Analyzing and Reducing Silent Data Corruptions using Software-Level Techniques

Thesis Proposal

Siva Kumar Sastry Hari
Department of Computer Science
University of Illinois at Urbana-Champaign
shari2@illinois.edu

Contents

Abstract

Hardware reliability is becoming a major challenge with technology scaling and increasing in-field device failure rates. Commodity systems must be made resilient to such device failures, motivating very low-cost resiliency solutions. Software-level symptom detection techniques have emerged as low-cost and effective solutions with low Silent Data Corruption (SDC) rate. However, eliminating or significantly lowering the user-visible SDC rate is crucial for these solutions to become practically successful.

The goal of this thesis, therefore, is to provide programmers and system designers with tools and techniques to evaluate resiliency solutions and tailor them to achieve significantly higher reliability for a given performance (and/or power) budget with low-effort. So far, my work addresses the challenges in identifying all application locations that are susceptible to produce SDCs when subjected to soft-errors. It then focuses on converting these undesirable silent corruptions to visible detections through program-level cost-effective error detectors.

To identify a comprehensive list of SDC-causing program locations, this work presents an approach called Relyzer that systematically analyzes all application fault sites. Instead of performing fault injections on all possible application level fault sites, which is impractical, Relyzer carefully picks a small subset. It employs novel fault pruning techniques that reduce the number of faults sites by either predicting their outcomes or showing them equivalent to others. Our results show that over 99.78% of faults are pruned across twelve studied workloads. Validation experiments also show that the selected faults represent the rest of the faults with an average of 96% accuracy for the studied fault models and applications.

Analyzing the comprehensive list of SDC-causing instructions, obtained by Relyzer, we identified a few program properties that are responsible for most SDCs. Exploiting this analysis we developed low-cost program-level error detectors for these application sections. These detectors are effective in significantly reducing the reliance on instruction-level redundancy for full SDC coverage. Overall, our detectors provide practical and flexible choice points on the performance vs. reliability trade-off curves. For example, for an average of 90%, 99%, or 100% reduction of the baseline SDC rate, the average execution overheads of our approach versus redundancy alone are respectively 12% vs. 30%, 19% vs. 43%, and 27% vs. 51%.

Future directions include developing simple program-level resiliency metrics that can identify SDC-causing instructions with low profiling effort and developing a fast and systematic fault simulation framework that can significantly lower Relyzer's complete resiliency evaluation time.

1 Introduction

As process technology scales, the increasingly smaller devices become susceptible to a variety of in-field hardware failure sources; e.g., high-energy particle strikes (or soft-errors), voltage droops, wear-out, and design bugs [5, 10]. This increases the likelihood of a hardware failure in the field. Soft-errors in logic, among these failure sources, are expected to increase at a high rate with scaling (Section 3.1 in [2]). Therefore, future systems must deploy low-cost in-field resiliency solutions to guarantee continuous error-free operation.

Hardware fault detection mechanisms currently form a crucial part in devising such reliability solutions. Traditional solutions use heavy amounts of redundancy (in space or time) to detect hardware faults. Owing to their prohibitive costs, such detection mechanisms are increasingly unacceptable for modern commodity systems. Instead, there is a growing recognition that a wide spectrum of the commodity space will accept only much lower cost solutions (in area, power, and performance), perhaps at the cost of tolerating very occasional failures.

Recently, there has been a surge of research in software-level symptom based fault detection techniques [14, 16, 24, 27, 29, 37, 40, 53] that provide promising such low-cost alternatives. These techniques detect only those hardware faults that corrupt software execution by monitoring for anomalous software behavior using simple, low-cost monitors. Despite the simplicity of their detectors, these techniques have demonstrated impressively high detection rates. Unfortunately, some fraction of faults do escape the detection mechanism and silently impact the correctness of the program output. Such faults are called silent data corruptions or SDCs. As an example, SWAT (SoftWare Anomaly Treatment) [20, 24, 47], a state-of-the-art reliability solution, reports an SDC rate of <0.5% across several (compute-intensive, media, and distributed client-server) workloads for both permanent and transient faults in all microarchitectural units studied except the data-centric FPU. Since faults resulting in SDCs produce corrupted application output without leaving any trace of failure behind, such a small SDC rate is still a hindrance for this approach to become a practically successful one. Hence it is crucial to significantly lower the SDC rate in a cost-effective manner and provide a mechanism to tune for user-visible SDC rate vs. performance. .

Whether a hardware fault will produce an SDC is highly dependent on the application; therefore, it is likely that the most cost-effective mechanism to reduce SDCs will be application-specific. Moreover, with the ever-increasing constraints on power, performance, and area, application-specific customization of resiliency solutions is desirable. Such a customization would require a detailed resiliency profile of applications to identify potentially SDC-causing

program locations. This profile can be obtained through a comprehensive resiliency analysis that performs rigorous injection campaign on all faults that can possibly affect program execution. For most applications, this translates to trillions of (time-consuming) injection experiments which is clearly infeasible. Hence there is a need for a practical resiliency analysis technique that can identify *all* vulnerable (SDC-causing) application locations.

Such an analysis would allow us to focus on potentially SDC-producing application sites and develop targeted cost-effective mechanisms to convert SDCs to detections. This requires developing and placing application-level (and in some cases application-specific) error detectors that incur low overheads. This guided and customized approach can also provide programmers and system designers the ability to effectively tune for resiliency vs. performance overhead, allowing them to target any SDC rate or overhead.

1.1 Application Resiliency Analysis

A comprehensive evaluation of an application's resiliency, with a detection mechanism in-place, requires studying the impact of all faults of interest injected at each cycle of the application's execution (one at a time). Here, when we refer to a *fault*, we include both the hardware site *where* a fault is injected as well as *when* it is injected in an application's execution (the application site). After injecting a fault (at a given cycle), the application must be allowed to run potentially to completion to determine if the fault is masked, results in a SDC, or is detected. For application benchmarks with billions of instructions, this translates (conservatively) into trillions of fault injection runs for most benchmark suites and fault models of interest. Such a comprehensive fault injection campaign is clearly infeasible.

Most fault-injection based evaluation techniques [24, 27, 29] bound the experiment time by studying a randomly selected sample of faults (typically of the order of thousands of faults per application) out of all the faults possible (more than a trillion faults for this study). While these methods may provide statistical guarantees on SDC rates, certain faults that may be important to the application may be sampled out. Equally important, such statistical sampling provides virtually no feedback on which parts of the application remain vulnerable (other than the few instructions where faults were injected) and might need protection in other ways to reduce the SDC rate. Therefore, it is important to better identify the remaining vulnerable portions of the program and to reduce the SDC rate with cost-effective detectors.

We therefore developed *Relyzer*, a resiliency analyzer that systematically analyzes all (dynamic) application fault-sites to determine a minimal set for when transient faults need to be injected.[1] Figure 1 briefly summarizes how Relyzer works. It first lists all application sites that can be directly affected by our chosen transient faults.[2] For some of these faults, Relyzer can directly predict their outcome (detection, masking, or SDC) through simple static analysis and dynamic profiling of the fault-free execution. These faults do not need detailed fault injection experiments. For the remaining faults, Relyzer primarily uses the insight that faults propagating "similarly" through the program are likely to result in similar outcomes. We propose novel heuristics based on static and dynamic control flow and data flow no capture the notion of "similar" for faults in different types of instructions. Using these heuristics, we categorize application fault-sites into equivalence classes. We then select a representative from each equivalence class and thoroughly study it through a detailed fault injection experiment.

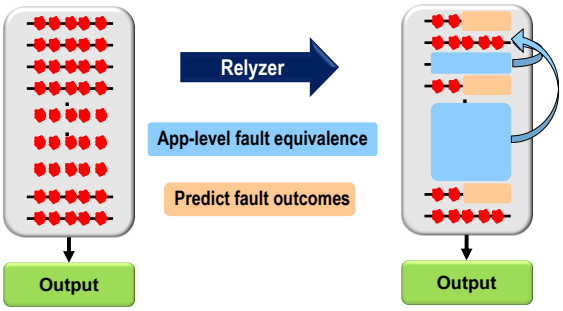


Figure 1: Relyzer Overview: Relyzer analyzes all application-level fault-sites (shown as small red-colored clouds on the left) and prunes a significant fraction of them through fault pruning techniques. Only the few remaining faults-sties (shown on the right) require fault injection experiments for complete application resiliency analysis

[1] We only consider single-threaded applications in this work and leave the exploration of multithreaded applications to future work.

[2] Since our focus is not on the hardware sites, we choose two examples: transient single-bit flip faults in (architectural) integer registers and in output latches of address generation units.

Our results show that Relyzer significantly reduces the number of faults that require fault injection experiments. Relyzer pruned about 99.78% of the total faults in the twelve applications studied here (three to six orders of magnitude reduction for all but one application). We validated the pruning techniques that use heuristics by matching the fault injection outcomes of the representative faults against outcomes of a sample of faults they each represent. Each pruning technique and fault model combination individually gave an accuracy of >92%, averaged across all studied applications. Overall, with the combination of all pruning techniques, Relyzer was able to correctly determine the outcomes of 96% of all faults, averaged across all twelve applications studied.

1.2 Reducing Silent Data Corruptions

We identified virtually all SDC-causing fault-sites by performing fault injections in the Relyzer-identified sites. Investigating these SDC-causing fault sites revealed that only a small fraction of static instructions cause most SDCs (virtually all the SDCs were caused by approximately 20% of static instructions for the studied applications). Prior work has made similar observations and applied selective instruction-level duplication to increase the detection rate [15, 46]. However, we observed that the small fraction of SDC-causing static instructions consume a much higher fraction of the execution time (in number of dynamic instructions). Our results indicate that protecting all SDC-causing instructions through instruction-level duplication may incur 50% overhead on average, assuming a conservative one cycle overhead per covered instruction. This high overhead is consistent with that reported for previous selective instruction-based redundancy techniques, and motivates selective detection techniques that are much more cost effective than instruction-level redundancy.

We therefore focus on developing an error detection scheme that converts SDCs to detections with low overhead. But first, we need to answer the following two key questions: *where* to place the error detectors and *what* detectors to use. We address the first question by placing the detectors in program locations where errors from many SDC-causing instructions propagate into few quantities (or variables). For detectors in these locations, we exploit program-level properties that hold true for these (potentially) error carrying quantities. In our work, we place our detectors at the end of loops and function calls that contain SDC-causing instructions, and detect errors by testing program-level properties on variables that are live at these points (e.g., comparing the outcomes of similar computations and known value equalities). We also achieve the ability to obtain a set of near-optimal detectors for any SDC coverage target,[3] which allows us obtain continuous SDC coverage vs. performance trade-off curves. Results show that our program-level detectors significantly lower the average overhead needed by selective instruction-level redundancy at all SDC coverage targets.

1.3 Contributions

- **Relyzer: Application resiliency analyzer [19]:** Relyzer systematically analyzes all application fault-sites to obtain a detailed reliability profile of applications. It employs novel fault pruning techniques that reduce the fault-injection experiments needed by either predicting their outcomes or showing them equivalent to others. This smart selective fault evaluation technique prunes 99.78% of fault-sites across twelve studied workloads. We further show that only 0.004% represent 99% all fault-sites (providing a coverage of 99%) and performing fault injections in these sites is practically viable.[4] This allows identifying all the application sites that are vulnerable to SDCs (the equivalence classes whose representatives result in SDCs). To our knowledge, Relyzer is the first work to develop such a notion of fault equivalence for application fault-sites (analogous to that of hardware fault equivalence). Validation experiments show that the selected faults represent the rest of the faults with an average accuracy of 96% for the studied fault models and applications.

- **Understanding program properties that lead to SDCs [18]:** We analyzed program instructions that cause >90% of the SDCs and identified a few program properties that appear repeatedly within the same application and even across different applications.

- **Developing low-cost program-level detectors [18]:** Using the above analysis, we developed low-cost detectors that are placed at the end of loops and function calls. These detectors invoke program-level property checks on a few variables that potentially carry errors from a large number of SDC causing

[3] SDC coverage is defined as the fraction of SDCs converted to detections.

[4] With our existing simulation speeds, faults in every 8^{th} bit of a given dynamic instruction can be simulated in approximately 21 days on a cluster of 200 cores.

instructions. We find that our detectors provide an average SDC coverage of 84% with an average execution overhead of 10%.

- **Effective SDC coverage vs. execution overhead trade-off curves [18]:** Using our low-cost detectors, we present continuous SDC coverage vs. execution overhead trade-off curves for our applications. These curves fall back to instruction-level redundancy for the sites that our program-level detectors cannot cover. Compared to similar curves with instruction-level redundancy alone, our approach yields much better execution overheads for all SDC coverage targets of interest on average; e.g., 12% vs. 30% overhead for 90% SDC coverage and 19% vs. 43% for 99% coverage. The ability to quantify such curves, achieved for the first time, enables programmers and system designers to effectively tune for resiliency vs. overhead, allowing them to target any SDC coverage with the lowest cost combination of our detectors.

1.4 Limitations and Future Work

- Relyzer evaluation time: Relyzer, for the first time, systematically analyzes all application-level fault sites for a thorough resiliency profile. The evaluation time needed by Relyzer is practical (approximately 48 hours to profile all applications + 15 hours for fault injection campaign per application), but it may still be high for some situations. The current speed of our fault simulation framework is one of the main factors that increases the evaluation time needed to identify SDC-causing instruction by Relyzer, especially when we include several applications to our study.

  Moreover, the requirement of detailed and input-specific dynamic profiles of the applications may also hinder Relyzer's practicality by increasing its evaluation time.

- Limited applications and fault models: Including more applications to our study would allow us to develop a generic set of detectors (both lossless and lossy with their coverage statistics) for different class of applications. These generic set of detectors (with a wide variety of application-specific detectors) would allow the software designers/architects to easily tune for resiliency for any given performance budget.

  Our work, so far, focuses on single-bit-flips (or soft-errors) in instruction-level architectural registers. Extending Relyzer to microarchitecture- or gate-level fault models (which may show up as multi-bit or multi-value errors at architecture-level) and evaluating our detectors with them would be interesting.

- Recovery and detection latency: We do not evaluate the detection latency of our detectors. The tolerable latency of error detection is dictated by the recovery window a checkpointing-and-rollback solution can support at low-cost. Hence, evaluating the detection latency offered by our program-level detectors and tailoring them to detect errors within tolerable recovery window is important future work.

In future I plan to address the first limitation – lowering the application resiliency evaluation time – with either of the two following approaches: (1) develop simple program-level resiliency metrics that can identify SDC-causing instructions with low profiling effort or (2) develop a fast systematic fault simulation framework that can significantly lower Relyzer's complete resiliency evaluation time. If time permits, I will also address the remaining limitations.

1.5 Other Contributions

In addition to this work, I led the mSWAT [20] project that focused on detecting and diagnosing hardware faults in multicore systems while running multithreaded workloads. I was involved in a fault recovery project that addressed an important, but commonly ignored, aspect of recovery - the output commit problem. I was also a part of the team that developed a novel fault injection infrastructure called SWAT-Sim [23] that uses hierarchical simulation to study the system-level manifestations of gate-level faults. Recently, I also contributed to the CrashTest'ing SWAT project which was done in collaboration with University of Michigan [39]. This project validated the effectiveness of SWAT detectors on an FPGA implementation of an industry strength processor running an unmodified commercial operating system and modern applications.

In the following section we outline the related work. Next, in Section 3, we explain Relyzer and how it prunes application fault sites. It also evaluates it effectiveness. In Section 4 we explain the low-cost program-level error detectors we developed to reduce SDCs. An evaluation of these detectors is also presented here.[5] In Section 5, we explain the future directions.

[5]Most of the text in Sections 3 and 4 is borrowed from [19] and [18] respectively.

2 Related Work

The traditional approach for reliability is to use coarse-grained system-level redundancy (e.g., replicating an entire processor or a major portion of the pipeline) [8, 31]. There has also been substantial microarchitecture-level work that exploits redundancy at a finer microarchitectural granularity [6, 11, 13, 17, 33, 42, 43, 48, 44]. These techniques incur significant overhead in area, performance, power, and/or wear-out that is paid almost all the time. In contrast to the above, we seek a reliability solution that incurs minimal overhead in the common case where there are no errors, and potentially higher cost in the uncommon case when an error is detected.

Symptom-based fault detection techniques [14, 16, 37, 53, 40, 24, 47, 20] provide one such low-cost alternative. These mechanisms treat anomalous software behavior as symptoms of hardware faults and detect them by placing very low-cost symptom monitors in hardware or software. Researchers have shown that this approach is effective in detecting both permanent and transient hardware faults with only a small fraction resulting in SDCs through statistical fault injections on microarchitecture-level models [24, 20]. Such techniques form the baseline for our work.

2.1 Resiliency Evaluation

There has been much work in evaluating resiliency solutions [29, 24, 23, 38, 39, 34]. Much of this work is evaluated using statistical fault injection campaigns on architecture-, microarchitecture-, or gate-level simulators or FPGA emulators running various benchmark applications. The hardware and software locations are typically randomly selected to achieve some statistical confidence. This approach does not provide any insight on the parts of the application that remain vulnerable to SDCs (other than for the relatively few application sites where faults were actually injected and simulated).

SymPLFIED [36] and Shoestring [15] share our high-level goal of finding faults that escape detection and lead to SDCs. SymPLFIED uses a powerful symbolic execution method to abstract the state of erroneous values in the program. It injects such a symbolic error at all possible application sites (one at a time) and uses model checking with the abstract execution technique to explore all possible paths with the symbolic error and determines the outcomes of such paths (masking, detection, or SDC). The focus of SymPLFIED is to reduce the number of fault values per application site that need to be injected (hence the symbolic fault). Our focus, so far, has been on reducing the number of application sites where the fault is injected (we restrict the values by simply restricting our hardware fault models since that is not the focus of our work). Comibining SymPLFIED with Relyzer is an interesting future direction. However, it is unclear whether the model checking techniques used in SymPLFIED can scale to large applications; so far, it has been applied to only a few small benchmarks (e.g., a Siemens benchmark).

Shoestring provides a pure static analysis that identifies static instructions where faults are likely to be detected quickly enough; e.g., there is a short-enough path in the data-flow graph from such a fault to enough potentially symptom generating instructions. The rest of the faults are considered vulnerable and the important ones among these (currently, stores) are protected by duplicating any instructions that produce data that feeds into them. Shoestring succeeds in its goal of reducing the SDC rate by about 34% to 1.6% at 15.8% performance cost. One shortcoming of Shoestring is that it only employs static analysis to identify vulnerable instructions. Our approach (Relyzer), on the contrary, exploits information known only at execution time (e.g., store and load addresses) and applies a set of dynamic analyses that can distinguish between different instances of a static instruction. Our technique bins application fault locations into equivalence classes and then performs accurate fault simulation of the representative fault to identify the outcome. This allows Relyzer to account for masking of a fault, quantify a program's SDC rate, and enumerate the dynamic conditions that make code sections SDC prone. Shoestring's static analysis cannot achieve any of these.

Benso et al. [7] proposed a solution that performs runtime analysis of the application variables to obtain the criticality behavior of every variable. That work developed an analytical model to obtain this criticality behavior considering three variables – lifetime of the application variable, number of reads to it, and whether it is a pointer or not. The work proposed that the contribution of each of these variables is application independent and once the parameters of the model are set they remain fixed for all other applications. The results show that this solution observes low inaccuracies in predicting the criticality of variables. However, the results were shown on small applications with few variables. Relyzer, on the other hand, captures the fault propagation behavior from the fault-free execution of the application and uses it to categorize faults into different equivalence classes. It, however, relies on fault injection experiments on the pilots to estimate the outcome of the population (as opposed to estimating the outcomes based on static application-independent parameters).

Sridharan et al. [50] quantify the reliability behavior of an application using a metric called Program Vulnerability Factor (PVF). PVF is a microarchitecture-independent method to quantify architectural fault masking inherent to a program. PVF focuses on identifying only those faults that are masked by the application. It does not attempt to distinguish faults that lead to SDCs from the ones that result in detection, but this distinction becomes crucial with symptom-based detection technique in place. Hence, our work focuses on distinguishing SDCs from detections. It is unclear whether PVF can make this distinction.

2.2 Detectors to Reduce SDCs

SWIFT [44] is a fully compiler-based software solution that inserts redundant code to compute duplicate versions of all register values, and validation code for checking the two versions. SWIFT more than doubles the number of dynamic instructions, relying on underutilized hardware resources for performance. CRAFT [45] later improved the performance of SWIFT through hardware support. PROFiT [46] improved upon both SWIFT and CRAFT by adding techniques to manage the desired levels of performance and reliability. It uses the programs performance and reliability profile (obtained by statistical fault injections) to identify the code sections that need duplication to meet the given performance and reliability constraints. Due to the lack of fine grained knowledge of the applications reliability profile, it considered duplication only at the function granularity. Moreover, their expensive (in time) evaluation strategy limited them to apply selective redundancy on just a few applications. Shoestring [15], however, obtained a list of high-value (or SDC-prone) instructions through a static analysis and applied SWIFT-like selective instruction-level redundancy to protect these high-value instructions. In our work, we obtain a detailed reliability profile through Relyzer and use selective redundancy only on the SDC causing instructions as our baseline.

Software-level invariant based fault detection has been applied for fault detection [16, 47, 37]. In particular, range-based likely program invariants (inserted at all stores) have been employed for reducing SDCs produced by hard faults [47]. Results show a reduction in SDCs of up to 74% for a microarchitecture-level permanent fault model, but with an execution overhead of 14% on SPARC machines. Moreover, this technique suffers from false positives which can further increase the overheads. For a transient fault model, our technique provides a better SDC coverage vsperformance trade-off through a more selective placement of a broader range of detectors. Combining insights from these two studies for both fault models is part of our future work.

Pattabiraman et al. [35], developed metrics, namely *fanout* and *lifetime*, to identify what application variable to protect and where to place detectors. The goal, however, was to prevent or limit fault propagation and avoid system crashes with minimum possible detector locations, not particularly to reduce SDCs. Subsequently, they also proposed a technique to automatically derive application-specific detectors to be placed at these locations [37]. This technique tries to dynamically associate a property check for the identified variable from a set of pre-defined checks. The properties they used are similar in some respects to a few of our observations. However, they differ significantly because detectors in [37] do not considered complex properties spanning across multiple variables like the loop based detectors presented in this paper. Moreover, detectors in [37] produce false positives, whereas our detectors never fire in fault-free executions. In future, however, we plan to utilize *fanout* and *lifetime* metrics and combine them with other information like *is a pointer or not* and *length of error propagation chain* to identify SDC-causing instructions with little effort.

2.3 Other Related Work

Prior work explored the inherent error tolerant behavior of applications [25, 41]. They showed that a large class of applications depict inherent error tolerance – multimedia, artificial intelligence, compute-intensive SPEC, and I/O-intensive server workloads.

Software-level symptom-based detection mechanisms (like SWAT) rely on full-system checkpointing schemes for recovery. SafetyNet [49] and ReVive [32, 51] are the two state-of-the-art checkpointing solutions.

3 Relyzer: Application Resiliency Analyzer

3.1 Fault Pruning Techniques

Relyzer systematically analyzes all application fault sites and carefully selects a small subset for thorough fault injection experiments such that it can still estimate the outcomes of all the faults in the application. To achieve this goal, Relyzer applies a set of pruning techniques that are classified as *known-outcome* and *equivalence-based*

pruning techniques. The known-outcome techniques largely use static (and some dynamic) program analyses to predict the outcome of a fault. The equivalence-based techniques prune faults by showing them equivalent to others using static and dynamic analyses and/or heurstics.

Relyzer first enumerates all the faults that can impact the application. We require a fault-free execution trace for a given application (and input) for this step. Each dynamic instruction instance in the trace forms a potential application fault site. At this site, we consider injecting hardware faults that would be exercised by this instruction (one fault at a time). Since our focus is on the choice of application fault sites and not on exhaustively studying all hardware faults, we choose to study transient faults (single bit flips) in architectural integer registers and in output latches of address generation units. As an example, consider an add instruction with register operands g1, g2, and l1 as an instruction appearing in the dynamic instruction trace. We consider injecting single-bit-flips in the integer registers g1, g2, and l1 that are accessed by this instruction (one at a time, in different bits). Faults in the address generation unit will be considered only when it is exercised; i.e., in load and store instructions.

While enumerating the list of all application fault sites, Relyzer stores all the fault related information in a data structure called the fault database. Relyzer next applies the fault pruning techniques on this initial set of faults. While the fault pruning is being performed, all the required information for successful computation of overall SDC rate and the SDC rate for each static instruction is logged in the fault database. Details of the fault database can be found in [19].

3.1.1 Known-outcome pruning techniques

Bounding addresses: Transient hardware faults can make applications access memory locations that fall out of the range of the allocated address space. Such accesses are likely to result in detectable symptoms (e.g., fatal traps, segmentation faults, application aborts, and kernel panic). SWAT employs detectors specifically to detect such scenarios within recoverable latencies (e.g., out-of-bounds detectors [41]). We do not need injection experiments to identify the outcome of most such faults and can directly prune them as follows.

We determine the range of valid addresses, for both the stack and the heap, by studying the dynamic memory profile of the application. To keep our implementation simple, we monitor global and heap addresses together. This also eliminates the problem of distinguishing them from each other while profiling. This approximation only makes our technique conservative if we assume that the out-of-bounds detector can also detect faults in addresses that make accesses cross the global-heap boundary.

Once we identify the range of the valid addresses, we prune faults that allow a memory instruction to access an invalid address (e.g., faults in high order bits of the address when the fault-free trace shows valid addresses are within lower order bits). This technique is applicable to memory instructions (both loads and stores).

Bounding branch targets: Analogous to the bounding addresses case, a fault that causes a control instruction to jump to a location that is not in the application instruction space is likely to result in a detectable symptom (e.g., SWAT's fatal trap and app-abort detectors or an out-of-bounds detector analogous to that for data addresses can detect such faults).

The address range that contains all possible targets can be obtained by noting the start and the end of the text section of an application. Typically, the text section is small (for applications with under million instructions; i.e., under 32 bits) and hence a large fraction (over 50% on 64-bit machines) of faults in branch targets can be predicted as detected and pruned by this technique. This bounding technique may not be directly applicable to jumps to shared libraries because the registers used by these operations may already contain addresses that are out of the text section.

We performed an optimistic experiment on the studied fault models and applications and found that it only provides a pruning of approximately 0.5%. Hence, we do not include it in our study here. However, this technique may be effective for other faults models, e.g., faults in immediate operands. Many branch instructions specify PC-relative displacements as immediate operands, and could benefit from this technique.

Constant-based: In some logical operations, only a fraction of the bit locations in the source operands are used to produce the destination register value. Several of these bit locations in the source operand are usually discarded and hence faults in these bits get masked. Currently we apply this technique only on logical shift operations, where the shift count is a constant. We prune faults in the bit locations of the source register (non-constant operand) that are not be used to produce the destination register value and treat them as masked.

This technique provides a modest benefit for unoptimized applications. We do not not report it for the optimized applications because preliminary experiments showed that it provided insignificant benefit for them.

Several other instruction specific pruning techniques (similar to the one above) have a potential of providing added pruning. This is, however, a part of our future work.

3.1.2 Equivalence-based pruning techniques

This class of pruning techniques eliminates faults that are equivalent to each other from the initial set of fault sites and retains only the representative faults (pilots) for thorough fault injection experiments. We further categorize pruning techniques in this section as *precise* and *heuristic-based* based on whether they use accurate analyses or heuristics to form the equivalence classes.

3.1.2.1 Precise techniques

Def-use analysis: A register definition is created whenever a register is used as a destination operand in an instruction. Faults in the definition of a register have similar behavior to that of faults in the first use of this definition. Therefore, we prune out faults in the definition and retain faults in the first use. Note that this technique prunes faults only in the definition and not in the uses. There can be multiple uses of a definition, and faults in different uses may have different fault propagation. Whenever a definition is pruned, we record the information of the first use at the definition such that the outcomes of the faults in the first use can be related to that of definition's at a later stage (details of the recorded information is presented below).

Ideally, the destination register operands of all the instructions should be pruned by this technique. In our experiments, however, we prune faults in only those destination registers that have a first use within the same basic-block. Since we implemented this technique as a static program pass, accounting for this equalization in the fault database (i.e., associating faults in of the first use to that of the definition's) was non-trivial for the cases where the def to first-use chains spanned across multiple basic blocks. Moreover, in the presence of conditional move operations it was unclear whether a static pass can still prune faults without compromising on the precise association of a definition with the first-use. Hence, we limited ourselves to a conservative but precise implementation.

Constant-based: We applied the principal of constant propagation to prune faults from the operands of instructions that use constants. For such instructions, the effect of a fault in the source register (non-constant) operand can be studied directly in the destination operand; therefore, we can prune faults from such source operands. This pruning technique is currently limited to only those logical operations where a single-bit fault in the source operand propagates as a single-bit fault in the destination (e.g., logical `xor`). This technique provides negligible benefits for optimized applications. Hence we report it only for the unoptimized applications.

3.1.2.2 Heuristic-based techniques

Control-equivalence: This heuristic pruning technique uses the observation that faults propagating through similar code sequences are likely to behave similarly. It also uses the observation that a majority of the faults appear in code sequences that are executed many times. Consider a static instruction I with many dynamic instances in the fault-free execution under consideration. The pruning technique attempts to partition all these dynamic instances of I into equivalence classes, based on the control flow path followed after the dynamic instance.

It is convenient to describe and implement the algorithm at the basic block level. The technique uses the fault-free application execution to enumerate all possible control flow paths up to a depth n starting at the basic block that contains the instruction of interest. Depth is defined as the number of branch or jump instructions encountered. For the paths that were exercised multiple times in the execution, it randomly selects one dynamic occurrence, a pilot. It prunes all other unselected executions of such paths (population) and assumes that faults in those dynamic executions are represented by the selected ones (pilots). More precisely, a dynamic instruction instance on a pilot path serves as a pilot for other instances with the same PC on the other paths in its population.

Figure 2 explains through an example how this pruning technique selects pilots. The figure presents a control flow graph of a small program, with the basic blocks represented by the black and grey circles with numbers on their sides. Assume the grey basic block is not exercised by the dynamic execution of interest. Assume $n = 5$ (depth until which control flow is tracked). Suppose we are interested in finding the representative pilots for an instruction in basic block 1. We enumerate all control flow paths starting at basic block 1 up to a depth of 5 that are executed in the dynamic fault-free execution of interest. Basic block 4 is never executed and hence it does not appear in the list of dynamically exercised paths. We identify each path as forming a new equivalence class. There will be potentially many instances of such paths in the dynamic execution trace. We randomly select

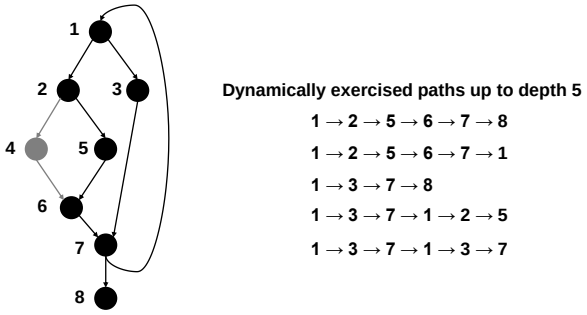


Figure 2: Control-equivalence. The figure shows a CFG for a small program starting at basic block 1 and ending at basic block 8. We enumerate all dynamically exercised control paths up to a depth, say 5. Here basic block 4 (showed in grey) never gets exercised. Therefore control flow paths through this node do not appear on the list of dynamically exercised paths. The executions along each of these paths form the equivalence classes for similar fault outcomes.

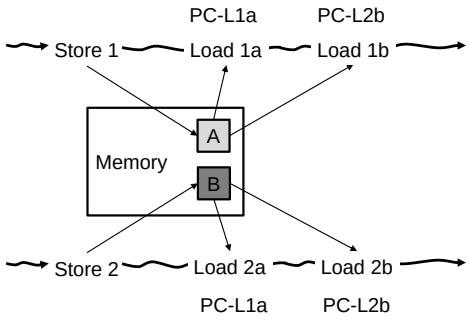


Figure 3: Store-equivalence. Store 1 and Store 2 are two store instructions from the same static instruction writing to addresses A and B respectively. Load 1a with program counter PC-L1a and Load 1b with program counter PC-L1b are two load instructions reading the value from address A. Similarly, Load 2a and Load 2b are two loads from address B with program counters PC-L2a and PC-L2b respectively. The store-equivalence heuristic requires that PC-L1a equal PC-L2a and PC-L1b equal PC-L2b.

one dynamic execution sequence for each equivalence class and name it as the pilot for that class. As mentioned before, a dynamic instruction instance on a pilot path serves as a pilot for other instances with the same PC on the other paths in its population.

We apply this technique to prune faults in all instructions other than stores and those that affect stores within a basic block. In other words, we do not apply this technique on stores and the instructions that stores depend on. This is because the propagation of a fault in a store also depends on the addresses of the loads in the control flow path taken (only loads to the same address as the store will propagate the fault). The next technique described deals with this distinction. An exception to the above are a few SPARC specific instructions; namely, *save*, *restore*, *call*, *return*, and *read state register*. In this study, we do not inject faults in these instructions and therefore do not consider them any further. We also do not inject faults in the dead instructions and do not consider those any further either. Overall, this technique has the potential of pruning a large fraction of the faults by softening the constraint on evaluating all dynamic occurrences from a specific code section.

Store-equivalence: A fault in a store instruction propagates through the loads that read the faulty values. Load addresses are not entirely captured by the control flow path taken after the store. We therefore developed an alternate heuristic, called store-equivalence, for faults in store instructions or in instructions that a store depends on. This heuristic captures the fault propagation behavior by observing the addresses that a store writes in a fault-free execution and recording all read accesses to this address. It treats the the faults in stores differently whenever a different permutations of loads instructions read the stored value.

Figure 3 illustrates our heuristic with an example. Consider Store 1 and Store 2 as two dynamic store instruction instances from the same static instruction. To determine if the faults in these two store instrucitons will have the same outcomes, we examine all the loads that return the values written by these stores in the fault-free execution, i.e., Load 1a and Load 1b for Store 1 and Load 2a and Load 2b for Store 2 from the figure. We first check whether the number of such loads is the same (two for each store in the figure). If this is the case, then we check whether the static instructions (program counters) of the corresponding loads are the same (e.g., if the program counters of Load L1a and Load L2a are the same and if those of Load L1b and Load L2b are the same in the figure). If these match, then we conclude that the two dynamic store instructions are very likely to have similar fault outcomes and we place them both in the same equivalence class.

3.1.3 Implementation

Relyzer implements the pruning techniques using static and dynamic program analyses. For example, the precise equivalence-based and known-outcome pruning techniques are implemented as a static program pass (they, how-

ever, use basic dynamic application profile). Store- and control-equivalence based pruning techniques are heavily dependent on the dynamic program information and, hence, are implemented as dynamic analyses.

As a first step, Relyzer initializes the fault database and computes the initial set of faults. The information about the number of dynamic instances of each static instruction is required to compute the size of initial set of faults and is obtained through a dynamic profile of the application. Relyzer then applies the first pruning technique – known-outcome address bounding, which is implemented as a static program pass. This technique requires the knowledge about the boundaries of stack and heap addresses accessed during the entire course of the program and this information is obtained through dynamic profiling. Relyzer next performs the def-use analysis (first equivalence based pruning technique), which is also implemented as a static program pass. (For applications with unoptimized codes, it also applies constant-based pruning technique at this step.)

Once these static techniques are applied, Relyzer prepares the application codes for the employment of the dynamic control- and store-equivalence based pruning techniques. It labels the static instructions to mark the pruning techniques that will be applicable to them. For instructions within a given basic block, all stores and the instructions that any store is dependent on are labeled to be pruned by store-equivalence based technique. Since the dynamic store-equivalence based analysis is performed only on store instructions, the identify of the store instruction is recorded with the instructions that affect his store (in other words, the instructions that any store depends on also record the identify of the store instruction.). Once the instructions for store-equivalence based pruning technique are marked, all other remaining instructions (with the earlier mentioned exceptions) are marked to be pruned by control-equivalence based technique.

Relyzer then profiles the instructions and memory in more detail to obtain dynamic control- and store-equivalence classes as discussed in section 3.1.2. As a last step it uses the store-equivalence classes (for store instructions) and associates them with the instructions that recorded the identity of these stores (as mentioned above) to obtains the respective classes for all instructions that stores depends on. At this point Relyzer computes the remaining number of pruned faults by analyzing the updated fault database and terminates.

3.2 Methodology

3.2.1 Workloads

We implemented the pruning techniques described in Section 3.1 for single-threaded applications compiled for the SPARC V9 [54] architecture, assuming the hardware fault models previously described. We selected twelve applications – four each (randomly selected) from the SPLASH-2 [55], PARSEC [9], and SPEC CPU2006 [21] benchmark suites. Table 1 provides a brief description of these applications, including the inputs used, the dynamic instruction count, number of exercised static instructions, and the number of faults prior to applying any pruning. We do not include the initialization and the output phases of the applications in our study – these phases are usually dominated by file reads and writes, memory allocation and deallocation, etc. We found that the effectiveness of the developed pruning techniques varies significantly depending on whether the applications are optimized. We focus our results primarily on the optimized versions of the applications. Section 3.3.1, however, briefly summarizes the impact of optimizations for a subset of the above applications. The dynamic instruction counts and the number of faults in table 1 pertain to the optimized version of the applications.

3.2.2 Fault injection framework

As previously mentioned, the fault models we study are single bit flips in architectural integer registers and in the output latches of the address generation units (for loads and stores). Our fault injection simulation infrastructure uses a full system simulation environment comprising of Wind River Simics [52] and the GEMS microarchitectural and memory timing simulator [28], running our applications on the OpenSolaris operating system and compiled to the SPARC V9 ISA. This framework is similar to that used in the previous work on SWAT (e.g., [24]) with some modifications.

Our framework allows us to inject faults at any point in the application execution. This is the chosen application fault site, as represented by a dynamic instruction in the fault-free execution. To inject a fault, we start the application and execute it in functional mode (Simics-only) up to 500 cycles before the chosen application fault site. Then we start detailed timing simulation (Simics+GEMS) and inject the fault when the application fault site is reached. Thus, for address generation unit faults, we flip the specified bit in the unit's output latch when it generates the address for the specified dynamic instruction. For integer register faults, we flip the specified bit in the specified register when the specified dynamic instruction reads the register (for a source) or writes the register (for a destination). The flipped bit retains its state until the latch or register is overwritten. We then

Benchmark Suite	Application	Description	Input	Num. executed instrns. after init & before finish phase		Num. Faults
				Dynamic	Static	
Parsec 2.1	Blackscholes	Calculates prices of options with Black-Scholes partial differential equation	sim-large	22.3 Million	538	1.9 Billion
	Fluidanimate	Simulates an incompressible fluid for interactive animation purposes	sim-small	611.4 Million	1,297	102.5 Billion
	Streamcluster	Solves the online clustering problem	sim-small	1.44 Billion	3,318	106 Billion
	Swaptions	Computes prices of a portfolio of swaptions using Monte Carlo simulations	sim-small	922.2 Million	1,696	97.3 Billion
SPLASH-2	FFT	1D Fast Fourier Transform	64K points	548 Million	1,483	48.7 Billion
	LU	Factors a matrix into the product of a lower & upper triangular matrix	512×512 matrix 16×16 blocks	402.8 Million	1,124	33.2 Billion
	Ocean	Simulates large-scale ocean movements based on eddy and boundary currents	258×258 ocean	358 Million	20,322	21.7 Billion
	Water	Evaluates forces and potentials that occur over time in a system of water molecules	512 molecules	504.3 Million	3,812	36.6 Billion
SPEC-Int 2006	Gcc	Based on gcc Version 3.2, generates code for Opteron	test	3.8 Billion	248,391	500.4 Billion
	Libquantum	Simulates a quantum computer running Shor's polynomial time factorization algorithm	test	235.4 Million	2,922	27.4 Billion
	Mcf	Vehicle scheduling using a network simplex algorithm	test	4.57 Billion	1,346	485.4 Billion
	Omnet++	Uses the OMNet++ discrete event simulator to model a large ethernet campus network	test	1.35 Billion	6,913	146 Billion

Table 1: Applications studied. The number of dynamic instructions, exercised static instructions, and faults pertain to the optimized versions of the applications.

simulate the application for another 500 instructions in the detailed mode before switching to the functional mode and running it to completion.

We check for all SWAT symptoms [41] (fatal traps, application aborts, and kernel panics) in the detailed mode and a reduced set of symptoms (fatal traps, kernel panics, and system error messages) in the functional simulation phase. If a symptom is detected or a timeout condition is met (the application executes more than twice its expected runtime before producing the output), then we terminate the simulation and the outcome is recorded as detected. Otherwise, the output of the application is collected and compared with the fault-free output. We record the outcome as masked or an SDC depending on whether the two outputs are or are not the same respectively.

Note that when we inject a fault, there is always an instruction that consumes a faulty value or uses a faulty address. Thus, compared to pure microarchitecture-level injection simulations, we see no microarchitectural masking and very limited architectural masking. This is by design since we wish to maximize the injections that might lead to SDCs.

3.2.3 Pruning techniques

The pruning techniques require both static and dynamic analyses of the application. The static analyses study the binary and extract several properties that are either directly applied towards fault pruning or are later used by the dynamic technique. Since our fault injection infrastructure is developed for the SPARC V9 ISA model, we restrict our study to SPARC V9 binaries. We could not find any publicly available tools to analyze SPARC binaries, so we developed our own static binary analyzer that performs basic control flow and data flow analyses.[6] Using this static infrastructure, we traverse the application and create the set of all transient fault sites. We then apply the static pruning techniques, compute the pruned fault set, and collect information for dynamic analyses. The dynamic analyzer profiles the branches, the memory access patterns (for store-equivalence technique), instruction

[6]We use the dynamic branch profile to create a correctly connected control flow graph because jump and link instructions create broken edges in the graph that may not be completed through static information alone.

control flow patterns (for control-equivalence technique), etc. We use Wind River Simics [52] to implement these dynamic profilers. Finally, we use the information from both the static and dynamic analyses to generate the final pruned fault set.

For store-equivalence pruning, we dynamically observe every store instruction, the addresses they write to, and record all loads that read the stored value (as explained in Section 3.1.2). For mcf, however, we record only the first ten loads instead of all loads for forming the store-equivalence classes such that our store-equivalence algorithm finishes in a reasonable time of <10 hours.

To quantify the impact of the pruning techniques, we report the percentage of total faults that are pruned (in total and by the individual techniques) and the absolute number of remaining faults (pilots) that must be simulated to determine the resiliency of an application.

3.2.3.1 Validating pruning techniques

The control- and store-equivalence based fault pruning techniques use heuristics and require validation. Each of these pruning techniques chooses a dynamic instruction (*pilot*) to represent the outcome of several other dynamic instructions (the *population*). We quantify the validity of these techniques by quantifying the extent to which the pilots correctly represent the population. For example, suppose the injection of a fault in a pilot results in masking the fault. Suppose the injection of an analogous (hardware) fault in all members of the population results in 98% of the outcomes being masked and 2% detected or SDC. Then we say that the prediction rate of the pilot is 98%. The overall prediction rate for the pruning techniques is the weighted average of the prediction rate for all the pilots for that technique, weighted by the fault populations represented by the pilots.

To find the exact misprediction rate, ideally, we would run fault injections for all the pilots and all their associated populations. Simulating this combination is clearly prohibitive in simulation time. To reduce this time, we first restrict our validations only to the optimized applications. Further, for a given pilot, we randomly sample its population to determine the prediction rate. We select the sample size such that the 99% confidence interval for prediction is within 5% of the actual prediction rate.[7] We then inject transient faults in the pilot and the selected samples to obtain the prediction rate. Ideally, we would inject faults in all bit locations in the appropriate faulty units for the pilot, but the simulation time would be prohibitive. We instead injected faults in every 8th bit (bits 0, 8, 16, 24, 32, 40, 48, and 56 for a 64-bit register or the output latch of the address generation unit) that was not already pruned by the known-outcome pruning technique (e.g., if the known-outcome pruning technique prunes higher-order 32 bits, then we inject faults only in bits 0, 8, 16, and 24).

Sampling the population for a given pilot still leaves the problem that there are many pilots, each of which would require a large number of simulations for validation. We therefore restricted the number of pilots such that it was feasible to simulate all of them (and their sampled populations) in the available time. We selected enough pilots such that the total number of fault injections we had to perform for validation (for pilots and the population) was over one million for each of control- and store-equivalence (1,378,000 for control and 1,093,000 for store) across all fault models. In particular, for validating control-equivalence, we performed approximately 1,092,000 and 286,000 injections for integer register and address generation unit fault models respectively. For store-equivalence, he corresponding number of injections are 835,000 and 258,000. Further, each selected pilot represented a population of at least 1,000. For the 99% confidence interval, our average validation results for control-equivalence pruning have error bars of 1.84% and 3.67% for the integer register and address generation unit fault models respectively. For store-equivalence pruning, the corresponding error bars are 2.85% and 4.61%.

3.3 Evaluation

3.3.1 Effectiveness of pruning techniques

3.3.1.1 Overall pruning effectiveness

Tables 2(a) and (b) show the overall effectiveness of Relyzer's pruning techniques by presenting the percentage of total faults pruned for the optimized and unoptimized applications respectively. The tables also show the absolute number of total faults and the faults remaining after pruning. The applications are ordered according to the total number of original faults.

For optimized applications, we find that Relyzer prunes an aggregate of 99.78% of all the studied faults across all applications. The total number of faults that need to be simulated reduces from 1.6 trillion to 3.52 billion, a

[7] The pilot requires only one fault injection experiment to obtain the outcome A. We can formulate the fault injection experiments for the population as a Bernoulli trial with outcomes being either A or not A. Assuming all the experiments are independent, we can apply the principals of confidence intervals used for normal distributions.

Application	Initial faults (in billion)	Total pruning	Remaining faults (in millions)
Blackscholes	1.9	99.99%	0.07
Ocean	21.7	99.99%	2.9
Libquantum	27.4	99.98%	4.1
LU	33.2	99.99%	1.1
Water	36.6	99.99%	2.1
FFT	48.7	99.99%	0.3
Swaptions	97.3	99.99%	0.6
Fluidanimate	102.5	99.91%	92
Streamcluster	106	99.99%	8.6
Omnet++	146	99.99%	2.2
Mcf	485.4	99.43%	2,781
Gcc	500.4	99.88%	627.5

(a) Optimized applications.

Application	Initial faults (in billion)	Total pruning	Remaining faults (in millions)
Blackscholes	4.01	99.99%	0.03
FFT	61.18	99.99%	0.16
Libquantum	127.03	99.93%	3.40
LU	175.36	99.99%	0.80
Swaptions	318.66	99.99%	0.08

(b) Unoptimized applications.

Table 2: Effectiveness of Relyzer's pruning on (a) optimized and (b) unoptimized applications. The applications are in increasing order of total number of original faults.

three to five orders of magnitude reduction for all applications except mcf. The lowest pruning rate for a single application was 99.43% (for mcf) while most applications saw a pruning rate of 99.99%. For mcf,[8] two stores observed a pruning of 20%, bringing down mcf's overall pruning rate. The number of remaining faults in these two stores and the instructions that these stores depend on alone accounted for 83% of the total remaining faults for mcf.

3.3.1.2 Pruning effectiveness of individual techniques

Figures 4(a) and (b) show the effectiveness of Relyzer's individual pruning techniques for the optimized and unoptimized applications respectively. The stacks in each bar show the contributions of the individual pruning techniques when applied in the order shown (bottom to top) for all the faults in the application. There is no stack for constant-based pruning techniques in part (a) because our preliminary experiments showed these techniques provide limited benefit for those applications.

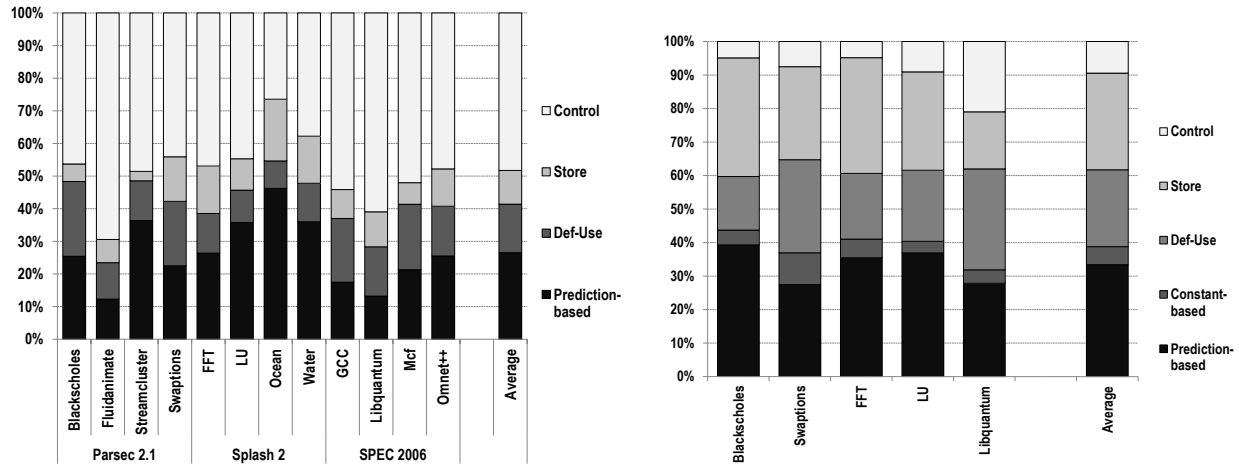
Focusing on the optimized applications, we found that the known-outcome pruning technique pruned an average of approximately 27% of all the faults. Def-use analysis prunes 15% of all the faults on average. Thus, the above mostly static techniques alone provided approximately 42% of pruning across our applications. Control-equivalence is overall the most effective individual technique for these applications, providing 48% of the pruning on average. Finally, store-equivalence technique pruned about 10% of all the faults.

The unoptimized applications show slightly different behavior. First, store-equivalence provides notably more pruning than in the optimized applications. A likely reason is that there are more memory operations in unoptimized codes since they use the stack heavily and the registers poorly. Moreover, these store operations are often represented by a small number of pilots because they observe few permutations of loads during store-equivalence pruning. Second, the constant-based techniques provide significantly more pruning for the unoptimized applications (about 6.5% of total faults). We observed that the number of remaining faults increases significantly (by about 100%) when this technique was excluded for the unoptimized applications. However, it had negligible impact on optimized applications. Overall, figure 4 shows significant differences in the relative effectiveness of the pruning techniques between the optimized and unoptimized codes, showing that compiler optimizations do impact the behavior of fault propagation.

3.3.1.3 Trading off simulation time with coverage

Although Relyzer is able to prune faults effectively, there are still a relatively large number of remaining faults that need to be simulated, especially for the longer applications. Relyzer allows a systematic method to trade off simulation time with coverage, revealing sweet spots that dramatically reduce simulation time with modest reduction in coverage.

[8]For forming the store-equivalence classes for mcf, we accounted for the first ten loads instead of all loads so that our algorithm finished in a reasonable time of <10 hours.



(a) Optimized applications

(b) Unoptimized applications

Figure 4: Effectiveness of the individual pruning techniques for (a) optimized and (b) unoptimized applications.

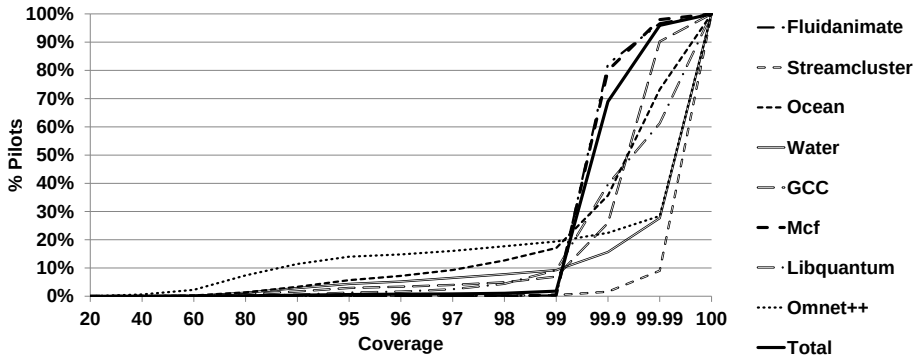


Figure 5: The percentage of pilots (y-axis) required to provide a desired amount of fault coverage (x-axis) for optimized applications. The x-axis shows the percentage of the total initial number of faults that are covered by the corresponding percentage of pilots. This includes the faults from the known-outcome category which are always considered covered. Note that the scale on the x-axis is not linear. Only individual applications that have more than 1 million remaining (not pruned) faults are shown, along with a curve for the aggregate faults across all applications.

Figure 5 shows the percentage of pilots (y-axis) needed to provide a desired coverage of the faults across the entire application (x-axis) after applying all pruning techniques for the optimized applications. These pruning techniques include the known-outcome class, which is considered to be always covered. For readability, we plot only the individual applications that had more than 1 million remaining faults that need simulation. For the full picture, we also plot the data for the total number of faults.

It is evident from the figure that only a small fraction of the pilots cover most of the faults. For example, 99% of all the faults across all the studied applications can be covered by 1.81% of the pilots. This corresponds to approximately 64 million faults. We can reduce this set further by compromising on the bit locations; e.g., injecting a fault in only every eighth bit of a given dynamic instruction, as in our validation experiments. With our existing simulation speeds, this set of faults can be simulated in approximately 21 days on a cluster of 200 cores.

We observed similar results for unoptimized codes as well, but do not present them here because most of the unoptimized applications we studied have under 1 million remaining faults.

3.3.2 Validation of heuristics-based pruning techniques

3.3.2.1 Prediction rate for control- and store-equivalence

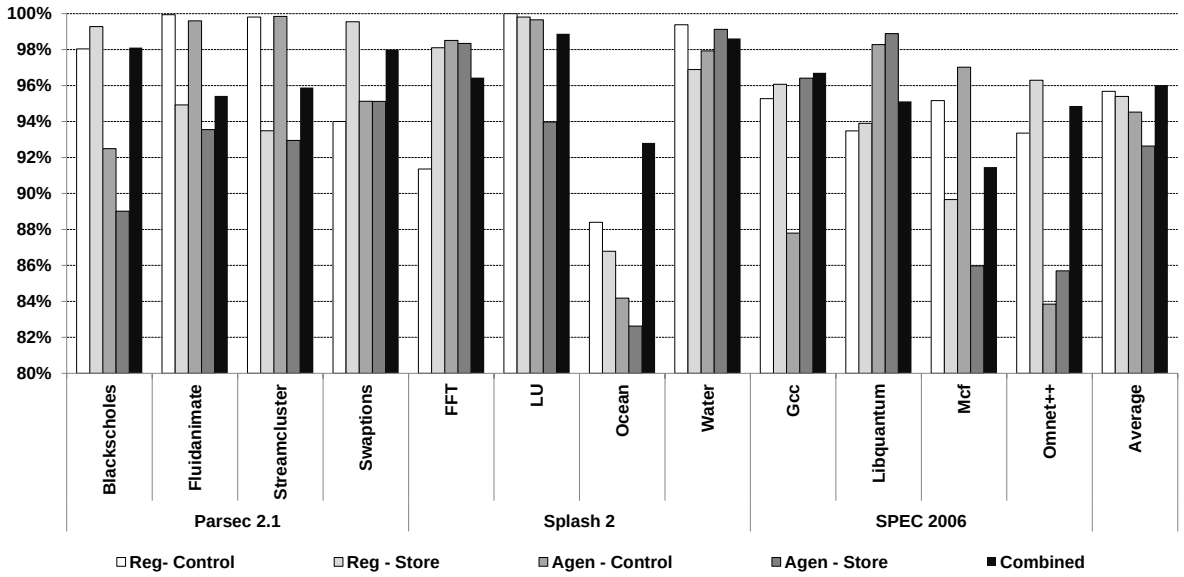


Figure 6: Validation of control- and store- equivalence for integer register (reg) and output latch of address generation unit (agen) faults for optimized applications. The *combined* bars for each application show the prediction rate across all fault models and pruning techniques.

We validated the heuristics-based pruning techniques, namely control- and store-equivalence, for the optimized applications as described in Section 3.2.3.1. Figure 6 shows the prediction rate of the pilots for all twelve applications and both the studied hardware fault models (integer register or *reg* and output latch of address generation unit or *agen*). The combined bar for each application shows the observed prediction rate across all studied fault models after applying all pruning techniques. For each application, the combined bar is the average of the prediction rate of each pruning technique and fault model combination, weighted by the fraction of faults pruned by that combination. Specifically, in addition to accounting for control- and store-equivalence, this bar also accounts for faults pruned by def-use pruning (by associating a def-use pruned fault's prediction rate with that of its representative faults' rate). It also accounts for known-outcome based pruning, assuming a 100% prediction rate for that technique.

Figure 6 shows that the pilots selected through control-equivalence predict the outcome of their populations with an average (across all applications) accuracy of 95.7% for reg faults and 94.5% for agen faults. The pilots selected by store-equivalence predict their population's outcomes with an average accuracy of 95.4% for reg faults and 92.6% for agen faults. The figure also shows that for each individual application, Relyzer predicts the outcome across all fault models and pruning techniques with an accuracy of >91% (shown by the *combined* bar). This prediction rate averaged across all applications is 96%.

Integer register faults observe a prediction accuracy of approximately 90% or higher for all applications except Ocean. On the other hand, agen faults showed <90% (the lowest is about 82%) for some cases for five applications – Blackscholes, Ocean, Gcc, Mcf, and Omnet++. We examined a few of these cases to understand why the prediction rate was not higher, and believe many of these can be eliminated by refining our heuristics.

For example, for Omnet++, a notable contributor to the mispredictions with control-equivalence for agen faults was a load instruction that should have been labeled for store-equivalence pruning. The instruction directly affected a store (and nothing else), but in the next basic block. Since our analysis looks only within the basic block for such data dependences to select instructions for store-equivalence pruning, it could not find this dependency. We plan to extend our static techniques in the future to enable correct labels in such cases.

As another example, we examined Blackscholes for store-equivalence pruning for agen faults. The major contributor to the misprediction rate was a load instruction loading values from faulty addresses. In several cases, it so happened that the faulty and correct address had the same value; therefore, the fault was masked rightaway. On the other hand, sometimes this was not the case, and the fault led to an SDC. A common pilot represented both classes of cases, leading to mispredictions. The fundamental issue is that our heuristics do not examine the faulty value. For Blackscholes, we could easily modify our implementation so that during our profiling step (fault-free execution), for every potential fault in a load address, we check the faulty address to determine if the value is different. This leads to a known-outcome based technique that can immediately determine if such a fault

would be masked.

Examining the mispredicted cases in Ocean for agen faults pruned by store-equivalence exposed a more difficult limitation of Relyzer. A store instruction with a faulty address was one of the major sources of the high misprediction rate in Ocean. Such a store corrupts the intended address by not writing the value from the source register and also the faulty address by writing an unintended value. We found the root cause of the misprediction to be the writing of the source register value in the faulty address. Since Relyzer cannot examine fault propagation through faulty addresses, this becomes a fundamental limitation and overcoming it is an interesting future direction.

3.3.2.2 SDC vs. non-SDC prediction rate

A key application of Relyzer is identifying SDC causing fault sites; therefore, we would like to ensure that Relyzer's prediction rate for SDC causing faults is also high. The data in figure 6 showed the prediction accuracy for masked and detected faults as well. Here, we distinguish only between SDC and non-SDC outcomes, treating the masked and detected outcomes the same. With just the SDC and non-SDC categories in mind, we revisited the validation results in figure 6. We observed that the average (across all applications) prediction rate for control-equivalence for reg and agen faults is 96.6% and 96.2% respectively, slightly higher than the overall prediction rate which also distinguished between the two non-SDC outcomes. Similarly, the average prediction rate for store-equivalence for reg and agen faults is also a higher 95.9% and 94.9% respectively.

4 Low-cost Program-level Detectors for Reducing SDCs

Since Relyzer significantly reduced the number of faults that require detailed fault injection experiments for complete resiliency analysis, we performed fault injections in the Relyzer-identified sites and obtained a comprehensive list of SDC-causing instructions in the entire application. Investigating the SDC-causing fault sites revealed that only a small fraction of static instructions cause most SDCs. Figure 7(a) shows that virtually all the SDCs for the studied applications were caused by just 20.6% of the static instructions on average; 90% of the SDCs were caused by a mere 5.4% of the static instructions (Section 4.2 provides the detailed methodology for these results). This observation motivates using selective instruction-level detection techniques. Prior work has made similar observations, but has used selective instruction-level redundancy for detection [15, 44, 45].

Figure 7(b) shows the execution time (in number of dynamic instructions) consumed by the static instructions that cause SDCs. We find that the small fraction of SDC-causing static instructions consume a much higher fraction of the execution time. The figure shows that protecting all SDC-causing instructions through instruction-level redundancy may incur 50% overhead on average, assuming a conservative one cycle overhead per covered instruction (33% overhead on average for covering 90% of the SDCs). This high overhead is consistent with that reported for previous selective instruction-based redundancy techniques [15], and motivates selective detection techniques that are much more cost effective than instruction-level redundancy.

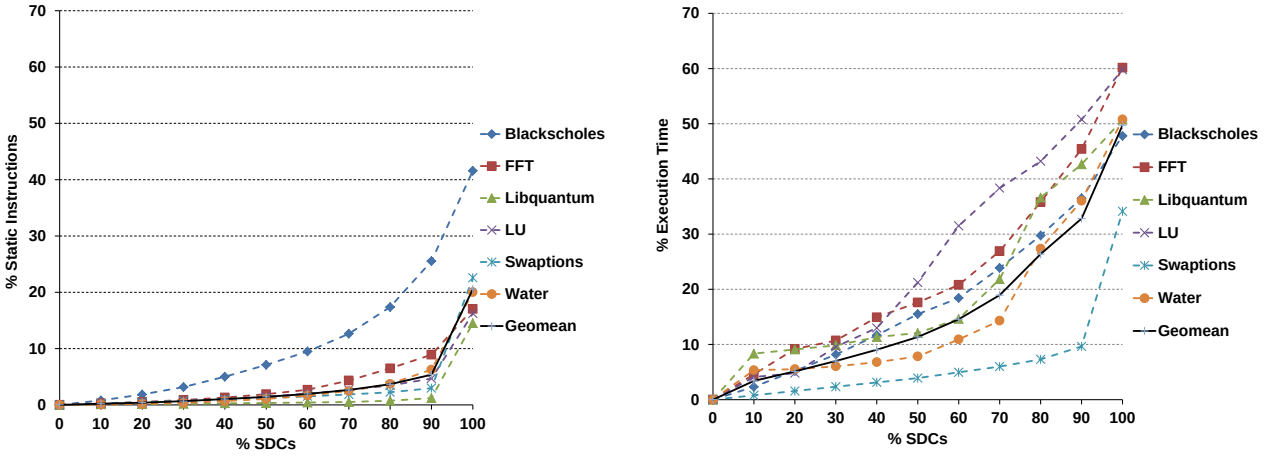
Hence with the goal of reducing and possibly eliminating the reliance on instruction-level redundancy, we focus on finding alternate low-cost program-level error detectors. Our approach is to move up from the instruction-level to understand the program behaviors and properties that are responsible for producing SDCs.

4.1 Analyzing SDC-Causing Program Sections and Developing Program-level Detectors

We first analyze the list of the SDC-causing fault sites. We sort the SDC-causing static instructions in decreasing order of the number of SDCs they can produce, and analyze them in that order. For each instruction, we inspect the disassembled binary code around it to associate an application code (C code) section with it.[9] To our surprise, we observed a few code properties appearing repeatedly across different locations in the same application and even across different applications.

Given the SDC-causing sites, the next goal is to identify *where* to place the detectors and *what* detectors to use. For placement (where), the program locations should be selected such that many faults propagate to these points in a few variables. We used the end of loops and function calls that contain the SDC-causing instructions. For the detectors (what), we exploit a range of program-level properties: (1) comparing similar computations,

[9] Compiler optimizations often make a direct association harder. However, we were able to identify the section of application code that contains the instruction of interest in most cases.



(a) (b)

Figure 7: SDC-causing instructions and their impact on execution time. For a given application and input, part (a) shows the percentage of (executed) static instructions that cause a given percentage of silent data corruptions (SDCs). Part (b) shows the fraction of execution time (on a 1 IPC machine) taken by the static instructions in part (a) (for the given percentage of SDCs). For example, in FFT, 2% of the static instructions cause 60% of the SDCs and take 21% of the execution time. (The detailed methodology is in Section 4.2.)

(2) checking value equality, (3) range checks, and (4) performing mathematical tests. While devising these program-level detectors, we also ensure that they are low-cost.

Our approach of placing the detectors at the end of loops and function calls can potentially increase detection latencies because the errors are allowed to propagate until a detection point. However, these latencies can be tolerated by the state-of-the-art full-system checkpoint and rollback mechanisms [41, 49]. A further exploration of the relationship between such detection latencies and recovery is part of our future work.

The rest of this section describes the program code sections that we identified as SDC prone with examples, and explains the low-cost program-level detectors we devised to detects SDCs in these code sections using the above mentioned insights.

4.1.1 Incrementalization in loops

We observed that a significant fraction of SDC-causing fault sites directly affect computations in loops. These application sites often correspond to the loop index variables and/or addresses referring to array elements that are accessed in every loop iteration. For example, Figure 8(a) and (c) give the source and compiled code respectively for a single loop in the LU application from the SPLASH2 benchmark suite. Almost all instructions operating on integer registers in this code section were listed high in the sorted list of SDC-causing fault sites.[10] In particular, these faults alone produced over 50% of all the SDCs in LU. Faults in this compiled code can result in SDCs in the following two ways: (1) A fault affecting i can either terminate the loop early or cause it to go back in the iteration space. Since there is no loop-carried dependence, the latter effect will always result in masking the fault. (2) Faults in addresses A and B can result in detection if the faulty address is unallocated. If the faulty address points to a valid but incorrect memory location then the fault may be masked or result in an SDC. In this scenario, we observed that faults in several low-order bits in A and B resulted in SDCs because faulty addresses pointed to incorrect locations in arrays a and b.

Analyzing this code further, we observe that it uses the loop incrementalization optimization [26]. This optimization is typically applied on programs that perform computations on array elements in loops. Addresses to access these array elements must be computed in every iteration. This can be expensive if computed from scratch from the initial value (involving a multiplication and an addition). Modern compilers, therefore, apply the loop incrementalization optimization where the new value of the address is computed from the value in the previous iteration, involving just an addition. This optimization is shown to produce significant performance benefits for array-based codes [26]. Figure 8(b) and (c) show the assembly codes without and with the incrementalization optimization respectively for the C code shown in Figure 8(a).

[10] Our fault model (explained in Section 4.2) considers faults only in integer architecture register operands of executing dynamic instructions.

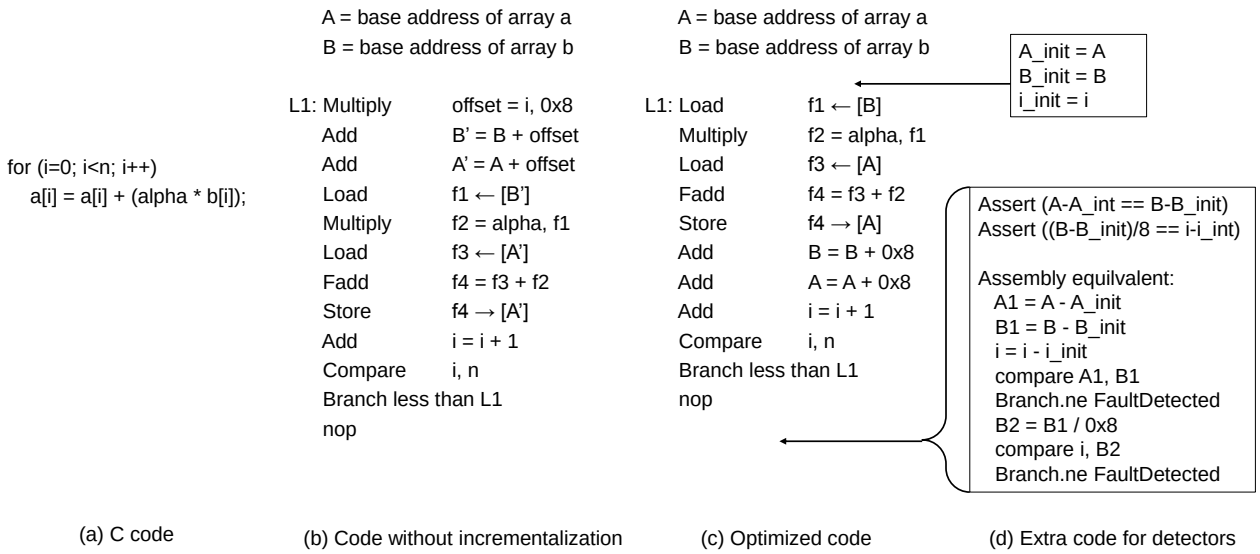


Figure 8: An "SDC-hot" code section with loop incrementalization in LU from the SPLASH2 benchmark suite: (a) C code, (b) unoptimized assembly without loop incrementalization, (c) optimized assembly with loop incrementalization, and (d) detector for the optimized code. Faults in this (optimized) loop alone produce >50% of all SDCs in LU. The extra code in part (d) detects errors affecting i, A, and B in the optimized code. Initial values of these registers are collected at the beginning of the loop. These values are later used at the end of the loop to test the program-level properties.

Detecting errors in incrementalized loops: Incrementalization makes errors in index variables and addresses used to access array elements propagate until the end of the loop. Hence, a property check at this location on these accumulated quantities can detect faults impacting these variables across all the iterations of this loop. Often the incrementalization in a loop is performed on multiple variables such that they all are incremented in every loop iteration with a value that is constant across iterations. We utilize this inherently similar computation to derive a property check at the end of the loop.

Figure 8(d) shows such a detector for our LU example. First, the initial values of A, B, and i are copied into different registers (or predefined memory locations). If the initial value of a register is predetermined as a constant then we can skip this step. For example, we do not have to collect the initial value of i because it is always 0. The values A and B are incremented with the same constant value in all the iterations. Hence the difference between their final and initial values should be the same. This property check can detect all single-event-upsets in these variables in all iterations of the loops. A similar check for variable i can also be performed by accounting for the different amount of increments used for i and A or B (also shown in Figure 8(d)). Since these detectors do not compromise coverage, we call them "lossless."

Codes that do not use the incrementalization optimization may produce intermediate values (offset, A', and B') in every loop iteration as shown in Figure 8(b). Since faults affecting these intermediate values do not propagate to the end of the loop in a few variables, deriving a low-cost error detector is hard for non-incrementalized versions.

4.1.2 Registers with long life

We observed that a sizable chunk of SDCs were caused by faults in registers with long life, with multiple uses through this life. For example, we observed that the register holding the value n in Figure 8(c) is SDC prone. This register stays alive until the end of the loop and is used in every iteration of the loop. Other prominent examples are the registers that hold stack and frame pointers. These registers are typically set at the beginning of a function call and stay alive until the last instruction in the function body is executed.

Detecting errors in a register with long life: Errors in such a register remain alive until the end of the life of the register. Hence, the location to place a detector is, trivially, just after the last use of this register. If the register is used in many instructions through its life, then the cost of the detector is amortized across all of those uses. For this detector, we first attempt to identify another register or a constant such that its value can be compared to our target register. If this attempt fails, then we record the register's initial value (created at the definition of this register) in a different register (or a predefined memory location). At the detection location, we compare the initial value with the latest value in the register. An example of this is detecting faults in the

register that stores the value of n in Figure 8(c). The value of n at the end of the loop can be tested with its earlier recorded value (from the beginning of the loop or its definition point). These detectors, like the previous ones, are also "lossless."

4.1.3 Application-specific behavior

For some applications, a large chunk of the SDC-causing fault sites belong to a few procedures. These procedures often do not have any side effects; i.e., the only output of the procedure is the return value. The exponential function from the math library, the BitReverse function from the FFT application from the SPLASH2 benchmark suite, and the RanUnif function (uniform random number generator) from the Swaptions application from the Parsec benchmark suite are few examples.

Detecting errors in the exponential function: A significant fraction of SDC-causing sites in Blackscholes and Water from Parsec and SPLASH2 benchmark suites respectively belong to the exponential function. The output of this function depends only on the input and no other previously stored data. All the errors created by the static fault sites in this function body, therefore, propagate through the output at the end of this function. We therefore place our detector at the end of the function.

Naively testing the output for correctness at this location can be expensive due to the nature of the function. We utilized a basic mathematical property of this function such that the errors can propagate through accumulating quantities over different invocations. This allows us to perform the test infrequently and still cover all the fault sites in these invocations.

From the definition of the exponential function, we know that $\exp(i1+i2) = \exp(i1) \times \exp(i2)$ and $\exp(i1-i2) = \exp(i1) \div \exp(i2)$, where $i1$ and $i2$ are inputs and $\exp(i1)$ and $\exp(i2)$ are outputs of two invocations respectively. This property allows us to accumulate inputs using addition or subtraction and outputs using multiplication or division respectively. To detect errors, we re-execute this function with the accumulated input and compare its result with the accumulated output. The cost of this re-execution will be amortized across several invocations of this function. To detect errors in tolerable latencies, the frequency of the invocation of this detector can be dictated by the recovery solution (by specifying the tolerable detection latency).

Since a floating point operation on all hardware inherently generates an error and the exponential function on large or small inputs can exacerbate this error, we decided to apply this test only on relatively smaller inputs; i.e., when the absolute value of the input is < 25. For the remaining inputs, we rely on redundancy. We observed that very few invocations in our applications use inputs that are $\leq$ -25 and ≥ 25. Moreover, we use a combination of addition and subtraction on input such that the absolute value of the accumulated input is closer to zero and accordingly we use multiplication or division to accumulate output. This detector may show a loss in detection coverage if the error caused by the fault is within the estimated precision error of the floating point operations. We therefore call this detector "lossy."

Detecting errors in BitReverse function: In the FFT application from the SPLASH2 benchmark suite, nearly half of the SDC-causing sites belong to a function called BitReverse. This function takes an integer value as input and reverses its bits in the boolean representation. For example, if the input is 3 (0011), a 4-bit value, then the output should be 12 (1100).

The output of this function depends only on the input and no other previously stored data. Hence all the errors generated within this function body propagate through the output at the end of this function making it an ideal location for detector placement. Since this procedure does not show any accumulating behavior, we resort to checking parity on both the input and output. Since they both have the same number of bits set, the computed parities should match and detect faults that makes output and input differ by an odd number of bits. Naive software implementation for parity generation, however, can be expensive. One of the most optimized ways is to compute it in parallel [1]. Another way is to use the parity flag in Intel 64 architectures [3] that is generated on every logical and arithmetic operation on the low-order byte of the result. These implementations take <10 instructions to compute the parity of a 32-bit value (Exact implementations can be found in [18]). This detector may lose coverage if the corrupted output has a multi-bit error, and is therefore "lossy."

Detecting errors affecting registers with a fixed upper bound: A significant number of SDCs in the Water application from SPLASH2 were generated by errors in the variable KC in the code segment shown in Figure 9. To detect faults affecting the variables K and KC (directly and/or indirectly) in different iterations of this loop, we placed a detector at the end of the loop. From this code, it is evident that $KC \leq 9$ and $K = 9$ hold at the end of loop; we therefore used these invariants as detectors. Since all faults affecting K cannot be detected by testing $KC \leq 9$ alone, we also add $K = 9$ to the detector. Faults that affect KC alone (without corrupting K) such that $KC \leq 9$ may remain undetected. Since a loss in detection coverage can be observed, this detector is again "lossy."

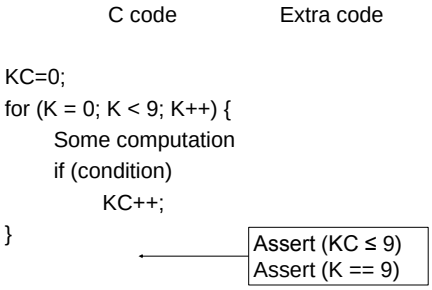


Figure 9: A detector for a register with a fixed upper bound. The figure shows a code section from the Water application. Faults affecting this code eventually corrupt the value of KC and produce SDCs. The assertions show how these faults can be detected.

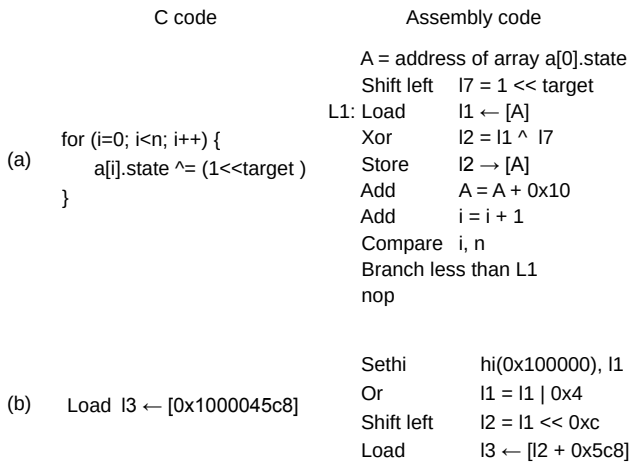


Figure 10: SDCs due to local computations: Faults in short-lived registers, $l1$ and $l2$, produce non-negligible fraction of SDCs. (a) Code from the Libquantum application. (b) Instructions generated by the Sun *cc* compiler to compute a static address.

Detecting errors in the random number generator from Swaptions: Over 90% of the SDCs in the Swaptions applications from the Parsec suite were caused just by a uniform random number generator function. This function takes a seed as the input and performs a series of integer operations to update the seed. This updated value is then used to generate the random number which ranges between 0 and 1. Since errors always propagate through the output, we place the detector at the end of this function call and it tests whether the output follows the specification; i.e., $0 \leq output \leq 1$. Since this detector cannot detect all the errors affecting the output of this function, it is "lossy."

4.1.4 Local computations or registers with short life

We observed that a non-negligible fraction of SDCs were caused by faults in local computations with short register data flow chains. One example of this scenario is shown in Figure 10(a). Registers $l1$ and $l2$ store intermediate results and have short lives. Faults affecting these registers eventually corrupt the values stored in memory locations pointed by A. Another example of this pattern is the sequence of instructions that compute the static addresses known at compile time. In SPARC V9 systems (our target machine), the global data section is stored above 1GB point in the virtual address space layout [4] and hence addresses of global variables require >32 bits. Multiple instructions are needed to generate these addresses because the ISA lacks instructions that can move constants of required sizes of >32 bits directly.

Since errors in the locally computed values do not propagate to a few values at an easily identifiable location in the program, deriving detectors and placing them for cost-effective detection is hard. Hence, we rely on instruction-level redundancy for these computations.

4.2 Experimental Methodology

We analyzed application resiliency by performing fault injection experiments in the fault sites that are selected by Relyzer (Section 3). For our fault model, we consider transient faults or single bit flips in every bit in each of integer architecture register operands (one at a time) of executing instructions. Since this fault model considers fault sites that are highly likely to be architecturally live, it inherently filters a large fraction of masked faults (faults that do not affect application output). This allows us to focus more on faults that impact application output (and potentially cause SDCs).

We selected a mix of six applications from the SPLASH2 [55], Parsec [9], and SPEC CPU2006 [21] benchmark suites for this study (Blackscholes, FFT, Libquantum, LU, Swpations, and Water from Table 1). All the selected applications were compiled using Sun C/C++ compiler version 5.9 with the highest level of optimization. We performed fault injections such that 99% of all the fault sites were analyzed (as reported by Relyzer). Overall we performed 890,000 fault injections across all the studied applications. These experiments were completed in

approximately 3 days on a cluster of 175 compute nodes. The fault injection framework used in this evaluation is similar to the one described in Section 3.2.2.

4.2.1 Detectors and overhead evaluations

We implemented our program-level detectors (described in Section 4.1) in Simics using breakpoints. Simics provides a framework to set breakpoints on various processor events and perform desired computations on these events. Our program-level detectors usually have two parts - one for collecting the information (typically at the beginning of loops or functions) and the other for executing a specified check. At these points, we also collect information needed to measure the execution overheads. We measure the overheads in terms of the increase in the number of dynamic instructions. Table 3 shows the number of instructions we add to the application's total number of dynamic instructions on every invocation of collection or testing point of a detector. We measure the overheads for instruction-level redundancy by estimating that one instruction can be protected by one extra instruction even though the requirement is often more.

4.2.2 Evaluating the lossy detectors

The expected coverage of a detector is obtained by analyzing SDC causing sites and checking whether the detector can catch errors originating from these sites. Since the actual coverage observed by the lossy detectors may differ from the expected coverage, their effectiveness must be evaluated experimentally. Hence we performed a statistical fault injection campaign for the fault sites that are expected to be covered by these detectors. Overall we performed approximately 10,000 injections such that the error bars on our results are $< 2.8\%$ at 99% confidence level.

4.2.3 Determining the lowest overhead detectors for a target SDC coverage

Our detectors from Section 4.1 coupled with instruction-level redundancy-based detectors provide a range of choices to achieve a given SDC coverage (fraction of SDCs detected). We would like to determine the lowest overhead set of detectors for each target SDC coverage, and understand the consequent trade-off between execution overhead and SDC coverage. Such SDC coverage vs. overhead curves also enable a fair comparison with instruction-level redundancy based detectors.

To generate the above curves, we used a dynamic programming algorithm similar to one that solves the 0-1 knapsack problem. We start by labeling all (mutually exclusive) detectors of interest (redundancy based and/or our program-level detectors) with the fraction of SDCs they cover and their execution overheads (as discussed in Section 4.2.1). We then run the optimization algorithm to find the combination of detectors with the minimum combined overhead with a constraint that the sum of the SDC coverage provided is at least equal to the target.

We generate execution overhead vs. SDC coverage curves for different classes of detectors: instruction-level redundancy only, our lossless detectors, and our lossless+lossy detectors. For the last two curves, some SDC causing static instructions of an application may not be covered (or may be only partially covered) by our program-level detectors. We therefore add instruction-level redundancy-based detectors for those static instructions to our dynamic programming problem. For the partially covered static instructions, the SDC coverage assigned to the redundancy based detectors (for the purposes of our optimization algorithm) is the number of SDCs not covered by our detectors. For the lossless+lossy curves, the dynamic programming algorithm assumes there is no coverage loss in the lossy detectors when determining which redundancy based detectors to consider (but the SDC coverage attributed to the lossy detectors when plotting the curves does take into account the loss using the method in Section 4.2.2). Thus, these curves may still terminate without covering all SDCs. Finally, the overall optimal solution for a target SDC coverage is to select the set of detectors that incur the least execution overhead among the above three trade-off curves.

4.3 Results

4.3.1 Sources of SDCs

For reference, Figure 11 shows the absolute SDC rates obtained by our Relyzer-driven fault injection experiments as described in Section 4.2. The SDC rates of our applications range between 8% to 32%. These are much higher than prior evaluations [24, 41], primarily because of the difference in the fault model. Our fault model considers faults in only those architectural registers that are highly likely to be alive, whereas prior work uses microarchitecture (and lower) level fault models which have a much higher masking rate [24, 23]. We chose the higher level fault model because our focus is on uncovering all possible SDCs with as few fault injections as

	Operations	Estimated number of instrns.
Collecting a reg value	$reg' = reg$	1
Lossless detectors	$r1 - r1' == r2 - r2'$	4
	$(r1 - r1')/const == r2 - r2'$	5
	$(r1 - r1')/r3 == r2 - r2'$	5
	$r1 == r2$	2
	$r1 == const$	2
	$r1 == r2 - const$	3
Lossy detectors	$r1 \leq const$	2
	Testing BitReverse functionality	20
	Accumulated check for *exp* function	20
	Range checking for RandUnif $0 \leq reg \leq 1$	4

Table 3: Extra instructions used for measuring execution overhead

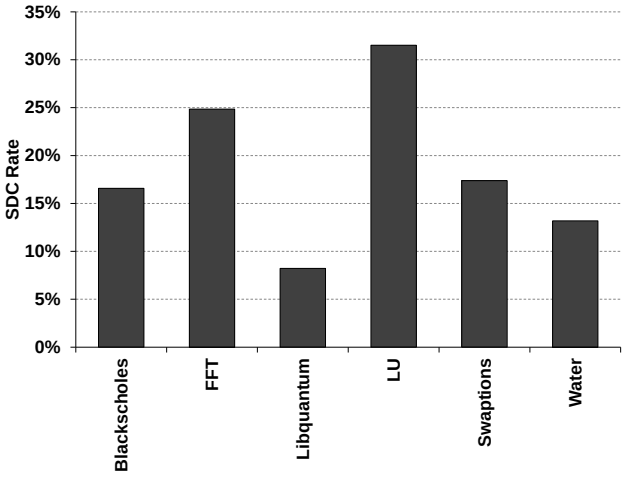


Figure 11: Baseline absolute SDC rates. These absolute rates are higher than previously reported for symptom-based detectors [24, 41], largely because of the different fault models used.

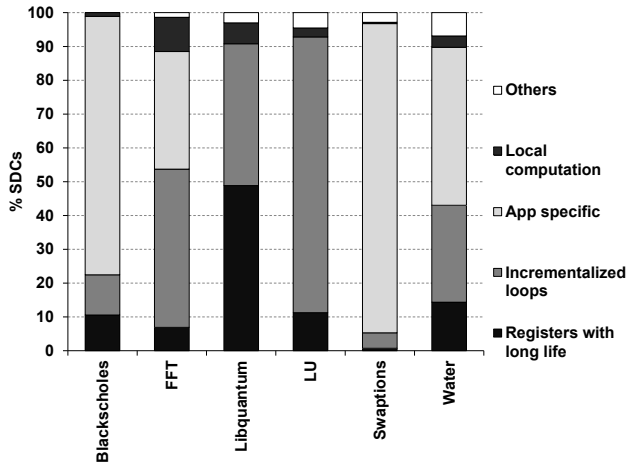


Figure 12: Contribution of code patterns from Section 4.1 to SDCs.

Applications	Num app. locations	Lossless		Lossy
		Loop based	Long lived reg. based	App. specific
Blackscholes	2	4	4	1
FFT	10	15	12	1
Libquantum	10	8	18	
LU	13	12	16	
Swaptions	9	12	5	1
Water	15	13	17	2

Table 4: Number of detectors placed in the static application code

possible and then reducing those SDCs. While we report the absolute SDC rate here for reference, the rest of this section focuses on the fraction of the baseline SDCs that are detected by our detectors (or SDC coverage).

To understand where in the program the SDC causing instructions come from, Figure 12 categorizes them based on the code patterns we identified in Section 4.1. Figure 12 shows this categorization. We observe that fault sites that correspond to registers with long life and incrementalized loops produce a significant fraction of SDCs for FFT, Libquantum, LU, and Water (>90% of SDCs in Libquantum and LU). Application-specific behavior was a major contributor for Blacksholes, FFT, Swaptions, and Water. The figure shows that only a small fraction of SDC producing fault sites (up to 11.5%) were either categorized as local computations or not categorized at all (labeled as others in the figure) for all applications. This indicates that our detectors can potentially cover a large fraction of SDCs.

4.3.2 Static overhead of the program-level detectors

Table 4 shows the program-level detectors placed in the static code for our applications. The second column shows the number of static application locations where our detectors were placed. The remaining columns show the number of detectors placed for covering faults in incrementalized loops, registers with long life, and application specific behavior. The sum of the last three columns may not add up to the value in the second column because multiple detectors can be placed in one static code location. The relatively small number of static code locations that require modifications shows that our devised detectors are not intrusive on the application.

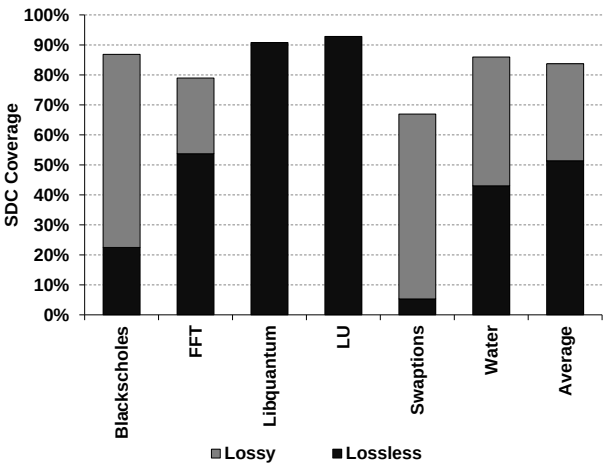


Figure 13: SDC coverage obtained by our program-level detectors, separated into coverage from the lossless and lossy detectors.

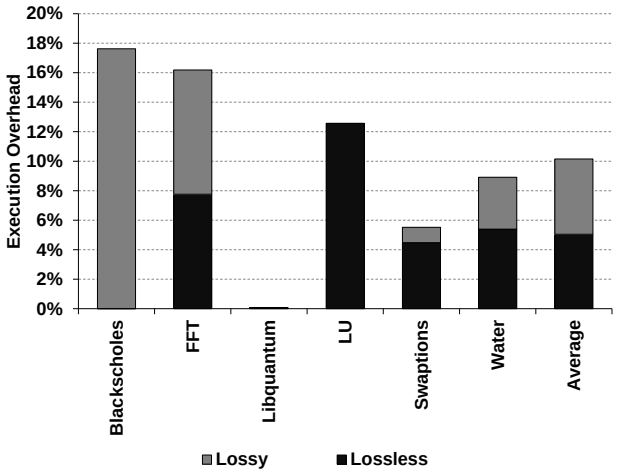


Figure 14: Execution overheads incurred by the program-level detectors, separated into coverage from the lossless and lossy detectors. The overhead of LU can be lowered to 3.4% with a small change in an input parameter without loss of performance or SDC coverage.

Moreover, the small number of application specific detectors means that limited program knowledge is required to implement them. This reinforces the benefit of Relyzer in pinpointing the SDC-vulnerable code sections that need examination.

4.3.3 SDC coverage of the program-level detectors

Since the program-level detectors were placed based on the SDC vulnerability of the fault sites, the corresponding reduction in the SDCs (SDC coverage) is known a priori, assuming that the added detectors are perfect. Thus, for the lossless detectors, the corresponding areas marked in Figure 12 (*incrementalized loops* and *registers with long life*) directly give the SDC coverage. We observe that on average, these detectors alone provide an SDC coverage of 50%. These detectors do not need further evaluation – they are sound and do not compromise coverage of their corresponding SDC sites.

Figure 12 shows that the application specific or lossy detectors also potentially cover a significant fraction of SDCs. Since these detectors can observe a coverage loss, their actual SDC coverage cannot be derived from their area in Figure 12. Instead we use a statistical fault injection campaign as explained in Section 4.2.2. Our detectors for the exponential function, BitReverse function, values with upper bounds, and uniform random number generator show a coverage loss of 16%, 27%, 3%, and 33% respectively, relative to their expected or potential coverage indicated by Figure 12. For faults in the exponential function, we observed that most of the undetected faults produced outputs that could be tolerated by the application. For the random number generator, we observed that for our input set, the number of iterations of the corresponding Monte Carlo simulation executed is small and not yet convergent; preliminary experiments showed that with a large enough number of iterations, the errors may be tolerated in this case as well. In this work, however, we treat all undetected faults that result in output deviation as loss in coverage.

Figure 13 shows the total actual SDC coverage of our program-level detectors, combining both the lossy and lossless detectors. The figure shows that our detectors are highly successful, converting 67-92% of the original SDCs into detections (average of 84%), with both the lossy and lossless detectors contributing significantly.

4.3.4 Execution overhead from the program-level detectors

Figure 14 shows the runtime overheads of our program-level detectors, separating the contributions from the lossy and lossless detectors. The overheads range from 0.08% to 18%, with an average of 10%.

The largest overheads come from the lossy application-specific detectors. Specifically, the exponential function in Blackscholes and the BitReverse function in FFT take the overheads for these applications to over 10%. Libquantum, Swaptions, and Water see much lower overheads of under 10%. Libquantum in particular sees almost zero overhead because of its use of loop-based detectors placed at the end of long running loops.

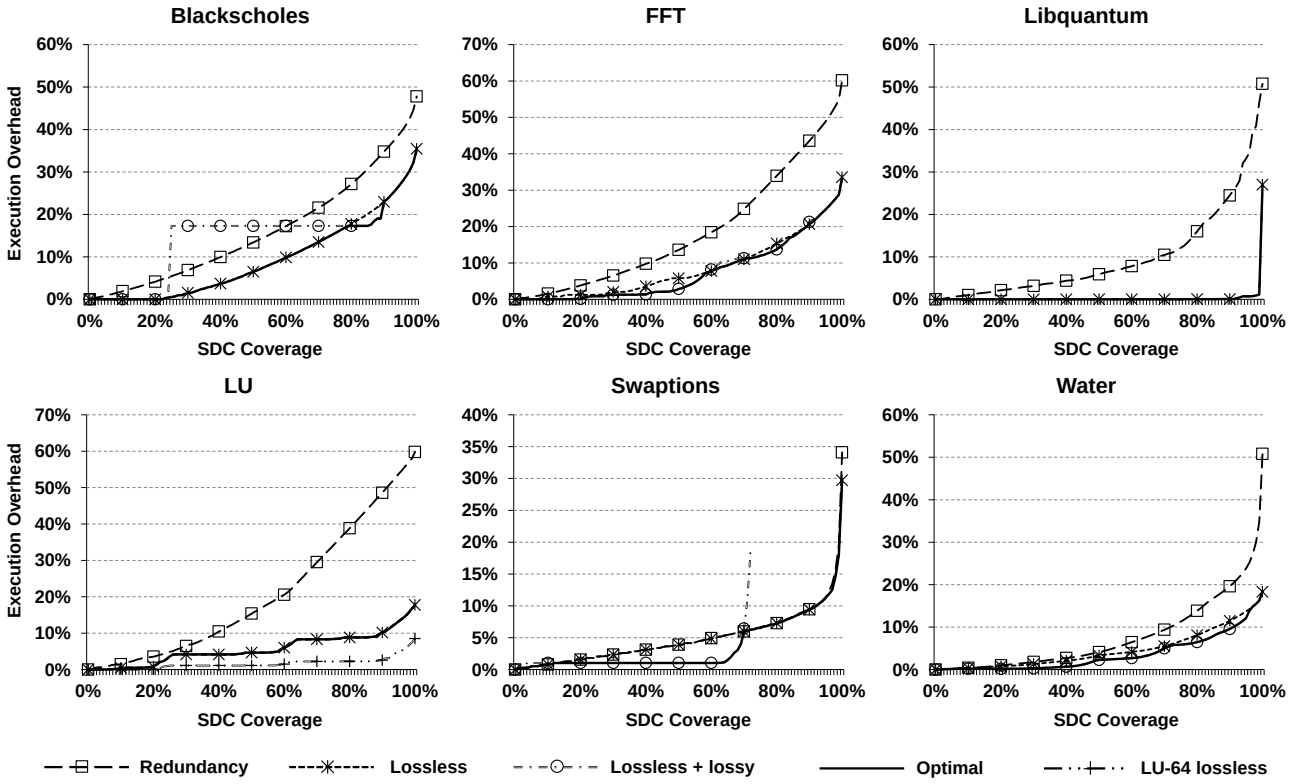


Figure 15: SDC coverage vs. execution overhead for each application for different classes of detectors.

Although LU shows an overhead of 12.57%, a closer look showed that it can be lowered significantly. One of the loop based detectors (shown in Figure 8) executes with high frequency because the loop terminates after a small number of iterations (16 in particular). The number of iterations of this loop is dictated by a parameter that controls the block-size used by the blocking optimization for improving the effectiveness of memory hierarchies. This parameter can be increased to 64 on modern processors without any loss of performance [9]. When we deployed our detectors on this application with the block-size parameter set to 64, we observed that the overheads reduced to a much lower 3.24%. Since all the detectors used in this application are lossless, there is no compromise on SDC coverage with this modification.

4.3.5 SDC coverage vs. execution overheads

Figure 15 plots, for each application, SDC coverage vs. execution overhead trade-off curves for different classes of detectors: instruction-based redundancy, lossless program-level, lossless+lossy program-level, and optimal that combines the best of the above. For LU, the figure also shows a curve for the version with the block size of 64 (using the same SDC profile as for the base LU since the application binary and inputs other than block size are unchanged). The methodology used is as described in Section 4.2.3. In particular, the curves for program-level detectors add (selective) instruction-level redundancy for the SDC targets they cannot otherwise reach. The above curves serve two purposes: (1) they provide a fair way to compare the redundancy based and program-level detectors by allowing overhead comparisons for a fixed SDC coverage target and (2) they enable programmers and system designers to systematically trade off SDC coverage and performance.

The graphs show that our program-level detectors can reduce overhead relative to instruction-level redundancy alone at all target SDC coverage points for most of the applications. Focusing on Libquantum and LU, which do not use lossy detectors, we observe that in both cases, the overhead reduction relative to redundancy-only is quite high for the most part. The gains for LU are magnified when a larger block size of 64 is used (the "LU-64 lossless" curve). For Libquantum, the program-level detectors see near zero overheads to cover up to 91% of the SDCs. For both applications, the optimal curves fully overlap the program-level detector (lossless) curves.

Among the applications that use lossy detectors, all but Swaptions see significant overhead improvements for most of the interesting SDC coverage targets. In Blackscholes, the lossless+lossy curve shows a step behavior at 25% SDC coverage because the detector used to cover the SDCs in the exponential function with overhead of

about 18% was required to achieve the target SDC coverage. This detector could have potentially capped the overhead for high SDC coverage points but its lossy behavior limited its coverage.

For FFT and Water, the use of the lossy detectors along with the lossless ones consistently provided lower execution overheads than lossless detectors alone. In Swaptions, the simple lossy detector provides a low-cost alternative to redundancy up to an SDC coverage of up to 70%. For higher coverage the optimal solution was, however, to use redundancy for the most part. The lossless detectors provided limited benefit in reducing the overhead needed to cover all SDCs.

Our approach consistently yields much better execution overheads for all SDC coverage targets of interest on average. The optimal solution at 90%, 99%, and 100% average SDC coverage incur execution overhead of 12%, 19%, and 27% respectively, whereas the corresponding overheads for the redundancy-only solution are 30%, 43%, and 51% (which are 2.5X, 2.26X, and 1.89X higher).

5 Future Work

5.1 Reducing Reliability Evaluation Time

5.1.1 Developing profile-driven metrics to find SDC-causing application sites

Relyzer makes it possible to list SDC-causing instructions with high accuracy. It requires detailed dynamic profiles of the applications (majority of them are input specific). These input-specific profiles may hinder its practicality. To overcome this challenge, we plan to identify simple program metrics that can list the SDC-causing instructions with high accuracy and low-effort. Prior work proposed using program-level metrics obtained from simple program profiles to identify vulnerable program variables and/or instructions [35, 7, 15]. These techniques relied on limited error propagation and mostly static analyses reducing their accuracy.

Pattabiraman et al. [35] developed metrics, namely fanout and lifetime, to identify what application variable to protect and where to place detectors. The goal, however, was to prevent or limit fault propagation and avoid system crashes with minimum possible detector locations, not particularly to reduce SDCs.

Benso et al. [7] proposed metrics to obtain criticality behavior of every program variable with simple runtime analysis. They developed an analytical model for this purpose that considers three variables – lifetime of the application variable, number of reads to it, and whether it is a pointer or not. The work proposes that the contribution of each of these variables is application independent. The results were, however, shown on small applications with few variables and the application independent behavior was not thoroughly evaluated.

Shoestring [15] is a purely static technique that identifies static instructions where faults are likely to be detected quickly enough; e.g., there is a short-enough path in the data-flow graph from such a fault to enough potentially symptom-generating instructions. The rest of the faults are considered vulnerable and the important ones among these (currently, stores) are protected by duplicating any instructions that produce data that feeds into them. One shortcoming of Shoestring is that it only employs static analysis to identify vulnerable instructions and does not distinguish between different instances of a static instruction and exploit information known only at execution time (e.g., store and load addresses). It would be interesting to combine the analyses of Shoestring and Relyzer such that SDC-prone application sections can be identified in limited evaluation time.

We plan to investigate these metrics (develop new ones) and tailor them to identify SDC-causing instructions with low-effort. Evaluating these metrics, so far, was time consuming and tedious because statistical fault injections were required to test every new metric. With Relyzer, however, this process is significantly simplified because we now have the list of virtually all SDC-causing program instructions. This allows us to evaluate the effectiveness of new metrics (and old ones) with relatively little effort.

5.1.2 Systematic fault simulation framework

Relyzer is the first techniques, to our knowledge, that systematically analyzes all application-level fault sites for a thorough resiliency profile. The resiliency evaluation time of Relyzer[11] is practical, but it may still be high in some situations. The current speed of our fault simulation framework is one of the main factors that increases the evaluation time needed to identify SDC-causing instruction by Relyzer. A fault injection experiment simulates the entire application from the fault injection point to identify its impact (SDC or application masking) making

[11] Dynamic application profiling and fault injection campaign require approximately 2 days and 15 hours per application respectively. The dynamic profile runs, however, can be parallelized and require <2 days approximately for all the applications together. On the other hand, the time needed for the fault injection campaign is only going to increase with applications because it is already parallelized (utilizing all 175 nodes on our cluster).

it expensive in time. To significantly lower the fault simulation time, we propose to develop a systematic fault simulation framework where all faults are simulated only for a short duration, until a test point (or t-point) in the application is reached. Only the faults that corrupt the *live* application state at these points in unique ways need further simulation. In other words, faults that produce same live state corruptions at a t-point are equivalent. Hence, just one fault requires further simulation. Moreover, faults that do not corrupt the live program state at a t-point can be declared as *masked* and the simulation can terminate. A series of t-points in an application can potentially reduce the number of full fault simulations significantly. We explain the t-points, their generation, and their potential benefits in the following two subsections.

It may seem that the goal of this approach is identical to the one described in Section 5.1.1 - lowering the time needed for resiliency evaluation and identifying SDC-causing instructions. Finding the most-accurate metric in Section 5.1.1, however, requires the list of SDC-causing instructions making this approach valuable. Moreover, if accurate low-effort metrics (from Section 5.1.1) cannot be identified for a class of applications then this approach will be significantly beneficial.

5.1.2.1 T-points

T-points are primarily used to check for fault masking and equivalence. This can be achieved by comparing the entire application state at these points during a fault simulation with the fault-free application state at the same point (obtained from fault-free execution) and/or previously collected faulty application state at these points (obtained from previous fault simulations). Comparing the entire application state, however, can be inefficient and prohibitive in time. Hence, the t-points should be selected such that the application state being affected by a fault is confined to few application variables. Interestingly, we found that the SESE (single-entry single-exit) regions [22], from the compiler literature, have properties closest to what we desire. A SESE region is defined in a graph with an ordered edge pair (a,b) of distinct control flow edges a and b where a dominates b; i.e., every path from start to b includes a; b postdominates a; every path from a to exit includes b; and every cycle containing a also contains b.

Since there is only one exit point in a SESE region, all the data has to flow though that point. Therefore, if we inject a transient fault inside a SESE region then the error should propagate though the exit point of the region, assuming that the dynamic control flow follows the static control flow graph. Moreover, we also know that the errors originating in this region will flow through the *live variables* at the exit point of the SESE region. The set of live values at these points can be obtained by live variable analysis (a data flow analysis) of the program [30]. This approach can potentially limit the amount of state needed for comparison to identify fault masking and equivalence significantly at the exit points of the SESE regions.

5.1.2.2 T-points generation algorithm

Our algorithm to identify t-points is inspired by and similar to the SESE regions identification algorithm [22]. In a straight-line code, the region between any two points is a SESE region; we will ignore these regions and focus only on the block-level CFG where the straight-line code has been coalesced into basic blocks. For a basic block with multiple entry edges, we split it in to two blocks such that the first one with multiple entry edges has as few instructions as possible and second block has a single entry edge. Similarly, we also split the blocks with multiple exit edges such that the first block has a single exit edge and the second one has very few instructions. We then obtain the SESE regions using the algorithm presented in [22]. For adjacent SESE regions a and b, where exit edge of a is same as the entry edge of b, we obtain c by combining a and b (which is also a SESE region).[12] We then remove b and add c to the list of SESE regions. This allows us to enumerate more t-points for any node in the program than the original SESE region identification algorithm.

Once all the SESE regions are obtained, they are organized in to a program structure tree (explained in the next section). This tree structure can simplify the identification of the next t-point during a fault simulation.

Figure 16 shows a CFG and the extracted SESE regions obtained by applying the above mentioned algorithm on it. In this example, the checks for masking and equivalence to shorten the simulations for faults in basic block numbered 3 are performed at the exit edges (after the last instructions) of the basic blocks numbered 7, 8, and 16. Similarly, for simulations for faults in basic block 13 the checks are performed at the exit edges of the basic blocks 14, 15, and 16.

[12] If the edge pair (p,q) is a SESE region and (q,r) is a SESE region, then (p,r) is also a SESE region

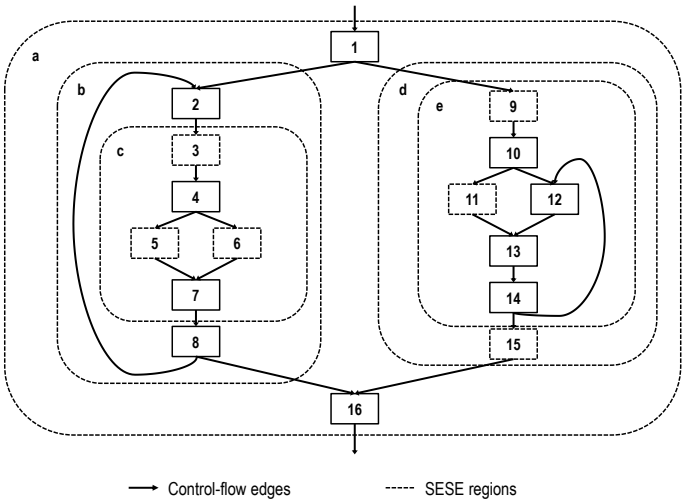


Figure 16: SESE regions: This example shows a program's control flow graph and lists the SESE regions (enclosed by dotted lines). The exit points of the SESE regions form t-points.

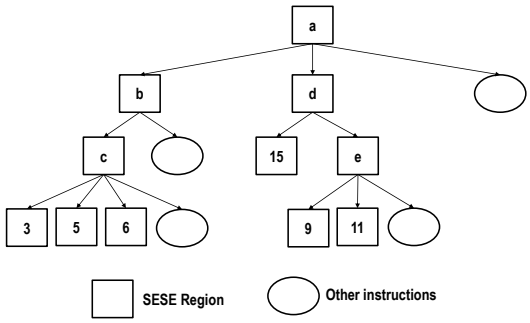


Figure 17: Modified program structure tree (PST) for the CFG shown in Figure 16.

5.1.2.3 Methodical Resiliency Analysis

The program structure tree or PST [22] that organizes the SESE regions can provide a structure to the resiliency profile of the entire program. We plan to extend the PST by adding a new leaf node type to it such that the resiliency of the entire program can be analyzed (by including non-SESE regions). Every leaf node in this modified PST will hold a resiliency summary of the program region that it represents (a SESE or non-SESE region). These summaries, which outline the detection, masking, and SDC rates, can then be combined with each other to obtain the summaries for every node in the tree. An example of this modified PST is shown in Figure 17.

The information in the root node in this PST would represent the resiliency summary of the entire program. Resiliency summaries of the sub-blocks can be obtained by navigating the PST downwards (a top-down approach). This can be helpful in better identifying (and perhaps ranking) the program sections that require attention. The bottom-up approach, on the other hand, can help in identifying the program locations where low-cost detectors can be placed (i.e., finding program locations where potentially many errors can propagate to few variables). Since the values that are alive at SESE region exit points are potentially limited, they may be optimal points for the detector placement.

This structured representation of the resiliency profile of the application may have other advantages, apart from the ones mentioned above. We plan to explore this further.

5.2 Long-term future directions

I also plan to address the following challenges if time permits.

5.2.1 Enhancing the program-level detectors

The detectors developed in Section 4.1 lowered the SDC rate significantly. However, these detectors can further be enhanced in the following ways:

- Expanding the set of applications: Evaluating these detectors on more applications can further strengthen our confidence. Studying more applications can provide us the opportunity to develop new (application-specific) detectors if the current set of detectors do not provide high coverage. Ideally we would like to develop a set of detectors (both lossless and lossy with their coverage statistics) for the software designers/architects to select from.

- Lower-level fault models: The goal is to develop a set of detectors that are independent of the fault model. Hence, evaluating these detectors on micro-architecture fault model (or other low-level fault models) would be interesting.

  Moreover, Relyzer was developed for single-bit-flips (or soft-errors) in instruction-level architectural registers. Extending Relyzer to microarchitecture- or gate-level fault models (which can potentially show up as multi-bit or multi-value errors at architecture-level) is interesting future direction.

- Accounting for application-level error tolerance: It has been shown that a large class of applications are inherently error tolerant [25, 41]. Utilizing this observation and tailoring detectors' placement would be interesting.

- Automatic development and placement of the detectors: Currently the process of identifying and placing a detector for an SDC-prone code section is mostly manual. Developing techniques (or compiler passes) for automatic identification and placement of these program-level detectors can significantly lower the programmer's effort.

- Providing feedback to programmers: Developing detectors for all SDC-prone sections may require programmer's feedback in some applications. Hence a framework that provides inputs to the programmer about the SDC-vulnerable code sections would be valuable. We envision such a framework to provide effective means to develop and place application-specific detectors with low-effort.

5.2.2 Fault recovery

An important piece of any resiliency scheme is fault recovery. The existing SWAT system relies on full system checkpointing schemes like SafetyNet [49] and ReVive [51] for recovery. Recently, it has been shown that the I/O activity can limit the tolerable recovery window to 100K instructions as the execution overheads increase significantly with larger checkpoint windows [41].

This dictates the guarantees that a detection scheme should provide for detection latency. In our work, however, we did not evaluate the latency provided our program-level detectors (from Section 4.1). Hence, measuring these detection latencies and tailoring our detectors such that they convert SDCs to detections within the recoverable window is important future work.

5.2.3 Relyzer for power

A significant fraction of soft-errors get masked at circuit, gate, microarchitecture, architecture, and application level producing correct application output [39, 23, 24, 41]. For example, over >90% microarchitecture-level single-bit soft faults are masked at architecture-level and >30% of instruction-level architecture register faults are masked at application-level. (Li et al.[23] also showed that >30% gate-level permanent faults are masked at application-level.) This indicates that the current hardware, optimized for performance, is being significantly under-utilized in space and time.

Relyzer may be utilized to solve this problem because it can identify all application-level fault sites that potentially get masked. Since single-bit faults in these sites do not have any impact on the application output, we may be able to dynamically drop the computations (in hardware or software) that create these values. A technique that drops dynamically dead instructions [12] from the execution has been proven beneficial. Investigating how Relyzer can assist in increasing the power characteristics of current systems (by potentially finding all application-level dynamically dead bit-locations) is interesting.

6 Conclusion

With technology scaling, the hardware reliability problem is becoming increasingly challenging for a wide class of systems, motivating low-cost reliability solutions. Software-level symptom detection techniques have emerged as low-cost and effective solutions with low Silent Data Corruption (SDC) rates. However, for some cases, the SDC rate is still non-negligible. Hence, techniques to identify program locations that produce SDCs and convert them to detections at low-cost are needed to eliminate or significantly lower the user-visible SDC rate.

This thesis proposes a mechanism to identify and understand the application locations that cause SDCs. It then develops program-level error detectors to convert SDCs to detections at low-cost.

This work developed Relyzer, a technique to systematically analyze all fault injection sites in an application for transient faults. It employs a set of novel fault pruning techniques that dramatically reduce the number

of faults (application sites) that require thorough fault simulations. Relyzer predicts the outcome of several faults, eliminating the need for thorough fault injection experiments for them. It then exploits the fact that several application fault sites are impacted in a similar way by certain hardware faults, and develops heuristics to identify such application-level fault equivalence. Relyzer employs a series of static and dynamic techniques to categorize equivalence classes of faults, such that only one pilot fault from an equivalence class needs to be thoroughly studied through fault injection experiment. Through these techniques, we show that Relyzer prunes the set of faults by 99.78% across the twelve studied applications. With fault injections in the remaining fault sites, Relyzer can identify all SDC causing instructions.

Utilizing Relyzer, we identified SDC-causing application instructions and understood program-level properties around a large fraction of these SDC-causing instructions. This analysis facilitated the development of low-cost program-level error detectors. We find that these detectors can to convert an average of 84% of the SDCs to detections across our applications, at an average execution overhead of 10%. Compared to instruction-level redundancy alone, our program-level detectors (with instruction-level redundancy as backup) show, on average, significantly lower execution overheads at all SDC coverage targets of interest; e.g., 19% vs. 43% for 99% SDC coverage. Thus, the program-level detectors, owing to their lower cost and efficacy in detecting SDCs, provide practical and flexible choice points in the performance vs. reliability trade-off curve.

Future directions include developing simple program-level resiliency metrics that can identify SDC-causing instructions with low profiling effort and developing a fast and systematic fault simulation framework that can significantly lower Relyzer's complete resiliency evaluation time.

References

[1] Bit Twiddle Hacks. Website. `http://graphics.stanford.edu/~seander/bithacks.html`.

[2] Final Report of Inter-Agency Workshop on HPC Resilience at Extreme Scale. Website. `http://institute.lanl.gov/resilience/docs/Inter-AgencyResilienceReport.pdf`.

[3] *Intel 64 and IA-32 Architectures Software Developer Manuals*.

[4] Solaris 64-bit Developer's Guide. Website. `http://docs.oracle.com/cd/E19253-01/816-5138/advanced-2/index.html`.

[5] *International Technology Roadmap for Semiconductors*, 2009.

[6] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *International Symposium on Microarchitecture*, 1998.

[7] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Taghaferri. Data criticality estimation in software applications. In *International Test Conference*, 2003.

[8] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. In *International Conference on Dependable Systems and Networks*, 2005.

[9] C. Bienia and K. Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proc. of 5th Workshop on Modeling, Benchmarking and Simulation*, 2009.

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

[11] F. Bower, D. Sorin, and S. Ozev. Online Diagnosis of Hard Faults in Microprocessors. *ACM Transactions on Architecture and Code Optimization*, 4(2), 2007.

[12] J. A. Butts and G. Sohi. Dynamic Dead-Instruction Detection and Elimination. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

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

[14] M. Dimitrov and H. Zhou. Unified Architectural Support for Soft-Error Protection or Software Bug Detection. In *International Conference on Parallel Archtectures and Compilation Techniques*, 2007.

[15] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[16] O. Goloubeva et al. Soft-Error Detection Using Control Flow Assertions. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003.

[17] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *International Symposium on Computer Architecture*, 2003.

[18] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost Program-level Detectors for Reducing Silent Data Corruptions. In *International Conference on Dependable Systems and Networks*, 2012.

[19] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[20] S. K. S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *International Symposium on Microarchitecture*, 2009.

[21] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006.

[22] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. *SIGPLAN Not.*, 29:171–185, June 1994.

[23] M.-L. Li, P. Ramachandran, R. U. Karpuzcu, S. K. S. Hari, and S. V. Adve. Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults. In *International Symposium on High Performance Computer Architecture*, 2009.

[24] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[25] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *International Symposium on High Performance Computer Architecture*, 2007.

[26] Y. Liu and S. Stoller. Loop optimization for aggregate array computations. In *Proceedings of International Conference on Computer Languages*, pages 262 –271, May 1998.

[27] G. Lyle, S. Cheny, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. An End-to-end Approach for the Automatic Derivation of Application-Aware Error Detectors. In *International Conference on Dependable Systems and Networks*, 2009.

[28] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4), 2005.

[29] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *International Symposium on Microarchitecture*, 2007.

[30] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

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

[32] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReVive I/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *International Symposium on High Performance Computer Architecture*, 2006.

[33] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer. An Architectural Framework for Detecting Process Hangs/Crashes. In *European Dependable Computing Conf*, 2005.

[34] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam. Sampling + DMR: Practical and Low-overhead Permanent Fault Detection. In *International Symposium on Computer Architecture*, 2011.

[35] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Application-based metrics for strategic placement of detectors. In *Proc. of Pacific Rim Intl. Symp. on Dependable Computing (PRDC)*, 2005.

[36] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer. Symplfied: Symbolic program-level fault injection and error detection framework. In *International Conference on Dependable Systems and Networks*, 2008.

[37] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In *European Dependable Computing Conference*, 2006.

[38] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin. CrashTest: A Fast High-Fidelity FPGA-based Resiliency Analysis Framework. In *ICCD*, 2008.

[39] A. Pellegrini, R. Smolinski, X. Fu, L. Chen, S. K. S. Hari, J. Jiang, S. V. Adve, T. Austin, and V. Bertacco. CrashTest'ing SWAT: Accurate, Gate-Level Evaluation of Symptom-Based Resiliency Solutions. In *Proceedings of the Conference Design, Automation, and Test in Europe*, 2012.

[40] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee. Perturbation-based Fault Screening. In *International Symposium on High Performance Computer Architecture*, 2007.

[41] P. Ramachandran. *Detecting and Recovering from In-Core Hardware Faults Through Software Anomaly Treatment*. PhD thesis, University of Illinois at Urbana Champaign, 2011.

[42] V. K. Reddy, A. S. Al-zawawi, and E. Rotenberg. Assertion-Based Microarchitecture Design for Improved Fault Tolerance. In *ICCD*, 2006.

[43] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *International Symposium on Computer Architecture*, 2000.

[44] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *Proc. of Intl. Symp. on Code generation and optimization*, Washington, DC, USA, 2005. IEEE Comp. Society.

[45] G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *International Symposium on Computer Architecture*, 2005.

[46] G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-Controlled Fault Tolerance. *ACM Transactions on Architecture and Code Optimization*, 2(4), 2005.

[47] S. Sahoo, M.-L. Li, P. Ramchandran, S. V. Adve, V. Adve, and Y. Zhou. Using Likely Program Invariants to Detect Hardware Errors. In *International Conference on Dependable Systems and Networks*, 2008.

[48] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[49] D. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *International Symposium on Computer Architecture*, 2002.

[50] V. Sridharan and D. R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *International Symposium on High Performance Computer Architecture*, 2009.

[51] M. P. J. Torrellas. ReVive: Cost-Effective Arch Support for Rollback Recovery in Shared-Mem Multiprocessors. In *International Symposium on Computer Architecture*, 2002.

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

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

[54] D. L. Weaver and T. Germond, editors. *The SPARC Arch. Manual*. Prentice Hall, 1994. Version 9.

[55] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, 1995.