© 2013 Siva Kumar Sastry Hari

PRESERVING APPLICATION RELIABILITY ON UNRELIABLE HARDWARE

BY

SIVA KUMAR SASTRY HARI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Sarita V. Adve, Chair, Director of Research
Professor Vikram S. Adve
Associate Professor Valeria Bertacco, University of Michigan
Doctor Helia Naeimi, Intel Labs
Associate Professor Samuel T. King
Professor Rob A. Rutenbar

Abstract

According to Moore's law, technology scaling is continuously providing smaller and faster devices. These scaled devices are, however, becoming increasingly susceptible to in-field hardware failures from sources such as high-energy particle strikes (or soft-errors). This reliability threat is expected to affect a broad computing market motivating the need for very low cost resiliency solutions.

Software anomaly based hardware error detection has emerged as an effective low cost solution. A small fraction of hardware errors, however, escape these anomaly detectors and produce Silent Data Corruptions (or SDCs). Eliminating or significantly lowering the user-visible SDC rate is crucial for software-driven reliability solutions to become practically successful. The goal of this thesis, therefore, is to provide programmers and system designers with tools and techniques to evaluate software-driven resiliency solutions, identify application locations that are susceptible to producing SDCs, and provide application-centric SDC mitigation techniques to achieve significantly lower SDC rates for a given performance (and/or power) budget with low effort.

The first part of this thesis presents an approach called Relyzer that addresses the challenges in identifying virtually all application locations that are susceptible to producing SDCs when subjected to soft errors. Instead of performing expensive error injections on all possible application-level error sites, which is impractical, Relyzer carefully picks a small set of representatives called pilots. It employs novel error site pruning techniques to reduce the number of detailed error injections. The key insight is to show equivalence between application-level error sites, as apposed to only predicting their outcomes. Relyzer uses program structure and dynamic information to show equivalence between error sites from different dynamic instances of the same static instruction or variable. Results show that 99.78% of error sites are pruned across twelve studied workloads, requiring only a few expensive error injections to determine the vulnerability of all application-level error sites.

While performing error injection experiments on the remaining error sites (one at a time) is practical, it still requires significant simulation time. Therefore, Relyzer proposes and employs a gang error simulator called GangES as a performance enhancement technique. GangES bundles multiple error (pilot) simulations and periodically compares simulation states to allow early termination of equivalent ones, saving simulation time that is otherwise needed to run the application to completion and verify the output. GangES attempts to show equivalence between error sites from different static instructions and variables that were not considered by Relyzer earlier. Results show that GangES provides a total error simulation time savings of 51%.

The second part of the thesis employs Relyzer's capability of identifying virtually all SDC-causing program locations for three different purposes.

- Developing SDC-targeted application-centric error detectors is a primary application of Relyzer. To achieve this goal, this thesis employs Relyzer to identify and analyze program properties that appear around most SDC-producing program locations. Exploiting this analysis, it then develops low cost program-level error detectors that are shown to be effective in reducing the reliance on expensive (instruction-level) redundancy for full SDC coverage.

- Architects often over-provision systems for higher resiliency, trading off power or performance, due to lack of efficient techniques that allow such tuning. Relyzer enables tuning for resiliency by identifying virtually all SDC-vulnerable program locations and selectively adding error detectors. Employing Relyzer and the SDC-targeted error detectors (developed as a part of the first application), this thesis obtains practical and flexible points on performance vs. resiliency trade-off curves. For example, for an average SDC reduction of 90% and 99%, the average execution overheads of this approach versus selective redundancy alone are respectively 12% vs. 30% and 19% vs. 43%.

- This thesis also studies (previously proposed) pure program analyses based metrics and some derivatives that do not need error injection experiments as faster alternatives to identify SDC-causing program locations. Although the results are largely negative, they provide evidence that such models are not straightforward to determine and signify the importance of Relyzer.

To my brother and parents

Acknowledgments

Many people have supported and encouraged me in the process of accomplishing my most ambitious goal thus far. I only wish I had the space to list and thank each and every one of them in this small section. I will, however, enumerate a few here.

Six years ago, when I was applying for graduate school, I was extremely fortunate that my adviser, Sarita Adve, believed in me even though I was naïve and knew little about computer architecture research. Throughout the years of working with her, I have acquired a plethora of skills required to conduct research independently. I have been constantly amazed by her optimism, enthusiasm, focus, and tenacity. Her expert guidance, encouragement, and support have helped me grow into an efficient researcher, for which I would remain indebted to her. I would also like to express my deepest gratitude to her for the immeasurable efforts and countless hours she spent supporting my research and helping me attain my professional goals. It was a great honor, privilege, and pleasure to work with Sarita.

I would also like to thank my PhD committee, Vikram Adve, Helia Naeimi, Valeria Bertacco, Sam King, and Rob Rutenbar for their time and valuable feedback. I would like to specially thank Helia for providing me an extraordinary internship experience at Intel Labs and being a great mentor. Research conducted during this internship laid a foundation for my PhD dissertation. I would also like to thank Ishwar Parulkar for providing me an unforgettable internship experience at Sun Microsystems early in my PhD program, which has helped me understand the role of computer architecture research in advancing microprocessor industry.

I would like to mention special thanks to my lab-mates for a lively and healthy research environment. I have learned several skills from Alex and Pradeep that have helped me grow quickly during my early years as a PhD candidate. Countless interaction and collaborations with several

of my colleagues, Hyojin, Rakesh, Byn, Rob, Matt, Lei, Xin, and Radha have made my workplace fun and enjoyable, and I thank them for that.

I would also like to thank the friendly staff in our department, especially Molly, Michelle, Andrea, and Mary Beth, who have made my life much simpler by managing all the complicated administrative tasks from conference room bookings, to travel arrangements, to thesis deposit.

My research at University of Illinois has been supported in part by the Gigascale Systems Research Center (funded under FCRP, an SRC program), the National Science Foundation under Grants NSF CCF 0541383, and CCF 0811693, and an OpenSPARC Center of Excellence at Illinois supported by Sun Microsystems.

My friends at Champaign, in an otherwise monotonous town, have made my six years of graduate life rewarding, enjoyable, and seem short. I thank Sreekanth, Gasak, Nikhil, Sandeep, Silpa, Siva, Shikhar, Raj, Vijay, Meghna, and several others for the fun times and selfless support.

Last but not the least, I would like to thank my parents for having faith in me and unconditionally supporting me in every step I take. No amount of thanks are sufficient for my brother who has been extremely supportive and encouraging throughout. I would also like to thank my extended family members for being a great cheering team and constantly fueling my aspirations.

Table of Contents

List of Tables . . . . **x**

List of Figures . . . . **xi**

Chapter 1 Introduction **1**
1.1 Relyzer: Application resiliency analyzer 3
1.1.1 GangES: Speeding up resiliency analysis 5
1.2 Applications of Relyzer 7
1.2.1 Reducing silent data corruptions 7
1.2.2 Tuning resiliency at low cost 8
1.2.3 Evaluating pure program analyses based metrics to find SDCs 9
1.3 Summary of contributions 9
1.4 Organization 11

Chapter 2 Background: SWAT: SoftWare Anomaly Treatment **13**
2.1 Error detection 14
2.2 Error diagnosis 14
2.3 Error recovery 15
2.4 Detailed evaluation of SWAT detectors 16
2.4.1 Hierarchical error simulation 17
2.4.2 CrashTest'ing SWAT 17
2.4.3 SWAT detection summary 18
2.5 Summary 19

Chapter 3 Relyzer: Application Resiliency Analyzer for Transient Errors **21**
3.1 Error pruning techniques 22
3.1.1 Known-outcome pruning techniques 23
3.1.2 Equivalence-based pruning techniques 25
3.1.3 Implementation 30
3.1.4 Computing SDC rates 31
3.2 Methodology 32
3.2.1 Workloads 32
3.2.2 Error injection framework 32
3.2.3 Pruning techniques 34
3.3 Evaluation 36
3.3.1 Effectiveness of pruning techniques 36

3.3.2 Validation of heuristics-based pruning techniques . . . 41
3.4 Summary . . . 44

Chapter 4 GangES: Reducing Error Simulation Time . . . 46
4.1 A new error simulation framework . . . 46
4.1.1 What state to compare? . . . 47
4.1.2 When to compare executions? . . . 48
4.2 Methodology for GangES . . . 52
4.2.1 Static program structure identification . . . 52
4.2.2 Dynamic error injection and system state comparison . . . 52
4.2.3 Evaluation . . . 55
4.3 Results for GangES . . . 55
4.4 Summary . . . 60

Chapter 5 Low Cost Program-level Detectors and Tuning SDC Reduction . . . 62
5.1 Analyzing SDC-causing program sections and developing program-level detectors . . 64
5.1.1 Incrementalization in loops . . . 64
5.1.2 Registers with long life . . . 67
5.1.3 Application-specific behavior . . . 67
5.1.4 Local computations or registers with short life . . . 70
5.2 Experimental methodology . . . 71
5.2.1 Detectors and overhead evaluations . . . 71
5.2.2 Evaluating the lossy detectors . . . 72
5.2.3 Determining the lowest overhead detectors for a target SDC coverage . . . 72
5.3 Results . . . 73
5.3.1 Sources of SDCs . . . 73
5.3.2 Static overhead of the program-level detectors . . . 75
5.3.3 SDC coverage of the program-level detectors . . . 75
5.3.4 Execution overhead from the program-level detectors . . . 76
5.3.5 SDC coverage vs. execution overheads . . . 77
5.4 Summary . . . 80

Chapter 6 Evaluating Pure Program Analyses Based Metrics to Find SDCs . . 81
6.1 Pure program analyses based metrics for finding SDCs . . . 81
6.2 Evaluation methodology . . . 82
6.3 Results for pure program analyses based metrics . . . 84
6.3.1 Effectiveness of individual metric in predicting SDCs . . . 84
6.3.2 Linear Regression . . . 85
6.4 Summary . . . 89

Chapter 7 Related Work . . . 90
7.1 Resiliency evaluation . . . 90
7.2 Detectors to reduce SDCs . . . 92

Chapter 8 Conclusion and Future Directions **94**
8.1 Summary and conclusions 94
8.2 Limitations and future directions 96
8.2.1 Enhancing the program-level detectors 96
8.2.2 Lower-level error models 97
8.2.3 Developing profile-driven metrics to find SDC-causing application sites 97
8.2.4 Methodical resiliency analysis 98
8.2.5 Exploiting program indexing schemes 98
8.2.6 Error recovery and detection latency 99

References **100**

List of Tables

2.1 Error injection locations. 18

3.1 Applications studied for Relyzer. The number of dynamic instructions, exercised static instructions, and errors pertaining to the optimized versions of the applications. 33
3.2 Effectiveness of Relyzer's pruning on (a) optimized and (b) unoptimized applications. The applications are in increasing order of total number of original errors. 37

4.1 Applications studied for GangES 55

5.1 Extra instructions used for measuring execution overhead 73
5.2 Number of detectors placed in the static application code 74

6.1 Correlation Matrix: Correlation coefficients between *metrics* and *sdc* for different workloads 84
6.2 Linear regression summary. Numbers in brackets correspond to the respective numbers for linear regression on polynomials. 87

List of Figures

1.1 Relyzer overview. Relyzer first lists all application-level error sites (left box). Each horizontal line represents a dynamic instruction. Each red circle is an error site; e.g., a bit of a source register (not all sites for an instruction are shown for simplicity). Sites marked as P are pruned by known-outcome pruning technique. Sites marked with the same digit (from 0 to 5) are classified as the same equivalence class by equivalence-based pruning. Only one site from each equivalence class is chosen for error injection (right box). Error injections in just these few sites enable a complete application resiliency analysis. 4

1.2 Outlining additional error site equivalence that GangES provides over Relyzer. GangES attempts to show equivalence between multiple error simulations from different instances of static instructions with (same or) different variables that are not considered by Relyzer. 6

2.1 SDC rates from permanent and transient errors injected into non-FP units in server, SPEC, and media workloads. Numbers on the top of each bar show the percentage of injected errors that were categorized as SDC-not-tolerated. The low rates show that the SWAT detectors are highly effective in detecting hardware faults. 19

3.1 Error database. The first three fields in the first table store basic information regarding an error site and the static instruction. The bit min and bit max fields indicate the amount of pruning performed by the known-outcome pruning techniques – errors in the bits between bit min and bit max are pruned and their predicted outcome is recorded in the expected outcome field. The equivalent instruction and the erroneous unit ID fields store information for the def-use equivalence pruning technique and the constant-based technique. The information regarding pilots that are obtained by the store- and control-equivalence based pruning techniques is stored in a separate table. It also stores some extra information that can aid the development of low cost detectors. 23

3.2 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 error outcomes. 28

3.3 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. 29
3.4 Effectiveness of the individual pruning techniques for (a) optimized and (b) unoptimized applications. 39
3.5 The percentage of pilots (y-axis) required to provide a desired amount of error coverage (x-axis) for optimized applications. The x-axis shows the percentage of the total initial number of errors that are covered by the corresponding percentage of pilots. This includes the errors 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) errors are shown, along with a curve for the aggregate errors across all applications. 40
3.6 Validation of control- and store- equivalence for integer register (reg) and output latch of address generation unit (agen) errors for optimized applications. The *combined* bars for each application show the prediction rate across all error models and pruning techniques. 41
3.7 Breakdown of outcomes obtained from error injections in the sampled pilots. 44

4.1 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 the comparison points. 51
4.2 Modified program structure tree (PST) for the CFG shown in Figure 4.1. This example shows a hierarchical representation of nested SESE regions. It also adds new leaf nodes to non-terminal SESE regions that contain non-SESE blocks and instructions in these non-SESE blocks are added to the new leaf nodes. 51
4.3 Explaining how errors are simulated in GangES. This figure explains when to start an error simulation, how many error simulation to bundle together, and what state to collect to compare simulations at comparison points. 53
4.4 Effectiveness of GangES in reducing the total wall clock time needed for error simulations and the number of full error simulations. For each application, the bars in Figure (a) show the total wall-clock time needed for simulating all Relyzer-identified errors, GangES framework, and the time spent in simulating errors that need full executions (after GangES). The bars in Figure (b) show the fraction of error simulations that GangES identifies as detections, that were *saved* from full execution (i.e., terminated early due to a state match with another execution), and that needed full simulation (*need full*). 56
4.5 The SESE exit where the *saved* simulations (from Figure 4.4(b)) are equalized. As an example, 94% of saved simulations are shown equivalent to some other simulation at the first SESE exit. 58
4.6 Distance of the successful comparison point (categorized by SESE region exit number) from the point of error injection. 58

4.7 Categorizing the errors that need full simulations based on whether register state, memory state, or both mismatched during comparisons or no comparison was made before timeout condition was met. 59
4.8 Amount of time needed by GangES and error simulations that need full executions when all processor registers were compared at SESE exits (vs. comparing just the live registers at these exit points) is shown here. 60

5.1 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 5.2.) 63
5.2 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. . . . 66
5.3 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 errors can be detected. . 69
5.4 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. 69
5.5 Baseline absolute SDC rates. These absolute rates are higher than previously reported for symptom-based detectors [28, 46], largely because of the different error models used. 74
5.6 Contribution of code patterns from Section 5.1 to SDCs. 75
5.7 SDC coverage obtained by our program-level detectors, separated into coverage from the lossless and lossy detectors. 77
5.8 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. 77
5.9 SDC coverage vs. execution overhead for each application for different classes of detectors. 78

6.1 SDC reduction vs. execution overhead curves. The X axis for the above graphs plots *% execution overhead* (in terms of increase in dynamic instructions) and the Y axis represents *% SDC covered* 86

Chapter 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 [2, 6, 11]. This increases the likelihood of a hardware failure in the field. Therefore, future systems must deploy low cost in-field resiliency solutions to guarantee continuous error-free operation.

Traditional resiliency solutions use heavy amounts of redundancy (in space or time) to detect, diagnose, and recover from hardware errors. Owing to their prohibitive costs, such 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 anomaly based error detection techniques [15, 17, 28, 32, 34, 42, 45, 60] that provide such promising low cost alternatives. These techniques detect only those hardware errors that corrupt software execution by monitoring for anomalous software behavior using simple, low cost monitors. Since error detection occurs at a high level and relatively rarely, diagnosis and recovery can be more complex and have a higher overhead (but within the constraints of mean time to repair and recovery). Despite the simplicity of their detectors, these techniques have demonstrated impressively high detection rates. SWAT (SoftWare Anomaly Treatment) [22, 27, 28, 29, 46, 53], a state-of-the-art resiliency solution that detects both permanent and transient errors and diagnoses at the faulty core/microarchitectural unit granularity, is an example of such a solution.

Unfortunately, some fraction of errors do escape the detection mechanism and silently impact the correctness of the program output. Such errors are called silent data corruptions or SDCs.

SWAT [22, 46] reports an aggregate SDC rate of <0.71% across several (compute-intensive, media, and distributed client-server) workloads for both permanent and transient errors in all microarchitectural units studied except the data-centric FPU. Most of these evaluations were conducted using a microarchitecture level simulator. Similar SDC rates were also observed by SWAT-Sim [26], a hierarchical simulation framework that evaluated SWAT with (more accurate) gate-level error models, and CrashTest'ing SWAT [43], an FPGA-based evaluation of SWAT with an industry strength processor running an unmodified commercial operating system and modern applications.

Errors that result in SDCs produce corrupted application outputs without leaving any trace of failure behind. Hence such a small SDC rate is still a hindrance for approaches like SWAT to become practically successful. It is therefore 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 error 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 error injection campaign for all errors 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 Relyzer: Application resiliency analyzer

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

Most error-injection based evaluation techniques [28, 32, 34] bound the experiment time by studying a randomly selected sample of errors (typically of the order of thousands per application) out of all the errors possible (more than a trillion errors for this study). While these methods may provide statistical guarantees on SDC rates, certain errors 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 errors 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 error sites to determine a minimal set for when transient errors need to be injected.[1] Figure 1.1 briefly summarizes how Relyzer works. It first lists all application sites that can be directly affected by our chosen transient errors.[2] For some of these errors, Relyzer can directly predict their outcome (detection, masking, or SDC) through simple static analysis and dynamic

[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 errors in (architectural) integer registers and in output latches of address generation units.

profiling of the error-free execution. These errors do not need detailed error injection experiments. For the remaining errors, Relyzer primarily uses the insight that errors 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 to capture the notion of "similar" for errors in different types of instructions. Using these heuristics, we categorize application error sites into equivalence classes. We then select a representative, which we call pilot, from each equivalence class and thoroughly study it through a detailed error injection experiment.

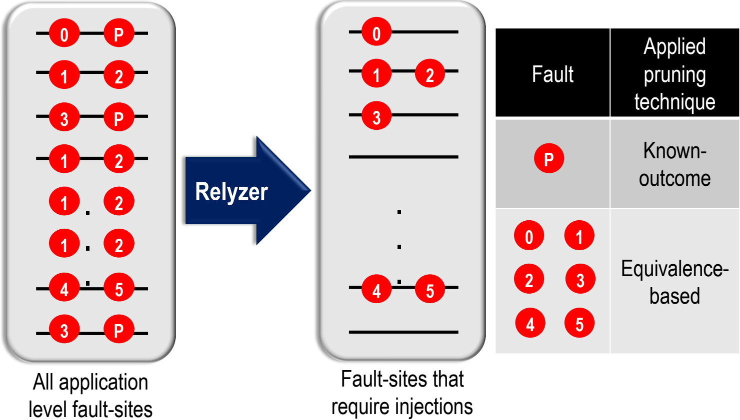


Figure 1.1: Relyzer overview. Relyzer first lists all application-level error sites (left box). Each horizontal line represents a dynamic instruction. Each red circle is an error site; e.g., a bit of a source register (not all sites for an instruction are shown for simplicity). Sites marked as P are pruned by known-outcome pruning technique. Sites marked with the same digit (from 0 to 5) are classified as the same equivalence class by equivalence-based pruning. Only one site from each equivalence class is chosen for error injection (right box). Error injections in just these few sites enable a complete application resiliency analysis.

Our results show that Relyzer significantly reduces the number of errors that require error injection experiments [21]. Relyzer pruned about 99.78% of the total errors 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 error injection outcomes of the representative errors against outcomes of a sample of errors they each represent. Each pruning technique and error 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 errors, averaged across all twelve applications studied.

1.1.1 GangES: Speeding up resiliency analysis

While Relyzer is certainly more practical than comprehensive pure error injections, it still needs significant running time (about 72 hours for a set of eight applications on a cluster of 172 nodes). Most of this time (about 65%) is spent in simulating errors injected in the pilot instructions that result in masking or SDCs (the remaining are detected, and majority of detections occur soon after error injection). Each error injection experiment simulates the entire application from the point where the error is injected and compares the output with the error-free output to categorize the outcome of the error as an SDC or application-level masking.

We propose a mechanism and tool that determines which of Relyzer's pilots will produce the same error outcomes. This technique bundles and runs multiple error simulations (of Relyzer's pilots) and periodically compares the state of these simulations at certain application points – those with identical states will produce the same outcomes and only one in each such set needs to continue and complete its full simulation. The early termination of equivalent simulations can potentially save significant time. We call this tool Gang Error Simulator or *GangES*[3] and employ it to enhance the performance of Relyzer. While Relyzer mostly focused on showing equivalence between error sites from different dynamic instances of the same static instruction or between error sites of instructions with a def and use of the same variable, GangES looks for equivalence between error sites in dynamic instances of *different static instructions using (same or) different variables* (Figure 1.2). GangES is not able to entirely eliminate simulating an equivalent error site; instead, it enables terminating the simulation much faster (for several error injection runs in a gang of error simulations) than it would take to naturally complete the simulation. The two key challenges for GangES are: (i) *when* to compare the state of two error simulations and (ii) *what* state to compare.

A judicious choice for when to compare (which execution points) is critical because the instruction sequences executed by multiple error simulations may temporarily diverge but merge again. A judicious choice of what to compare is also critical because naively comparing all register and memory locations can be prohibitively inefficient.

For when to compare, we select program locations where multiple error simulations would reach

[3] Pronounced as gan-jeez.

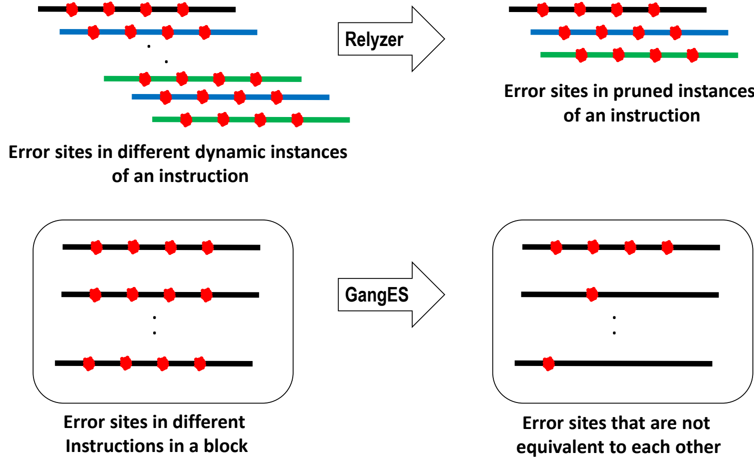


Figure 1.2: Outlining additional error site equivalence that GangES provides over Relyzer. GangES attempts to show equivalence between multiple error simulations from different instances of static instructions with (same or) different variables that are not considered by Relyzer.

even if the error (temporarily) exercised different system events and branch directions. We leverage the previously proposed program structure tree (PST) from the compiler literature for this purpose [25]. A program structure tree organizes an application's control flow graph (CFG) into nested single-entry single-exit (SESE) regions.[4] For every execution that exercises the entry edge of a region, the exit edge will be exercised, irrespective of the data and the error (as long as the error path follows the CFG). Therefore, for every error simulation where errors are injected in a particular SESE region, the corresponding SESE exit edge would be exercised, providing a definite program location to compare execution states. We also perform the state comparison whenever the SESE exit edge of the immediately encapsulating SESE region is reached. This structure also identifies program locations where potentially few program variables are alive, limiting the amount of state to compare. We compare all the (potentially) live registers at these points and any memory locations that have been stored until these points.

Our results show that a large fraction of error simulations do not need full application simulation and checking of the output to identify the outcome of the error injection [23]. Only 47% required full simulation. 94% of the error simulations that were saved by our approach required only 2,850

[4]Formally, 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.

number of instruction executions, averaged across our applications. Overall, we found that GangES replaced Relyzer's error simulation time of approximately 12,600 hours for eight of our workloads with 6,110 hours, providing a wall-clock time savings of 51.5%.

1.2 Applications of Relyzer

We employed Relyzer to achieve the following three objectives: (1) analyzing SDC-causing program locations and adding error detectors to prevent SDCs, (2) tuning application resiliency at low cost allowing system architects to efficiently trade off reliability vs. performance, and (3) evaluating pure (static and dynamic) program analyses based metrics (and a combination of them) that aim to identify SDC-causing program locations with significantly less effort and runtime when compared to Relyzer.

1.2.1 Reducing silent data corruptions

We identified virtually all SDC-causing error sites by performing error injections in the Relyzer-identified sites. Investigating these SDC-causing error 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 [16, 52]. 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 find that our detectors provide an average SDC coverage of 84% with an average execution overhead of 10% [20].

1.2.2 Tuning resiliency at low cost

Hardware and software architects continuously balance performance, power, and resiliency to meet ever increasing design constraints. However, selecting a design choice and evaluating it remains a tedious process. Since efficient tools and techniques to tune resiliency are unavailable, architects often over-provision systems for higher resiliency (trading off performance or power).

Relyzer, for the first time, enables tuning resiliency by identifying virtually all SDC-vulnerable program locations. This allows architects to target any resiliency budget by protecting just the desired set of vulnerable program locations. Utilizing this approach we achieved the ability to obtain a set of near-optimal detectors for any SDC coverage target,[5] allowing us to obtain continuous SDC coverage vs. performance trade-off curves. Using our low cost detectors and selective instruction-level duplication for instructions not covered by our detectors, we present continuous SDC coverage vs. execution overhead trade-off curves for our applications [20]. Compared to similar curves with selective 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. 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.

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

1.2.3 Evaluating pure program analyses based metrics to find SDCs

Relyzer makes it possible to list SDC-causing instructions but it requires detailed dynamic profiles of applications (majority of which are input specific). This requirement may hinder its widespread use. Alternatively, using simple (static and dynamic) pure program analyses based metrics to identify program locations that are susceptible to producing SDCs is an attractive approach. These techniques are much faster than Relyzer (and statistical error injection based techniques), but their accuracy has not been previously validated.

Relyzer, for the first time, enables determining the accuracy of previously proposed pure program analyses based techniques (because it provides error outcomes for virtually all instructions). We studied and evaluated the approach proposed by [41] and some derivatives.[6] Specifically, the metrics we evaluated are cumulative and average *fanout* and *lifetime* and dynamic instruction count, which are defined for every static instruction as described in Section 6.1. We also evaluate combinations of these metrics using linear models based on regression techniques that use these metrics to predict SDCs produced by the instructions.

Our results were largely negative, meaning that we could not identify a common metric that finds SDC-producing instructions with high accuracy. It is possible that other pure program analyses techniques provide better results, but a comprehensive study is beyond the scope of this thesis. We believe our results here provide evidence that such models are not straightforward to determine and signify the importance of Relyzer.

1.3 Summary of contributions

- **Relyzer: Application resiliency analyzer [21]:** Relyzer systematically analyzes all application error sites to obtain a detailed reliability profile of applications. It employs novel error pruning techniques that reduce the error-injection experiments needed by either predicting their outcomes or showing them equivalent to others. This smart selective error evaluation technique prunes 99.78% of error sites across twelve studied workloads. We further show that only 0.004% represent 99% all error sites (providing a coverage of 99%) and performing error

[6] This effort was led by my colleague Radha Venkatagiri.

injections in these sites is practically viable.[7] 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 error equivalence for application error sites (analogous to that of hardware error equivalence). Validation experiments show that the selected errors represent the rest of the errors with an average accuracy of 96% for the studied error models and applications.

  - **GangES: A gang error simulator to reduce error simulation time [23]:** We proposed, implemented, and evaluated the effectiveness of GangES as a performance enhancement mechanism for Relyzer. GangES bundles multiple error simulations (of Relyzer pilots) and periodically compares the state of these simulations at certain application points – those with identical states will produce the same outcomes and only one among them needs to complete to identify the outcome (SDC or masking). Our results show that GangES replaced Relyzer's error simulation time of approximately 12,600 hours for eight of our applications with 6,110 hours, providing a wall-clock time savings of 51.5%.

- **Applications of Relyzer**

  - **Understanding SDC-causing program properties and developing cost effective error detectors [20]:** 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. Using this analysis, we developed low cost detectors that are effective in reducing SDCs. These detectors were placed at the end of loops and function calls. They invoke program-level property checks on a few variables that potentially carry errors from a large number of SDC causing instructions. We find that our detectors provide an average SDC coverage of 84% with an average execution overhead of 10%.

[7] With our existing simulation speeds, errors in every 8th bit of a given hardware error site (e.g., architectural register) can be simulated in approximately 21 days on a cluster of 200 cores.

- **Effective SDC coverage vs. execution overhead trade-off curves [20]:** 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 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.

- **Evaluating program analysis based metrics to find SDCs [23]:**

  Using simple (static and dynamic) program properties to identify program locations that are susceptible to producing SDCs (instead of using Relyzer) is an attractive alternative. This approach is much faster, but its accuracy has not been previously validated. Relyzer, for the first time, enables determining the accuracy of (previously proposed) program analysis based techniques. We studied and evaluated multiple metrics such as cumulative and average *fanout* and *lifetime* and dynamic instruction count. We also evaluate combinations of these metrics using linear models based on regression techniques.

  Our results were largely negative, meaning that we could not identify a common metric that finds SDC-producing instructions with high accuracy. Our results provide evidence that such models are not straightforward to determine and signify the importance of Relyzer.

1.4 Organization

The following chapter outlines SWAT. Chapter 3 proposes and evaluates Relyzer. Chapter 4 proposes and evaluates how GangES further reduces Relyzer's application resiliency evaluation time. Chapter 5 explains and evaluates the low cost program-level error detectors we developed to reduce SDCs. It also presents the SDC coverage vs. performance curve we obtained to tune resiliency at

low cost.[8] Chapter 6 applies Relyzer to evaluate simple (static and dynamic) program properties that aim to identify SDC-causing program locations with little effort and runtime when compared to Relyzer. Finally, Chapter 8 summarizes the work and outlines future directions.

[8] Most of the text in Chapters 3 and 5 is taken from the original publications [21] and [20] respectively.

Chapter 2

Background: SWAT: SoftWare Anomaly Treatment

This chapter presents a software anomaly based hardware resiliency solution called SWAT, which forms a baseline for the remaining of the techniques presented in this thesis. SWAT [28, 27, 26, 22, 43] deploys software-centric hardware error detection, diagnosis, and recovery techniques for treating in-field hardware failures and providing application resiliency. SWAT's detailed evaluations showed the effectiveness of the anomaly based error detectors in detecting both permanent and transient errors, which are also outlined in this chapter. Towards the end, a summary of SWAT's results is also presented which shows the observed SDC rates.[1] Readers familiar with anomaly based error detection techniques can skip this chapter.

Two high level observations that drive SWAT work are: (1) any hardware resiliency solution should handle only those errors that affect software execution and (2) despite the growing reliability threat, error-free operation remains the common case and must be optimized. Hence, SWAT detects hardware errors by watching for anomalous software behavior using zero to low overhead hardware and software monitors. This error detection mechanism is largely oblivious to the underlying hardware error mechanism, treating hardware errors analogous to software bugs and potentially leveraging solutions for software reliability to amortize overhead. Given that error detection occurs at a high level, diagnosis and recovery are more complex. However, since diagnosis and recovery are invoked only in the relatively rare event of an error, a higher overhead is acceptable (but within the constraints of mean time to recovery).

[1] I played an important role in developing several (but not all) techniques presented in this chapter, but the contents of this chapter are largely presented as background and are not among the primary contributions of this thesis.

2.1 Error detection

The SWAT work proposed hardware error detection that employs very low cost monitors to detect anomalous software behavior with very little hardware or software support [28, 46]. Speficially, SWAT deploys detectors for *fatal traps*, which are not thrown in normal execution and are indicators of anomalous software behavior (e.g., division by zero), *hangs*, *high-OS*, which detect abnormally high amount of OS activitiy (more than 50,000 instructions spent contiguously) except for system calls and interrupts, and *out-of-bounds* accesses to addresses that are outside legal boundaries. Results showed that these detectors achieve an aggregate SDC rate of $<0.71\%$ across several (compute-intensive, media, and distributed client-server) workloads for both permanent and transient errors in all microarchitectural units studied except the data-centric FPU [46, 22]. Subsequently, the iSWAT framework proposed using mined likely program invariants from the application to augment these hardware-only detectors [53]. Although these invariants further reduce the SDC rate, this thesis does not explore these detectors as they required changes to the application binary and significant runtime in capturing the invariants.

2.2 Error diagnosis

After an error detection, control is transferred to SWAT firmware that initiates diagnosis. The diagnosis mechanism identifies whether the source of the error is a software bug, transient or permanent hardware error, or a false positive (from iSWAT and a heuristic detector). In the case of a permanent error, it identifies the faulty core and the faulty microarchitectural component for repair or reconfiguration.

A novel diagnosis algorithm called mSWAT was developed to identify the faulty core. mSWAT deterministically replays (and records) the anomaly activating trace of each thread (in a multi-threaded application) in isolation and looks for divergence among the two (original and replayed) executions [22]. If no divergence is observed then the source of the error is diagnosed as either a deterministic software bug or an iSWAT false positive and an appropriate software layer is notified. On an event of a divergence, mSWAT performs a third replay of the divergent thread and compares

its execution with the earlier recorded execution. This comparison leads us to the diagnosis decision, identification of the faulty core. To reduce the hardware overhead, mSWAT emulates the isolated deterministic replays (in software through firmware support). It also implements an iterative algorithm that repeatedly records and replays executions for 100K instruction, limiting the record storage size. Overall, mSWAT implemented emulated Triple-Modular Redundancy (TMR) to diagnose the faulty core with zero performance overhead in error-free cases. Our results show that this technique achieves high diagnosability of over 95% with low diagnosis latency (98% diagnosed within 10 million cycles) and low hardware support.[2]

Based on the reconfigurable granularity present in the system, it may be desirable to diagnose the faulty microarchitectural component. Trace based fault diagnosis (TBFD) achieves this goal with the help of another error-free core, availability of which is already known through mSWAT [27]. The key insight here is that the execution that generated the software anomaly can be used as a *test trace* to repeatedly activate any errors present in order to incrementally perform diagnosis.

2.3 Error recovery

Since the system state may be corrupted before an error detection, SWAT relies on backward error recovery schemes like checkpointing for system recovery. Recent work has proposed low overhead checkpoint and restore mechanisms that efficiently record the changes since the previous checkpoint in hardware with recovery intervals ranging from hundreds of thousands to millions of cycles and rollback the changes on an error detection to reach a pristine system state [58, 55].

An important, but commonly ignored, aspect of error recovery is the *output commit problem*. Since external outputs cannot be rolled back after commit, they should be released only when they are known to be error-free. These outputs must therefore be buffered for the duration of the recovery interval, potentially impacting error-free performance. Although ReVive-I/O handles output buffering in a modern system with relatively low performance impact [37], it relies on dedicated software to buffer and drain high-level output commands to the devices. While such a software solution avoids hardware overhead, buffering outputs in software has some significant

[2] An earlier version of this technique [22] is presented in my masters thesis [19].

limitations. For example, the software that buffers the outputs is itself is vulnerable to in-core errors, resulting in possibly committing erroneous external outputs. Further, the buffering software needs to be written while accounting for device-specific output semantics, requiring fairly sophisticated modifications for every new class of devices supported.

Therefore, SWAT proposed a solution that employs dedicated hardware to buffer low-level external outputs (stores to I/O space) in a hardware structure until they are verified to be error-free [29, 46, 47]. This solution is device-oblivious and can easily support a variety of devices, providing a practical approach. An evaluation of the error-free overheads shows that at short checkpoint intervals of up to 100K instructions, this technique incurs a performance overhead of $< 5\%$ and an area overhead of $< 2KB$. At a recovery interval of 100K instructions for server workloads, 94% of the injected permanent and transient errors are either masked or recovered without affecting application output, 4% are detected but not recovered, and only 44 (0.25%) of the injected 17,920) errors corrupt the application outputs and result in SDCs. Results also indicate that in the presence of I/O, this high system recovery rate cannot be achieved without output buffering and device recovery.

2.4 Detailed evaluation of SWAT detectors

SWAT has been evaluated using microarchitectural simulators, which simplify many of the low-level design features of a processor. Hence, it is important to evaluate SWAT's detectors on a realistic system to validate the results obtained through less detailed models. An accurate evaluation of SWAT needs to model a processor at gate-level because real faults manifest at the gate level and can impact software behavior differently from microarchitecture-level error models. Such an evaluation environment must have the ability to quickly run thousands of instructions and a full software stack (e.g., OS and application) because these layers can influence error propagation and thus software behavior.

2.4.1 Hierarchical error simulation

SWAT-Sim presents an error injection infrastructure that uses hierarchical simulation to study the system-level manifestations of permanent (and transient) gate-level errors [26]. To achieve speed close to a microarchitectural simulator and minimize overhead, SWAT-Sim only simulates the component of interest (the faulty component) at gate-level by invoking a gate-level simulation of the component *on-demand*. SWAT-Sim observes a small average performance overhead of under 3x for the components it simulated, when compared to pure microarchitectural simulations.

SWAT was evaluated using SWAT-Sim by studying the system-level manifestations of errors injected in a Decoder, an Arithmetic and Logic Unit (ALU), and an Address Generation Unit (AGEN) of a superscalar processor. Results indicate that SWAT detectors are effective in detecting permanent hardware errors even for gate-level error models. This evaluation approach, however, is complex to implement because it has to interface both microarchitectural and gate-level models of each component and has been proposed for only a limited number of components.

2.4.2 CrashTest'ing SWAT

An approach to achieve higher accuracy, speed, and processor coverage is to use reconfigurable hardware, such as Field- Programmable Gate Arrays (FPGAs), to emulate faults in digital designs. The CrashTest'ing SWAT project built a platform on the OpenSPARC project [4], augmented with SWAT detectors and the CrashTest infrastructure [44] to allow for error injection experiments with detailed error models, thereby achieving performance or accuracy. This platform (OpenSPARC + SWAT detectors + CrashTest on an FPGA) allows accurate gate-level evaluation of hardware errors across the entire processor design and runs complete applications on top of a full OS and hypervisor [43]. This platform was used to evaluate the effectiveness of SWAT's detectors with a total of 30,800 stuck-at and 20,830 path-delay error injection experiments across five SPECInt 2000 benchmarks. Overall, these experiments validate the results previously reported by software-based simulations of SWAT, but also reveal some interesting differences: **(1)** a high error masking rate (62.56% of experiments); **(2)** silent data corruptions (0.79% of experiments) are concentrated within a handful of functional units in the processor data path; **(3)** the range of software anomalies

detected is much wider than recognized in previous evaluations of SWAT.

2.4.3 SWAT detection summary

Results from one of the most comprehensive evaluations of SWAT detectors are summarized in Figure 2.1 [46, 22]. In this study, the following SWAT detectors are employed – Fatal traps, Hangs, Kernel panic, App-abort, High-OS, and Out-of-bounds. These results are obtained by injecting permanent errors (stuck-at-0 and stuck-at-1) and transient errors (bit-flips) in latches of a representative mix of microarchitectural structures in a modern out-of-order processor (Table 2.1) using a microarchitecture level simulator. These errors were injected while simulating 4 server workloads (apache, sshd, squid, and mysql), all 16 SPEC CPU 2000 benchmarks, and 6 multi-threaded media applications (ray tracing, face recognition, mpeg encoder, mpeg decoder, lu, and bodytrack) running OpenSolaris OS. In this study, the High-OS detector was disabled for the server workloads as these daemons predominantly operate in privileged mode. The out-of-bounds detector was not employed for media workloads. A total of about 46,000 error injections were performed such that the error bars in the results (in each bar in Figure 2.1) are <2% at 99% confidence interval.

μarch structure	**Error location**
Instruction decoder	Input latch
Integer ALU	Output latch
Register bus	Bus on reg file write
Physical int reg file	An int physical register
Reorder Buffer (ROB)	Entry's src/dst reg id
Reg Alias Table (RAT)	Log→ phys register map
Address unit (AGEN)	Virtual address output

Table 2.1: Error injection locations.

The outcome of an error injection is categorized as *detected* if a deployed SWAT monitor was fired. For injections where the application outputs were produced, these outputs were compared with the expected (error-free) outputs. If there is no mismatch then the outcome is declared as *masked*. If the output is different from the expected output but is tolerated by the application (such as PSNR >50dB for MpegEnc and MpegDec and lossless compressed output for gzip), the outcome is categorized as *SDC-tolerated* [46, 22]. Outputs that are not tolerated by the applications are categorized as *SDC-not-tolerated*.

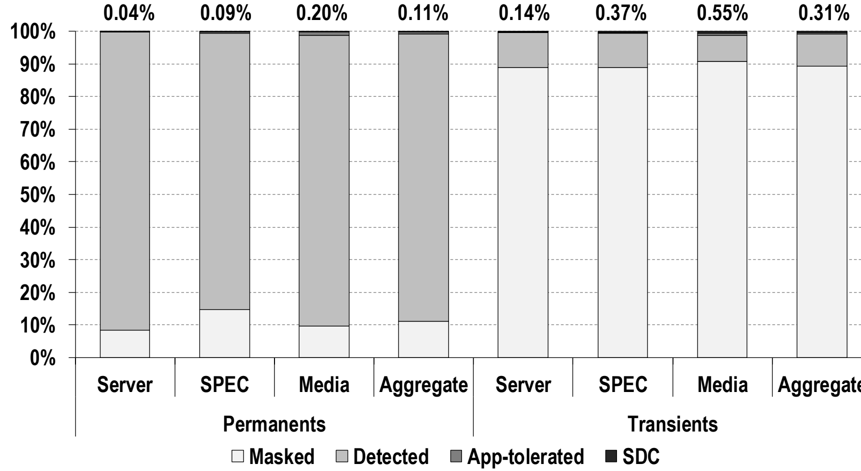


Figure 2.1: SDC rates from permanent and transient errors injected into non-FP units in server, SPEC, and media workloads. Numbers on the top of each bar show the percentage of injected errors that were categorized as SDC-not-tolerated. The low rates show that the SWAT detectors are highly effective in detecting hardware faults.

Figure 2.1 summarizes the results and shows that low cost SWAT detectors achieve low rates for errors categorized as SDC-not-tolerated. Combining SDC-tolerated and SDC-not-tolerated categories, aggregate SDC rates of <0.71% were observed across all the studied (compute-intensive, distributed client-server, and media) workloads for both permanent and transient errors in all microarchitectural units studied except the data-centric FPU.

2.5 Summary

This chapter presented SWAT, a complete resiliency solution that detects, diagnoses, and recovers from in-field hardware errors. SWAT deploys near zero cost anomaly detectors that achieve an aggregate SDC rates of <0.71% across several workloads for both permanent and transient errors in all microarchitectural units studied except the data-centric FPU based on microarchitectur level simulations.

Since errors that produce SDCs escape the placed detectors and corrupt application outputs, even the small SDC rates as reported in Figure 2.1 are a hindrance in deploying SWAT as a low cost resiliency solution. All (prior) SWAT evaluations are performed using statistical error injection

experiments where thousands of error sites were randomly selected for evaluation from trillions of possible sites (number of applications $\times$ application execution points $\times$ number of hardware units $\times$ number of bit locations). Such an evaluation cannot provide insight into all the remaining SDC-prone error sites, which can be crucial in developing a more effective error detection mechanism.

To overcome this challenge and as a significant step towards achieving the goal of full resiliency, this thesis focuses on developing techniques to fully evaluate an application's resiliency for transient errors in the following chapters. Expanding the study to permanent errors requires future research.

Chapter 3

Relyzer: Application Resiliency Analyzer for Transient Errors

As explained in the previous chapter, SWAT presents a low cost resiliency solution that detects transient (and permanent) hardware errors with high efficacy. SWAT reports that only a small fraction of hardware errors remain undetected and corrupt application outputs, producing silent data corruptions or SDCs (Section 2.4.3). Specifically, an aggregate of $<0.71\%$ of studied errors produce SDCs across several (compute-intensive, media, and distributed client-server) workloads based on microarchitectural simulations in all units studied except the data-centric FPU.

Since errors that result in SDCs produce corrupted application output without leaving any trace of failure behind, even such a small SDC rate is still a hindrance for approaches like SWAT to become practically successful. It is therefore 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 error 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 a rigorous error injection campaign for all errors 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.

This chapter presents *Relyzer*, a resiliency analyzer that aims to obtain a complete application resiliency profile for transient hardware errors. It systematically analyzes all (dynamic) applica-

tion error sites to determine a minimal set for when transient errors need to be injected, instead of performing rigorous injection campaign on all errors that can possibly affect program execution. Relyzer achieves this goal by employing a series of novel application level error site pruning techniques that either predict outcomes of errors (without performing detailed error injection experiments) or categorize application error sites into equivalence classes and select only a representative from each equivalence class for error injection experiments.

3.1 Error pruning techniques

Relyzer systematically analyzes all application error sites and carefully selects a small subset for thorough error injection experiments such that it can still estimate the outcomes of all the errors 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 an error. The equivalence-based techniques prune errors by showing them equivalent to others using static and dynamic analyses and/or heuristics.

Relyzer first enumerates all the errors that can impact the application. We require an error-free execution trace for a given application (and input) for this step. Each dynamic instruction instance in the trace forms a potential application error site. At this site, we consider injecting hardware errors that would be exercised by this instruction (one error at a time). Since our focus is on the choice of application error sites and not on exhaustively studying all hardware errors, we choose to study transient errors (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). Errors 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 error sites, Relyzer stores all the error related information in a data structure called the error database, shown in Figure 3.1. In particular, it stores

Fault Site			For known-outcome pruning techniques			For precise equivalence-based pruning techniques		For store- and control-equivalence pruning
Static Instruction (PC)	Faulty unit ID	Number of dynamic instances	Bit min	Bit max	Expected outcome	Equivalent instruction (PC)	Faulty unit ID	List of pilots

Pilot ID	Faulty unit ID and bit location	Population	Outcome	Extra information

Figure 3.1: Error database. The first three fields in the first table store basic information regarding an error site and the static instruction. The bit min and bit max fields indicate the amount of pruning performed by the known-outcome pruning techniques – errors in the bits between bit min and bit max are pruned and their predicted outcome is recorded in the expected outcome field. The equivalent instruction and the erroneous unit ID fields store information for the def-use equivalence pruning technique and the constant-based technique. The information regarding pilots that are obtained by the store- and control-equivalence based pruning techniques is stored in a separate table. It also stores some extra information that can aid the development of low cost detectors.

the identity of the static instruction (program counter), the hardware error sites that can affect the instruction (e.g., names of registers), and the number of dynamic instances of the instruction. The rest of the fields in Figure 3.1 apply to specific pruning techniques and are described with those techniques.

Relyzer next applies the error pruning techniques on this initial set of errors. While the error 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 error database (Figure 3.1). Additionally, some dynamic information to assist the later development of low cost detectors can also be logged, but we leave the exploration of such information and detectors to future work.

3.1.1 Known-outcome pruning techniques

Bounding addresses: Transient hardware errors 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 errors, application aborts, and kernel panic).

SWAT employs detectors specifically to detect such scenarios within recoverable latencies (e.g., out-of-bounds detectors [46]). We do not need injection experiments to identify the outcome of most such errors 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 errors in addresses that make accesses cross the global-heap boundary.

Once we identify the range of the valid addresses, we prune errors that allow a memory instruction to access an invalid address (e.g., errors in high order bits of the address when the error-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, an error 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 errors).

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 errors 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 error 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 errors models, e.g., errors 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 errors in these bits get masked. Currently we apply this technique only on logical shift operations, where the shift count is a constant. We prune errors 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.

Information for error database: Known-outcome based techniques prune errors by declaring them as detected or masked. Hence, very little information is needed to be recorded in the error database. We record the range of the bits that are pruned in the bit min and bit max field along with the estimated outcome in the expected outcome field of the error database (Figure 3.1).

3.1.2 Equivalence-based pruning techniques

This class of pruning techniques eliminates errors that are equivalent to each other from the initial set of error sites and retains only the representative errors (pilots) for thorough error 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.

Precise techniques

Def-use analysis: A register definition is created whenever a register is used as a destination operand in an instruction. Errors in the definition of a register have similar behavior to that of errors in the first use of this definition. Therefore, we prune out errors in the definition and retain errors in the first use. Note that this technique prunes errors only in the definition and not in the

uses. There can be multiple uses of a definition, and errors in different uses may have different error propagation. Whenever a definition is pruned, we record the information of the first use at the definition such that the outcomes of the errors 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 errors 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 error database (i.e., associating errors 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 errors 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 errors from the operands of instructions that use constants. For such instructions, the effect of an error in the source register (non-constant) operand can be studied directly in the destination operand; therefore, we can prune errors from such source operands. This pruning technique is currently limited to only those logical operations where a single-bit error in the source operand propagates as a single-bit error in the destination (e.g., logical `xor`). This technique provides negligible benefits for optimized applications. Hence we report it only for the unoptimized applications.

Information for error database: The entries of the error sites that are pruned by this technique record the information of the error sites that now represent them. The pruned error site records the identification of the representative error site, i.e., the program counter of the instruction and the erroneous unit id from Figure 3.1. For example, a definition that is pruned by the def-use analysis records the program counter of the instruction where the first use is found along with the erroneous unit id of the first use.

Heuristic-based techniques

Control-equivalence: This heuristic pruning technique uses the observation that errors propagating through similar code sequences are likely to behave similarly. It also uses the observation that a majority of the errors appear in code sequences that are executed many times. Consider a static instruction I with many dynamic instances in the error-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 error-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 errors 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 3.2 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 error-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 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 errors 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

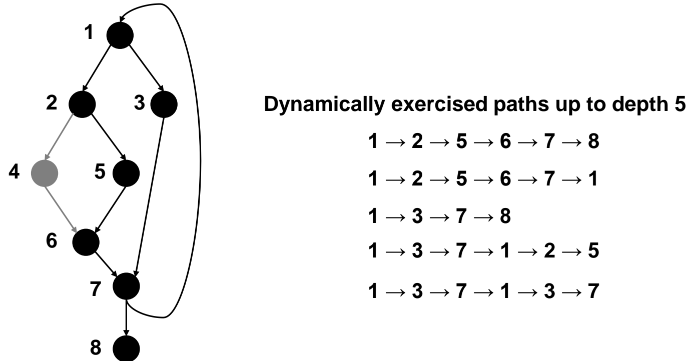


Figure 3.2: Control-equivalence. The figure shows a CFG for a small program starting at basic block 1 and ending at basic block 8. We enumerate all dynamically exercised control paths up to a depth, say 5. Here basic block 4 (showed in grey) never gets exercised. Therefore control flow paths through 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 error outcomes.

the instructions that stores depend on. This is because the propagation of an error 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 error). 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 errors in these instructions and therefore do not consider them any further. We also do not inject errors 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 errors by softening the constraint on evaluating all dynamic occurrences from a specific code section.

Store-equivalence: An error in a store instruction propagates through the loads that read the erroneous 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 errors in store instructions or in instructions that a store depends on. This heuristic captures the error propagation behavior by observing the addresses that a store writes in an error-free execution and recording all read accesses to this address. It treats the errors in stores differently whenever a

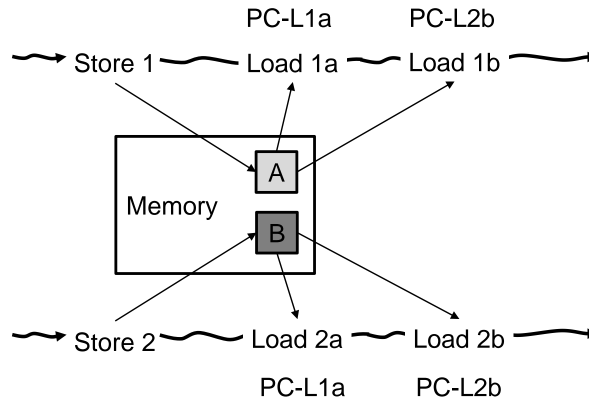


Figure 3.3: Store-equivalence. Store 1 and Store 2 are two store instructions from the same static instruction writing to addresses A and B respectively. Load 1a with program counter PC-L1a and Load 1b with program counter PC-L1b are two load instructions reading the value from address A. Similarly, 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.

different permutations of loads instructions read the stored value.

Figure 3.3 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 errors in these two store instructions will have the same outcomes, we examine all the loads that return the values written by these stores in the error-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 error outcomes and we place them both in the same equivalence class.

Information for error database: We record the information regarding the pilots in the error database (Figure 3.1). In particular, we record the unique pilot id and the size of the population it represents. We can also record additional information such as the control path (sequence of instructions) this pilot represents for control-equivalence based technique and usage pattern of the

stored value (e.g., number of loads and their program counters) in the extra information field. This information may be helpful for the applications of Relyzer (e.g., understanding and protecting SDC causing error sites).

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, however, 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 error database and computes the initial set of errors. The information about the number of dynamic instances of each static instruction is required to compute the size of initial set of errors 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 errors by analyzing the updated error database and terminates.

3.1.4 Computing SDC rates

Once all the pruning techniques are applied, the error injection experiments on the remaining error locations are performed. The SDC rate of the application can now be computed by using the information stored in the error database. Figure 3.1 shows the information we collected during the pruning phase.

To compute the SDC rate for an error site, the outcomes of the pilots corresponding to that error sites are observed. If no pilot produced an SDC, then the SDC rate for that pilot is zero. If one or more pilots produce SDCs, then the sizes of the populations of all these pilots are added and multiplied with the number of remaining bits in for that error site. This value is then divided by the product of number of dynamic occurrences and the initial number of bits (prior to applying pruning techniques) for that error site. The extra recorded information can also be utilized to understand the execution conditions that might have led to an SDC. The SDC rate for the error sites that were pruned by the precise equivalence-based techniques is obtained by simply computing the SDC rate of the error site that represents this it.

The SDC rate at the (static) instruction level can be calculated by simply performing an arithmetic average of the SDC rate at the error site level. A weighted average of the SDC rate at the instruction level with weights being the number of dynamic occurrences of each instruction can be performed to obtain the SDC rate of entire application.

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 [61] architecture, assuming the hardware error models previously described. We selected twelve applications – four each (randomly selected) from the SPLASH-2 [62], PARSEC [10], and SPEC CPU2006 [24] benchmark suites. Table 3.1 provides a brief description of these applications, including the inputs used, the dynamic instruction count, number of exercised static instructions, and the number of errors 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 errors in Table 3.1 pertain to the optimized version of the applications.

3.2.2 Error injection framework

As previously mentioned, the error 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 error injection simulation infrastructure uses a full system simulation environment comprising of Wind River Simics [59] and the GEMS microarchitectural and memory timing simulator [33], 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., [28]) with some modifications.

Our framework allows us to inject errors at any point in the application execution. This is the chosen application error site, as represented by a dynamic instruction in the error-free execution. To inject an error, we start the application and execute it in functional mode (Simics-only) up to 500 cycles before the chosen application error site. Then we start detailed timing simulation

Benchmark Suite	Application	Description	Input	Num. executed instrns. after init & before finish phase		Num. Errors
				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 3.1: Applications studied for Relyzer. The number of dynamic instructions, exercised static instructions, and errors pertaining to the optimized versions of the applications.

(Simics+GEMS) and inject the error when the application error site is reached. Thus, for address generation unit errors, we flip the specified bit in the unit's output latch when it generates the address for the specified dynamic instruction. For integer register errors, 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 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 [46] (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 error-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 an error, there is always an instruction that consumes an erroneous value or uses an erroneous 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 error pruning or are later used by the dynamic technique. Since our error 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.[1] Using this static infrastructure, we traverse the application and create the set of all transient error sites. We then apply the static

[1] 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.

pruning techniques, compute the pruned error set, and collect information for dynamic analyses. The dynamic analyzer profiles the branches, the memory access patterns (for store-equivalence technique), instruction control flow patterns (for control-equivalence technique), etc. We use Wind River Simics [59] to implement these dynamic profilers. Finally, we use the information from both the static and dynamic analyses to generate the final pruned error 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 errors that are pruned (in total and by the individual techniques) and the absolute number of remaining errors (pilots) that must be simulated to determine the resiliency of an application.

Validating pruning techniques

The control- and store-equivalence based error 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 an error in a pilot results in masking the error. Suppose the injection of an analogous (hardware) error 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 error populations represented by the pilots.

To find the exact misprediction rate, ideally, we would run error 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.[2] We then inject transient errors in the pilot and the selected samples to obtain the prediction rate. Ideally, we would inject errors in all bit locations in the appropriate erroneous units for the pilot, but the simulation time would be prohibitive. We instead injected errors 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 errors 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 error 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 error models. In particular, for validating control-equivalence, we performed approximately 1,092,000 and 286,000 injections for integer register and address generation unit error 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 error 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

Overall pruning effectiveness

Tables 3.2(a) and (b) show the overall effectiveness of Relyzer's pruning techniques by presenting

[2]The pilot requires only one error injection experiment to obtain the outcome A. We can formulate the error 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 errors (in billion)	Total pruning	Remaining errors (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 errors (in billion)	Total pruning	Remaining errors (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 3.2: Effectiveness of Relyzer's pruning on (a) optimized and (b) unoptimized applications. The applications are in increasing order of total number of original errors.

the percentage of total errors pruned for the optimized and unoptimized applications respectively. The tables also show the absolute number of total errors and the errors remaining after pruning. The applications are ordered according to the total number of original errors.

For optimized applications, we find that Relyzer prunes an aggregate of 99.78% of all the studied errors across all applications. The total number of errors that need to be simulated reduces from 1.6 trillion to 3.52 billion, a 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,[3] two stores observed a pruning of 20%, bringing down mcf's overall pruning rate. The number of remaining errors in these two stores and the instructions that these stores depend on alone accounted for 83% of the total remaining errors for mcf.

[3]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.

Pruning effectiveness of individual techniques

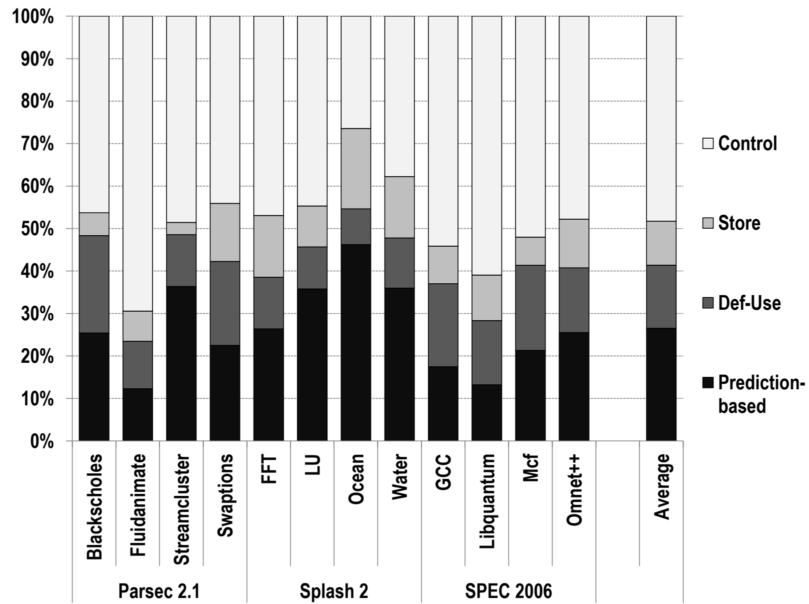
Figures 3.4(a) and (b) show the effectiveness of Relyzer's individual pruning techniques for the optimized and unoptimized applications respectively. The stacks in each bar show the contributions of the individual pruning techniques when applied in the order shown (bottom to top) for all the errors in 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 errors. Def-use analysis prunes 15% of all the errors 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 errors.

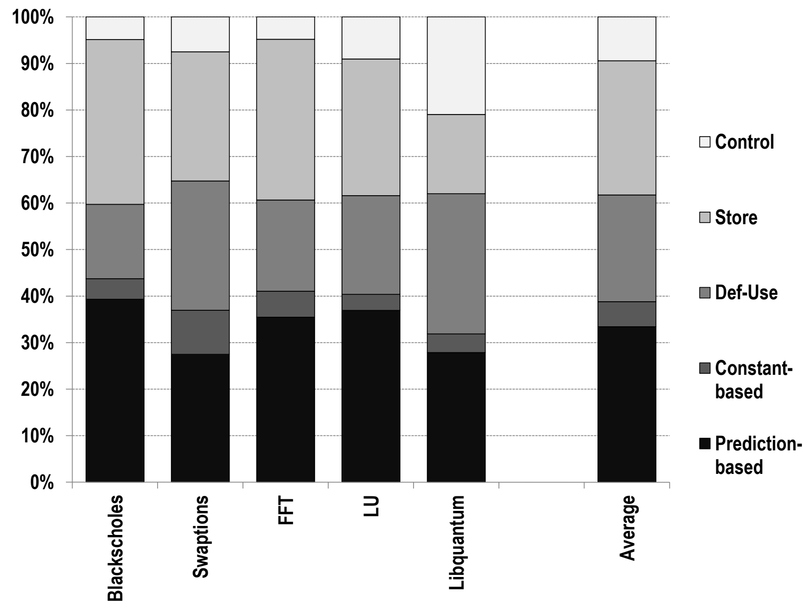
The unoptimized applications show slightly different behavior. First, store-equivalence provides notably more pruning than in the optimized applications. A likely reason is that there are more memory operations in unoptimized codes since they use the stack heavily and the registers poorly. Moreover, 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 errors). We observed that the number of remaining errors increases significantly (by about 100%) when this technique was excluded for the unoptimized applications. However, it had negligible impact on optimized applications. Overall, figure 3.4 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 error propagation.

Trading off simulation time with coverage

Although Relyzer is able to prune errors effectively, there are still a relatively large number of remaining errors that need to be simulated, especially for the longer applications. Relyzer al-



(a) Optimized applications



(b) Unoptimized applications

Figure 3.4: Effectiveness of the individual pruning techniques for (a) optimized and (b) unoptimized applications.

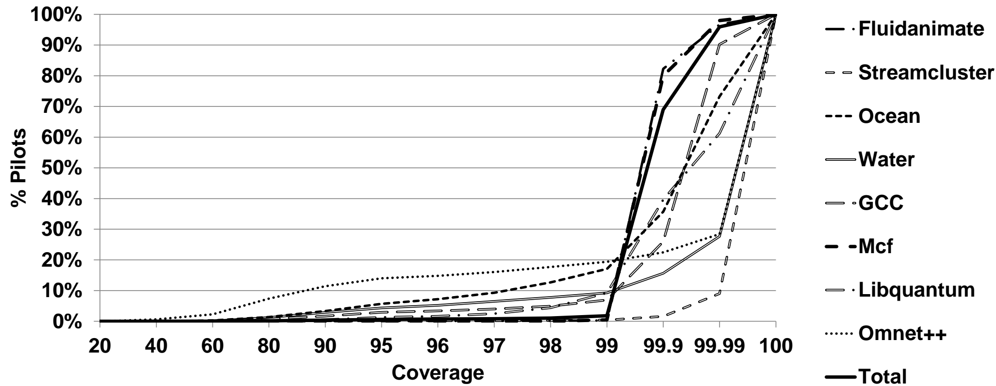


Figure 3.5: The percentage of pilots (y-axis) required to provide a desired amount of error coverage (x-axis) for optimized applications. The x-axis shows the percentage of the total initial number of errors that are covered by the corresponding percentage of pilots. This includes the errors from the 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) errors are shown, along with a curve for the aggregate errors across all applications.

lows a systematic method to trade off simulation time with coverage, revealing sweet spots that dramatically reduce simulation time with modest reduction in coverage.

Figure 3.5 shows the percentage of pilots (y-axis) needed to provide a desired coverage of the errors 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 errors that need simulation. For the full picture, we also plot the data for the total number of errors.

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

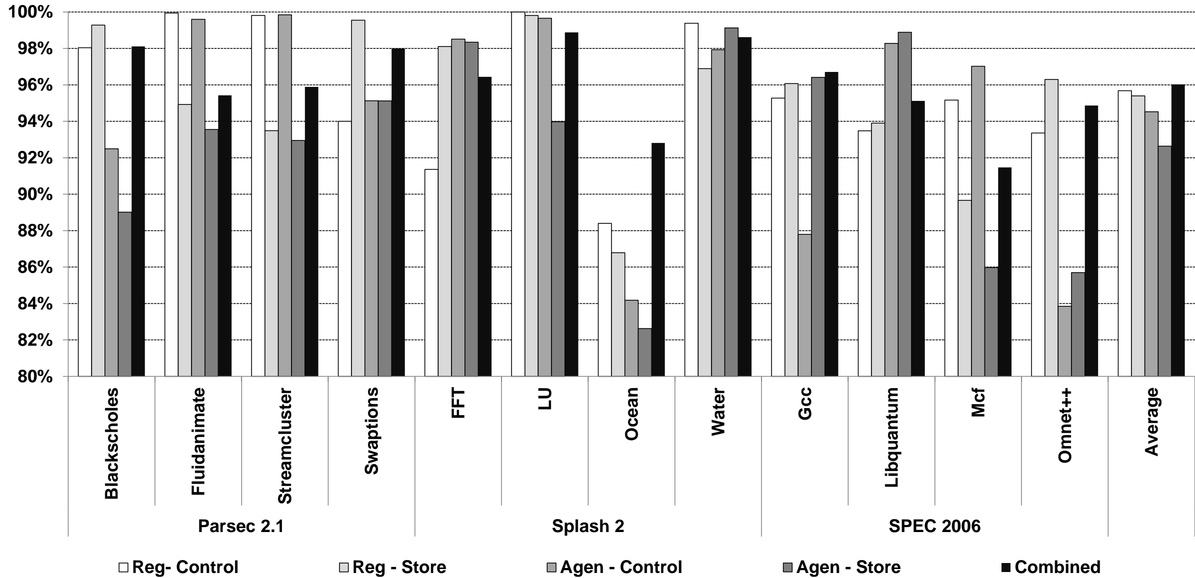


Figure 3.6: Validation of control- and store- equivalence for integer register (reg) and output latch of address generation unit (agen) errors for optimized applications. The *combined* bars for each application show the prediction rate across all error models and pruning techniques.

3.3.2 Validation of heuristics-based pruning techniques

Prediction rate for control- and store-equivalence

We validated the heuristics-based pruning techniques, namely control- and store-equivalence, for the optimized applications as described in Section 3.2.3. Figure 3.6 shows the prediction rate of the pilots for all twelve applications and both the studied hardware error 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 error models after applying all pruning techniques. For each application, the combined bar is the average of the prediction rate of each pruning technique and error model combination, weighted by the fraction of errors pruned by that combination. Specifically, in addition to accounting for control- and store-equivalence, this bar also accounts for errors pruned by def-use pruning (by associating a def-use pruned error's prediction rate with that of its representative errors' rate). It also accounts for known-outcome based pruning, assuming a 100% prediction rate for that technique.

Figure 3.6 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 errors and 94.5% for agen errors. The pilots selected by store-equivalence predict their population's outcomes with an average accuracy of 95.4% for reg errors and 92.6% for agen errors. The figure also shows that for each individual application, Relyzer predicts the outcome across all error 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 errors observe a prediction accuracy of approximately 90% or higher for all applications except Ocean. On the other hand, agen errors 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 errors 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 dependencies 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 errors. The major contributor to the misprediction rate was a load instruction loading values from erroneous addresses. In several cases, it so happened that the erroneous and correct address had the same value; therefore, the error was masked rightaway. On the other hand, sometimes this was not the case, and the error 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 erroneous value. For Blackscholes, we could easily modify our implementation so that during our profiling step (error-free execution), for every potential error in a load address, we check the erroneous address to determine if the value is different. This leads to a known-outcome based technique that can immediately determine if such an error would be masked.

Examining the mispredicted cases in Ocean for agen errors pruned by store-equivalence exposed

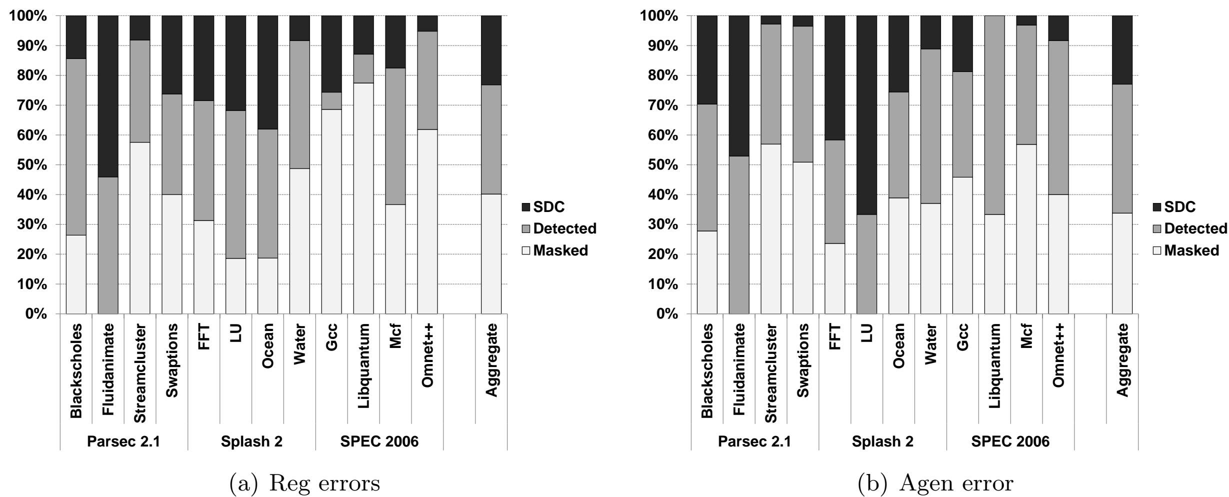
a more difficult limitation of Relyzer. A store instruction with a erroneous address was one of the major sources of the high misprediction rate in Ocean. Such a store corrupts the intended address by not writing the value from the source register and also the erroneous address by writing an unintended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

SDC vs. non-SDC prediction rate

A key application of Relyzer is identifying SDC causing error sites; therefore, we would like to ensure that Relyzer's prediction rate for SDC causing errors is also high. The data in figure 3.6 showed the prediction accuracy for masked and detected errors 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 3.6. We observed that the average (across all applications) prediction rate for control-equivalence for reg and agen errors 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 errors is also a higher 95.9% and 94.9% respectively.

Outcomes for error injections in pilots

We next show that our choice of pilots is not biased towards any specific error outcome. We plotted the outcomes from the error injection experiments of the pilots that were selected for validation (described in Section 3.2.3). Figure 3.7 shows the distribution of the outcomes for reg errors in part (a) and agen errors in part (b). These two figures represent just 3,298 error injection experiments. Each bar in figure 3.7 has between 129 and 320 injections for part (a) and between 27 and 102 for part (b). Even with such small sample sizes, the error outcomes have a significant fraction of SDCs. In aggregate, about 23% of the pilots result in SDCs for both reg and agen errors. This



(a) Reg errors

(b) Agen error

Figure 3.7: Breakdown of outcomes obtained from error injections in the sampled pilots.

result indicates that Relyzer can be effective in finding SDC causing error sites.

3.4 Summary

This chapter presented Relyzer, a technique to systematically analyze all transient error injection sites in an application. Relyzer seeks to identify all SDC causing instruction instances, both to enable quantifying the application's true SDC vulnerability and to motivate low cost application-specific protection mechanisms for the desired SDC-vulnerable cases.

Relyzer employs a set of novel error pruning techniques that dramatically reduce the number of errors (application sites) that require thorough error simulations. Relyzer predicts the outcomes of several errors, eliminating the need for thorough error injection experiments for them. It then exploits an observation that errors in several application error sites have similar behavior at application-level, and develops heuristics to identify such application-level error equivalence. Relyzer employs a series of static and dynamic techniques to categorize error sites into equivalence classes, such that only one representative (pilot) error from an equivalence class needs to be thoroughly studied through error injection experiment. Through these techniques, we show that Relyzer prunes the set of errors by 99.78% across twelve studied applications.

This chapter also evaluates the accuracy of the heuristics-based error pruning techniques by

matching the results from error simulations for the pilots with results from error simulations with samples of the represented error populations. Averaged across all the studied pruning techniques (heuristics- and analysis-based), errors models, and applications, Relyzer correctly determined the outcomes of 96% of the errors. Overall, Relyzer significantly reduces the application-level error sites that require time-consuming simulations, making it feasible to study resiliency of a complete application through a relatively small number of error injection experiments.

Chapter 4

GangES: Reducing Error Simulation Time

Relyzer provides a practical approach to evaluate an entire application's resiliency. It is 2 to 6 orders of magnitude faster than comprehensive error injection. However, Relyzer still requires significant running time (about 4 days for our eight applications on our cluster of 172 nodes). Most of this time (about 75%) is spent on error injections. An error error injection experiment is a time-consuming process because it simulates the entire application from the point where the error is injected and verifies the output by comparing it with the error-free output to categorize the outcome.

This chapter, therefore, proposes a gang error simulation framework called GangES that aims to reduce the number of full transient error injections needed to evaluate the outcomes of Relyzer's pilots. GangES groups and runs multiple error simulations (of Relyzer's pilots) and periodically compares the state of these simulations at certain application points – those with identical states will produce the same outcomes and only one in each such set needs to continue and complete its full simulation. The early termination of multiple errors in a group can potentially save significant time.

4.1 A new error simulation framework

GangES presents a systematic transient hardware error simulation techniques that takes as input a set of errors to be simulated for an application and outputs the outcome for each error (masked, detected, or SDC). Each error in the error set specifies a dynamic instruction instance, a hardware resource used by that instruction instance, and the type of error to be injected in that hardware resource for that instruction instance. In our experiments, we focus on single-bit flips in the integer architectural registers used by the specified instruction instance (one bit flip per simulation). For

error detection, we use detectors similar to those used in Relyzer (Section 5.2). In our experiments, the input set of errors for GangES consists of the errors that Relyzer is not able to prune (i.e., the pilots of all instruction equivalence classes as categorized by Relyzer). Relyzer performs error injections for all of these errors – for those not detected, Relyzer must execute the application to the end and compare the output of the execution with that of the error-free execution to determine masked or SDC outcomes (Figure 1.2).

GangES aims to reduce the overall evaluation time for its input error set by terminating as many error simulations as possible soon after error injection and well before the end of the application (including those simulations that would eventually be masked or SDCs). Our approach is to repeatedly compare execution states of a group or gang of multiple simulations in progress (hence the name Gang Error Simulator). Any simulations in the group that reach identical states will produce the same result and all but one of them can be terminated.

A naive implementation would compare the entire system state (processor and memory state) at every cycle to identify the earliest point in the execution to terminate an error simulation. The entire system state often consists of megabytes to gigabytes of data, comparing which on every cycle can be very expensive in time. Moreover, such comparisons may not identify error masking or equivalence effectively because even one trace of mismatch in temporarily divergent state (due to temporarily divergent control flow) or dead values will result in non-equivalent outcome. Hence, the challenge in developing a time-effective simulation framework for GangES is in identifying what state to compare and when to compare.

4.1.1 What state to compare?

The state to compare at an execution point can be divided into two components – processor register state and memory state. The size of the entire memory state at a given point in the execution is significantly larger than the memory state relevant to the processor. Ideally, we want to identify only the live memory state (the memory locations that will be read in the future before being overwritten) – this is potentially much smaller than the full memory state. However, the live memory state for different erroneous executions and for the error-free execution may be different.

Moreover, identifying the live memory state is known to be a complex problem [35].

Our approach is to compare the memory addresses and data that are *touched* (written) by the multiple error simulations. This significantly reduces the amount of memory state to compare. In this approach, the simulations that are being compared need to start at the same execution point, prior to error injection. Starting the simulation from the beginning of an application can result in an unmanageable number of touched memory addresses. Hence we start the simulation only a few instructions before the error injection point, limiting the number of touched addresses to compare. This also allows us to save the simulation time from the beginning of the application to the point of error injection using a single checkpointed state for the start of the simulation. Another implication of this optimization is that we group errors that need to be injected in closeby instruction instances together for comparison.

For processor register state, our goal is to compare live architectural registers at comparison points. Comparing the entire register state would be fast but it may not be effective in showing error equivalence or masking. The live register state at a given point in a program can be obtained statically. However, an error in an execution may result in a different control flow changing the live state for that execution. Hence, we obtain a conservative live set of registers dynamically by fast forwarding the execution to hundreds to thousands of instructions and removing the registers that are written before being read from the set of all the registers in this fast-forward phase.

4.1.2 When to compare executions?

For when to compare execution states, our approach is to select program locations where all executions would reach even if different system events take place or different branch directions are exercised during error simulations. To select such program locations, we identify single-entry single-exit (SESE) regions from the control flow graph of the application [25]. Formally, a SESE region is defined as 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 [25]. For every execution that exercises the entry edge of a region, the exit edge will be exercised, irrespective of the data and error (assuming control follows

the static CFG[1]). Therefore, for every error simulation where errors are injected in a particular SESE region, the corresponding SESE exit edge would be exercised, providing a definite program location to compare execution states.

Our algorithm to identify the comparison points is inspired by and similar to the SESE regions identification algorithm by Johnson et al. [25]. They provide a linear-time algorithm for finding SESE regions and for building a hierarchical representation of program structure based on SESE regions called the program structure tree (PST). This algorithm works by reducing the problem to that of determining a simple graph property called cycle equivalence: two edges are cycle equivalent in a strongly connected component iff for all cycles C, C contains either both edges or neither edge. They provide a fast, linear-time algorithm based on depth-first search for solving the cycle equivalence problem, thereby finding SESE regions in linear time.

In 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. However, we modify the CFG to potentially obtain more comparison points for a given error site as follows. For a basic block with multiple entry edges,[2] we split it into two blocks such that the first one with multiple entry edges has as few instructions as possible and the 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 the minimal instructions. We then apply Johnson et al.'s algorithm to obtain the SESE regions [25].

For adjacent SESE regions, a and b, where the exit edge of a is the same as the entry edge of b, we obtain c by combining a and b (which is also a SESE region).[3] We then remove a and b and add c to the list of SESE regions. This provides error simulations (in a) with more opportunity to mask the effect of the error or become equivalent to another error injection by executing a few extra instruction (in b). Figure 4.1 shows a CFG and the extracted SESE regions obtained by applying the above mentioned algorithm on it.

Once all the SESE regions are obtained, which are typically nested (see Figure 4.1), they are

[1]If an error results in a branch to a target not specified in the CFG, our technique may be less effective but still correct.

[2]Note that a basic block has a single entry instruction, but there may be multiple edges into the instruction.

[3]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.

organized in a hierarchical representation to obtain a program structure tree (PST) using the algorithm proposed by Johnson et al. A SESE region that immediately contains other regions is considered parent for the containing regions (e.g., a becomes the parent for regions b and d from Figure 4.1). We modified this tree by adding a new leaf node to SESE regions that immediately contain non-SESE blocks. A new leaf node contains instructions that do not belong to any of its sibling SESE regions (SESE regions that are immediately contained in its parent). For example, a new leaf node containing instruction from blocks 1 and 16 is added to a (Figure 4.2). This modified PST is utilized to identify next comparison points during an error simulation. By traversing up the tree from a node where error injection is being considered, exit points of subsequent parent nodes become the comparison points.

In the example shown in Figure 4.1, the checks for masking and equivalence to shorten the simulations for errors in basic block numbered 3 are performed when the exit edge of the current SESE region is exercised and when exit edges of all ancestors of the current SESE region according to the modified PST are exercised (Figure 4.2), i.e., when edges $7 \rightarrow 8$, $8 \rightarrow 16$, and $16 \rightarrow end$ are exercised. Similarly, for simulations for errors in basic block 13 the checks are performed at exit edges $14 \rightarrow 15$, $15 \rightarrow 16$, and $16 \rightarrow end$.

Advantage of (modified) PST: The modified PST (Figure 4.2) adds new leaf nodes that represent non-SESE regions allowing the resiliency of the entire program to be analyzed. Every leaf node in this modified PST can 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.

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

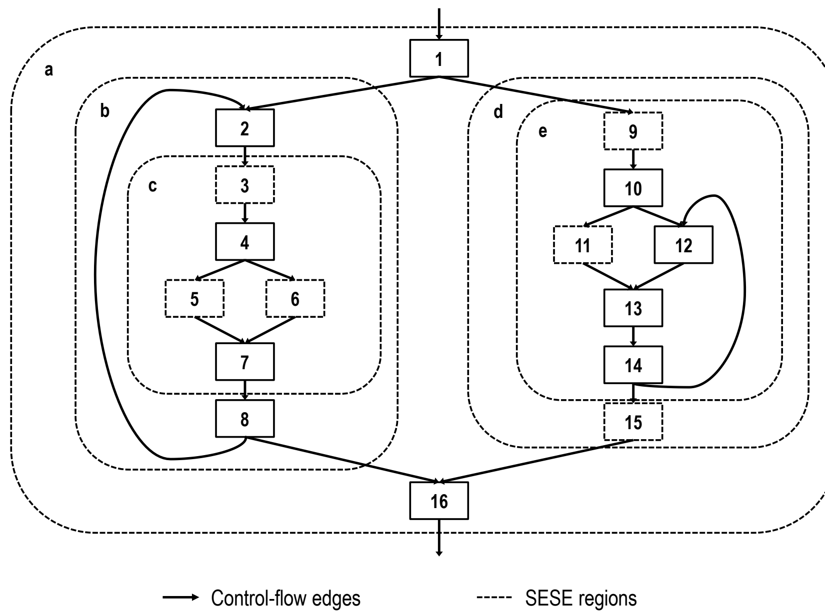


Figure 4.1: SESE regions. This example shows a program's control flow graph and lists the SESE regions (enclosed by dotted lines). The exit points of the SESE regions form the comparison points.

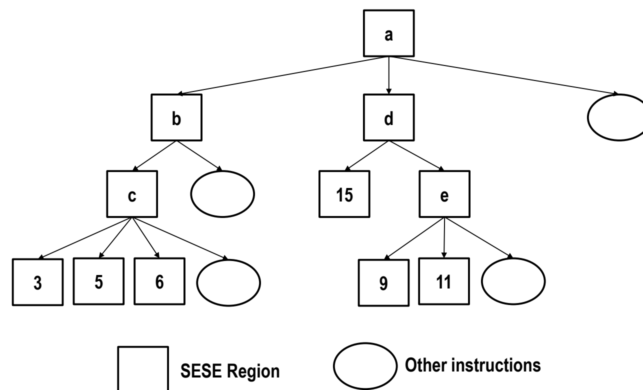


Figure 4.2: Modified program structure tree (PST) for the CFG shown in Figure 4.1. This example shows a hierarchical representation of nested SESE regions. It also adds new leaf nodes to non-terminal SESE regions that contain non-SESE blocks and instructions in these non-SESE blocks are added to the new leaf nodes.

placement.

4.2 Methodology for GangES

Our systematic error simulation framework has two components – (1) static program structure identification and (2) dynamic error injection and state comparison.

The error model we used injects transient errors or single bit flips in architectural integer registers that are accessed by dynamic instructions. Since it is infeasible to inject errors in all dynamic instructions of an application, we first applied Relyzer to significantly reduce the error sites to evaluate but keeping the ability to reason about all application level error sites [21]. However, our approach is applicable to any set of input error sites.

4.2.1 Static program structure identification

We implement our static program analyses at the binary level because our error model studies errors at instruction level. We used the SPARC V9 binary analyzer we developed as described in Section 3.2.3. This tool constructs a control flow graph from the binary and performs basic control flow analyses. We implemented the intra-procedural SESE region identification and PST generation algorithms (Section 4.1.2) in this infrastructure.

4.2.2 Dynamic error injection and system state comparison

Once we identify when to compare execution states, the next steps are to (1) identify when to start an error simulation and how many simulations to group together for efficiency, (2) inject the error, (3) collect state for comparing executions, and (4) compare simulations at the comparison points. Figure 4.3 explains how our technique works with an example. We use Wind River Simics [59] to implement our error injection and simulation state comparison algorithm.

Identifying when to start an error simulation and how many injections to group together: We take several application checkpoints (using Simics' checkpointing feature) at different points in the application. This allows us to start simulations from intermediate execution points(① in Figure 4.3), instead of starting from the beginning of the application for every simulation, saving running time.

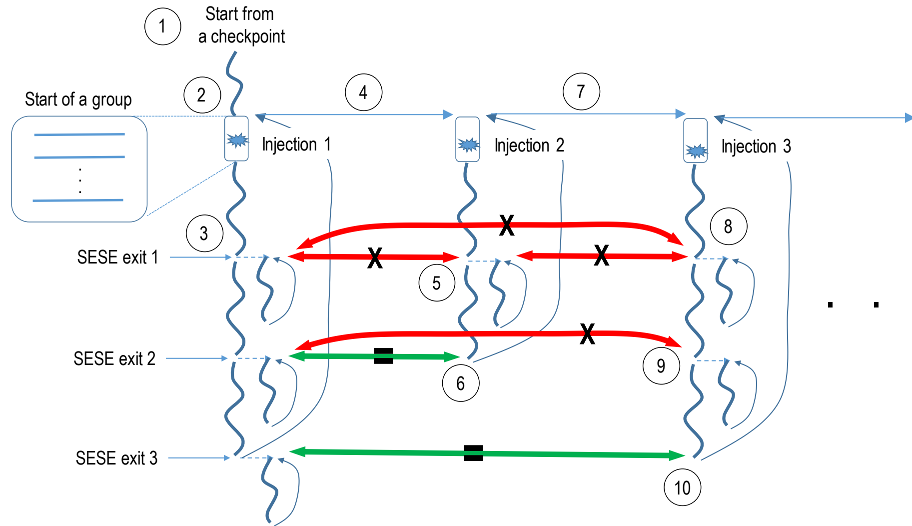


Figure 4.3: Explaining how errors are simulated in GangES. This figure explains when to start an error simulation, how many error simulation to bundle together, and what state to collect to compare simulations at comparison points.

We identify a set of error injection sites such that all the simulations can be started from a single stored checkpoint. Since we collect touched memory addresses from the first injection for comparison purposes for every injection in the group, we restrict the distance between two injections and the number of injections in each group to control the amount of data being collected. In our setup, we only group error sites that are <100,000 instructions apart and restrict the group size to 1,000. We observe an average group size of 327 with these parameters. We did some sensitivity analysis and found that the average group size reduced rapidly to 100 when we lowered the maximum distance from 100,000 to 1,000, and when we increased the maximum group size to 2,000, the average group size increased to only 398.

Error injection: We start execution from an application checkpoint for a group of error injections sites and create a Simics bookmark[4] just before the first injection point in the group (② in Figure 4.3). We inject errors directly into the architecture registers by flipping the bit value at the specified error location, which is a tuple of cycle number, program counter of the instruction that exercises the error, register operand, and bit location.

[4]A bookmark set at a particular point in the execution allows Simics to move the simulation to that point from anywhere in the application, restoring the execution state at that point. This feature allows us to move backwards in an execution.

Collecting execution state for comparing simulations: After an error injection, we continue simulation until a comparison point, the next SESE exit, is reached. We set a breakpoint at the program counter of the instruction that immediately follows the current SESE region's exit edge (③ in Figure 4.3). We also set breakpoints at the ancestor SESE region exits according to the PST (for example, we set breakpoints at the exits of regions *3*, *c*, *b*, and *a* while considering error injection in region *3* from Figure 4.2). Whenever a breakpoint is reached, we remove it to avoid further interrupts in the simulation (which may be caused by regions within loops) and to avoid complicated control logic to match the program location (just the number of instances of a breakpoint activation may not be sufficient because error may allow the program to skip or repeat some loop iterations) for comparing states. At each comparison point, we compare the live register state and touched memory state with other simulations that previously reached this point (example, ⑤, ⑥, and ⑧-⑩ in Figure 4.3). Since different error injection sites (in a group) may belong to different sub-trees in a PST, we ensure that we only compare states when simulations reach the same program location by comparing the program counter of the breaking instruction.

We compare live register state at every comparison point, which we obtain by listing all processor registers, executing the next thousand instructions, and removing the registers that are written before being read (we use the Simics bookmark utility to execute forward and return to the same execution point). We also compare the touched memory addresses and values at a comparison point. We observe memory operations (reads/writes) through a memory module attached to Simics (which is added to the simulation just before the first injection, at ② in Figure 4.3). In the first error simulation, we only collect the touched memory and live register state for future comparisons. We continue this state collection until a fixed number of SESE exits have reached (5 in our setup) or a threshold number of instructions have executed (100,000 in our setup[5]) and then continue to the next error injection (④ in Figure 4.3). Whenever a state match is identified (⑥ and ⑩ in Figure 4.3) we terminate that simulation and declare it as equivalent to a previous simulation and continue to the next error injection (⑦ in Figure 4.3). We also terminate a simulation if an error

[5]We selected the threshold number of instructions such that most bundled error simulations have a memory footprint of <2GB. For a small number of cases that exceed this limit, we adjust the threshold to 100 once this memory limit is reached for the remaining injections in that bundle.

	Application	Description	Input
PARSEC 2.1	Blackscholes	Calculates prices of options with Black-Scholes partial differential equation	sim-large
	Fluidanimate	Simulates an incompressible fluid for interactive animation purposes	sim-small
	Streamcluster	Solves the online clustering problem	sim-small
	Swaptions	Computes prices of a portfolio of swaptions using Monte Carlo simulations	sim-small
SPLASH 2	FFT	1D Fast Fourier Transform	64K points
	LU	Factors a matrix into the product of a lower & upper triangular matrix	512×512 matrix 16×16 blocks
	Ocean	Simulates large-scale ocean movements based on eddy and boundary currents	258×258 ocean
	Water	Evaluates forces and potentials that occur over time in a system of water molecules	512 molecules

Table 4.1: Applications studied for GangES

detection occurs (e.g., fatal processor exceptions, application assertion failure, application abort, out-of-bounds access, or timeout).

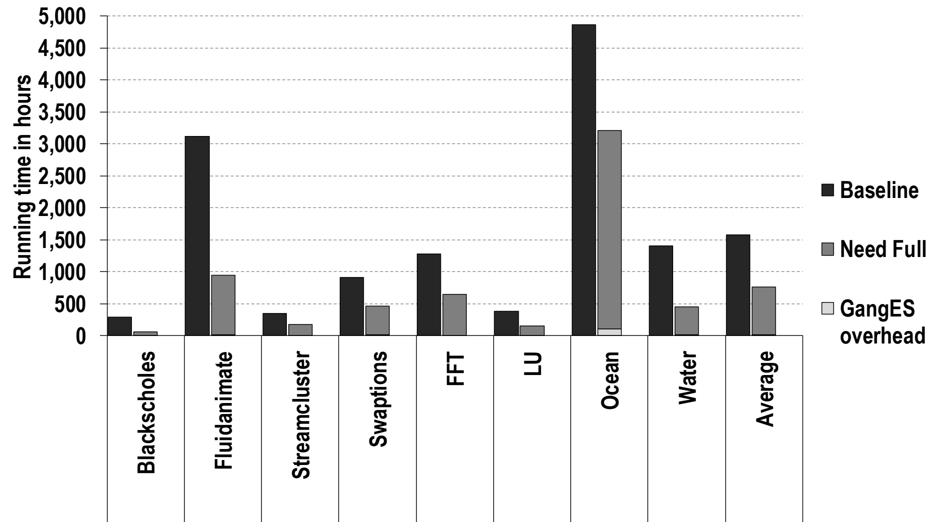
4.2.3 Evaluation

We evaluate our technique using a subset of applications we used for Relyzer (Section 3.2.1). Particularly, we used eight single-threaded applications – four each from the SPLASH-2 [62] and PARSEC [10] benchmark suites. Table 4.1 provides a brief description of these applications and inputs used (which are similar to what we used earlier).

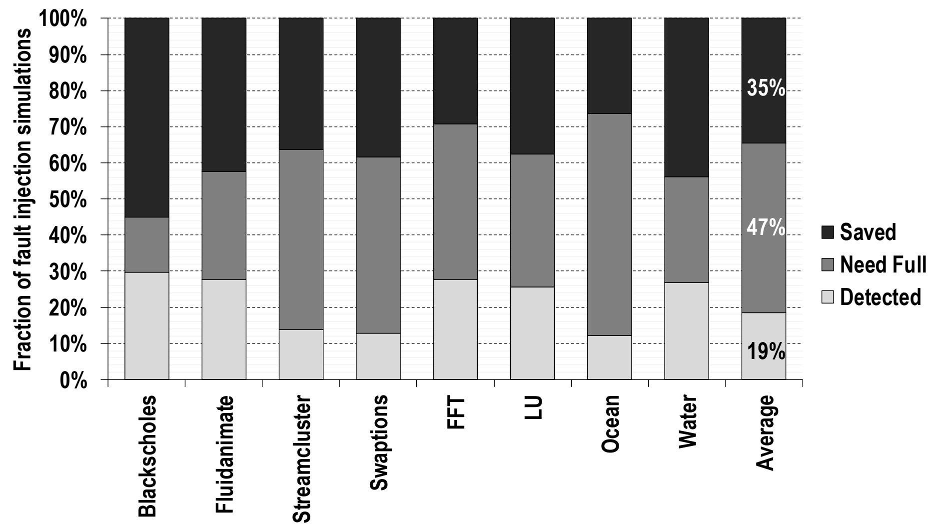
To evaluate GangES, we quantify the savings obtained by terminating the full application executions early in the error simulation framework. We obtain two quantities named *sim-savings* and *need full* which specify the number of full simulations that were saved and the number of error injections that need full simulation after applying the above mentioned screening technique, respectively. In cases where equivalence was showed between simulations, we also measure when the equalization occurred.

4.3 Results for GangES

After employing Relyzer, 1.08 million application error sites needed simulation for our workloads, which required approximately 12,600 hours of running time to perform error injection experiments (approximately 3 days on our cluster of 172 compute nodes). To reduce the evaluation time and the number of full simulations, we employ GangES.



(a)



(b)

Figure 4.4: Effectiveness of GangES in reducing the total wall clock time needed for error simulations and the number of full error simulations. For each application, the bars in Figure (a) show the total wall-clock time needed for simulating all Relyzer-identified errors, GangES framework, and the time spent in simulating errors that need full executions (after GangES). The bars in Figure (b) show the fraction of error simulations that GangES identifies as detections, that were *saved* from full execution (i.e., terminated early due to a state match with another execution), and that needed full simulation (*need full*).

Figure 4.4(a) shows the effectiveness of GangES. For each application, it first shows the baseline, which is the time needed for simulating Relyzer's errors. It also shows the time consumed by GangES in identifying which simulations need full simulations (*GangES overhead*) and the time that was needed to run such simulations to completion (*need full*). This figure shows that we obtain high simulation time savings of 51% on average across our workloads. These savings translate to hundreds of hours of simulation time savings for all our workloads (ranging from 231 hours for Blackscholes to 2,170 hours for Fluidanimate). This figure also shows that GangES consumes a small fraction of the total simulation time (only 170 hours for all our workloads together).

In Figure 4.4(b), we show the fraction of the total error simulations that result in a detection (these would be terminated early regardless of GangES), that were *saved* from full execution, and that need full simulation (*need full*). On average, approximately 35% of the total error simulations were saved; i.e., they were shown equivalent to another execution, saving the simulation time of running them to completion and comparing their output to the error-free output. Overall, 47% of the total input error set required full simulation.

Figure 4.5 shows when the equalization was performed for the *saved* simulations (from Figure 4.4(b)). It shows that on average about 94% of saved simulations were terminated at the first SESE region exit from the point of error injection. Approximately 5% and 1% of the saved simulations were equalized at the second and third SESE exits respectively.

Figure 4.6 shows the average distance in the number of executed instructions from the point of error injection to the simulation state equalization (at a SESE exit). Specifically, it shows that the average distance to first successful comparison (averaged across our applications) is approximately 2,850 instructions (where 94% of saved simulations are equalized). Distance to successful comparisons varied significantly with applications because the comparison points are identified according to the PST, which is application-specific. The average distance to successful comparison at the kth exit can be smaller than that of the qth, where $k > q$, because varying number of simulations from different procedures reach different SESE exits (note our PST generating algorithm is intra-procedural).

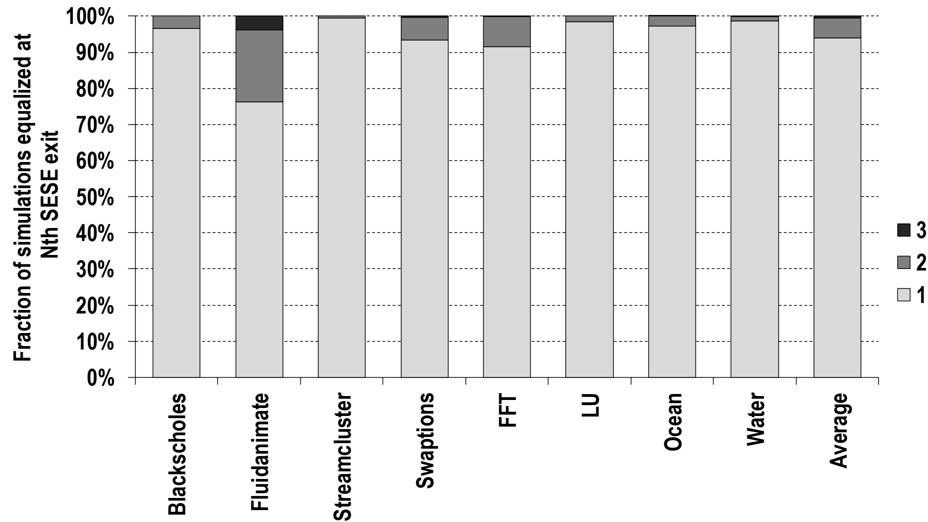


Figure 4.5: The SESE exit where the *saved* simulations (from Figure 4.4(b)) are equalized. As an example, 94% of saved simulations are shown equivalent to some other simulation at the first SESE exit.

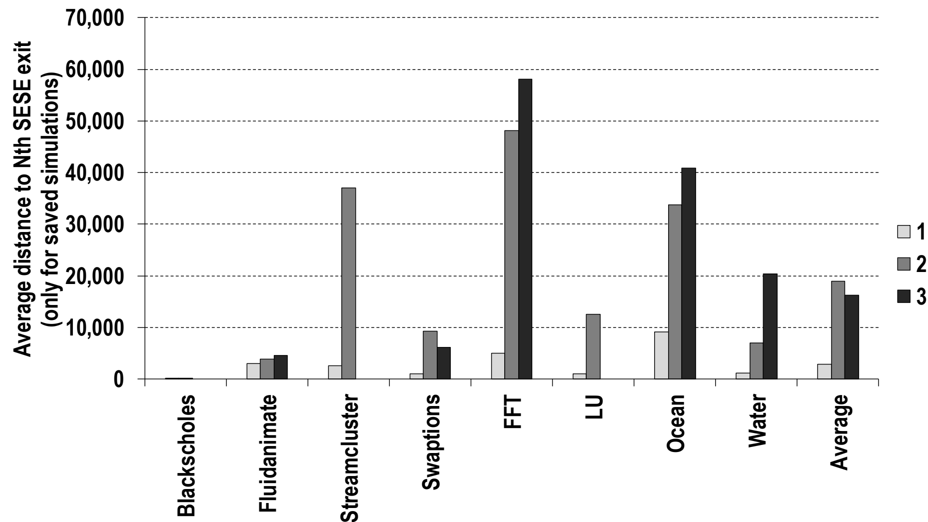


Figure 4.6: Distance of the successful comparison point (categorized by SESE region exit number) from the point of error injection.

Figure 4.7 explains why the simulations categorized as *need full* in Figure 4.4(b) were not equalized to other simulations. We categorize these simulations based on the following criteria: (1) Register state mismatched at all exercised SESE exits (Reg Only), (2) Register state matched but memory state mismatched at all exercised SESE exits (Mem Only), (3) Combination of categories

(1) and (2) occurred at different SESE exits, i.e., register state mismatched at some SESE exits and when it matched the memory state did not match (Reg+Mem), (4) No comparison was performed prior to a threshold number of instructions[6] executed (Timeout). From Figure 4.4(b), we observed that Ocean requires the highest number of full simulations, requiring 61% of all the simulations to be run until completion. From Figure 4.7, we note that over 58% of simulations that were marked as *need full* were never compared to any other simulation for Ocean. Hence, we need to identify more reachable comparison points and a compact way to store simulation state to allow more comparisons to increase the savings of GangES. Increasing our threshold for comparison should allow more comparison points and employing compression or encoding techniques should lower the simulation state storage overhead. These alternatives are a subject of our future research.

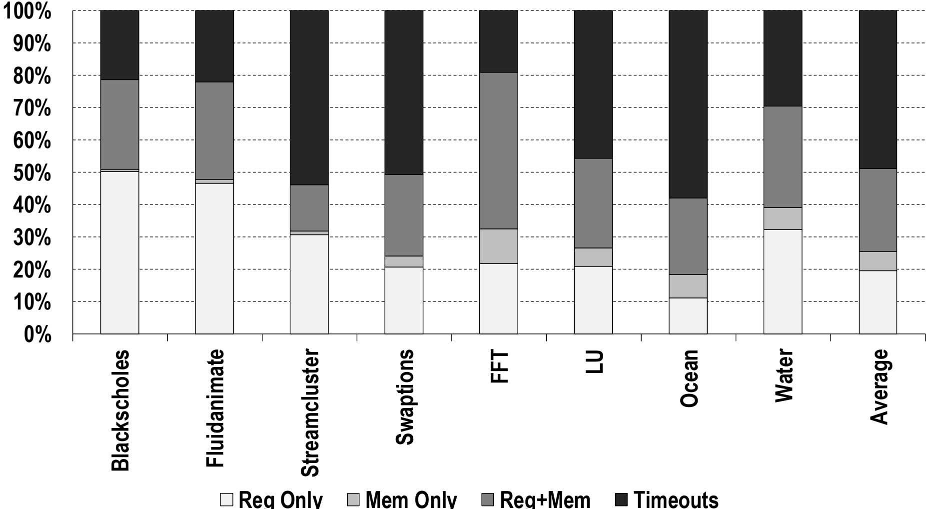


Figure 4.7: Categorizing the errors that need full simulations based on whether register state, memory state, or both mismatched during comparisons or no comparison was made before timeout condition was met.

We also evaluated the sensitivity of comparing all registers vs. live registers at SESE exits by observing the impact on the savings obtained by GangES (Figure 4.8). Recall that we fast forward 1000 instructions to obtain a conservative live processor register state at a comparison point (Figure 4.3). When we disallowed this step and compared all processor registers, the average wall clock time needed for GangES and full simulations increased to 911 hours from 764 hours for

[6]Recall that we use 100,000 as the default threshold. However, once the memory footprint of the executing bundled simulation exceeds 2GB, we adjust the threshold to 100 for the remaining injections in that bundle.

our workloads, reducing the average amount of simulation time savings to 42% (from 51%). This clearly shows that comparing live processor registers provides significant simulation savings over comparing all registers.

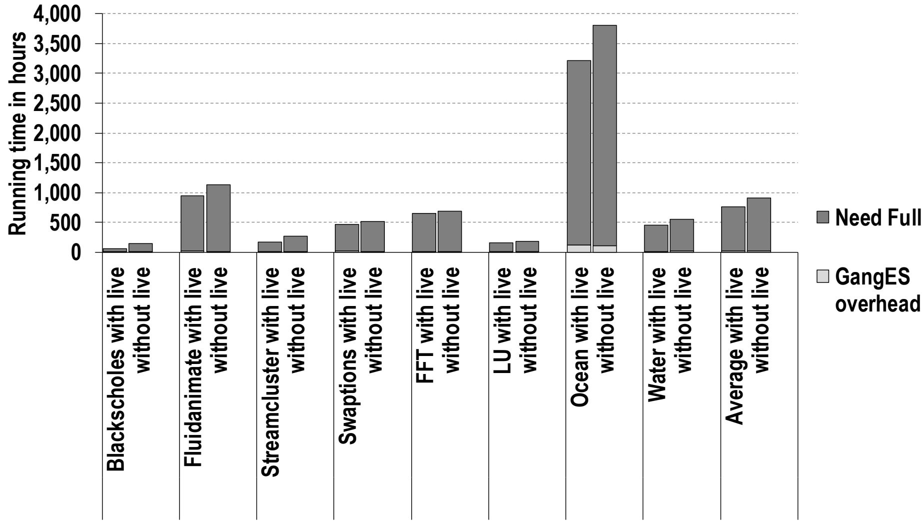


Figure 4.8: Amount of time needed by GangES and error simulations that need full executions when all processor registers were compared at SESE exits (vs. comparing just the live registers at these exit points) is shown here.

4.4 Summary

Relyzer provides a practical approach to evaluate an entire application's resiliency. It employs simple dynamic analyses to predict error behavior and shows equivalence between several error sites. It then performs error injections on the remaining sites to obtain a complete application resiliency profile. This approach is practical but it still requires significant error simulation time to obtain an application's resiliency profile.

This chapter presents a gang error simulator called GangES, as a performance enhancement mechanism for Relyzer, that aims to significantly reduce the error simulation time needed to evaluate the outcomes of Relyzer's pilots. GangES bundles multiple error simulations together and periodically compares states to identify similarities between executions to allow early termination, saving significant evaluation time. Identifying when to compare executions and what state to compare is challenging because instruction sequences can be different between multiple error simulations

and comparing the entire system state can be expensive in time, respectively. To overcome this challenge, we leverage the static structure of a program, identify single-entry single-exit (SESE) regions, and use SESE region exit edges as comparison points. This approach provides limited and effectively spaced comparison points. We compare a small amount of system state comprising of live processor registers and limited touched memory addresses at these comparison points.

Our results show that after applying GangES, only 47% of the error simulations originally identified by Relyzer required running the application to completion and the checking the output to determine the fault outcome. 94% of the error simulations that were terminated early by GangES required an average of only 2,850 instructions to be executed before termination. Overall, we found that GangES replaced Relyzer's error simulation time of approximately 12,600 hours with a total time of 6,110 hours, providing a wall-clock time savings of 51.5% for our workloads and error model.

Chapter 5

Low Cost Program-level Detectors and Tuning SDC Reduction[1]

Since Relyzer significantly reduced the number of errors that require detailed error injection experiments for complete resiliency analysis, we performed error injections in the remaining sites and obtained a list of virtually all SDC-causing instructions in an application. In this chapter, we present an analysis of these SDC-causing program locations and the low cost selective program-level error detectors, we developed, that were are shown to be effective in reducing SDCs. Since hardware and software architects continuously trade off performance, power, and resiliency to meet ever increasing design constraints, developing a low cost error detection mechanism without providing the flexibility to balance costs is not sufficient.

Relyzer, for the first time, enables tuning resiliency by identifying virtually all SDC-vulnerable program locations. This allows architects to target any resiliency budget by protecting just the desired set of vulnerable program locations. Utilizing this approach we achieved the ability to obtain a set of near-optimal detectors for any SDC coverage target.[2] This allowed us to obtain continuous SDC coverage vs. performance trade-off curves. We present this approach (as the second application of Relyzer) and our results in this chapter.

Investigating the SDC-causing error sites revealed that only a small fraction of static instructions cause most SDCs. Figure 5.1(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. (We only considered errors in integer register operands of executing instruction in our work and Section 5.2 provides the detailed methodology for these results.) This observation motivates using selective instruction-level detection techniques. Prior work has made

[1]Most of the text, figures, and tables in this chapter are taken from the original publication [20] and are copyrighted by IEEE.

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

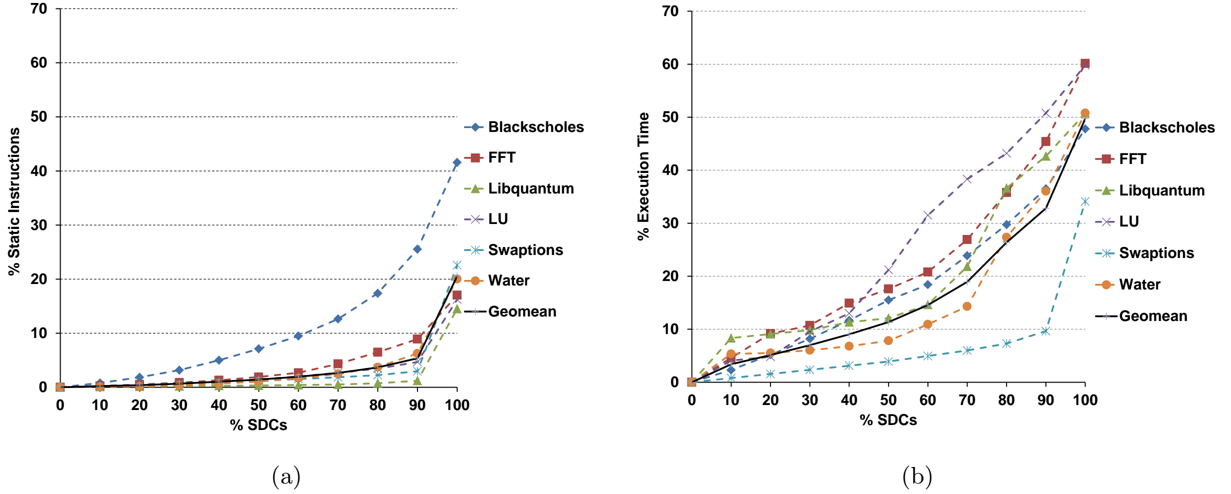


Figure 5.1: SDC-causing instructions and their impact on execution time. For a given application and input, part (a) shows the percentage of (executed) static instructions that cause a given percentage of silent data corruptions (SDCs). Part (b) shows the fraction of execution time (on a 1 IPC machine) 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 5.2.)

similar observations, but has used selective instruction-level redundancy for detection [16, 50, 51].

Figure 5.1(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 [16], 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 next 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.

5.1 Analyzing SDC-causing program sections and developing program-level detectors

We first analyze the list of the SDC-causing error 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.[3] 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 errors 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, (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. A further exploration of the relationship between such detection latencies and recovery is left to 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 detect SDCs in these code sections using the above mentioned insights.

5.1.1 Incrementalization in loops

We observed that a significant fraction of SDC-causing error 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 5.2(a)

[3] 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.

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 error sites.[4] In particular, these errors 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) An error 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 error. (2) Faults in addresses A and B can result in detection if the erroneous address is unallocated. If the erroneous address points to a valid but incorrect memory location then the error may be masked or result in an SDC. In this scenario, we observed that errors in several low-order bits in A and B resulted in SDCs because erroneous addresses pointed to incorrect locations in arrays a and b.

Analyzing this code further, we observe that it uses the loop incrementalization optimization [31]. 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 [31]. Figure 5.2(b) and (c) show the assembly codes without and with the incrementalization optimization respectively for the C code shown in Figure 5.2(a).

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 errors 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.

[4] Our error model (explained in Section 5.2) considers errors only in integer architecture register operands of executing dynamic instructions.

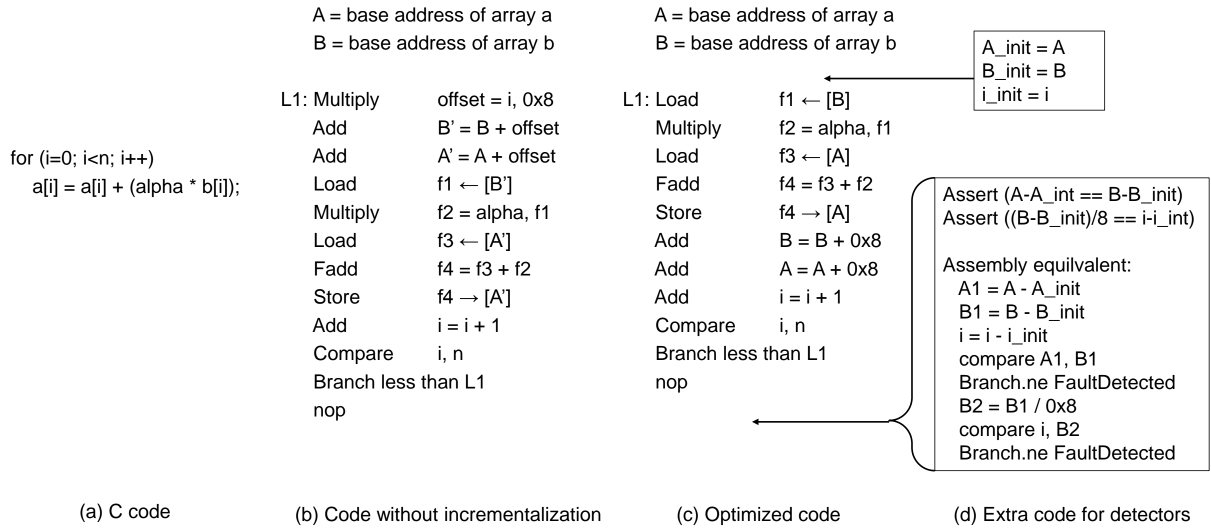


Figure 5.2: An "SDC-hot" code section with loop incrementalization in LU from the SPLASH2 benchmark suite: (a) C code, (b) unoptimized assembly without loop incrementalization, (c) optimized assembly with loop incrementalization, and (d) detector for the optimized code. Faults in this (optimized) loop 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.

Figure 5.2(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 5.2(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 5.2(b). Since errors 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.

5.1.2 Registers with long life

We observed that a sizable chunk of SDCs were caused by errors in registers with long life, with multiple uses through this life. For example, we observed that the register holding the value n in Figure 5.2(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 errors in the register that stores the value of n in Figure 5.2(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."

5.1.3 Application-specific behavior

For some applications, a large chunk of the SDC-causing error 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 error 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 error 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 error 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

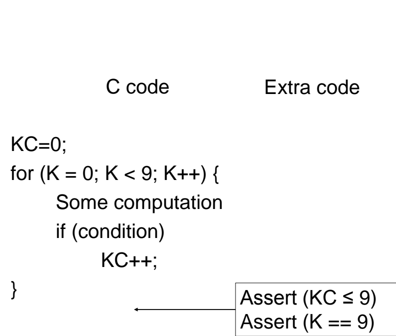


Figure 5.3: A detector for a register with a fixed upper bound. The figure shows a code section from the Water application. Faults affecting this code eventually corrupt the value of KC and produce SDCs. The assertions show how these errors can be detected.

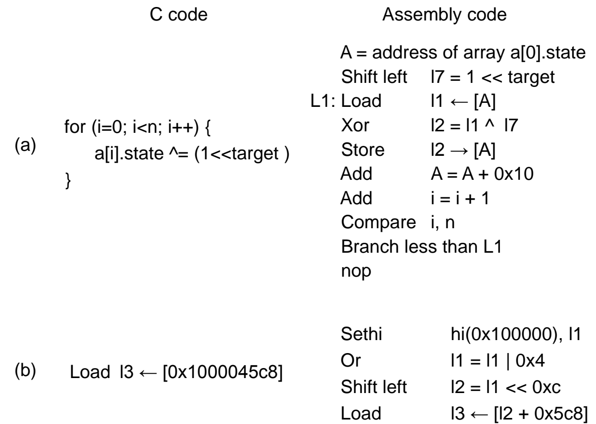


Figure 5.4: SDCs due to local computations: Faults in short-lived registers, $l1$ and $l2$, produce non-negligible fraction of SDCs. (a) Code from the Libquantum application. (b) Instructions generated by the Sun *cc* compiler to compute a static address.

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 errors 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 [20]). 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 5.3. To detect errors 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 errors 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."

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."

5.1.4 Local computations or registers with short life

We observed that a non-negligible fraction of SDCs were caused by errors in local computations with short register data flow chains. One example of this scenario is shown in Figure 5.4(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 [5] 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.

5.2 Experimental methodology

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

We selected a mix of six applications from the SPLASH2 [62], Parsec [10], and SPEC CPU2006 [24] benchmark suites for this study (Blackscholes, FFT, Libquantum, LU, Swpations, and Water from Table 3.1). All the selected applications were compiled using Sun C/C++ compiler version 5.9 with the highest level of optimization. We performed error injections such that 99% of all the error sites were analyzed (as reported by Relyzer). Overall we performed 890,000 error injections across all the studied applications. These experiments were completed in approximately 3 days on a cluster of 175 compute nodes. The error injection framework used in this evaluation is similar to the one described in Section 3.2.2.

5.2.1 Detectors and overhead evaluations

We implemented our program-level detectors (described in Section 5.1) 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 5.1 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 over-

heads for instruction-level redundancy by estimating that one instruction can be protected by one extra instruction even though the requirement is often more.

5.2.2 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 error injection campaign for the error 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.

5.2.3 Determining the lowest overhead detectors for a target SDC coverage

Our detectors from Section 5.1 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 5.2.1). 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.

	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 5.1: Extra instructions used for measuring execution overhead

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 5.2.2). 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.

5.3 Results

5.3.1 Sources of SDCs

For reference, Figure 5.5 shows the absolute SDC rates obtained by our Relyzer-driven error injection experiments as described in Section 5.2. The SDC rates of our applications range between 8% to 32%. These are much higher than prior evaluations [28, 46], primarily because of the difference in the error model. Our error model considers errors in only those architectural registers that are highly likely to be alive, whereas prior work uses microarchitecture (and lower) level error models

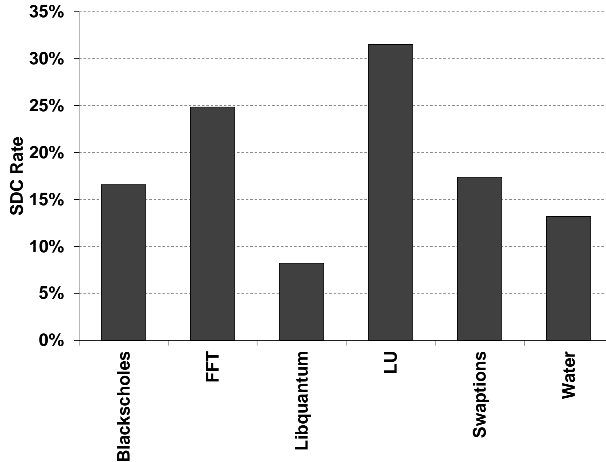


Figure 5.5: Baseline absolute SDC rates. These absolute rates are higher than previously reported for symptom-based detectors [28, 46], largely because of the different error models used.

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 5.2: Number of detectors placed in the static application code

which have a much higher masking rate [28, 26]. We chose the higher level error model because our focus is on uncovering all possible SDCs with as few error injections as 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 5.6 categorizes them based on the code patterns we identified in Section 5.1. Figure 5.6 shows this categorization. We observe that error 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 error 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

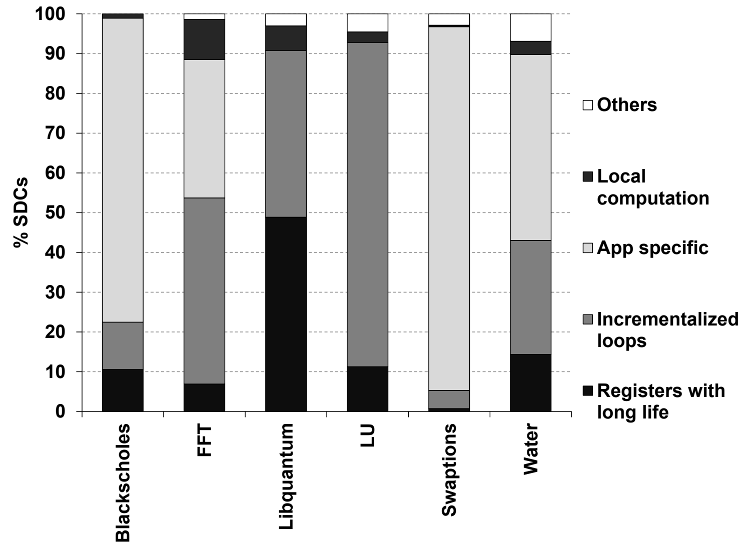


Figure 5.6: Contribution of code patterns from Section 5.1 to SDCs.

a large fraction of SDCs.

5.3.2 Static overhead of the program-level detectors

Table 5.2 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 errors 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. 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.

5.3.3 SDC coverage of the program-level detectors

Since the program-level detectors were placed based on the SDC vulnerability of the error 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 5.6 (*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 5.6 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 5.6. Instead we use a statistical error injection campaign as explained in Section 5.2.2. 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 5.6. For errors in the exponential function, we observed that most of the undetected errors 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 errors that result in output deviation as loss in coverage.

Figure 5.7 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.

5.3.4 Execution overhead from the program-level detectors

Figure 5.8 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

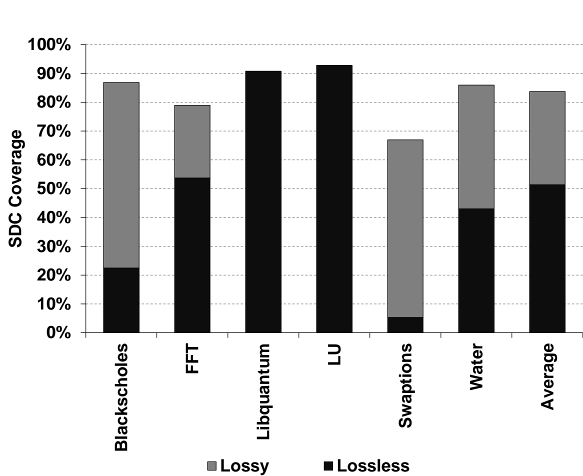


Figure 5.7: SDC coverage obtained by our program-level detectors, separated into coverage from the lossless and lossy detectors.

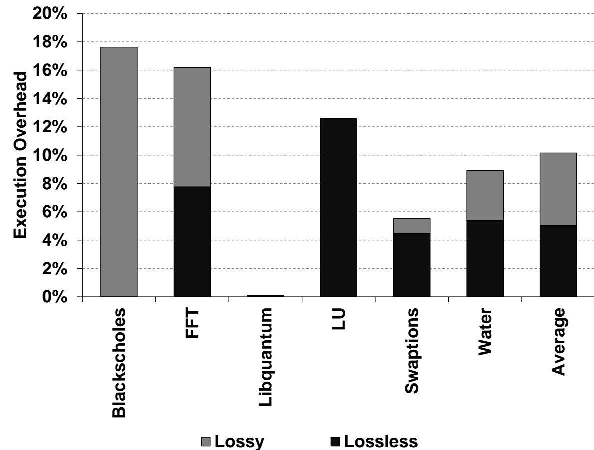


Figure 5.8: Execution overheads incurred by the program-level detectors, separated into coverage from the lossless and lossy detectors. The overhead of LU can be lowered to 3.4% with a small change in an input parameter without loss of performance or SDC coverage.

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.

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 5.2) 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 [10]. 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.

5.3.5 SDC coverage vs. execution overheads

Figure 5.9 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

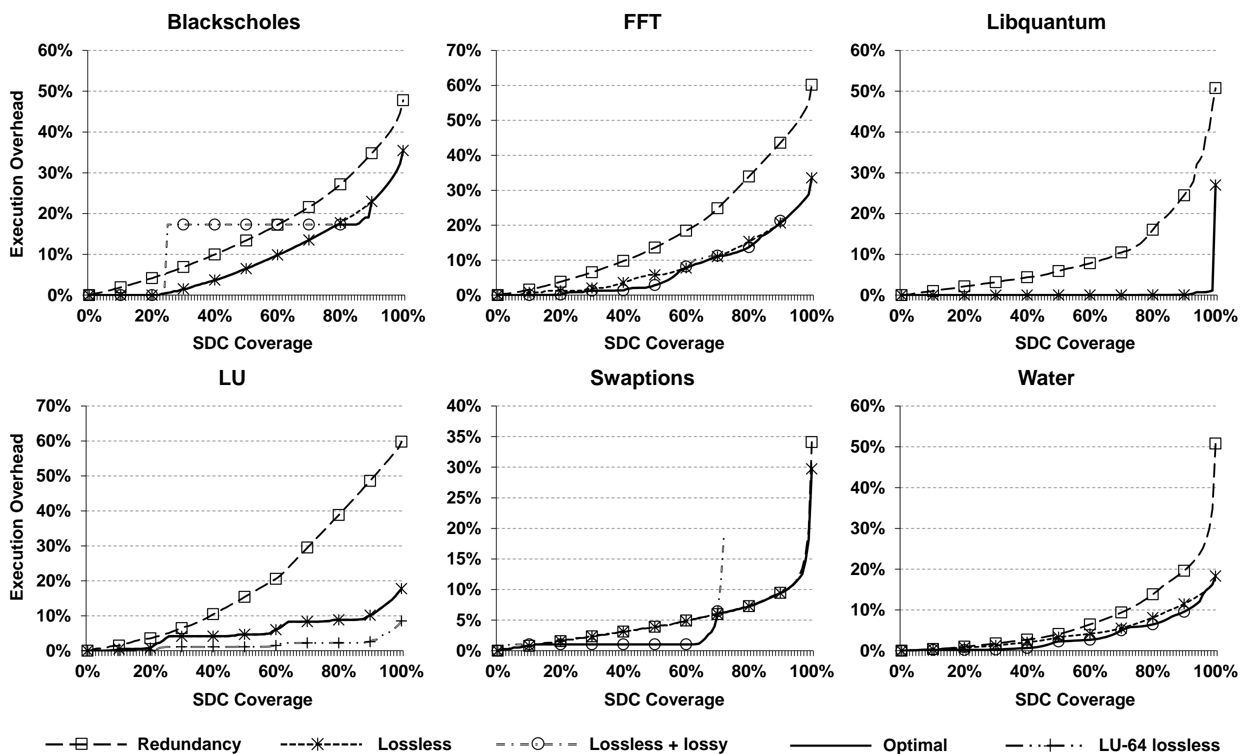


Figure 5.9: SDC coverage vs. execution overhead for each application for different classes of detectors.

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 5.2.3. 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).

5.4 Summary

Relyzer enables finding virtually all SDC-causing error sites in an application and this chapter exploits this capability to develop cost-effective SDC-targeted error detectors, as the first application of Relyzer. To achieve this goal, this chapter first presents an understanding of the program-level properties that repeatedly appear across different SDC-causing locations in the same application and across different applications. This analysis facilitated the placement and development of low cost program-level error detectors. For placement, the program locations were selected such that many errors propagate to these points in a few variables. For detectors, a range of program-level property checks were exploited such as checking for value equality between variables, range checks, and checking similar computations. Our results show that these SDC-targeted detectors were able to convert an average of 84% of the SDCs to detections across the studied applications, at an average execution overhead of 10%.

This chapter also employs Relyzer and these SDC-targeted detectors to tune resiliency at low cost, as the second application of Relyzer. Relyzer enables finding near-optimal cost error detectors for any SDC reduction target, allowing us to obtain continuous SDC coverage vs. performance trade-off curves. Using the developed low cost program-level detectors and selective instruction-level duplication based detectors for instructions that are not covered by program-level detectors, this chapter presents SDC coverage vs. execution overhead trade-off curves for our application. Results show that the developed program-level detectors (with instruction-level redundancy as backup) show significantly lower execution overheads on average when compared to instruction-level redundancy alone at all SDC coverage targets of interest; e.g., 19% vs. 43% for 99% SDC coverage, presenting system architects with effective design choices.

Chapter 6

Evaluating Pure Program Analyses Based Metrics to Find SDCs

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 be hard to obtain. Previous work has used simple (static and dynamic) program properties to identify program locations that are susceptible to producing SDCs. These techniques are much faster than Relyzer (and statistical fault injection based techniques), but their accuracy could not be validated.

Relyzer, for the first time, enables determining the accuracy of previously proposed pure program analyses based techniques (because it provides error outcomes for virtually all instructions). In this chapter, we study the approach proposed by [41] (and some derivatives) as an example to evaluate pure program analyses based techniques. Although our results were largely negative, we present them for completeness. It is possible that other pure program analyses techniques provide better results, but a comprehensive study is beyond the scope of this chapter. We believe our results here provide evidence that such models are not straightforward to determine and signify the importance of Relyzer. This effort was led by my colleague Radha Venkatagiri.

6.1 Pure program analyses based metrics for finding SDCs

Based on the results from [41], we explore the following two metrics for a given static instruction, as an indicator of its vulnerability to SDCs: (1) *Fanout* is defined for a dynamic instruction that writes to a register R as the number of uses of R before the next dynamic write to R. For a given static instruction, the fanout metric describes the cumulative fanout of all the dynamic instances of the instruction in the program. (2) *Av.lifetime* is defined for a static instruction that writes to

a register as the average of the lifetimes of dynamic instances of the static instruction. Lifetime for a dynamic instruction I_d that writes to a register R is defined as the number of cycles from the execution of I_d to the last use of R before the next dynamic write to R.

Additionally, we also explore the following three metrics: (1) *Av.fanout*, which is the fanout averaged over all dynamic instances of an instruction. (2) *Lifetime*, which is the cumulative lifetime over all dynamic instances of an instruction. (3) *Dyn.inst*, which is the total number of instances of the static instruction. The last metric was also explored in the prior work, but did not give good results – we explore it here because it performed better than other metrics in some cases for our results.

For a given static instruction, we also obtain the number of SDCs it produces by employing Relyzer and use it as the golden metric (*sdc*). We evaluate our five metrics using five of our applications (Blackscholes, Swaptions, FFT, LU, and Water). We collect the values of these metrics at the instruction level using Simics. We normalize all our metric values to one. For our error model, we consider transient errors or single bit flips in every bit in destination integer architecture registers (one at a time) of executing instructions.

6.2 Evaluation methodology

To evaluate our metrics, we first quantify how accurately they predict SDCs in isolation using two methods. First, for each application, we measure Correlation Coefficients[1] between individual metrics and *sdc* (golden metric).

Our second method of comparison observes that the objective of estimating SDCs with a metric is to find the best instructions to place error detectors to convert the SDCs to detections. We therefore first employ a 0/1 knapsack algorithm to find the optimal set of detectors that will provide the largest SDC coverage at a given cost – we assume duplication for detectors and charge one instruction as cost for duplicating and comparing results for one instruction on average. Thus, we determine SDC reduction vs. cost graphs using the known SDC count for each instruction from Relyzer (the *sdc*

[1]Correlation Coefficients *cc* are a standard measure of the linear relationship between two variables X and Y giving a value between +1 and -1 inclusive. $|cc|$ gives the strength of the correlation (1 indicates a perfect linear correlation and 0 indicates no correlation between X and Y). We use Pearson's Correlation Coefficients for our analysis.

golden metric). We call this curve the Relyzer curve (RC), which is similar to the curves shown in Figure 5.9 but with flipped axes.

We then apply the same knapsack algorithm using the metric of interest instead of the SDC count and plot a curve that we call the Prediction curve (PC). This curve is the predicted SDC reduction vs. cost curve if the metric were accurate. We also plot an Actual Curve (AC) as follows. For each point on the PC curve, we calculate and plot the actual number of SDCs (from Relyzer) covered by the instructions actually identified by the metric in the PC curve. This gives the actual SDC reduction vs. cost of the metric. For a given cost, the gap between AC and RC tells us how well the metric estimates SDCs (the smaller the gap, the better).

We also evaluate combinations of the above metrics using linear models based on regression techniques that use these metrics to predict SDCs being produced by the instructions. We use the statistical tool R to build (least square) linear regression models for each of the benchmarks, which take the following form

$$sdc_i = \beta_0 lifetime_i + \beta_1 fanout_i + \beta_2 av.lifetime_i + \beta_3 av.fanout_i + \beta_4 dyn.inst_i + \epsilon_i \tag{6.1}$$

We also attempt to evaluate a non-linear combination of our metrics. Since some non-linear relationships between variables (or metrics) can be approximated using linear regression on polynomials,[2] we evaluated another linear regression to model the following:

$$\begin{aligned} sdc_i = \beta_0 lifetime_i + \beta_1 (lifetime_i)^2 + \beta_3 (lifetime_i)^3 + \\ \beta_4 fanout_i + \beta_5 (fanout_i)^2 + \beta_6 (fanout_i)^3 + \\ \beta_7 av.lifetime_i + \beta_8 (av.lifetime_i)^2 + \beta_9 (av.lifetime_i)^3 + \\ \beta_{10} av.fanout_i + \beta_{11} (av.fanout_i)^2 + \beta_{12} (av.fanout_i)^3 + \\ \beta_{13} dyn.inst_i + \beta_{14} (dyn.inst_i)^2 + \beta_{15} (dyn.inst_i)^3 + \epsilon_i \end{aligned} \tag{6.2}$$

[2] The more complex the non-linearity, the higher the order of polynomials required.

Applications	vs	*lifetime*	*fanout*	*av.lifetime*	*av.fanout*	*dyn.inst*
Blackscholes	*sdc*	0.25	0.56	-0.05	-0.04	0.62
Swaptions		-0.03	0.23	-0.03	-0.02	0.27
FFT		0.08	0.49	-0.03	-0.01	0.80
LU		0.21	0.59	-0.02	-0.01	0.82
Water		0.02	0.12	-0.02	-0.01	0.18
All		0.13	0.44	-0.02	-0.01	0.54

Table 6.1: Correlation Matrix: Correlation coefficients between *metrics* and *sdc* for different workloads

6.3 Results for pure program analyses based metrics

6.3.1 Effectiveness of individual metric in predicting SDCs

We measure the Correlation Coefficients between *sdc* and individual metrics for all our metrics and applications, as described in Section 6.1. Table 6.1 shows that for our studied workloads, *av.lifetime* and *av.fanout* have no correlation with *sdc*. *Lifetime* displays weak to no correlation with *sdc* and *fanout* exhibits moderate correlation for Blackscholes and LU. Although *dyn.inst* is the only metric that shows high correlation with *sdc* for a few applications (FFT and LU), there is no single metric that uniformly demonstrates a strong linear relationship with *sdc* for all our workloads.

SDC vs. cost curves

Here we compare the optimal SDC coverage vs. cost for instruction duplication curves using our metrics vs. Relyzer by plotting the *RC*, *PC*, and *AC* curves as described in Section 6.1. For brevity, we present the SDC vs. cost curves for a representative subset from our workload and metric combinations in Figure 6.1 (the remaining graphs have patterns similar to one of the graphs shown here).

For graphs that use *av.lifetime* and *av.fanout*, the PC curve immediately goes up to very close to 100%. This is because the static instructions that have very large values for *av.lifetime* and *av.fanout* have few dynamic instruction. Hence, these static instructions they account for a large fraction of these metrics and the execution overhead of protecting them (based on dynamic instruction count) is very small.

Graphs for LU:*dyn.inst*, FFT:*dyn.inst*, and Blackscholes:*fanout* show a high correlation between

PC and AC, which was expected based on Table 6.1 (correlation coefficients >0.5). However, there is a significant gap in SDC coverage (in Y axis for a given value on X axis) between AC and RC curves. For example, at dynamic instruction overhead of 20%, the loss in SDC coverage for LU:*dyn.inst* and FFT:*dyn.inst* is 33% and 21%, respectively compared to Relyzer (RC). This is primarily because the instructions that do not produce SDCs were also selected by the metrics for protection. To understand this better, consider the FFT:*dyn.inst* graph from Figure 6.1, where the PC is a 45% line because the *dyn.inst* metric predicts that all instructions produce SDCs proportional to their dynamic instruction counts. The respective AC curve correlates well with the PC curve but the detectors added for any x-axis point (execution overhead target) also protects instructions that do not produce SDCs. On the other hand, RC places detectors for only those instructions that produce SDCs providing a much more cost effective set of detectors for the same execution overhead target.

Overall, the large gaps we observed in the SDC coverage for a given overhead are unacceptable, revealing that the individual metrics are poor predictors of SDCs. This also indicates that correlation coefficient alone is not a determining factor in predicting SDCs.

6.3.2 Linear Regression

Table 6.2 shows the result of the linear regression (using least squares) for our workloads (Equation 6.1 from Section 6.2). For each application we show the metrics that are significant in the model in the second column. The adjusted R^2 for the model is shown in the third column. Adjusted R^2 value estimates the percentage of variance in *sdc* that is explained by the *metrics*. If the adjusted R^2 is high then the derived model is considered robust. For example, 0.68 adjusted R^2 for LU implies that only 68% of the variance in *sdc* can be explained by the metrics, which leaves 32% as unexplained or caused by randomness. However, a low adjusted R^2 value can be interpreted either as the model is missing key additional explanatory variables (other metrics) that can improve the model, or a linear model is not sufficient to explain the relationship between the metrics and *sdc*. From our results in Table 6.2 we make the following observations:

- No common model (formed by a linear combination of our metrics) that offers a best fit for all our workloads was identified. For different applications different metrics were identified as be-

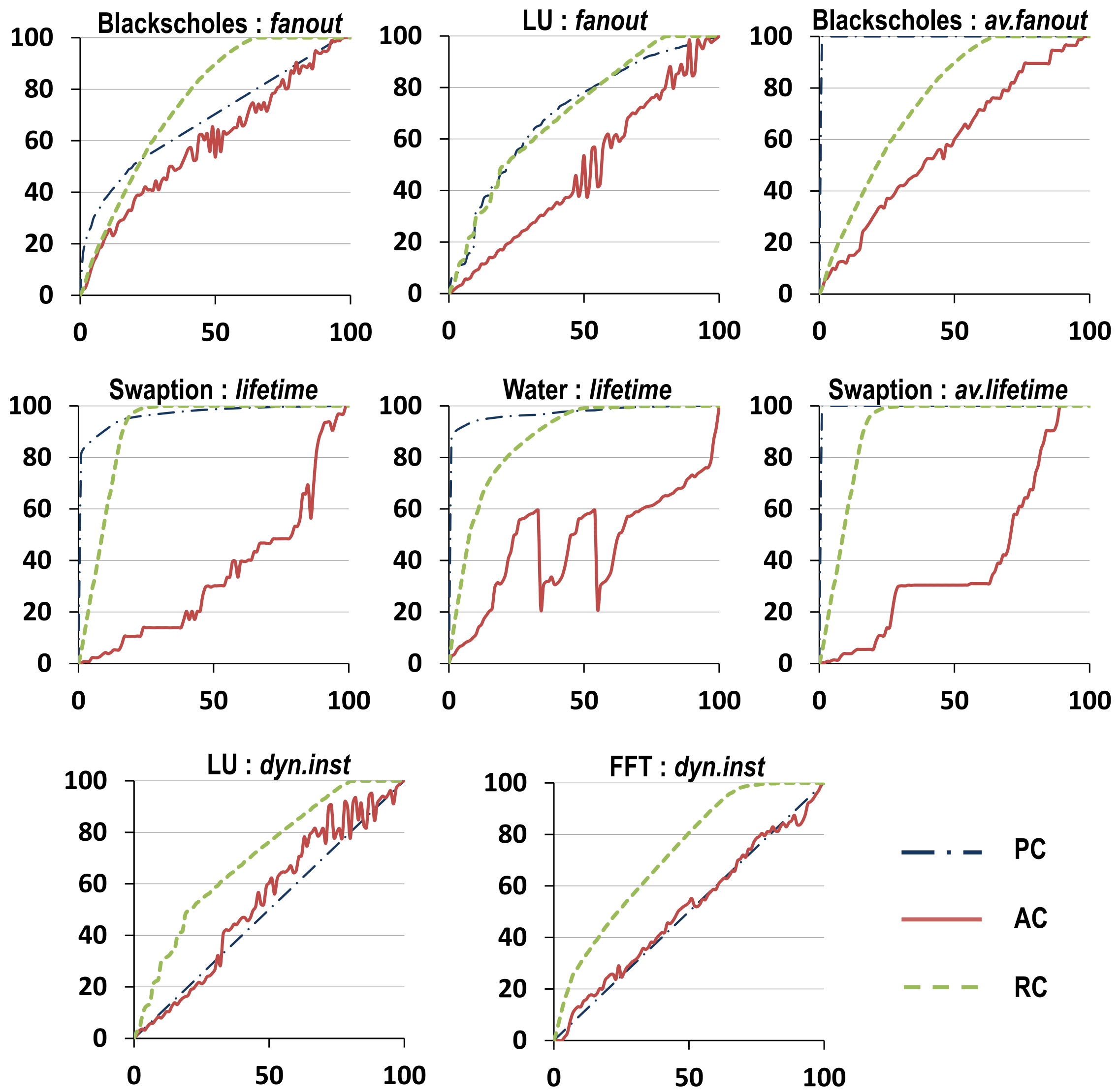


Figure 6.1: SDC reduction vs. execution overhead curves. The X axis for the above graphs plots *% execution overhead* (in terms of increase in dynamic instructions) and the Y axis represents *% SDC covered*

Applications	Significant metrics	Adjusted R^2	CV_{10}	CV_4	CV_2
			RMSE/Mean		
Blackscholes	*fanout*, *av.fanout*, *lifetime*, *dyn.inst*	0.58 [0.62]	1.62 [$> 10^4$]	62.65 [$> 10^4$]	203.3 [$> 10^4$]
Swaptions	*dyn.inst*, *lifetime*, *fanout*	0.08 [0.27]	4.39 [4.05]	4.49 [4.15]	4.60 [4.33]
FFT	*dyn.inst*, *lifetime*, *av.lifetime*	0.65 [0.66]	8.36 [$> 10^7$]	12.62 [$> 10^7$]	9.4 [$> 10^7$]
LU	*dyn.inst*, *fanout*, *lifetime*	0.68 [0.79]	4.64 [179.5]	4.42 [110.35]	4.58 [207.84]
Water	*lifetime*, *fanout*, *av.lifetime*, *av.fanout*, *dyn.inst*	0.04 [0.55]	6.62 [10^4]	6.31 [10^4]	13.22 [10^4]
All	*dyn.inst*, *lifetime*, *fanout*, *av.lifetime*	0.30 [0.484]	4.93 [5.25]	4.77 [6.47]	4.83 [37.9]

Table 6.2: Linear regression summary. Numbers in brackets correspond to the respective numbers for linear regression on polynomials.

ing significant contributors. For metrics that prove to be significant for multiple applications, the respective regression coefficients (β_i) were different. For example, even though *dyn.inst* is identified a significant metric for Swaptions, LU, and FFT, the regression coefficients (β_4) were 0.33, 1.29, and 0.96 respectively.

- The adjusted R^2 varied between 0.08 (for Swaptions) to 0.68 (for LU), and were mostly low. This implies that either our model is missing key additional metrics or a linear model is not sufficient to explain the relationship between our metrics and *sdc*.

- Since we could not find a common linear model that can explain the SDCs in our applications, we attempted to identify different linear model for different applications with limited number of error injection experiments (without requiring full Relyzer results). Hence we performed the cross validation experiments (shown in last three columns of Table 6.2). The Root Mean Square Error (RMSE) ratios for K-fold cross validations (CV_K) with K = 10, 4 and 2 is shown in the last tree columns (the reported number is RMSE/Mean). A K fold Cross Validation splits the population randomly into K parts. K-1 parts are used for training the model and one part is used for testing. This is done K times till all the parts have been used for testing.

  For models that have relatively high adjusted R^2 value (e.g., for LU), the cross validation showed high errors (values >1) in the predicted and observed SDCs. For example, for LU, the average error for CV_{10} is 4.64 times the mean.

Hence, we conclude that simple linear models based on our metrics may not be uniformly explanatory or predictive of SDC for our workloads.

We conducted another experiment to evaluate more complex non-linear models to predict SDCs. Since some non-linear relationships can be approximated using linear regression on polynomials, we performed linear regression to model Equation 6.2 (shown in Section 6.2). Our results, presented in brackets in Table 6.2, show that a trend similar to that of linear regression (using Eq. 6.1) was obtained for linear regression on polynomials. No common model for our studied workloads was identified and the error from cross-validation was also high. The the adjusted R^2 has improved for all our workloads, which indicates that non-linear combination of metrics can perform better than

a linear combination in predicting SDCs. However, for several applications the adjusted R^2 value is still poor, indicating that other metrics and/or different non-linear regression models are required.

6.4 Summary

This chapter employs pure program analyses based techniques to study application's resiliency. These techniques examine certain (static or dynamic) program properties to identify program locations that are susceptible to producing SDCs [16, 41]. These techniques are much faster than pure error injection based techniques or Relyzer, but their accuracy has not been previously validated.

Relyzer, for the first time, enables determining the accuracy of such (previously proposed) program analysis based metrics and some derivatives. Hence, we studied multiple metrics such as cumulative and average *fanout* and *lifetime* and dynamic instruction count. We also studied combinations of these metrics using linear models based on regression techniques. Results presented in this chapter show that these metrics and their combinations are unable to adequately predict an instruction's vulnerability to producing SDCs. Since the objective of finding such metrics is to find lowest cost detectors to reduce SDCs, we also compared the optimal SDC coverage vs. cost curves for these metrics with Relyzer's. However, results show that there are significant differences in the SDC coverage for a given cost budget, indicating that the studied metrics are poor predictors of SDC-causing instruction. Although it is possible that other (unexplored) pure analyses based techniques may be more accurate, the results presented in this chapter demonstrate that pure program analyses based techniques to identify SDC-causing program locations will be hard to develop, which further motivates future work on improving Relyzer (and GangES).

Chapter 7

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) [9, 36, 7]. There has also been substantial microarchitecture-level work that exploits redundancy at a finer microarchitectural granularity [7, 12, 14, 18, 38, 48, 49, 54, 50]. 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.

Software anomaly based error detection techniques [15, 17, 42, 60, 45, 28, 53, 22] provide one such low cost alternative. These mechanisms treat anomalous software behavior as symptoms of hardware errors and detect them by placing very low cost anomaly monitors in hardware or software. Researchers have shown that this approach is effective in detecting both permanent and transient hardware errors with only a small fraction resulting in SDCs through statistical error injections on microarchitecture-level models [28, 22]. Such techniques form the baseline for our work.

7.1 Resiliency evaluation

There has been much work in evaluating resiliency solutions [34, 28, 26, 44, 43, 39]. Much of this work is evaluated using statistical error 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 errors were actually injected).

SymPLFIED [40] and Shoestring [16] share our high-level goal of finding errors 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 error values per application site that need to be injected (hence the symbolic error). Our focus, so far, has been on reducing the number of application sites where the error is injected (we restrict the values by simply restricting our hardware error models since that is not the focus of our work). Combining 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 errors are likely to be detected quickly enough; e.g., there is a short-enough path in the data-flow graph from such an error to enough potentially anomaly generating instructions. The rest of the errors 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. Shoestring employs only 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 error locations into equivalence classes and then performs accurate error simulation of the representative error to identify the outcome. This allows Relyzer to account for masking of an error, 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. [8] 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. This work was, however, evaluated using small applications with few variables. Relyzer, on the other hand, captures the error propagation behavior from the error-free execution of the application and uses it to categorize errors into different equivalence classes. It relies on error injection experiments on the pilots (representatives of the equivalence classes) to estimate the outcome of the population and we developed GangES to speed up the error simulation time. In Chapter 6, we also evaluated several program metrics (lifetime, fanout, and dynamic instruction count of instruction writing to registers) and their combinations in predicting the SDC-causing program instructions. However, our findings were largely negative and we could not find any single model that can uniformly predict SDCs for our workloads.

Sridharan et al. [56] quantify the reliability behavior of an application using a metric called Program Vulnerability Factor (PVF). PVF is a microarchitecture-independent method to quantify architectural error masking inherent to a program. PVF focuses on identifying only those errors that are masked by the application. It does not attempt to distinguish errors that lead to SDCs from the ones that result in detection, but this distinction becomes crucial with anomaly based detection techniques in place. Hence, our work focuses on distinguishing SDCs from detections. It is unclear whether PVF can make this distinction.

7.2 Detectors to reduce SDCs

SWIFT [50] 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 [51] later improved the performance of SWIFT through hardware support. PROFiT [52] 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 error 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

application's 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 [16], 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 error detection has been applied for error detection [17, 53, 42]. In particular, range-based likely program invariants (inserted at all stores) have been employed for reducing SDCs produced by hard errors [53]. Results show a reduction in SDCs of up to 74% for a microarchitecture-level permanent error model, but with an execution overhead of 14% on SPARC machines. This technique suffers from false positives which are handled with diagnosis related hardware. For a transient error model, our technique provides a better SDC coverage vs. performance trade off through a more selective placement of a broader range of detectors. Combining insights from these two studies for both error models is left to future work.

Pattabiraman et al. [41], 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 error 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 [42]. 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 do not consider complex properties spanning across multiple variables like the loop based detectors presented in this thesis (Chapter 5). Moreover, detectors in [42] produce false positives, whereas our detectors never fire in error-free executions.

Motivated by this approach, we explored fanout, lifetime, and dynamic instruction count, their derivatives and several combinations as metrics to identify SDC-causing instructions in Chapter 6. However, our findings were largely negative and we could not identify any single model that can explain the SDC-vulnerability of instructions for our workloads.

Chapter 8

Conclusion and Future Directions

8.1 Summary and conclusions

With technology scaling, the hardware reliability problem is becoming increasingly challenging for a wide class of systems, motivating low cost reliability solutions. Software anomaly based detection techniques have emerged as low cost and effective solutions with low Silent Data Corruption (SDC) rates. However, for some cases, the SDC rates are 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.

The first part of this thesis proposes Relyzer, a technique to systematically analyze all transient error injection sites in an application. It employs a set of novel error pruning techniques that dramatically reduce the number of errors (application sites) that require thorough error simulations. Relyzer predicts the outcome of several errors, eliminating the need for thorough error injection experiments for them. It then exploits the insight that several application error sites are impacted in a similar way by certain hardware errors, and develops heuristics to identify such application-level error equivalence. Relyzer employs a series of static and dynamic techniques to categorize equivalence classes of errors, such that only one pilot error from an equivalence class needs to be thoroughly studied through error injection experiment. Through these techniques, we show that Relyzer prunes the set of errors by 99.78% across the twelve studied applications. With error injections in the remaining error sites, Relyzer can identify all SDC causing instructions.

Relyzer presents a practical resiliency evaluation approach but it still requires significant running time. Most of this running time is spent in error injections. Hence, this thesis presents a new error simulation framework called GangES to further improve the evaluation time of Relyzer.

GangES bundles multiple error simulations and periodically compares states to identify similarities between executions to allow early termination, saving significant evaluation time. Identifying when to compare executions and what state to compare is challenging because instruction sequence can be different between multiple error simulations and comparing entire system state can be expensive in time, respectively. To overcome this challenge, we leverage the static structure of a program, identify single-entry single-exit (SESE) regions, and use SESE region exit edges as comparison points. This approach provides limited and effectively spaced comparison points. We compare a small system state comprising of live processor registers and limited touched memory addresses at these comparison points. Results show that majority of error simulations do not need full application simulation and checking of the output to identify the outcome of the error injection. Overall, we found that GangES replaced Relyzer's error simulation time of approximately 12,600 hours for eight of our workloads with 6,110 hours, providing a wall-clock time savings of 51.5%.

Relyzer has many applications and the second part of this thesis demonstrates three of them: (1) analyzing SDC-causing program locations and adding error detectors to prevent SDCs, (2) tuning application resiliency at low cost, allowing system architects to efficiently balance SDC reduction vs. execution overhead, and (3) evaluating pure (static and dynamic) program analyses based metrics (and a combination of them) that aim to identify SDC-causing program locations with significantly less effort and runtime when compared to Relyzer.

Utilizing Relyzer, we identified virtually all 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 SDC-targeted detectors can convert an average of 84% of the SDCs to detections across our applications, at an average execution overhead of 10%.

Relyzer also enables tuning resiliency vs. performance by identifying virtually all SDC-causing application locations. We considered instruction-level redundancy based detectors and our program-level detectors to obtain SDC reduction vs. execution overhead trade-off curves. Results show that our program-level detectors (with instruction-level redundancy as backup) achieve significantly lower overheads when compared to redundancy alone at all SDC reduction targets, on average; e.g.,

19% vs. 43% for 99% SDC coverage. This demonstrates our SDC-targeted application-centric resiliency approach, enabled by Relyzer, allows us to provide practical and flexible choice points in the performance vs. reliability trade-off curve.

We also evaluated (previously proposed) pure program analyses based metrics and some derivatives that do not need error injections as a faster alternative to find SDC-causing program locations. Our results were largely negative and we believe they provide evidence that such program analyses based models are not straightforward to determine, signifying the importance of Relyzer.

8.2 Limitations and future directions

8.2.1 Enhancing the program-level detectors

The detectors developed in Chapter 5 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.

- Accounting for application-level error tolerance: It has been shown that a large class of applications are inherently error tolerant [30, 46]. 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.

8.2.2 Lower-level error models

The goal is to develop a set of detectors that are independent of the error model. Hence, evaluating the detectors presented in Chapter 5 on a microarchitecture-level error model (or other low-level error models) would be interesting.

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

8.2.3 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 be hard to obtain. To overcome this challenge, we studied and evaluated previously proposed program metrics (and some derivatives) obtained from simple and pure program analyses to identify SDC-vulnerable program locations [41, 8]. We also evaluated a combination of these metrics using linear models based on regression techniques. From our investigation, we concluded that either our models were missing key additional program metrics or our linear model is not sufficient to explain the relationship between the metrics and *sdc*.

Exploring new metrics and more complex non-linear models to predict SDC-vulnerable program locations is an interesting future direction. From our experiments we observed that instructions that do not produce SDCs (but have high metric values) were among the reasons for inaccuracies. It would also be interesting to evaluate a two-step approach that first models the probability of an instruction producing SDCs using logistic regression and then builds a linear/non-linear model to quantify the amount of SDC for those instructions.

We observe that software/hardware correctness testing also has the same input-specificity issue. Leveraging solutions deployed to those fields, e.g., unit testing, is interesting future direction. The correctness testing techniques also have another issue of covering uncommon corner cases, which may not be a major concern for Relyzer because the faction of the runtime spent in executing such program sections is typically insignificant, lowering their impact on application's reliability.

8.2.4 Methodical resiliency analysis

The modified program structure tree or PST, as described in Section 4.1.2, can be utilized to analyze the resiliency of an application. Traversing this tree in a top-down fashion would allow one to identify SDC hot spots in a program. Traversing it in the bottom-up fashion, on the other hand, would 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). Moreover, most of the detectors that we identified in Chapter 5 were placed at the end of SESE exits (end of loops and function calls). Overall, this tree structure can provide an automated framework to identify and place program level-detectors to reduce SDCs.

8.2.5 Exploiting program indexing schemes

GangES performs system state comparisons at program points that are identified using the program structure tree or PST (Section 4.1.2) to show equivalence between multiple error simulations to allow early terminations (and simulation-time savings). This (indexing) scheme has the three following properties. (1) It provides program points where most executions eventually reach even if different system events take place or different branch directions are exercised. (2) It provides limited number of comparison points (which is the depth of PST, typically <14 [25]) to limit the runtime needed for comparing states. (3) It provides program locations where errors are typically consolidated in a few program variables (e.g., end of loops and function calls, which are also SESE exits, Section 5.1) because temporary variables typically become dead at SESE exits, providing effective comparison points.

This scheme, however, misses some opportunities to show equivalence between error simulations

(Section 4.3, Figure 4.7). Prior research in the field of software reliability has studied program and memory indexing schemes [63, 57] mainly for software bug diagnosis. The structural indexing scheme [63] provides a mechanism to uniquely identify individual execution points so that the correlation between points in one execution can be inferred and correspondence between execution points across multiple executions can be established. This scheme would provide GangES more comparison points. However, mechanisms to prune structurally similar points are needed to limit the potentially large number of comparison points this scheme provides (to obtain the properties (2) and (3) mentioned above).

A memory indexing scheme [57] aligns memory locations across different executions, as opposed to [63] which aligns executions. Aligned memory locations across runs share the same index. Although, GangES only looks for divergence in memory states and not for the specifics of the state mismatch, a memory indexing scheme can be leveraged to analyze error propagation behavior from lower level (microarchitecture or architecture) error models to application level. Such an indexing scheme can also be exploited to develop accurate application level error models (as opposed to using single-bit-flip model at application level [13]).

8.2.6 Error recovery and detection latency

An important piece of any resiliency scheme is error recovery. The existing SWAT system relies on full system checkpointing schemes like SafetyNet [55] and ReVive [58] 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 [46].

This dictates the guarantees that a detection scheme should provide for detection latency. In our work, however, we did not evaluate the latency provided by our program-level detectors (from Chapter 5). Hence, measuring these detection latencies and tailoring our detectors such that they convert SDCs to detections within the recoverable window is important future work.

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] OpenSPARC T1. Website. `http://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html`.

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

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

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

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

[9] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStop Advanced Architecture. In *International Conference on Dependable Systems and Networks*, 2005.

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

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

[12] Fred Bower, Daniel Sorin, and Sule Ozev. Online Diagnosis of Hard Faults in Microprocessors. *ACM Transactions on Architecture and Code Optimization*, 4(2), 2007.

[13] Daniel Chen, Gabriela Jacques-Silva, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Bruce Mealey. Error behavior comparison of multiple computing systems: A case study using linux on pentium, solaris on sparc, and aix on power. In *Proc. of Pacific Rim Intl. Symp. on Dependable Computing (PRDC)*, 2008.

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

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

[16] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

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

[18] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *International Symposium on Computer Architecture*, 2003.

[19] Siva Kumar Sastry Hari. Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems running Multithreaded Workloads. Master's thesis, University of Illinois at Urbana Champaign, 2009.

[20] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. Low-cost Program-level Detectors for Reducing Silent Data Corruptions. In *International Conference on Dependable Systems and Networks*, 2012.

[21] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep 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.

[22] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *International Symposium on Microarchitecture*, 2009.

[23] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V. Adve, and Helia Naeimi. Ganges: A hybrid error injection + program analysis technique for hardware resiliency evaluation. Submitted for publication.

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

[25] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: computing control regions in linear time. *SIGPLAN Not.*, 29:171–185, June 1994.

[26] Man-Lap Li, Pradeep Ramachandran, Rahmet Ulya Karpuzcu, Siva Kumar Sastry Hari, and Sarita V. Adve. Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults. In *International Symposium on High Performance Computer Architecture*, 2009.

[27] Man-Lap Li, Pradeep Ramachandran, Swarup Sahoo, Sarita Adve, Vikram Adve, and Yuanyuan Zhou. Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults. In *International Conference on Dependable Systems and Networks*, 2008.

[28] Man-Lap Li, Pradeep Ramachandran, Swarup Sahoo, Sarita Adve, Vikram Adve, and Yuanyuan 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.

[29] Manlap Li. *SWAT: Designing Resilient Hardware by Treating Software Anomalies*. PhD thesis, University of Illinois, Urbana Champaign, 2009.

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

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

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

[33] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4), 2005.

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

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

[36] 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.

[37] Jun Nakano, Pablo Montesinos, Kourosh Gharachorloo, and Josep Torrellas. ReVive I/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *International Symposium on High Performance Computer Architecture*, 2006.

[38] Nithin Nakka, Giacinto Paolo Saggese, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. An Architectural Framework for Detecting Process Hangs/Crashes. In *European Dependable Computing Conf*, 2005.

[39] Shuou Nomura, Matthew D. Sinclair, Chen-Han Ho, Venkatraman Govindaraju, Marc de Kruijf, and Karthikeyan Sankaralingam. Sampling + DMR: Practical and Low-overhead Permanent Fault Detection. In *International Symposium on Computer Architecture*, 2011.

[40] 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.

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

[42] Karthik 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.

[43] 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.

[44] Andrea Pellegrini, Kypros Constantinides, Dan Zhang, Shobana Sudhakar, Valeria Bertacco, and Todd Austin. CrashTest: A Fast High-Fidelity FPGA-based Resiliency Analysis Framework. In *ICCD*, 2008.

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

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

[47] Pradeep Ramachandran, Siva Kumar Sastry Hari, Man-Lap Li, and Sarita V. Adve. Hardware Fault Recovery for I/O Intensive Applications. Submitted for publication.

[48] Vimal K. Reddy, Ahmed S. Al-zawawi, and Eric Rotenberg. Assertion-Based Microarchitecture Design for Improved Fault Tolerance. In *ICCD*, 2006.

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

[50] George Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *Proc. of Intl. Symp. on Code generation and optimization*, Washington, DC, USA, 2005. IEEE Comp. Society.

[51] George Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *International Symposium on Computer Architecture*, 2005.

[52] George Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Software-Controlled Fault Tolerance. *ACM Transactions on Architecture and Code Optimization*, 2(4), 2005.

[53] Swarup Sahoo, Man-Lap Li, Pradeep Ramchandran, Sarita V. Adve, Vikram Adve, and Yuanyuan Zhou. Using Likely Program Invariants to Detect Hardware Errors. In *International Conference on Dependable Systems and Networks*, 2008.

[54] Smitha Shyam, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[55] Daniel 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.

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

[57] William N. Sumner and Xiangyu Zhang. Memory indexing: canonicalizing addresses across executions. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 217–226, 2010.

[58] Milos Prvulovic Josep Torrellas. ReVive: Cost-Effective Arch Support for Rollback Recovery in Shared-Mem Multiprocessors. In *International Symposium on Computer Architecture*, 2002.

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

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

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

[62] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, 1995.

[63] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *International Conference on Programming Language Design and Implementation*, pages 238–248, 2008.