Appears in Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, March, 2008.

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

Man-Lap Li, Pradeep Ramachandran, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, Yuanyuan Zhou

Department of Computer Science
University of Illinois at Urbana-Champaign
swat@cs.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 cooperative hardware-software solution that watches for anomalous software behavior to indicate the presence of hardware faults. 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, resulting in identifying low-cost detection methods and providing guidelines for implementation of the recovery and diagnosis components of such a 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. Our main results are: (1) we are able to detect 95% of the unmasked faults in 7 out of 8 studied microarchitectural structures with simple detectors that incur zero to little hardware overhead; (2) over 86% of these detections are within latencies that existing hardware checkpointing schemes can handle, while others require software checkpointing; and (3) a surprisingly large fraction of the detected faults corrupt OS state, but almost all of these are detected with latencies short enough to use hardware checkpointing, thereby enabling OS recovery in virtually all such cases.

Categories and Subject Descriptors B.8.1 [*Reliability, Testing and Fault-Tolerance*]

General Terms Reliability, Experimentation, Design

Keywords Error detection, Architecture, Permanent fault, Fault injection

* This work is supported in part by an IBM faculty partnership award, the Gigascale Systems Research Center (funded under FCRP, an SRC program), the National Science Foundation under Grants NSF CCF 05-41383, CNS 07-20743, and NGS 04-06351, and an equipment donation from AMD.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'08. March 1–5, 2008, Seattle, Washington, USA.

Copyright © 2008 ACM 978-1-59593-958-6/08/0003...$5.00

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, design defects, and so on [4]. 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 a recent workshop, an industry panel converged on a 10% area overhead target to handle all sources of chip errors as a guideline for academic researchers [40]. 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 significant overheads in performance and/or power [38].

Two high-level observations motivate our work. First, the hardware reliability solution needs to handle only the device faults that propagate through higher levels of the system and become observable to software. Second, despite the reliability threat, fault-free operation remains the common case and must be optimized, possibly at the cost of increased overhead after a fault is detected (in accordance with Amdahl's law).

These observations motivate a strategy where faults are detected by watching for anomalous software behavior, or *symptoms* of faults, using zero to low-cost hardware and software monitors. Such a strategy treats hardware faults analogous to software bugs, potentially leveraging solutions for software reliability to further amortize overhead. Detecting faults at the software level can incur a significant delay from the point the fault was first activated, potentially complicating the fault diagnosis process for repair/reconfiguration for hard faults. We claim that this is the right tradeoff to enable low-cost detection since diagnosis is required only in the infrequent case of a fault.

Such a combination of simple high-level detection and potentially more complex and low-level diagnosis assumes a checkpoint/replay mechanism for recovery, which is also required for various other proposals for reliability as well as for other functions (e.g., transactional memory and speculative multithreading). This mechanism can be leveraged by the diagnosis process to repeatedly rollback and replay the execution trace that produced the detected

symptom to iteratively narrow down the source of the fault. Although we use software symptoms to detect hardware errors, the diagnosis and recovery components prevent these symptoms from becoming visible externally (providing external observers the illusion of near-perfect hardware). We rely on a thin firmware layer to control the coordination of and among the detection, diagnosis, and recovery components of the system.

Our cooperative hardware-software approach naturally extends to incorporate backup detection techniques (e.g., hardware checkers, selective redundancy, online test) for the cases where the high-level symptom-based detection coverage is determined to be insufficient; e.g., for some mission-critical applications or in case of some faults in some structures that may not easily reveal detectable symptoms at the required cost. Compared to any one such technique used in isolation, the potential advantages of our approach are:

Generality. High-level symptom-based detection techniques are largely oblivious to specific low-level failure modes or microarchitectural/circuit details. Thus, in contrast to detection methods that are driven by specific device-level fault models (e.g., wear-out detectors), high-level detection techniques are more general and extensible to numerous failure mechanisms and microarchitectures.

Ignoring masked faults. Previous work has shown that a large number of faults are masked by higher levels of the system such as circuit, microarchitecture, architecture, and application levels [9, 18, 20, 28, 48]. High-level detection techniques naturally ignore faults that are masked at any of these levels, avoiding the corresponding overheads.

Optimizing for the common case. Total system overhead is potentially reduced by emphasizing minimal detection overhead (which is paid all the time), possibly at the cost of higher diagnosis overhead (which is paid only in the case of a fault).

Customizability. A software (firmware) controlled system with detection mechanisms driven by software behavior provides a natural way for application-specific and system-specific customization of the reliability vs. overhead tradeoff. For example, when a fault is detected in a video application, the system may consider dropping the current frame computation rather than recovering it. Further, the approach is amenable to selective cost-conscious use of different symptom-based and backup detection techniques.

Amortizing overhead across other system functions. Our view of monitoring for software symptoms of hardware bugs is inspired by work on on-line software bug detection [11, 15, 22, 51, 52, 53]. Our approach can leverage software bug detection techniques for hardware fault detection and vice versa, amortizing overheads for full system reliability.

Is such a cooperative hardware-software solution that detects hardware faults through anomalous software behavior feasible for hardware reliability? And how should it work? Those answers fundamentally depend on the answers to several key questions, which we investigate in this work:

- For which microarchitectural structures do hardware faults produce *detectable* anomalous software behavior with very high probability? Others may need specialized hardware protection.
- How long does it take to detect the fault from the time it corrupts the architectural state? This detection latency impacts the recovery strategy: short latencies allow simple hardware checkpoint/recovery, long latencies may require more complex hardware and/or software checkpointing/recovery, and excessively long detection latencies may not be recoverable.
- How frequently do hardware faults corrupt operating system state? What is the detection coverage and latency for such faults? The OS typically needs a very high level of reliability. Further, software checkpointing and recovery of the OS is complex and therefore low-latency detection will be important to make hardware checkpointing and recovery of OS state feasible.

The bulk of our experiments here focus 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 manifest to software faster than a transient (because it lasts longer), but for the same reason, it is less likely to be masked and more likely to corrupt the OS with an irrecoverable system failure (unless intercepted quickly). Further, after a permanent fault is exposed, the system must diagnose its source and repair or reconfigure around the faulty unit. This is generally expensive, limiting the number of affordable false positives (unlike some detection techniques for transients [49]). Nevertheless, for completeness, we summarize the main results of our experiments for transients.

To answer the above questions, we inject a total of 12,800 permanent faults (stuck-at and bridging faults) in several microarchitectural structures in a modern processor running SPEC benchmarks. We use a full system microarchitecture-level simulator and simulate the faulty hardware for about 10 million instructions after the fault is injected (one fault at a time). We monitor for symptoms indicating anomalous software behavior in this window. Faults that are not detected within this window are functionally simulated to completion to identify additional masking effects and Silent Data Corruptions. Ideally, we would use a lower-level simulator for fault injections (e.g., gate 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. Our primary findings are as follows:

- **Detection coverage:** Across 7 of the 8 microarchitectural structures studied, 95% of the unmasked faults are detected via simple symptoms (such as fatal hardware traps, hangs, high OS activity, and abnormal application aborts that can be intercepted by the OS) within the 10-million-instruction detailed simulation window. For the remaining faults, functional simulation to completion showed that only 0.8% result in silent data corruptions (SDC) (the rest eventually produce one of our symptoms, although we do not count them as "detected" for coverage). Overall, these results show that most permanent faults that propagate to software are easily detectable through simple symptoms.
- **Detection latency for applications:** The latency from the time that application state is corrupted to the time the fault is detected is $\leq$ 100K instructions (microseconds range for GHz processors) for 86% of the detected cases – this can be handled with hardware checkpointing schemes [29, 43], using simple buffering of persistent state output (input) to solve the output (input) commit problems. The higher latency cases can be handled using software checkpointing and recovery.
- **Impact on OS:** Surprisingly, a large fraction of the faults corrupt operating system state even for SPEC applications. Although in fault-free mode, SPEC applications spend negligible time in the OS, a fault often invokes the OS (e.g., by causing a TLB miss) and, because it is persistent, subsequently corrupts OS state, making it important to understand the impact of faults on the OS. We find the latency from an OS architectural state corruption to a fault detection is within 100K OS instructions for virtually all detected faults. This implies that hardware checkpointing of OS state is feasible to recover the OS from nearly all faults – this is important since it is difficult to recover the OS using mechanisms that involve external software.

This work is part of a larger project called **SWAT** (**S**oft**W**are **A**nomaly **T**reatment), where we are investigating the design of a resilient system driven by high-level detection as motivated above. The results in this paper clearly establish the feasibility of such an approach and provide key guidelines for implementing the SWAT system and future resilient systems (Section 6).

2. Related Work

Software-centric detection and fault injection and propagation studies.

There is a large body of literature on detecting hardware faults through monitoring software behavior [12, 30, 31, 33, 34, 36, 46, 49]. The majority of this work focuses on control flow signatures, crashes, and hangs. Recent work has also examined value based invariants extracted in hardware [33] and invariants in software that are extracted ahead-of-time [31] for detecting errors; these schemes are analogous to our preliminary work on such invariants in software discussed in Section 6. There is also a large body of work that performs hardware (and software) fault injections to characterize the fault tolerant behavior of a system [1, 14, 16, 17]. Both these classes of work perform fault injections and follow the propagation of the fault 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-centric techniques, and which would require more expensive hardware support. 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 intermittents are usually modeled like transients except that they last a small number of cycles (e.g., up to 4 cycles). We focus on permanent faults (and only summarize our results for transients) because they are predicted to become increasingly important with growing concern from phenomena such as aging and inadequate burn-in [4, 44, 50]. Permanent faults are significantly different from transients and intermittents that last a few cycles because of their lower masking rate, consequently higher potential to impact the OS, and higher complexity of diagnosis (and consequent requirement of low number of false positives) as described in Section 1.

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 [49]), our work is distinguished by our study of both the application and OS 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.

Concurrent with this work, Meixner et al. have proposed the use of data flow checkers for transient and permanent faults [26]. Subsequently, they proposed the use of these and previous checkers (e.g., control flow checkers) to detect all faults in simple single-issue, in-order pipelines, with no interrupts [25]. Our symptom-based detectors work at a much higher level – they are largely oblivious to the microarchitecture and require very little hardware overhead. In the future, it will be interesting to compare the coverage and detection latencies of these classes of checkers.

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) [3, 27]. 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 transients [2, 12, 13, 35, 36, 38, 49], recently, there has been substantial work on handling hard errors. We focus on that work here.

Austin proposed DIVA, an efficient checker processor that is tightly coupled with the main processor's pipeline to check every committed instruction for errors [2]. While DIVA can be used to provide detection of hard errors, it does not provide mechanisms for diagnosis or repair. Bower et al. incorporated hard error diagnosis with DIVA checkers [6], using hardware counters that identify hard faults through heuristics based on the usage of different structures.

Shyam et al. 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 [41]. Since these tests are run only when the structures are idle, the performance loss incurred is rather small. Constantinides et al. enhanced this scheme further in [8] by adding hardware support so that the software can control the online testing process, adding flexibility for choosing test vectors. However, the performance penalty incurred by software-controlled online testing is high for reasonable hardware checkpointing intervals. Furthermore, the continuous testing of hardware can accelerate the wear-out process.

All of the above schemes incur significant overhead in area, performance, power, and/or wear-out that is paid almost all the time; further, these are customized solutions for hardware reliability. In contrast to the above, we seek a reliability solution that pays minimal cost in the common case where there are no errors, and potentially higher cost in the uncommon case when an error is detected. For example, using fatal traps as a detection mechanism has zero detection overhead until there is actually an error. We also require checkpoint/rollback support; however, analogous support is assumed by previous schemes as well [7, 8, 25, 41]. Additionally, we allow for the possibility of checkpoint support in software and leveraging such support that may be already present for software reliability. Finally, since we detect at the software level, we only detect errors that are not masked by the hardware or the software.

3. SWAT System Assumptions

There are a few essential properties of the SWAT system that provide the context necessary to understand this work:

- As noted in Section 1, we assume that the firmware-controlled diagnosis and recovery (of OS and applications) prevents symptoms of hardware errors from becoming visible externally. The goal is to give the illusion that hardware is near-perfect.
- The diagnosis component assumes a multicore system where a fault-free core is always available, and also assumes a checkpoint/replay mechanism.

  When a symptom is detected, the diagnosis process re-executes the program from the last checkpoint on the same core. If the symptom does not recur, it is diagnosed as a transient and execution continues. If the symptom recurs, execution is rolled back and restarted on a different core. If no symptom is observed, the problem is identified as either a permanent fault in

the original core or a non-deterministic software fault. We then rollback and re-execute on the original core and if the fault recurs, we assume it is a hardware fault. To further diagnose this fault, we run more special-purpose diagnostics and use these to select appropriate repair/reconfiguration actions, e.g., either at the level of the entire core or specific microarchitectural structures (with appropriate hardware hooks, the diagnosis procedure can narrow a permanent fault to within a structure inside the core). If the symptom persists on the new core, it is considered likely a software fault and is left to external software as usual.

The overall diagnosis latency will depend on the symptom detection latency, consequent checkpoint/replay mechanisms used, and context migration latency. While this latency could potentially be large, it is only paid in the infrequent event of a fault, and we believe it to be an appropriate trade-off in exchange for the low-cost "always on" symptom-based detection.

- For recovery, the SWAT system again assumes some form of checkpoint/replay mechanism is available. Depending on the system and application requirements (e.g. cost, detection latency, etc.), hardware checkpointing, software checkpointing, or a hybrid of hardware/software checkpoint/replay can be used. Hardware checkpoint/replay has been proposed for many purposes apart from reliability (e.g., transaction memory, speculative parallelism). SafetyNet [43] and ReVive [29, 32] claim reasonably low overhead for fairly long windows for hardware checkpoint/replay. We therefore believe that hardware recovery overhead will be acceptable, especially as it is amortized for many causes. Similarly, many software reliability schemes already rely on software checkpointing, and we can leverage this technology by incorporating it as a transparent OS service [45]. Furthermore, the combination of recovery method can be customized to suit the system requirements.
- As with any system that tolerates permanent faults, we assume hardware with the ability to repair or reconfigure around such faults.

We emphasize that some of the above are design choices that are neither exhaustive (i.e., alternative designs are possible) nor definitive. Investigating the actual design for such a system is outside the scope of this work. The experimental results we present will provide valuable guidance in deciding these eventual design choices.

4. Methodology

4.1 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 trade-off 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 [23] in conjunction with the Virtutech Simics full system simulator [47]. Together, these simulators provide cycle-by-cycle microarchitecture level timing simulation of a real workload (6 SpecInt2000 and 4 SpecFP2000) running on a real operating system (full Solaris-9 on SPARC V9 ISA) on a modern out-of-order superscalar processor and memory hierarchy (Table 1). Although in

Base Processor Parameters	
Frequency	2.0GHz
Fetch/decode/execute/retire rate	4 per cycle
Functional units	2 Int add/mul, 1 Int div 2 Load, 2 Store, 1 Branch 2 FP add, 1 FP mul, 1 FP div/Sqrt
Integer FU latencies	1 add, 4 mul, 24 divide
FP FU latencies	4 default, 7 mul, 12 divide
Reorder buffer size	128
Register file size	256 integer, 256 FP
Unified Load-Store Queue Size	64 entries
Base Memory Hierarchy Parameters	
Data L1/Instruction L1	16KB each
L1 hit latency	1 cycle
L2 (Unified)	1MB
L2 hit/miss latency	6/80 cycles

Table 1. Parameters of the simulated processor.

the fault-free case, our simulated applications are not OS-intensive ($< 1\%$ OS activity in our simulated window), we show later that fault injection significantly increases OS activity. Thus, it is critical to model the OS and its interaction with the applications in our simulations. (More complex OS-intensive workloads such as databases would provide additional insight, and are part of our future work.)

To inject faults, we leverage the timing-first approach [24] used in the GEMS+Simics infrastructure. In this approach, an instruction is first executed by the cycle-accurate GEMS timing simulator. On retirement, the Simics functional simulator is invoked to execute the same instruction again and to compare the full architecture state in GEMS and Simics. This comparison allows GEMS the flexibility to not fully implement a small (complex and infrequent) subset of the SPARC ISA – GEMS uses the comparison to make its state consistent with that of Simics in case of a mismatch that would occur with such an instruction.

We modified this checking mechanism for the purposes of microarchitectural fault injection. We inject a fault into the timing simulator's microarchitectural state and track its propagation as the faulty values are read through the system. When a mismatch in the *architectural state* of the functional and the timing simulator is detected, we check if it is due to the injected fault. If not, we read in the value from Simics to correct GEMS' architectural state. However, if the mismatch is because of an injected fault, we corrupt the corresponding state in Simics (register and memory) with the faulty state from GEMS, ensuring that Simics continues to follow GEMS' execution trace, upholding the timing-first paradigm.

We say an injected fault is *activated* when it results in corrupting the architectural state, as above. If the fault is never activated, we say the fault is *architecturally masked* (e.g., a stuck-at-0 fault in a bit that is already 0 or a fault in a misspeculated instruction are trivially masked). Since we know the privilege mode of the retiring instruction that corrupts the state, we can determine if a fault leads to any corruption in the architectural state of the OS or the application. As discussed later, this information has important implications for recovery.

4.2 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 [4, 5, 50]. Precise fault models for wear-out are still a subject of research [41]. In this paper, we use the well established stuck-at-0 and stuck-at-1 fault models as well as the dominant-0 and dominant-1 bridging fault models. While the stuck-at fault models apply to faults that affect a single bit, the bridging fault models concern faults that affect adjacent bits. The dominant-0 bridging fault acts like a logical-AND between the adjacent bits that are marked faulty, while the dominant-1 bridging fault acts like a logical-OR. Prior work has suggested that

μarch structure	Fault location
Instruction decoder	Input latch of one of the decoders
Integer ALU	Output latch of one of the Int ALUs
Register bus	Bus on the write port to the Int reg file
Physical integer reg file	A physical reg in the Int reg file
Reorder buffer (ROB)	Src/dest reg num of instr in ROB entry
Register alias table (RAT)	Logical → phys map of a logical reg
Address gen unit (AGEN)	Virtual address generated by the unit
FP ALU	Output latch of one of the FP ALUs

Table 2. Microarchitectural structures in which faults are injected. In each run, either a stuck-at fault is injected in a random bit or a bridging fault is injected in a pair of adjacent bits in the given structure.

some wear-out faults may initially manifest as (intermittent) timing violations before resulting in hard breakdown [37]. Modeling such faults requires lower level simulation than our current infrastructure, along with its attendant trade-offs (Section 4.1). For future work, we are exploring a hybrid simulation model to get both fidelity and speed, but that is outside the scope of this paper.

Table 2 lists the microarchitectural structures and locations where we inject faults. For each structure, we inject a fault in each of 40 random points in each application (after initialization), one injection per simulation run. For each application injection point, we perform an injection for each of the 4 fault models (two stuck-at and two bridging). The injections are performed in a randomly chosen bit in the given structure for stuck-at faults. For bridging faults, the randomly chosen pair of adjacent bits are injected. This gives a total of 1600 fault injection simulation runs per microarchitectural structure (10 applications × 40 points per application × 4 fault models) and 12,800 total injections across all 8 structures.

After a fault is injected, we run for 10 million instructions in the detailed simulator, where we watch for software symptoms indicating the presence of a hardware fault. If a symptom does not occur in the detailed simulation, the potentially corrupted execution is functionally simulated to completion. Section 4.4 describes these cases in more detail.

For completeness, we also performed a total of 6400 transient fault injections (single bit flips) in the same microarchitectural structures. (The number of injections is fewer than for permanent faults because of fewer fault models.)

4.3 Fault Detection

We focus here on simple detection mechanisms that require little new hardware or software support. Our detection mechanisms look for four abnormal application or OS behaviors as symptoms of possible hardware faults: (1) fatal traps that would normally lead to application or OS crashes, (2) abnormal application exit indicated by the OS, (3) application or OS hangs, and (4) abnormally excessive OS activity. Each of these is discussed below. Note that detecting these symptoms implies that they are made transparent to the user. For example, on a fatal trap, the user will not see the crash; rather, the trap invokes the diagnosis and recovery components of the SWAT system as described in Section 3. We also note that faults injected in the application may be detected either in the application or OS since we consider permanent faults. Figure 1 summarizes the various outcomes of an injected fault in our study.

4.3.1 Fatal hardware traps

An easily detectable abnormal behavior due to a hardware fault is a *fatal* hardware trap in either the application or the operating system. A fatal trap is typically not thrown during a correct program execution. On Solaris, the following traps are denoted as fatal traps – RED (Recover Error and Debug) state trap (thrown when there are too many nested traps), Data Access Exception trap, Division by zero trap, Illegal instruction trap, Memory misaligned trap,

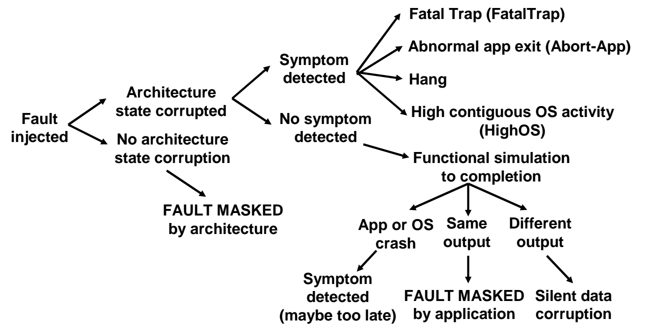


Figure 1. Outcomes of an injected fault. If the injected fault is not detected within 10M instructions, the fault is simulated to completion to identify its effect on the application's outputs.

and Watchdog reset trap (thrown when no instruction retires in the last 2^{16} ticks). Using these traps as symptoms of hardware faults requires no additional hardware overhead – in our proposed framework, such a trap would simply invoke a firmware routine that performs further diagnosis and recovery as needed (Section 3).

4.3.2 Abnormal application exit, indicated by the OS

Many application crashes are not visible through a hardware trap. For example, since the SPARC TLB is software-managed, hardware is unaware when the OS terminates an application due to a segmentation fault. However, the OS clearly knows this outcome. Similarly, an application may perform a graceful abort; e.g., it may exit after checking that the divisor is zero or the argument to a square root function is negative, or in general, after an assertion fires. Again, hardware is not informed of this abort, but the OS may know of the erroneous exit condition. In all of these cases, it is possible to modify the OS to first invoke the firmware routine that can diagnose the situation for a possible hardware fault and invoke recovery if needed.

Our simulation infrastructure is not yet set up to directly catch such OS invocations. Instead, for simulation purposes, once a state corruption is detected, we look for the OS idle loop - this indicates that the application was aborted as no other processes are running in the simulated system. We flag such an entry into the idle loop as a detected abnormal application exit (we verified that none of these were normal application exits).

4.3.3 Hangs

Another possible abnormal behavior due to a fault is a hang in the application or OS. Previous work has proposed hardware support to detect hangs with high fidelity, but with some area and power overhead [30]. Several optimizations to that work are possible. For example, a detector based on a simpler 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 our simulations, we use a heuristic based on monitoring all 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 implies the corresponding branch constitutes 1% of the total executed instructions), the detector flags a hang. The hang is in the OS if the detector flags a privileged branch instruction. We identified the threshold for flagging hangs through profiling the fault-free executions of the applications and masking out a handful of branches that did not satisfy this threshold. We did not optimize the threshold or the heuristic further because our results showed that hangs provided limited coverage.

4.3.4 High OS activity

An additional symptom we monitor is the amount of time the execution remains in the OS, without returning to the application. We profiled our applications and found that in a typical invocation of the OS, control returns back to the application after the OS executes for a few 10s of instructions, since trap handling routines are typically small pieces of code. We found two exceptions to this observation. First, on a timer interrupt after the allocated time quantum for the application expires, the OS scheduler may execute for much longer. Nevertheless, this duration did not exceed 10,000 instructions in any of our experiments, and we expect this number to be relatively application-independent (so it can be easily determined by profiling the OS). Second, for system calls (e.g., I/O), we observed that the OS may execute for much longer (10^5 or 10^6 instructions) before returning to the application.

Thus, as a symptom of abnormal behavior, we look for a threshold of over 30,000 contiguous OS instructions, *excluding* cases where the OS is invoked via a system call trap. This threshold corresponds to a conservative latency which is 3 times the maximum observed scheduler latency. This mechanism incurs low hardware overhead since it primarily uses a hardware instruction counter and can leverage already existing performance counters.

4.3.5 False positives

After a symptom is detected, if diagnosis (described in Section 3) determines that the symptom was not caused by a hardware fault, this symptom is deemed a false positive of the presence of a hardware fault. In these cases, symptoms such as fatal traps and application aborts are essentially software bugs and will simply be propagated to the appropriate software layer as usual. The additional diagnosis latency in these cases is acceptable since it is incurred in the case of a fault, albeit in software.

However, for symptoms such as hangs and high OS activity, the detection mechanisms themselves are prone to false positives as they are based on heuristics. When the diagnosis determines that one of these symptoms is determined to be a false positive of the presence of a hardware fault, the execution will simply continue (the diagnosis process may adjust the threshold for these detectors). In this case, the diagnosis latency is an overhead for fault-free execution and such cases must be reduced to an acceptable level. In general, there is a tradeoff between how aggressive the symptom detectors can get and the false positive rate.

4.4 Application Masking and Undetected Faults

A fault that corrupts the architectural state and does not invoke a detectable symptom in the 10M instruction detailed simulation window may be benign if it is masked by the application. Our detection mechanisms correctly do not detect such benign faults. To quantify these cases, we use functional (full-system) simulation to run the application to completion after 10M instructions (detailed simulation is too slow to run to completion). Note that the functional simulation does not inject any faults, and so the net effect for these cases is similar to an intermittent fault that lasts 10M instructions.

At the end of functional simulation, there are three eventual outcomes (for faults that are not architecturally masked or detected within 10M instructions) – the fault is masked by the application, causes a symptom with a latency >10M instructions, or results in a silent data corruption. We determine that a fault is masked by the application if the execution terminates gracefully and generates an output matching that of a correct (fault-free) execution. On the other hand, a fault could cause the application to abort or the system to crash during the functional simulation. These faults are categorized as symptom-causing faults with high latencies. Since we do not know the latencies and they may (or may not) be too long for recovery, we conservatively consider these faults as undetected

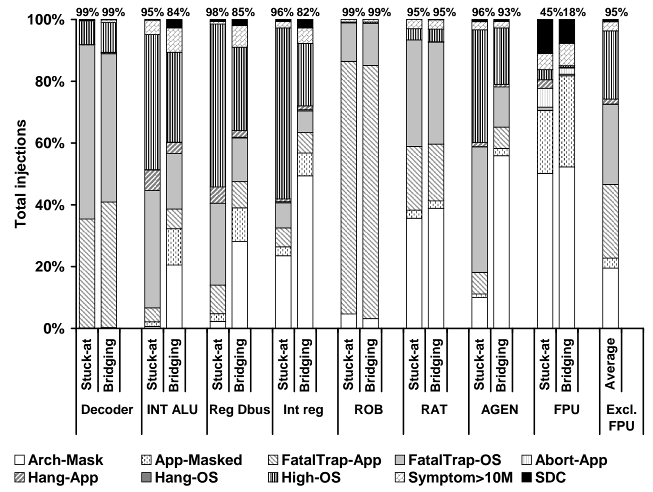


Figure 2. For each microarchitectural structure and fault model, the figure shows the impact of the injected faults. An injected fault may be masked by the architecture or the application. An unmasked fault may result in a Fatal Trap (from the application or the OS), Application Abort, Hang (of the application or the OS), or High-OS symptom. An unmasked fault not detected within 10M instructions is categorized as either Symptom>10M if it eventually exhibits a symptom or SDC if otherwise. The number above each bar is the coverage of our symptom-based detection scheme, conservatively assuming that the Symptom>10M faults are undetected. Our simple detectors show high coverage for permanent faults with only 0.8% of the injected faults resulting in SDCs.

when computing coverage (Section 4.5). In the worst case, the faulty execution terminates gracefully but generates a different outcome than that of a correct execution. We refer to this as silent data corruption or SDC.

4.5 Metrics

Coverage: The coverage of a detection mechanism is the percentage of non-masked faults it detects:

$$Coverage \quad = \quad \frac{Total\ faults\ detected}{Total\ injections\ -\ Masked\ faults}$$

where the Masked faults are faults masked by either the architecture or the application.

Detection latency: We report fault detection latency as the total number of instructions retired from the first architecture state corruption (of either OS or application) until the fault is detected.

As mentioned above, we consider only the faults that invoke our symptoms within the 10M instructions of detailed simulation as detected faults.

5. Results

5.1 How do Faults Manifest in Software?

We first show how the modeled permanent faults manifest in software, and the feasibility of detecting them with our simple detection mechanisms.

5.1.1 Overall Results

Figure 2 shows how permanent faults manifest in software for a given microarchitectural structure under each fault model. Stuck-at-0 and stuck-at-1 fault injections are combined under the *Stuck-at* bars and the dominant-0 and dominant-1 bridging faults are combined under the *Bridging* bars. The rightmost bar shows the average

data across all fault models in 7 of the 8 structures (excluding FPU). In each bar, the bottom two stacks represent the percentage of fault injections that are masked (by the architecture and the application, respectively), while the top-most (black) stack is the percentage of injections that result in SDCs. The Symptom>10M stack represents faults that result in symptoms (from either the application or the OS) after the detailed simulation window of 10M instructions.

The remaining stacks represent injections detected within 10M instructions using the symptoms discussed in Section 4.3. The figure separates the fatal hardware traps category into two, depending on whether the fatal trap was thrown by an application or OS instruction (FatalTrap-App and FatalTrap-OS, respectively). Similarly, it separates the hang category into Hang-App and Hang-OS, depending on whether the hang detector saw a hang in the application or OS code (determined by the privilege status of the instructions).

The number above each bar indicates the coverage for that structure under the given fault model. As mentioned in Section 4.4, we conservatively assume that the Symptom>10M stack is undetected for the coverage computation.

The key high-level results are:

- For the cases studied, permanent faults in most structures of the processor are highly software visible. 95% of faults that are not masked (except for the FPU) are detected using our simple detection mechanisms, demonstrating the feasibility of using high-level software symptoms to detect permanent hardware faults.
- For the FPU, 65% of the activated faults are not detected, suggesting that alternate techniques may be needed (e.g., redundancy in space, time, or information) for the FPU.
- Many of the faults are detected when running the OS code (the FatalTrap-OS, Abort-App, Hang-OS, and High-OS categories), even though the fault-free applications themselves are not OS intensive.
- The FatalTrap and High-OS categories make up the majority of the detections (66% and 30% respectively of all detected faults) while the Abort-App and Hang categories are the smallest ($\leq$2% each).
- For the faults not detected within the 10M instruction window, except for FPU, only 0.8% of the original injections result in silent data corruptions. The rest eventually lead to application/OS crashes or are masked by the application.

The rest of this section provides deeper analysis to understand the above results.

5.1.2 Analysis of Masked Faults

For stuck-at faults, Figure 2 shows a low architectural masking rate for many structures. This is because the injected fault is a permanent fault that potentially affects every instruction that uses these faulty structures during its execution. Exceptions are the integer register file, the RAT, and the FPU, where the architectural masking rate for stuck-at faults is about 25% to 50%. Architectural masking for an integer (physical) register occurs if it is not allocated in the simulated window of 10M instructions. Similarly, a RAT fault is masked if it affects the physical mapping of a logical register that is not used in this window. The high FPU masking rate occurs because of the integer applications.

Bridging faults also see the above phenomena for architectural masking. Additionally, most structures on the 64 bit wide data path (INT ALU, register DBus, integer register file, and AGEN) see a significantly higher architectural masking rate for bridging faults than for stuck-at faults. This difference stems from faults injected

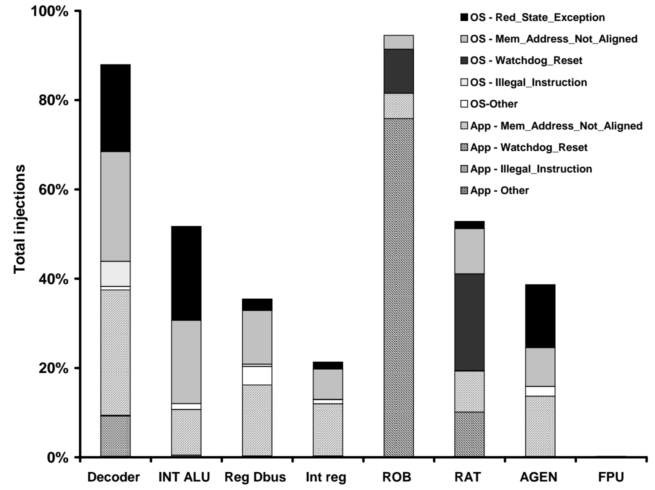


Figure 3. Distribution of detections by fatal traps. The *Other* category constitutes Data Access Exception, Protection Violation and Division by Zero traps, which make <8% of detections by fatal traps. The total height of a bar is the percentage of the total faults in the corresponding structure that caused fatal hardware traps.

in the upper 32 bits of the 64 bit fields (roughly half of total fault injections in those structures). Since many computations only use the lower 32 bits, the higher order bits are primarily sign extensions, with either all 0s (for positive numbers) or all 1s (for negative numbers). In either case, since adjacent bits are identical, bridging faults are rarely activated for higher order bits, resulting in a higher masking rate for these faults.

Relative to architectural masking, application masking is small but significant (6% of total injections). Many of these cases stem from faults injected in the higher order bits of the 64 bit data path – in some cases, these appear as architecture state corruptions (because the full 64 bit field is examined), but are actually masked at the application level due to smaller program level data sizes. These faults illustrate a benefit of our symptom-based detection approach since these benign faults are correctly ignored by our detectors.

5.1.3 Analysis of Detected Faults

Unmasked faults in many structures are highly visible as they are permanent in nature and are highly intrusive to the program's execution. Consequently, they often directly affect the control flow and memory access behavior of the program, which leads to detectable abnormal program behavior.

Large number of detections in the OS.

Surprisingly, in spite of the low OS activity for the fault-free runs of the simulated benchmarks, over 65% of the detected faults are detected through symptoms from the OS (Abort-App, FatalTrap-OS, Hang-OS and High-OS). Although the injected fault first corrupts the application, a common result of the fault is a memory access to an incorrectly generated virtual address. Since the address has not been accessed in the past, it invokes a TLB miss that would not have otherwise occurred. Because the SPARC TLB is software managed, this results in a trap invoking the OS. As the OS is executing on the same faulty hardware and, in general, is more control and memory intensive, the fault often will corrupt the OS state and result in a detectable symptom.

Fatal Hardware Traps.

66% of the fault detections are from fatal hardware traps. Figure 3 shows the distribution of the different types of these fatal traps. The height of a bar is the percentage of fault injections in the

corresponding structure that causes fatal traps. Fatal traps caused by the application are shown in the bottom (hatched portions) and those caused by the OS are shown on top (non-hatched portions).

Illegal instruction traps result when a fault changes the opcode bit in the instruction to an illegal opcode. As expected, these traps result mostly for decoder faults. However, they account for $<16\%$ of the fatal traps seen on decoder faults. This is because many injected faults in the instruction word either do not affect the opcode bits, or when they do affect opcode bits, they change the instruction into another valid instruction.

The watchdog timer reset trap is thrown when no instruction retires for more than 2^{16} ticks. These mostly occur in the ROB and RAT (over 90% and 59% of detected faults, respectively). ROB faults may change an instruction's source or destination register. If the source is changed to a free physical register, the instruction waits for data indefinitely. If the destination is changed, the dependent instructions indefinitely wait for their source operand. Faults in the RAT could also cause similar behavior. For example, the corrupted logical-to-physical register mapping could result in mapping a non-free physical register (say $preg_{23}$). Now that $preg_{23}$ is mapped to two logical registers (say r_2 and r_5), any subsequent instruction that writes to r_2 (r_5) will free $preg_{23}$ and instructions that read r_5 (r_2) wait for $preg_{23}$ indefinitely (since $preg_{23}$ is freed and marked *not ready*). However, since the ROB is a circular buffer and is heavily used, faults in the ROB are highly intrusive, frequently resulting in this trap. The RAT, however, is an array structure, some entries of which are never used in the simulated execution window. Hence, the number of such resulting watchdog timer reset traps are fewer from the RAT than from the ROB.

Misaligned accesses are common in all structures, accounting for over 44% of all the fatal traps thrown. Faults in most structures naturally affect the computation of memory addresses (e.g., all cases where a fault may affect the data or identity of a register used to compute an address). This often results in misaligned addresses, causing a misaligned access trap (Solaris requires addresses to be word aligned).

Red state exception is thrown when there are too many nested traps. The SPARC V9 architecture throws this exception when a trap at (maximum_trap_level - 1) occurs. The simulated processor has a maximum_trap_level of 5; i.e., at most four nested traps are allowed. This fatal trap situation constitutes roughly 15% of the fatal traps. The injected fault results in invoking the OS through a trap. When this trap handler executes, it re-activates the fault resulting in a nested trap, eventually leading to a RED state exception.

High OS.

The High-OS symptom has the next highest detection coverage after fatal traps (30%). In the majority of these cases, the application computes a faulty address invoking the OS on a TLB miss. The persistent hardware fault corrupts the TLB handler, resulting in the code never returning to the application.

This symptom has significant coverage overlap with fatal traps and hangs – removing this detector reduces the total coverage for all structures except FPU by about 15% (instead of the 30% if there were no overlap). This is because most of these cases eventually also lead to fatal traps and hangs. However, even for these cases, detection using the High-OS symptom significantly brings down detection latency (Section 5.3).

Hangs and application aborts.

The Abort-App symptom provides only 1% coverage. However, for the FPU, this symptom detects a high fraction of the detected faults (66%). In these cases, the application performs an illegal operation due to the injected fault (e.g., square root of a negative number), which causes the application to abort.

Hangs account for less than 3% coverage, with practically all hangs in the application code. An example of a hang is when a loop index variable is computed erroneously and the loop termination condition is never satisfied. While some hangs may result from the OS, the High OS symptom catches these before the hang detector can identify them as hangs. Thus, without the High-OS detector, hangs would provide higher coverage (but at a higher latency).

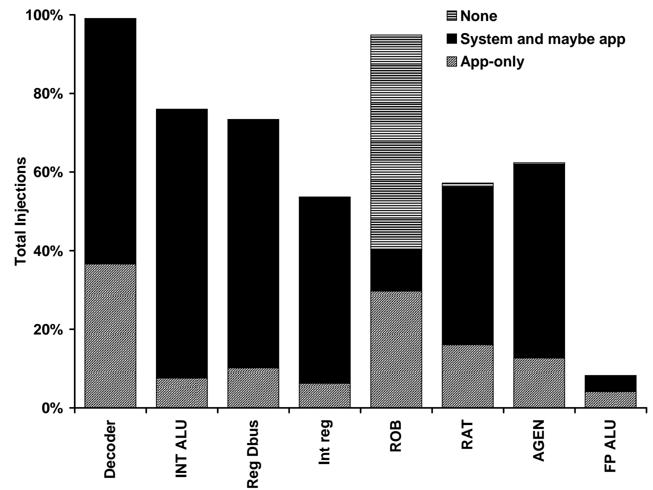


Figure 4. Application and system state integrity for the detected faults. The height of each bar gives the percentage of injected faults detected in that structure. We see that most faults corrupt the system state.

5.1.4 Analysis of Undetected Faults

Faults that are not masked and are not detected within the 10M instruction window of detailed simulation are divided into two categories – those that invoke a detectable symptom in the functional simulation portion of the execution (Symptom>10M) and those that terminate gracefully with a wrong output or silent data corruption (SDC). The detection latency for the former class of faults may or may not be short enough for full recovery (e.g., by rolling back to a software checkpoint). Nevertheless, eventual detection is better than the latter class of SDC-causing faults.

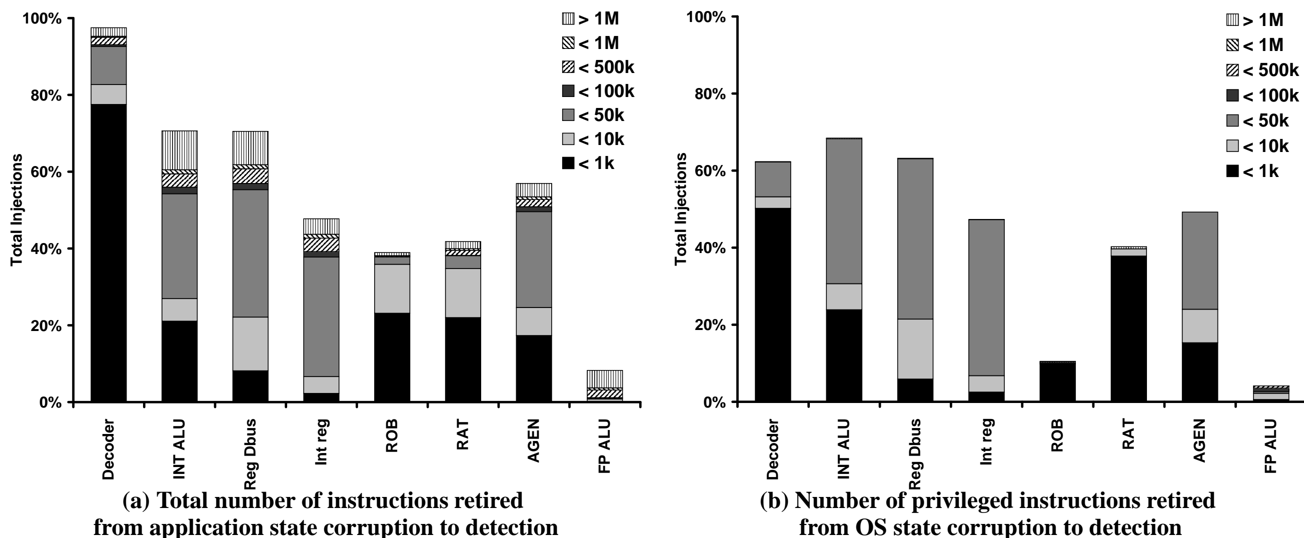
Figure 2 shows that for faults in all structures but the FPU, only 0.8% of the injected faults result in SDCs. This is a rather low number given our simple fault detectors, and shows that our symptom-based detection techniques are effective for these structures. Section 6 describes future work on more sophisticated symptom detection that has the potential to reduce this number even further.

For the FPU, 10% of the injected faults result in SDCs, largely because FPU computations less frequently affect memory addresses or program control (which are most responsible for detectable symptoms). Thus, our results show that the FPU requires alternate (potentially higher overhead) mechanisms to our simple symptom-based detectors. Section 6 discusses this further.

5.2 Software Components Corrupted

We next focus on understanding which software components (application or OS) are corrupted before a fault is detected (within the 10M instruction window of detailed simulation). This has clear implications for recovery. If only the application state is corrupted, it can likely be recovered through application-level checkpointing (for which there is a rich body of literature). However, OS state corruptions can potentially be difficult – software-driven OS checkpointing has been proposed only for a virtual machine approach so far [10] and the feasibility of hardware checkpointing would depend on detection latency.

For each structure, Figure 4 shows the percentage of fault injections that resulted in only application state corruption, OS (and



(a) Total number of instructions retired from application state corruption to detection

(b) Number of privileged instructions retired from OS state corruption to detection

Figure 5. Detection latencies for different structures, measured from (a) the first application state corruption and (b) the first OS state corruption. The latency is within 100K for 86% of the detected application state corruptions and for virtually all OS state corruptions, making hardware recovery feasible for the OS and for most application corruptions.

possibly application) state corruption, and corruption of neither the application nor the OS. The total height of each bar gives the percentage of faults injected into the given structure that resulted in a detected symptom.

Our main result here is that *over 65% of detected faults corrupt OS state before detection,* motivating exploration of checkpointing the OS and/or fault-tolerant strategies within the OS.

We note that whether the application/OS state was corrupted is not necessarily correlated with whether the fault was detected at an application/OS instruction (discussed in Section 5.1). A fault could be detected at an OS instruction, but may have already corrupted the application state. Similarly, a fault could be detected in application code, but meanwhile the application may have invoked the OS resulting in a (so far undetected) corruption in the OS state.

Additionally, there are a few detected fault cases where neither the application nor the OS state is corrupted (58% of detected faults in the ROB and 2% in the RAT). In all of these cases, the faults cause watchdog reset fatal traps to be thrown – the instruction at the head of the ROB never retires because its source physical register (say $preg_{head}$) never becomes available. These cases usually involve fairly complex interactions involving the ROB and the RAT. For example, consider a fault in the ROB that corrupts the destination field of a prior instruction that was supposed to write to $preg_{head}$. Because of the fault, the prior instruction writes to another physical register and never sets $preg_{head}$ as available. If the corrupted destination was previously free, then this does not corrupt the architectural state (our implementation of register renaming records the corrupted destination name in the retirement RAT (RRAT) when the corrupted instruction retires, thereby preserving the architectural state).

5.3 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 state 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 trade-offs. Software checkpointing of the OS is difficult and so far has only been proposed for a virtual machine approach [10]. Therefore, short detection latencies coupled with hardware support for checkpointing are likely to be more effective for OS recovery.

5.3.1 Latency from Application State Corruptions

For each structure, Figure 5(a) shows histogram data for detection latencies for fault injections that result in corrupting the application state. The latency is measured in terms of the number of retired instructions from the first application architecture state corruption to detection. The total height of each bar is the percentage of fault injections that corrupted the architecture state and were detected for that structure (within the 10M instruction window). Overall, about 39% of the detected faults that corrupt application architecture state have a latency of $<$1K instructions. These cases can be easily handled with simple hardware checkpoint and recovery techniques [42]. Further, 86% of the cases have a detection latency of $<$100K instructions (μs range for GHz processors). These cases can also be handled in hardware, albeit with more sophisticated support; e.g., SafetyNet supports multiple checkpoints with a checkpoint interval of 100K cycles [43]). Further, simple buffering can be used to replay persistent state output and input to solve the input/output commit problem.

On the other hand, the remaining application state corruptions (with detection latencies reaching millions of instructions) are currently infeasible for hardware recovery and will likely require software checkpointing techniques. These cases require considering a trade-off between complete recovery by buffering persistent state outputs and inputs for 100K to 10 million instructions (few 100's of microseconds to milliseconds for GHz processors) or risking incomplete recovery while immediately committing external outputs. Nonetheless, milliseconds of delay for many output operations (e.g., disks) do not violate software semantics and so should not pose a problem.

Hence, when the underlying hardware fault corrupts only the application, hardware- and/or software-level checkpoint and recov-

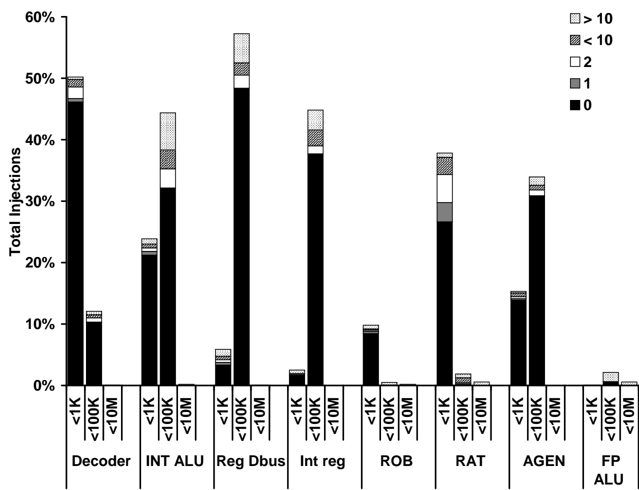


Figure 6. Number of times the OS-Application boundary is crossed from the first OS architecture state corruption to detection, for different detection latencies.

ery methods can be exploited, depending on the type of coverage vs. overhead trade-off desired.

5.3.2 Latency from OS State Corruptions

Figure 5(b) shows histograms of detection latency from OS state corruptions, measured as the number of OS instructions retired from the first OS architectural state corruption to detection. This is sufficient because an OS checkpoint/recovery mechanism need only keep track of OS instructions since applications cannot directly affect OS state.

The figure shows that over 42% of the detected faults in all structures are detected within 1K OS instructions, and virtually all (over 99%) are detected within 100K OS instructions. Thus, hardware checkpoint/recovery schemes (e.g., as in [29, 43]) can provide efficient OS recovery for our framework.

Finally, 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 application's execution and the OS execution within this interval governs the complexity of the OS recovery schemes. Figure 6 shows the histogram of the number of times the Application-OS boundary is crossed from the OS state corruption to detection. 80% of the detected OS corruptions were detected before the OS switched back to the application (zero crossings), suggesting that a naive checkpointing scheme that does not consider OS to application switches can provide system recovery for a large fraction of the cases once the fault is detected. Additionally, checkpoint/recovery hardware that handles a small number of OS-App crossings (<10) can recover the system in most (92%) cases.

5.4 Transient Faults

For our transient fault injection experiments, we found that over 94% of the faults are architecturally masked within the 10M instruction window. Of the remaining faults, 56% are detected within the 10M instruction window. We then simulated the rest of the cases to completion. 47% of these cases are masked by the application (bringing the overall masking rate to 96%) and 49% eventually raise detectable symptoms before termination. Overall, only 0.1% of the total injections result in SDCs. These results are consistent with previous studies [39, 49], and have the same implications for our approach as the results with permanent faults.

6. Implications for Resilient System Design

The findings in this paper provide several new and concrete guidelines for low-cost resilient system design.

Detection. Our results unequivocally show that for most microarchitectural structures, a large majority of permanent faults that propagate to software are detectable through low-cost monitoring of simple symptoms – 7 of 8 structures showed 95% coverage with detailed simulation spanning 10M instructions, and only 0.8% of injected faults result in Silent Data Corruptions (SDCs) (after running the applications to completion). The most powerful symptoms were fatal hardware traps (needing zero hardware cost) and high OS activity (needing a simple instruction counter). Further coverage was achieved with a hang detector (needing modest hardware support) and through detecting application aborts (needing very simple software support). These detection strategies would also be useful to detect software bugs.

The coverage and latency of our detection schemes are likely to improve further by using more sophisticated detectors. One powerful method is the use of program invariants, which have been previously studied for both (transient) hardware error detection [31, 33] and software bug detection and diagnosis [15, 21, 51]. To this end, we conducted preliminary experiments using sophisticated detectors derived from value-based invariants. We considered simple range-based invariants on integer function return values and values of integer loads and stores (i.e., invariants that specify constant upper and lower bounds on these values) and used the LLVM compiler infrastructure [19] to insert these invariants into the code. These experiments were done for three benchmarks: mcf, gzip and twolf. The results showed that value-based invariants significantly strengthened our detection scheme by improving coverage, shortening the detection latency for a majority of the faults, and (most importantly) *eliminating all but one of the SDC cases* for these 3 benchmarks. These results are encouraging for using more sophisticated symptoms when additional fault coverage is required by certain classes of applications.

Finally, for some structures like the FPU where faults were largely undetected, we will explore the alternatives above and classical mechanisms (e.g., residue codes, space/time redundancy).

Recovery and diagnosis. The relatively low detection latencies shown here facilitate checkpoint/replay based recovery and diagnosis. A specific challenge is the recoverability of the OS. Our results show that even for SPEC applications, which have low OS activity in fault-free runs, a large fraction of the faults corrupt the OS; therefore, much care is needed to make our system recoverable from OS failures. At the same time, we also see that the number of OS instructions executed from the time that the OS state is actually corrupted to the time of detection is less than 100K in virtually all cases. These results suggest that hardware checkpoint/replay techniques, such as ReVive [29] and SafetyNet [43] may be adequate for OS recovery, in terms of hardware state required, performance overhead, and simple solutions to the input and output commit problems.

For application recovery/replay, we find that detection latency is within the hardware recovery window for 86% of the cases. The higher latency cases need to be handled using software checkpointing, with an application specific trade-off between buffering persistent outputs/inputs (for ms) and full application recovery.

Other future work. Besides exploring the system implications mentioned above, we plan to refine the fault models used here, including studying intermittents and validating our insights with lower level simulators. We also plan to explore more OS intensive workloads, e.g., transaction processing and web servers.

Acknowledgments

We would like to thank Pradip Bose from IBM and Subhasish Mitra from Stanford University for many discussions on this work and insightful comments on previous versions of this paper. We also thank Ulya Karpuzcu for help with our simulation infrastructure.

References

[1] J. Arlat et al. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. *IEEE Computer*, 42(8), 1993.

[2] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *International Symposium on Microarchitecture (MICRO)*, 1998.

[3] David Bernick et al. NonStop Advanced Architecture. In *International Conference on Dependable Systems and Networks (DSN)*, 2005.

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

[5] Shekhar Borkar. Microarchitecture and Design Challenges for Gigascale Integration. In *International Symposium on Microarchitecture (MICRO)*, 2005. Keynote Address.

[6] Fred Bower et al. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In *International Symposium on Microarchitecture (MICRO)*, 2005.

[7] Fred A. Bower et al. Tolerating Hard Faults in Microprocessor Array Structures. In *International Conference on Dependable Systems and Networks (DSN)*, 2004.

[8] Kypros Constantinides et al. Software-Based On-Line Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation. In *International Symposium on Microarchitecture (MICRO)*, 2007.

[9] Edward W. Czeck and Daniel P. Siewiorek. Effects of Transient Gate-Level Faults on Program Behavior. In *International Symposium on Fault-Tolerant Computing (FTCS)*, 1990.

[10] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Symposium on Operating Systems Design and Implmentation (OSDI)*, 2002.

[11] Michael D. Ernst et al. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 2007.

[12] O. Goloubeva et al. Soft-Error Detection Using Control Flow Assertions. In *Proc. of 18th IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 2003.

[13] Mohamed Gomaa et al. Transient-Fault Recovery for Chip Multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, 2003.

[14] Weining Gu et al. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. In *International Conference on Dependable Systems and Networks (DSN)*, 2004.

[15] Sudheendra Hangal and Monica S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *International Conference on Software Engineering (ICSE)*, May 2002.

[16] Mei-Chen Hsueh et al. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4), 1997.

[17] G. Kanawati et al. FERRARI: A Flexible Software-based Fault and Error Injection System. *IEEE Computer*, 44(2), 1995.

[18] Hue-Sung Kim, Arun K. Somani, and Akhilesh Tyagi. A Reconfigurable Multi-function Computing Cache Architecture. In *International Symposium on Field Programmable Gate Arrays*, 2000.

[19] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Int'l Symposium on Code Generation and Optimization (CGO)*, 2004.

[20] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors. In *International Conference on Dependable Systems and Networks (DSN)*, June 2005.

[21] Ben Liblit, Mayur Naik, Alice Zheng, Alex Aiken, and Micael Jordan. Scalable Statistical Bug Isolation. In *Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[22] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[23] Milo Martin et al. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4), 2005.

[24] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-System Timing-First Simulation. *SIGMETRICS Performance Evaluation Rev.*, 30(1), 2002.

[25] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *International Symposium on Microarchitecture (MICRO)*, 2007.

[26] Albert Meixner and Daniel Sorin. Error Detection Using Dynamic Dataflow Verification. In *Parallel Architecture and Compilation Techniques (PACT)*, 2007.

[27] M. Mueller et al. RAS Strategy for IBM S/390 G5 and G6. *IBM Journal on Research and Development*, 43(5/6), Sept/Nov 1999.

[28] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *International Symposium on Microarchitecture (MICRO)*, 2003.

[29] Jun Nakano et al. ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2006.

[30] Nithin Nakka et al. An Architectural Framework for Detecting Process Hangs/Crashes. In *European Dependable Computing Conference (EDCC)*, 2005.

[31] Karthik Pattabiraman et al. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In *European Dependable Computing Conference*, 2006.

[32] Milos Prvulovic et al. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, 2002.

[33] Paul Racunas et al. Perturbation-based Fault Screening. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[34] V. Reddy et al. Assertion-Based Microarchitecture Design for Improved Fault Tolerance. In *International Conference on Computer Design* , 2006.

[35] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *International Symposium on Computer Architecture (ISCA)*, 2000.

[36] George A. Reis et al. Software-Controlled Fault Tolerance. *ACM Transactions on Architectural Code Optimization*, 2(4), 2005.

[37] R. Rodriguez et al. Modeling and Experimental Verification of the Effect of Gate Oxide Breakdown on CMOS Inverters. In *International Reliability Physics Symposium (IRPS)*, 2003.

[38] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *International Symposium on Fault-Tolerant Computing (FTCS)*, 1999.

[39] Giacinto P. Saggese et al. An Experimental Study of Soft Errors in Microprocessors. *IEEE Micro*, 25(6), 2005.

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

[41] Smitha Shyam et al. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[42] Daniel Sorin et al. Fast Checkpoint/Recovery to Support Kilo-Instruction Speculation and Hardware Fault Tolerance. Technical Report 1420, Computer Sciences Department, University of Wisconsin, Madison, 2000.

[43] Daniel Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *International Symposium on Computer Architecture (ISCA)*, 2002.

[44] Jayanth Srinivasan et al. The Impact of Scaling on Processor Lifetime Reliability. In *International Conference on Dependable Systems and Networks (DSN)*, 2004.

[45] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX Annual Technical Conference, General Track*, pages 29–44, 2004.

[46] Rajesh Venkatasubramanian et al. Low-Cost On-Line Fault Detection Using Control Flow Assertions. In *International On-Line Test Symposium*, 2003.

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

[48] Nicholas Wang et al. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *International Conference on Dependable Systems and Networks (DSN)*, 2004.

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

[50] David Yen. Chip Multithreading Processors Enable Reliable High Throughput Computing. In *International Reliability Physics Symposium (IRPS)*, 2005. Keynote Address.

[51] Pin Zhou, Wei Liu, Fei Long, Shan Lu, Feng Qin, Yuanyuan Zhou, Sam Midkiff, and Josep Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants. In *International Symposium on Microarchitecture (MICRO)*, 2004.

[52] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Simple, General Architectural Support for Software Debugging. IEEE Micro Special Issue: Micro's Top Picks from Computer Architecture Conferences, 2004.

[53] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2007.