent fault.

Across all structures, Int Reg faults tend to have longer OS latencies to crashes and hangs. This is because of the same reason discussed in Section 4.4.1. In order to recover from faults in this structure, more sophisticated techniques that can tolerate or buffer 10M instructions will be needed.

While the number of OS instructions is a good metric for guiding the design of an OS checkpointing scheme, the number of switches between the executions of the applications and the OS within this interval governs the complexity of the possible OS recovery schemes.

Figure 7 shows the histogram of the number of times the Application-OS boundary is crossed from the OS state corruption to the fatal trap across different fault models and different latencies. Each fault that leads to a crash is categorized by the number of times the boundary was crossed. For faults that result in hangs, we found that all of them do not cross

the App-OS boundary.

The figure shows that majority of the faults that result in crashes are contained within the system once the system state is corrupted (58% of Int Reg, 70% of ALU, and 80% or more of Decoder, Reg Dbus, ROB, and RAT). Thus, using a naive buffer which does not consider switches between the OS and application can give reasonable coverage for recovery.

Buffering techniques that consider two crosses (OS→App→OS) are cost-effective as the coverage rises to 88% and more for Decoder, ALU, Reg Dbus, ROB, and RAT and 64% for Int Reg. Thus, to design an OS recovery scheme with decent coverage, OS instruction buffering approaches that take OS-App switches in account are essential.

Assuming the buffering scheme can tolerate 10 App-OS crosses, high coverage can be achieved for all structures (97+% of the crashes) except Int Reg (73% of the crashes). While full OS recovery can be achieved for most of the structures, the low coverage in Int Reg shows that other forms of software-level detection (e.g., invariant violation detection, etc. [37]) and/or hardware support may be needed for faults of low intrusiveness in order to effectively shorten the detection latency and lower the number of boundary crossings for full recovery of the OS.

5 Conclusions and Implications for Resilient System Design

This work concerns understanding how permanent hardware faults in microarchitectural structures of modern processors propagate through software, including the application and operating system. The eventual goal of this project is to develop a low-cost, customizable hardware/software co-designed solution for hardware (and software) reliability. The findings in this paper provide several new and concrete guidelines for a complete low-cost reliability solution, including detection, recovery, and diagnosis.

Detection. Our results unequivocally show that for most microarchitectural structures, virtually all faults (more than 99% in all but two cases, and more than 93% in one of the remaining two) are detected through crashes and hangs. Crashes provide a zero-cost detection mechanism. For hangs, it suffices to invoke a (hardware) hang detector only in the OS. This hardware can initially be a very simple heuristic (e.g., backward branch frequency), which can be further refined in the rare case that it is invoked.

Our results also show that for the floating point unit, alternative mechanisms are required (e.g., residue codes, space/time redundancy), but these would potentially be much lower overhead than techniques such as full core redundancy. Further, even for the microarchitecture structures that show excellent coverage, there are still some faults that are not detected through crashes and hangs. For these cases, for applications/systems that require higher coverage, we plan to pursue other software level detection schemes (e.g., invariant violation detection, etc.) that can leverage software reliability techniques [37]. For mission-critical systems that require the highest coverage, we plan to explore limited on-line testing to backup our high-level detection.

Recovery. Our results show the feasibility of checkpoint/replay based recovery. Such an approach assumes a strategy for diagnosis, repair/reconfiguration and the ability to replay on correct hardware. We discuss this in the next portion of this section – first, we assume that such a strategy exists and note its implications on checkpointing.

A specific challenge is the recoverability of the operating system. Our results show that for both the OS-intensive initialization part and the non-OS intensive remaining part of our applications, a large fraction of the faults result in corrupting the OS; therefore, much care is needed to make our system recoverable from OS failures. At the same time, our results also show that for the microarchitectural structures considered for high-level detection, the number of OS instructions executed from the time that the OS state is actually corrupted to the time of a crash or hang is usually small. Except for faults in the register file, most faults are detectable within 100k OS instructions. Furthermore, for the longer latency faults, there are few crossings between the OS and application. These results suggest that hardware checkpoint/replay is feasible for the OS, in terms of hardware state required, performance overhead, and simple solutions to the input and output commit problems.

For application recovery, we find that the detection latencies are also often within the hardware recovery window; however, there are also many cases of higher latency. Regardless, in all cases, the latency is within 10M instructions (roughly a few ms for GHz processors). The high latency cases can be handled using software checkpointing, with an application specific tradeoff between buffering persistent outputs/inputs for milliseconds and full application recovery.

These results have clear and interesting implications for hardware/software co-designed recovery for the application and the OS, for both hardware and software reliability. For example, hardware checkpointing structures for hardware reliability can be leveraged for software reliability as well. Further, as mentioned above, software reliability driven detection techniques such as invariant violation detection, etc. can be used to reduce the detection latency if further needed.

Diagnosis and Repair/Reconfiguration. Although we do not deal with diagnosis directly here, the main implication of our approach is that diagnosis will likely be expensive (time intensive) and requires a potentially error-free processor to proceed. Expensive diagnosis is acceptable and is the right tradeoff for cheap detection since diagnosis overhead is only paid in the rare case that a fault is detected. For the latter issue (availability of an error-free processor), we assume we can leverage the multiple cores on current chip multiprocessors – we do the same for replay and forward progress of the workload in parallel with diagnosis. Diagnosis can use a replay/test strategy to incrementally determine where the fault occurred. Repair and reconfiguration can be done at the core level or within a core, exploiting the available redundancy in modern processors (albeit at some loss in performance).

Other future work. Besides exploring the system implications mentioned above, our future work will also include refining the fault models used here, including studying intermittents and integrating solutions for transients (e.g., as in [34]). We also plan to validate our insights with lower level simulators. Finally, this study has focused on SPEC benchmarks, which have little OS activity (although there is

significant OS interaction in the presence of faults). To get some understanding of OS intensive applications, we studied the initialization phases of the SPEC applications, which have significant OS activity. As mentioned in Section 3, our high-level results were similar for this part of the applications as well. However, in the future we need to validate our results for more realistic OS intensive workloads such as transaction processing and web server workloads.

References

[1] Design Panel, SELSE II - Reverie, 2006. `http://www.selse.org/selse2.org/recap.pdf`.

[2] J. Arlat, A. Costes, Y. Crouzet, J. Laprie, and D. Powell. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. *IEEE Trans. on Comp.s*, 42(8), 1993.

[3] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron MicroArch. Design. In *Proc. of 32nd Intl. Symp. on MicroArch. (MICRO-32)*, 1998.

[4] A. Avizienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE Trans. on Comp.s*, C-20(11), Nov. 1971.

[5] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. In *DSN '05: Proc. of 2005 Intl. Conf. on Dependable Systems and Networks (DSN'05)*, 2005.

[6] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), 2005.

[7] S. Borkar. MicroArchitecture and Design Challenges for Gigascale Integration. In *Keynote address at 37th Intl.. Symp. on MicroArch.*, 2005.

[8] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, 2004.

[9] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proc. of 38th Intl. Symp. on MicroArch. (MICRO-38)*, 2005.

[10] J. R. Carter, S. Ozev, and D. J. Sorin. Circuit-Level Modeling for Concurrent Testing of Operational Defects due to Gate Oxide Breakdown. In *DATE '05: Proc. of Conf. on Design, Automation and Test in Europe*, 2005.

[11] G. W. Dunlap et al. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[12] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. Soft-Error Detection Using Control Flow Assertions. In *DFT '03: Proc. of 18th IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 2003.

[13] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *ISCA '03: Proc. of 30th Intl. Symp. on Comp. Arch.*, 2003.

[14] W. Gu, Z. Kalbarczyk, and R. K. Iyer. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. In *DSN '04: Proc. of the 2004 Intl. Conf. on Dependable Systems and Networks*, page 887, Washington, DC, USA, 2004. IEEE Computer Society.

[15] R. Guo et al. Evaluation of Test Metrics: Stuck-at, Bridge Coverage Estimate and Gate Exhaustive. In *VTS '06: Proc. of 24th IEEE VLSI Test Symp.*, 2006.

[16] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault Injection Techniques and Tools. *Comp.*, 30(4), 1997.

[17] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: a flexible software-based fault and error injection system. *IEEE Trans. on Comp.s*, 44(2), 1995.

[18] M. Martin et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4), 2005.

[19] M. Mueller, L. C. Alves, W. Fischer, M. L. Fair, and I. Modi. RAS Strategy for IBM S/390 G5 and G6. *IBM J. Research and Development*, 43(5/6), Sept/Nov 1999.

[20] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer. An Architectural Framework for Detecting Process Hangs/Crashes. In *EDCC*, 2005.

[21] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA '02: Proc. of 29th Intl. Symp. on Comp. Arch.*, 2002.

[22] V. K. Reddy, A. S. Al-Zawawi, and E. Rotenberg. Assertion-Based MicroArch. Design for Improved Fault Tolerance. In *Proc. of 24th Intl. Conf. on Comp. Design (ICCD-24)*, 2006.

[23] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *ISCA '00: Proc. of 27th Intl. Symp. on Comp. Arch.*, 2000.

[24] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.*, 2(4), 2005.

[25] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *FTCS '99: Proc. of Twenty-Ninth Intl. Symp. on Fault-Tolerant Computing*, 1999.

[26] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer. An Experimental Study of Soft Errors in Microprocessors. *IEEE Micro*, 25(6), 2005.

[27] M. A. Schuette and J. P. Shen. Processor control flow monitoring using signatured instruction streams. *IEEE Trans. Comput.*, 36(3), 1987.

[28] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Proc. of 12th Intl. Conf. on Architectural support for programming lang. and operating systems*, 2006.

[29] D. Sorin, M. M. Martin, M. Hill, and D. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proc. of 29th Intl. Symp. on Comp. Arch.*, 2002.

[30] J. Srinivasan, S. V.Adve, P. Bose, and J. A. Rivers. The Impact of Scaling on Processor Lifetime Reliability. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, 2004.

[31] J. Srinivasan, S. V.Adve, P. Bose, and J. A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proc. of 32nd Intl. Symp. on Comp. Arch.*, 2005.

[32] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray. Low-Cost On-Line Fault Detection Using Control Flow Assertions. *iolts*, 00, 2003.

[33] Virtutech. Simics Full System Simulator. Website, 2006. `http://www.simics.net`.

[34] N. Wang and S. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *Dependable and Secure Computing, IEEE Trans. on*, 3(3), July-Sept 2006.

[35] E. F. Weglarz, K. K. Saluja, and T. M. Mak. Testing of Hard Faults in Simultaneous Multithreaded Processors. In *IOLTS '04: Proc. of Intl. On-Line Testing Symp., 10th IEEE (IOLTS'04)*, 2004.

[36] D. Yen. Chip Multithreading Processors Enable Reliable High Throughput Computing. In *Keynote address at Intl. Reliability Physics Symp.*, 2005.

[37] P. Zhou et al. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants. In *37th IEEE/ACM Intl. Symp. on Micro-architecture (MICRO)*, 2004.