Copyright
by
Chun-Kai Chang
2020

The Dissertation Committee for Chun-Kai Chang
certifies that this is the approved version of the following dissertation:

High-Fidelity Error Injection and Acceleration Techniques

Committee:

Mattan Erez, Supervisor

Earl E. Swartzlander

Nur A. Touba

Michael Orshansky

Michael B. Sullivan

High-Fidelity Error Injection and Acceleration Techniques

by

Chun-Kai Chang

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2020

To my family

Acknowledgments

This research would not have been possible without the support of my advisor, Mattan, who showed me how to conduct high-quality research and provided me the freedom and resources throughout the years. Whenever I went astray in research, it was Mattan who guided me to the right track.

I would also like to express my deepest appreciation to people in the LPH research group for the practical suggestions and invaluable insights into my research. Special thanks to Nick and Mike, who initiated the Hamartia project. Their implementations serve as solid building blocks for my research. Thanks also to Sangkug, who worked with me on this project, helped identify issues, and extended functionality of the tool. Also, I would like to recognize the assistance from Wenqi on our 2019 SC paper.

Additionally, I had great pleasure working with Guanpeng on evaluating the effectiveness of software detectors. Thanks should also go to Karthik, who made the collaboration possible.

Last but not least, I am deeply indebted to my family in Taiwan. They have been providing me encouragement and moral support. Thanks also to friends and teachers on the island that made me what I am today. Although we live in different time zones, we all know that we are still connected and have each other's back.

High-Fidelity Error Injection and Acceleration Techniques

Chun-Kai Chang, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Mattan Erez

As technology scales down, the likelihood of hardware errors that silently corrupt the results of applications is increasing. Evaluating the resilience of applications against hardware errors is thus of significant concern. Current evaluation techniques via error injection are either low-fidelity or inefficient in terms of using computing resources. This dissertation demonstrates that sophisticated integration of injectors across abstraction layers and novel sampling algorithms can significantly improve both the fidelity and efficiency. Specifically, this dissertation describes an open-source instruction-level error injector that generates high-fidelity hardware errors due to particle strikes and voltage droops. Two acceleration techniques, nested Monte Carlo and Injection-Point Overprovisioning, are proposed to speed up error injection campaigns by $1 - 2$ orders of magnitude. This dissertation also answers the question of when high-fidelity is needed to evaluate the impact of hardware errors on applications and the effectiveness of error detectors.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Tables **xi**

List of Figures **xii**

Chapter 1. Introduction **1**
1.1 Research Goals 3
1.2 Current Approaches 5
1.2.1 Hierarchical Injection 5
1.2.2 Error Pruning 6
1.3 New Approaches in This Dissertation 6
1.4 Contributions 7
1.5 Dissertation Structure 8

Chapter 2. Background **9**
2.1 Hardware Faults and Errors in Computer Systems 9
2.1.1 Fault Mechanisms 10
2.1.1.1 Particle Strikes 10
2.1.1.2 Voltage Droops 10
2.2 Reliability Evaluation Methodologies 12
2.2.1 Hardware-Based Methodologies 12
2.2.2 Software-Based Methodologies 13
2.2.3 Monte Carlo Methods 13
2.2.3.1 Inefficiency of Simple Random Sampling 14
2.3 Current Error Models 15
2.3.1 Random Bit-Flipping 15

2.3.2 Previous Value 15
2.3.3 Random Value 16

Chapter 3. Evaluation Methodology 17
3.1 Hamartia: An Open-Source Error Injection and Detection Suite 17
3.1.1 Motivation 17
3.1.2 Error Injection and Analysis Flow 19
3.1.3 Features 19
3.1.4 Implementation 21
3.1.4.1 Instruction-Level Injectors 21
3.1.4.2 The Error Context API 22
3.1.4.3 RTL Gate-Level Injectors 22
3.2 Experimental Setup 23
3.2.1 Injection Outcome Classification 23
3.2.2 Testbed 24

Chapter 4. Error Models for Particle Strikes and Acceleration with Nested Monte Carlo 25
4.1 Background and Motivation 25
4.1.1 Fault Propagation and Masking 26
4.1.2 The Modeling Gap of Existing Error Models 27
4.1.3 Characterization of Hidden Soft Errors 28
4.2 A High-Fidelity Error Injector for Particle Strikes 30
4.3 Nested Monte Carlo 32
4.4 Evaluation Methodology 35
4.4.1 Detector Models 35
4.4.2 Error Models 36
4.4.3 Benchmarks and Output Quality 38
4.5 Experimental Results 39
4.5.1 Validation of Nested Monte Carlo 39
4.5.2 Injector Overhead 42
4.5.3 Reliability Outcome Distributions 45
4.5.4 Impact of Hardware Residue Checkers 47

4.5.5 Impact of Error Models and Detector Models on Application Output Quality 49
4.5.6 Impact of Error Models on Resilience Overhead 53
4.6 Summary 54

Chapter 5. Error Models for Voltage Droops and Acceleration with Injection-Point Overprovisioning 55

5.1 Background and Motivation 55
5.1.1 Timing Errors in Digital Circuits 56
5.1.2 Impact of Timing Errors on Computer Systems 58
5.1.3 Existing Timing Error Models 58
5.2 A High-Fidelity Error Injector for Voltage Droops 59
5.2.1 The Instruction-Level Injector 59
5.2.2 The Gate-Level Injector 61
5.2.2.1 The Fault Driver 61
5.2.2.2 Modeling Temporal Masking 62
5.2.2.3 Modeling Logical Masking 63
5.2.3 Limitations 64
5.3 Injection-Point Overprovisioning 65
5.4 Evaluation Methodology 71
5.4.1 Error Models 71
5.4.1.1 High-Fidelity Timing Error Models 71
5.4.1.2 Experimental Settings 75
5.4.2 Benchmarks 76
5.5 Experimental Results 76
5.5.1 Injection-Point Overprovisioning 76
5.5.2 Injection Outcome and Analysis 78
5.5.2.1 Instruction-Level Error Patterns 78
5.5.2.2 Application-Level Impact 81
5.5.3 Sensitivity Studies 83
5.6 Summary 84

Chapter 6. Effectiveness of Software-Based Error Detectors on High-Fidelity Errors **85**
6.1 Motivation 85
6.2 Compiler IR-Level Instruction Duplication 86
6.2.1 LLVM IR-Level Instruction Duplication 87
6.2.2 Evaluation Methodology 88
6.2.3 Experimental Results 91
6.3 Application-Level Error Detection 94
6.3.1 CLAMR 94
6.3.2 HeatDist 96
6.4 Summary 99

Chapter 7. Summary and Concluding Remarks **100**
7.1 Broad Applicability 101
7.2 Hardware Errors Beyond This Research 101

Bibliography **103**

List of Tables

1.1 Open-Source Error Injectors. 5

4.1 Circuits used in the RTL error model, their pipeline stages, and the fault-masking probabilities of injecting a logic gate and a latch, respectively. The reported masking rates are averages of all applications in this work. 38

4.2 Benchmark, input, and injection overhead per experiment. . . 39

4.3 Benchmark output quality metric. 39

4.4 Illustration of the normalization process for comparing simple random sampling (*SRS*) with nested Monte Carlo (*NestedMC*). 41

4.5 Ratio of dynamic instructions protected by residue checkers. . 50

4.6 p-values of Chi-squared and Kolmogorov-Smirnov tests for application output quality between error models. Bold fonts represent output qualities are significantly different. 51

5.1 Prior work on timing error modeling. 59

5.2 Circuits and their fault injection overhead. 72

5.3 Benchmarks to evaluate timing errors 76

6.1 Benchmarks and their input. 91

6.2 SDC ratios under full IR-level instruction duplication. 93

6.3 SDC ratios of HeatDist without protection, with the detector using temporal prediction, and with the detector using spatial prediction. 99

List of Figures

2.1 Simple random sampling 14

4.1 Particle strikes that are not modeled by high-level error injection. 27
4.2 Distribution of bit-flip count at circuit output with RTL-level particle-strike model. 29
4.3 Correlations between bit-flip positions with RTL-level particle-strike model 29
4.4 Nested Monte Carlo 32
4.5 Validation of nested Monte Carlo 40
4.6 Speedup of Hamartia over PINFI 43
4.7 Benefits of nested Monte Carlo 44
4.8 Particle-strike injection outcome distributions 46
4.9 Comparison of bit-flip count 47
4.10 Coverage of residue checkers 48
4.11 Particle-strike injection outcome distributions with residue checkers 48
4.12 Application efficiency due to checkpointing overhead 53

5.1 An example timing error caused by a voltage droop. 57
5.2 Instruction-level timing error injector 60
5.3 Example binary snippet and error configuration for timing error injection 61
5.4 Per-stage timing fault injector. 62
5.5 Injection-point overprovisioning 66
5.6 Analytical speedup of injection-point overprovisioning 70
5.7 Timing error groups for a 3-stage pipeline 74
5.8 Experimental speedup of injection-point overprovisioning . . . 77
5.9 Distributions of the number of flipped bits at circuit output due to voltage droops 78

5.10 Distribution of bit-flip positions at circuit output with input from each benchmark under the ML error group. Note that these are not distributions of positions with timing violation. . 80
5.11 Injection outcome distributions of voltage droops 81
5.12 Sensitivity studies of voltage droops 83

6.1 Instruction duplication 87
6.2 Protection curves (x-axis: protection level; y-axis: SDC coverage). 92
6.3 Outcome distributions of CLAMR 95
6.4 Outcome distributions of HeatDist 97

Chapter 1

Introduction[1]

Due to technology scaling and the need of energy efficiency, hardware is increasingly susceptible to various run-time fault sources that may lead to *errors*, which corrupt architectural state. However, the impact of hardware errors on applications varies. Some errors have no impact on some applications, some lead to system crashes, and others, which are known as silent-data corruptions (SDC), silently corrupt the results of applications. Hence, understanding the application-level effects of hardware errors is crucial for evaluating the reliability of computer systems.

The error-injection Monte Carlo methodology is widely used to quantify the impact of hardware errors on applications and the effectiveness of detectors designed to catch errors. This methodology requires thousands of application runs for sufficient accuracy. Each run injects one error by perturbing the state at some level of the system (e.g., the transistor level, the gate level, the micro-architectural level, the instruction level, or the application level) and then application-level impact is observed. The statistically expected impact is

[1] Part of this chapter appears in [1]. The author of this dissertation is the main contributor of the idea, implementation, and evaluation. The other coauthors in [1] assist development of the idea and implementation.

derived by analyzing injection results.

As compute resources are limited, to obtain results in a reasonable amount of time, researchers heavily rely on instruction-level error injection. It models errors by corrupting an instruction of the application under test. Compared with low-level injection (e.g., injecting to flip-flops at the gate level) that requires significant compute resources for hardware simulation, instruction-level injection is faster by five orders of magnitude [2].

However, existing instruction-level injection methodologies have a modeling fidelity problem. Ideally, the way that instructions are corrupted should match the effects of actual hardware errors. Errors generated at the gate level are high-fidelity as they are close to realistic hardware errors. In contrast, as shown later in this dissertation, existing error models at the instruction level (e.g., randomly flipping a bit of an operand) are low-fidelity because they do not take operand values into account.

This dissertation demonstrates that it is possible to use high-fidelity errors, which model particle strikes and voltage droops, at the instruction level and still maintain the higher injection speed of prior tools. The key enablers are hierarchical injection and novel sampling algorithms. Hierarchical injection combines a faster high-level injector with another detailed injector that is launched on demand to generate high-fidelity errors. The novel sampling algorithms accelerate the entire injection process by $1-2$ orders of magnitude while keeping sampling quality equal.

The aforementioned error injection framework and acceleration techniques are bundled into the Hamartia hardware error analysis suite[2] that is open-source and available to the resilience community. Hamartia enables rapid evaluation of software-based error detectors against realistic hardware errors. This research conducts case studies to evaluate the effectiveness of compiler-level and application-level error detectors at mitigating SDCs.

1.1 Research Goals

A desirable instruction-level error injector should generate high-fidelity error patterns to model various hardware errors without impacting the efficiency of reliability evaluation. Since existing injectors are either low-fidelity or inefficient, this dissertation develops a high-fidelity instruction-level error injection methodology along with novel acceleration techniques in order to meet the following goals and obtain new insights (e.g., understanding the effects of fidelity on experimental results).

High-fidelity error injection: Existing instruction-level injectors commonly rely on simple error models (e.g., single-bit flips) even though it is already known that such low-fidelity models do not match error patterns in actual hardware [3, 4, 2]. An ideal injector should generate error patterns that are as close to realistic hardware errors as possible.

Rapid resilience evaluation: Evaluating the resilience of applications using

[2]Available at `https://lph.ece.utexas.edu/users/hamartia`

low-fidelity models is already time-consuming and requires significant compute resource (thousands of core-hours per application). Improving error fidelity must not exacerbate the evaluation overhead. Furthermore, as applications and software-based error detectors are evolving rapidly, the error injection methodology should be fast enough to keep up with them.

Open-source evaluation framework: Modern hardware development heavily relies on proprietary tools and so does detailed error injection methodology. Ideally, the parallelism of error injection experiments should not be artificially limited by the number of tool licenses. It is thus beneficial to have a resilience evaluation framework that is independent of proprietary tools and publicly available to the community. In addition, open sourcing makes scientific results reproducible and expedites the development of novel research ideas.

Comprehensive error modeling: Transient hardware errors result from various run-time fault sources, each having distinct impact at the system level. Existing instruction-level injectors assume particle strikes as the error model [5, 6, 7, 8, 9, 1], though researchers have pointed out that circuit timing uncertainty also poses challenges for low-power system reliability [10, 11, 12]. It is desirable for an injector to encompass as many error models as possible for evaluating different types of errors.

Table 1.1: Open-Source Error Injectors.

	Injection Level	High Fidelity	Acceleration Techniques	Beyond Particle Strikes	Platform
LLFI [13]	LLVM IR				LLVM
KULFI [14]	LLVM IR				LLVM
P-FSEFI [6]	Emulation				QEMU
gem5-Approxilyzer [15]	Micro-arch		✓		x86
FAIL* [16]	Micro-arch, binary				x86, ARM
PERSim [17]	Gate level	✓		✓	OpenRISC (FPGA)
Chiffre [18]	Gate level	✓			RISC-V (FPGA)
Hamartia	Gate level, binary	✓	✓	✓	x86

1.2 Current Approaches

Current resilience studies employ *hierarchical injection* to enhance error fidelity with low overhead and *error pruning* to accelerate instruction-level error injection. However, none of the existing open-source tools meet the above goals simultaneously (Table 1.1).

1.2.1 Hierarchical Injection

Although low-level error injection (e.g., at the gate level) leads to high error modeling fidelity, it is too slow to obtain results in a timely manner (five orders of magnitude slower than instruction-level injection [2]). To strike a balance between injection speed and fidelity, previous work proposes hierarchical injection where a faster injector invokes another detailed injector just in time [19, 20, 21, 22]. However, using hierarchical injection alone is not fast enough because many injected errors are masked within hardware without affecting the software layers, which significantly increases the number of Monte Carlo trials and the total evaluation time. This research introduces

new acceleration techniques to speed up resilience evaluation.

1.2.2 Error Pruning

Several acceleration techniques are proposed to prune error-injection paths that are unimportant [23, 24, 25]. However, they are only applicable when errors propagate to the micro-architectural level or to the instruction level. As described later in this research, many hardware errors are masked at the circuit level or the gate level without corrupting architectural state. Thus, additional techniques orthogonal to error pruning are needed for high-fidelity error injection.

1.3 New Approaches in This Dissertation

This dissertation not only adopts hierarchical injection to enhance error fidelity but also proposes two novel sampling algorithms to accelerate high-fidelity error injection at the instruction level. The general idea of these new algorithms is to save evaluation time by embedding multiple Monte Carlo trials per application run while keeping sampling quality equal. In addition to high-fidelity error models for particle strikes, this research also develops high-fidelity error models for voltage droops. All of these error models and acceleration techniques are bundled within an open-source resilience evaluation suite named *Hamartia*,[3] which meets the above design goals.

[3]Hamartia means *to err* in Greek.

Key discoveries from evaluation include: (1) without detectors, single-bit flips are a good approximation of high-fidelity errors resulting from particle strikes, (2) existing low-fidelity error models do not represent errors caused by voltage droops, and (3) software-based detectors can effectively detect errors caused by particle strikes but not voltage droops.

1.4 Contributions

To summarize, the objectives of this research are: (1) increasing error modeling fidelity, (2) reducing the number of application runs while keeping sampling quality equal, and (3) evaluating the effects of modeling fidelity on experimental results. This work makes the following contributions:

- This research develops Hamartia, an open-source hardware error analysis suite with high fidelity and low overhead.
- This research enhances Hamartia's modeling fidelity for particle strikes and voltage droops.
- This research introduces two novel sampling algorithms, nested Monte Carlo and injection-point overprovisioning, which speed up error injection by $1-2$ orders of magnitude while keeping sampling quality equal.
- This research evaluates the impact of errors on applications and the effectiveness of software-based error detectors at mitigating SDCs.

1.5 Dissertation Structure

Chapter 2 reviews the basic concepts of hardware transient errors and resilience evaluation techniques. Chapter 3 introduces Hamartia, an open-source and high-fidelity error injection and detection suite used throughout this research. Chapter 4 develops error models for particle strikes in tandem with an acceleration technique called nested Monte Carlo. Chapter 5 proposes error models for voltage droops along with another acceleration technique called injection-point overprovisioning. Chapter 6 evaluates the effectiveness of various software-based error detection techniques using error models developed in this research. Finally, Chapter 7 summarizes this research and sheds light on future directions of error injection and resilience studies.

Chapter 2

Background

Before describing the contributions of this research, some fundamental concepts are reviewed in this chapter. Section 2.1 introduces relevant terminology and assumptions in this dissertation. Section 2.2 surveys the different methodologies for resilience evaluation. Section 2.3 summarizes the simple yet low-fidelity error models widely used in resilience studies at the instruction level.

2.1 Hardware Faults and Errors in Computer Systems

This research denotes *faults* as physical events that affect hardware components. If a fault eventually changes the architectural state, it becomes an *error*. The focus is *transient (soft) errors* in particular since they are random and transient in nature and thus hard to detect. In contrast, permanent faults that frequently lead to errors are typically detected once they occur. As modern computers usually protect memory with error checking and correcting codes (ECC), this research focuses on transient errors resulting from faults in arithmetic and logic units. Furthermore, *this research is concerned with the impact of errors, but not the rate of errors.* This is because the fault rates for

current technologies are well-studied [26, 27, 28, 29], and the fault rates for future technologies are largely unknown.

2.1.1 Fault Mechanisms

This dissertation specifically discusses two types of fault mechanisms: particle strikes and voltage droops. Both mechanisms are expected to occur more frequently due to technology scaling and low-power design.

2.1.1.1 Particle Strikes

Energetic particles from the environment can interact with a sensitive node in a micro-electronic device and cause electron-hole pairs that perturb the state of the affected node until a new value is written. The major contributors are high-energy neutrons from cosmic rays [30]. Although these faulty events rarely occur, modern processors are sensitive to particle strikes as they contain billions of transistors. This is the fault mechanism assumed by most prior work on instruction-level error injection [5, 6, 7, 8, 9, 1]. Both memory components and combinational logic are susceptible to particle strikes. Chapter 4 in particular discusses the modeling of particle strikes.

2.1.1.2 Voltage Droops

Voltage variations (or droops) include static IR drops and dynamic $\frac{dI}{dt}$ droops [10]. The former is the result of current (I) passes through resistance (R) in the power distribution network. The latter is caused by abrupt changes

in switching activity that induce transient changes to current and voltage. As voltage fluctuates, circuits can experience timing violation and thus some flip flops may latch a wrong value. Since the magnitude of IR drops are static throughout the execution of applications, this dissertation focuses on $\frac{dI}{dt}$ droops.

Processors typically have pre-determined operating points (in terms of clock frequency, supply voltage, and temperature) that are known to be safe. Additionally, sufficient design margins (or guardbands) are added to account for various variations in the field such as process, voltage, and temperature variations, ensuring that the processor functions correctly even in the worst cases. Prior work has shown that such guardbands account for about 18% of total node power [31, 32].

A new design strategy known as better-than-worst-case (BTWC) design has emerged [33]. The rationale behind BTWC design is that in typical scenarios the processor can run with reduced guardbands without generating any errors, resulting in higher energy efficiency. As for non-typical cases, errors are either avoided with error prediction [34, 31] or corrected with detection-recovery mechanisms [35, 36, 37]. Avoidance-based BTWC design has its limits and can still lead to errors. On the other hand, existing correction-based BTWC designs conservatively correct all detected errors without exposing errors to the software stacks. However, studies have shown that a variety of applications are resilient to errors and suitable for more aggressive BTWC design [38, 39]. For instance, exposing errors to applications can lead to sig-

nificant energy savings (10% to 50%) at little accuracy cost [39]. Evaluating the impact of voltage droops on applications can help identify applications or code regions that are resilient to errors and lead to improvements for BTWC design. Chapter 5 specifically discusses the modeling of voltage droops.

2.2 Reliability Evaluation Methodologies

In this section, existing resilience evaluation techniques are summarized. Since much prior work assumes particle strikes as the fault model, evaluation methodologies for particle strikes are described first, followed by methodologies for evaluating voltage droops.

2.2.1 Hardware-Based Methodologies

An example methodology is beam testing, which exposes electronic devices to a heavy-ion radiation environment in order to quantify the rate and the impact of radiation effects on applications [40, 9]. This methodology is highly accurate and offers high fidelity because real devices are used. The analogous methodology for evaluating timing errors is directly undervolting or overclocking commercial processors [41, 42, 43, 44]. Disadvantages of this methodology include high cost, high experimental setup time, limited availability of testing equipment, and an inability to precisely control the fault locations of interest. Evaluating software detectors using this methodology is thus cumbersome. Also, the results are specific to the devices under test. It is impractical for each application developer to test their application susceptibility and detector

effectiveness.

2.2.2 Software-Based Methodologies

Software is used to perform fault/error injection at various abstraction layers. These methods can target specific layers of interest with different fault/error models. For instance, faults/errors can be injected into applications [9], the operating system [45, 46], the architecture level [5, 13, 8], the micro-architecture level [47, 25], RTL gate level [19, 2, 1], etc. Software-based methods are more flexible because the injection substrate is decoupled from the fault/error models such that tools can be repurposed to model different types of faults.

However, software-based methods have a fidelity problem because: (1) they can only inject faults/errors into the layer(s) modeled by software, and (2) the fault/error model assumed may differ from reality. As a result, prior work adopts hierarchical injection to balance injection speed and accuracy by integrating injectors at different levels [20, 19, 1]. The tool developed in this research is of this type.

2.2.3 Monte Carlo Methods

Transient faults are assumed to occur randomly and uniformly during execution because they depend on a large number of factors including the dynamics of the system and variations of the operating environment. To evaluate the resilience of applications against transient errors, the Monte Carlo method

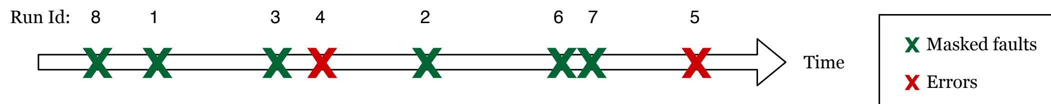

Figure 2.1: An example to show the inefficiency of simple random sampling.

(or sampling) is necessary because the error space is enormous.

2.2.3.1 Inefficiency of Simple Random Sampling

The conventional sampling algorithm known as *simple random sampling* is embarrassingly parallel, yet is inefficient in terms of collecting errors that are observable at the instruction level. Figure 2.1 shows an example of simple random sampling in which each application run is provided with one random fault injection point (a random dynamic instruction instance for instruction-level error injection). Faults that are masked at the circuit level or the gate level are marked in green, while errors that propagate to the instruction level are in red. In this example, eight application runs end up collecting only two observable errors. In reality, error injection experiments need to collect thousands of errors for statistical significance. Note that the higher the fault masking rate at the lower levels (i.e., beneath the instruction level), the more Monte Carlo trials are needed, which significantly increases evaluation time and enormously wastes resources. To reduce the number of application runs required for error injection campaigns, two novel acceleration techniques are proposed in this dissertation.

2.3 Current Error Models

Existing error models used at the instruction level are low-fidelity in general to allow rapid evaluation. This dissertation compares the impact of high-fidelity error models developed in this research with the current models summarized below.

2.3.1 Random Bit-Flipping

This type of models corrupts the instruction affected by a particle strike by flipping bits of an operand randomly. Single-bit flips are widely used in prior work because of their simplicity. The fidelity loss of single-bit flips results from the fact that particle strikes can corrupt multiple bits at the architectural level in a complex and application-dependent manner (discussed in Chapter 4).

2.3.2 Previous Value

This model is used by prior work to model the effect of voltage droops [39]. It assumes that when a droop occurs, all output pins of the affected circuit experience timing violation and thus latch the output value of the previous instruction using the same execution unit. The fidelity loss results from the fact that the impact of voltage droops on instructions is dependent on the application, the circuit, and its operating condition (discussed in Chapter 5).

2.3.3 Random Value

This model replaces the value of an instruction operand with a random value, possibly mimicking the worst-case errors. The fidelity loss results from the fact that it is data-independent.

Chapter 3

Evaluation Methodology

In order to increase error modeling fidelity, to reduce the number of runs while keeping sampling quality equal, and to evaluate the effects of fidelity on experimental results, this chapter introduces the Hamartia tool and the methodology for experiments. Section 3.1 describes Hamartia, an open-source tool for hardware error analysis.[1] Section 3.2 presents the common experimental setup used in this research.

3.1 Hamartia: An Open-Source Error Injection and Detection Suite

Section 3.1.1 motivates the creation of Hamartia. Section 3.1.2 describes the error injection and analysis flow. Section 3.1.3 introduces the key features of Hamartia. Section 3.1.4 describes the implementation details.

3.1.1 Motivation

Need for high-fidelity error models at the instruction level: Simple error models (e.g., single-bit flips) are currently used in resilience studies

[1] Available at `https://lph.ece.utexas.edu/users/hamartia`

at the instruction level and higher levels. On the other hand, prior work that evaluates the impact of high-fidelity errors on applications either does not model value dependency (i.e., error patterns are fixed across applications [48, 39, 49, 50]) or relies on special hardware platforms such as FPGAs [48, 2]. As shown later in this research, the impact of particle strikes and voltage droops on an application is determined by the values of the affected instruction(s). Since the number of FPGAs can artificially limit the parallelism of error injection experiments, it is thus desirable to model value dependency and to be able to perform resilience analysis on a general-purpose computer.

Need for low-overhead evaluation for high-fidelity errors: It is already time-consuming to perform error injection at the instruction level, let alone enhance error fidelity. Hamartia incorporates two novel acceleration techniques enabling high-fidelity error injection with low overhead.

Need for unified interface for instruction-level error injection: The resilience community lacks a common, shared interface for instruction-level error injection. Currently, error models and detector models have to be reimplemented for new projects. Hamartia addresses this issue by creating a shared interface which allows existing instruction-level injectors to integrate high-fidelity error models developed in this research and in the future.

Need for hassle-free error injection at scale: To take advantage of the embarrassingly-parallel property of the Monte Carlo method, error injection experiments need to be conducted on large-scale systems. Hamartia

is equipped with a set of scripts which automatically submit error injection experiments to the job scheduler of a large-scale system. Hamartia is an all-in-one suite that allows high-fidelity and low-overhead error analysis at scale. The design of Hamartia adopts modern software engineering practices for portability, extensibility, and usability.

3.1.2 Error Injection and Analysis Flow

The overall injection flow of Hamartia consists of three phases: *profiling*, *injection*, and *analysis*. As explained in Section 2.2.3, transient errors are assumed to affect each instruction with equal likelihood for modeling uniform-random particle strike times and dynamic variations in digital circuits. The profiling phase is used for this purpose and derives the upper bound of dynamic instructions in the program. The injection phase uses the Monte Carlo method to evaluate the impact of errors at the application level. In each Monte Carlo trial, the injector chooses a random instruction instance (based on the profile), injects an error into that instruction, and logs the behavior of the affected application. The analysis phase classifies the injection result for each Monte Carlo trial and generates a reliability report, which includes statistics such as SDC ratios and visualization results for further analysis.

3.1.3 Features

High-fidelity, low-overhead injection at the binary level: Hamartia performs error injection at the binary level because it results in the most ac-

curate results for instruction-level error injection. High-fidelity errors are generated with hierarchical injection. Overheads incurred by error injection are minimized through the novel sampling algorithms introduced in this research and other acceleration features of the underlying dynamic binary instrumentation tool (Section 3.1.4.1).

Error models and detector models: Hamartia implements a plethora of error models, including simple ones introduced in Section 2.3 and high-fidelity error models for particle strikes (Chapter 4) and voltage droops (Chapter 5). Additionally, several detector models such as arithmetic residual checkers [51, 52] are included.

Automatic injection and analysis pipeline: Hamartia takes only one input from the user: an error configuration file which specifies the binary of the application under test, the error model, and the detector model (if any). Error injection experiments are automatically submitted to the job scheduler. Once results are ready, Hamartia runs user-specified analysis scripts and generates a report.

Reusable and extensible design: The core of Hamartia is an Application Programming Interface (API) which allows error models and detector models to be used by other injectors even written in different languages.[2] Object-oriented programming is widely adopted in the design of Hamartia for better manageability and extensibility.

[2] Hamartia currently supports C++ and Python.

3.1.4 Implementation

For high error fidelity, Hamartia adopts hierarchical injection in which an instruction-level injector invokes a specific RTL gate-level injector at a specific instruction instance.[3] The RTL gate-level injector performs just-in-time fault simulation at the register-transfer level with input from the instruction-level injector. An error context API enables communication between the instruction-level injector and the gate-level injector.

3.1.4.1 Instruction-Level Injectors

The implementation of the instruction-level injector is based on Pin, a dynamic binary instrumentation tool [53], which allows modification to the architectural state of a dynamically-instrumented x86 application at runtime.[4]

Pin (and dynamic binary instrumentation in general) is selected for the following reasons: (a) it is much faster than micro-architecture-level simulation and lower-level injectors since instrumentation can be disabled after the injection point to run at native speed, (b) it is more accurate than compiler IR-level injectors [8], and (c) it allows injection into specific binary and source code regions. Using Pin, it is still possible to map injected instructions back to the program source lines (i.e., directly pointing out which lines are problematic).[5]

---

[3] The invoked injector depends on the error model specified in the error configuration.

[4] Although the Pin-based implementation limits the usage to x86 platforms, the framework itself can be easily generalized to the others.

[5] The application needs to be compiled with debugging information.

The design of the instruction-level injectors are object-oriented. The base injector is able to inject a single-bit flip to the output operand of a random instruction instance (as in prior work). Developers can inherit the base injector to develop new injectors. For instance, two injectors are implemented in this research to model particle strikes and voltage droops, respectively.

3.1.4.2 The Error Context API

One of the main challenges of integrating an RTL gate-level injector with a higher-level injector is software compatibility across abstraction layers. To solve the compatibility problem, a generic error context API is designed to carry essential information regarding a dynamic instruction instance. Some of the crucial components are: (1) instruction type, which can be used to select the target circuit for fault injection at the gate level, (2) input operands, which are used to drive the circuit, (3) the error-free output operands, which can be used to determine if an injected fault propagates to the ouptut of the circuit (necessary for the acceleration techniques proposed in this research), and (4) potentially corrupted output operands, which are filled by the gate-level injector and might be the same as the error-free output if the injected fault is masked.

3.1.4.3 RTL Gate-Level Injectors

The gate-level injector performs just-in-time fault simulation based on the error context built by the instruction-level injector. There are two gate-

level injectors developed in this research: one models transient faults due to particle strikes and the other models faults due to voltage droops. The details are presented in Chapter 4 and Chapter 5, respectively. When the just-in-time fault simulation finishes, the gate-level injector updates the potentially-corrupted output operands in the error context and returns the context to the instruction-level injector.

Each RTL gate-level injector is wrapped into a set of Python modules. The actual simulation is done by Icarus Verilog, an open-source Verilog simulation tool [54]. With the cross-language error context API, gate-level injectors can be easily integrated with other instruction-level injectors as well [14, 13, 6, 55, 56].

3.2 Experimental Setup

This section explains how the application-level injection outcomes are classified and the testbed on which the experiments are conducted in this research.

3.2.1 Injection Outcome Classification

Unless otherwise noted, the analysis phase classifies all injection outcomes into three primary categories:

- **Masked**: the injected error is masked by application, with output iden-

tical or similar to the error-free run.[6]

- **Detected Uncorrectable Error (DUE)**: errors that crash or hang the program are categorized as DUE_{crsh}. Errors that result in obviously erroneous application output (e.g., output is not finite or mismatch in matrix size) are also in this category and denoted as DUE_{test}.

- **Silent Data Corruption (SDC)**: the program ends normally with output errors that are hard to detect.

3.2.2 Testbed

A machine that runs OpenSUSE 42.3 on an Intel i5-6500 CPU with 16GB DRAM is used to develop tools and techniques proposed in this research. Once functionality is verified, error injection experiments are deployed on the Lonestar5 supercomputer at TACC (Texas Advanced Computing Center).

[6] The error tolerance of output is application-dependent.

Chapter 4

Error Models for Particle Strikes and Acceleration with Nested Monte Carlo[1]

This chapter develops high-fidelity error models for particle strikes along with an acceleration technique called nested Monte Carlo which speeds up evaluation by orders of magnitude. Section 4.1 describes the background and motivation of developing high-fidelity models. Section 4.2 presents the design of the high-fidelity injector for particle strikes in Hamartia. Section 4.3 introduces the nested Monte Carlo technique. Section 4.4 and Section 4.5 compare the results of error injection using high-fidelity errors vs. current low-fidelity ones.

4.1 Background and Motivation

Section 4.1.1 describes the propagation of transient (soft) errors due to particle strikes within digital circuits. Section 4.1.2 explains the modeling gap between existing simple error models and realistic soft errors. Section 4.1.3 shows the error patterns of realistic errors at the instruction level to motivate

[1]Part of this chapter appears in [1, 57]. The author of this dissertation is the main contributor of the idea, implementation, and evaluation. The other coauthors in [1, 57] assist development of the idea and implementation.

the need for high-fidelity models for particle strikes.

4.1.1 Fault Propagation and Masking

Soft errors that affect storage elements and combinational logic are triggered by random radiation events such as particle strikes. Only those strikes that are energetic enough can potentially lead to data corruption; otherwise, they are said to be electrically masked. When a strong particle strike hits a memory component (e.g., latches and flip-flops), the data is corrupted until new data is written into the component. On the other hand, if a logic gate is hit, a transient pulse known as a *single-event transient* is created. If the faulty signal (i.e., the single-event transient or the corrupted data signal from a memory component) reaches the next latching window, erroneous data can be written into a latch. The propagation process continues and might eventually corrupt data in architectural components such as the register file. At this point, the fault manifests as an error.

This research assumes that faults escape electrical masking and timing masking; that is, considered are particle strikes that carry sufficient charge and result in faulty signals that arrive on time at the next latch. However, logical masking is modeled (e.g., the faulty signal enters an AND gate with the other input being 0) because it depends not only on the circuit but also on the application. In other words, recall that *this research is concerned with the impact of errors, but not the rate of errors* (Section 2.1). Also, the focus is soft errors that affect execution units of the processor because errors occurring

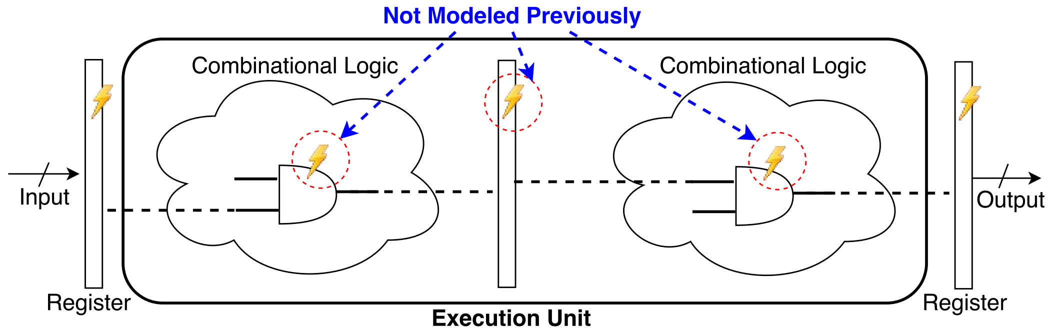

Figure 4.1: Particle strikes that are not modeled by high-level error injection.

within those circuits directly affect application data and thus more likely lead to SDC. The same is assumed in previous work [13, 14, 58].

4.1.2 The Modeling Gap of Existing Error Models

Consider a fault that occurs at a random location within a circuit module consisting of an input buffer, combinational logic, internal pipeline buffers, and an output buffer (Figure 4.1). Since the goal is to inject errors, this work assumes the circuit has inputs and/or outputs associated with architectural state (e.g., an ALU).

Note that soft errors can be grouped based on their initial fault site within the circuit. First, consider the faults that occur at either the input buffer or the output buffer. When a fault happens at the output buffer, it directly manifests as an error. On the other hand, when a fault occurs in the input buffer, it can be masked by the operation (e.g., erroneous bits are shifted out). Such errors can be modeled by bit flips because they either directly affect the output or logical masking can be modeled by running the operation with

erroneous inputs. These soft errors are already modeled in previous work via injecting errors into instruction operands [5, 13, 59, 60, 61, 8].

Next, consider the case where the fault site is at either a logic gate within combinational logic or internal pipeline buffers. Here the assumption is that the fault induces a pulse that flips the output of the affected unit. Although this faulty signal may be masked logically before propagating to output buffer, it is possible that it leads to a soft error that corrupts multiple bits of the output buffer. Because the exact impact of the soft error on the output buffer depends on the initial fault site, the circuit, and the input data vector, there is no corresponding simple architecture-level modeling for soft errors originating from these internal circuit nodes. These errors are termed *hidden soft errors* in this chapter. To quantify the impact of this modeling gap, gate-level fault injection is performed to study characteristics of these hidden soft errors. This gate-level injection is also called *RTL injection* because the *register-transfer level* (RTL) description of a circuit specifies the circuit's gates and latches.

4.1.3 Characterization of Hidden Soft Errors

Figure 4.2 shows the distribution of how many bits in the circuit output buffer are flipped if a fault from a circuit internal node is not logically-masked. On average, 78% of errors manifest as single-bit flips. Hence, the single-bit flip model does not accurately reflect 22% of errors, and these errors potentially cause more severe data corruption. Note that the distribution varies across

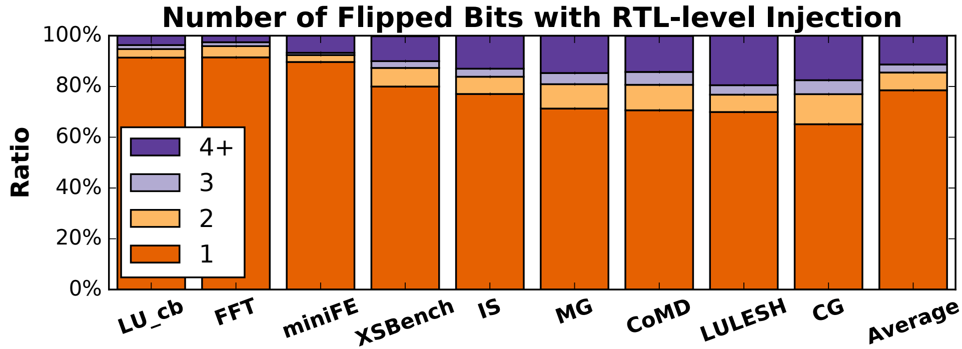

Figure 4.2: Distribution of bit-flip count at circuit output with RTL-level particle-strike model.

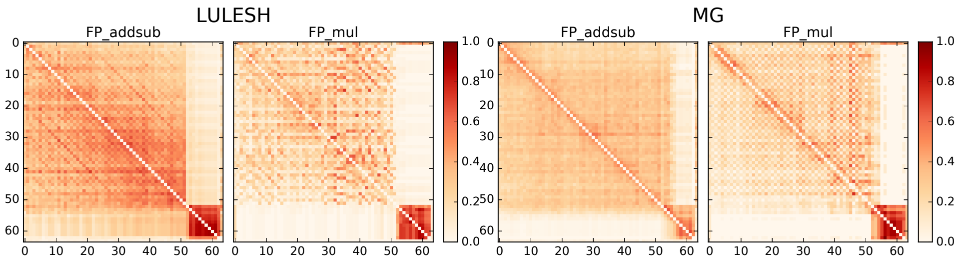

Figure 4.3: Correlations between bit-flip positions with RTL-level injection. Breakdown into two applications (`LULESH` and `MG`) and two circuits (64-bit floating-point adder and multiplier). Axes denote bit locations of circuit outputs.

applications since logical masking depends on circuit input.

Modeling multi-bit errors is challenging because of correlations between bit-flip positions at a circuit's output (Figure 4.3). First, correlations vary across circuits because the circuit's structure determines its logic operations, which in turn affect logical masking. Second, correlations are related to input data. For example, using input data from `LULESH`, the floating-point adder has strong correlations between bits in the exponent field (bit 52-62), while such phenomena are not observed with input from `MG`. This is because log-

ical masking depends on input data. Hence, not only do correlations vary across circuits, they also depend on input data and thus on applications. Such complex and data-dependent correlation is not modeled by existing random bit-flipping models.

Although realistic error patterns are different from single-bit errors at the instruction level, the impact on applications is still unknown due to application-level error masking. Thus, it is necessary to perform error injection to evaluate the end-to-end effects of hardware faults on applications. Next, the design of a high-fidelity instruction-level for modeling particle strikes is introduced.

4.2 A High-Fidelity Error Injector for Particle Strikes

The injector is developed on top of Hamartia introduced in Chapter 3. It is based on the same architecture for hierarchical injection, consisting of an instruction-level injector and a gate-level injector.

The instruction-level injector: The role of the instruction-level injector is to build the error context for the target instruction (Section 3.1.4), pass the error context to the gate-level injector, and then change the architectural state of the application based on the modified error context returned by the gate-level injector. Since the base instruction-level injector already implements the functionality of injecting a random dynamic instruction, the implementation of the instruction-level injector simply inherits the base injector without changes.

The gate-level injector: To model particle strikes at the gate level with low overhead, the implementation is split into two stages: pre-processing and runtime. In the pre-processing stage, an additional gate is inserted at each node of the circuit by modifying the RTL source code with Pyverilog [62]. For instance, to model a particle strike that flips a node's value, the circuit is augmented with XOR gates, each of which has one input connecting to an existing node and the other input as a trigger signal.[2] Note that the pre-processing stage is an one-time effort and is totally transparent to the end users.

The runtime stage consists of several steps. First, the tool selects a pre-processed circuit based on the instruction type in the error context. For instance, if the target instruction is an integer ADD, then a pre-processed integer adder is fetched. Next, gate-level simulation is performed using the circuit and the values of input operands in the error context. The tool randomly triggers an XOR gate (by setting its trigger signal to 1) to emulate a fault. Finally, the output of the circuit is written into the error context as the new output operands of the target instruction, and the error context is sent back to the instruction-level injector to modify the architectural state of the application.

Note that due to logical masking, the output of the circuit may still be

[2]The tool also supports modeling of stuck-at-0 and stuck-at-1 (by inserting AND gates and OR gates), but they are not evaluated in this research because the focus is transient errors instead of permanent errors.

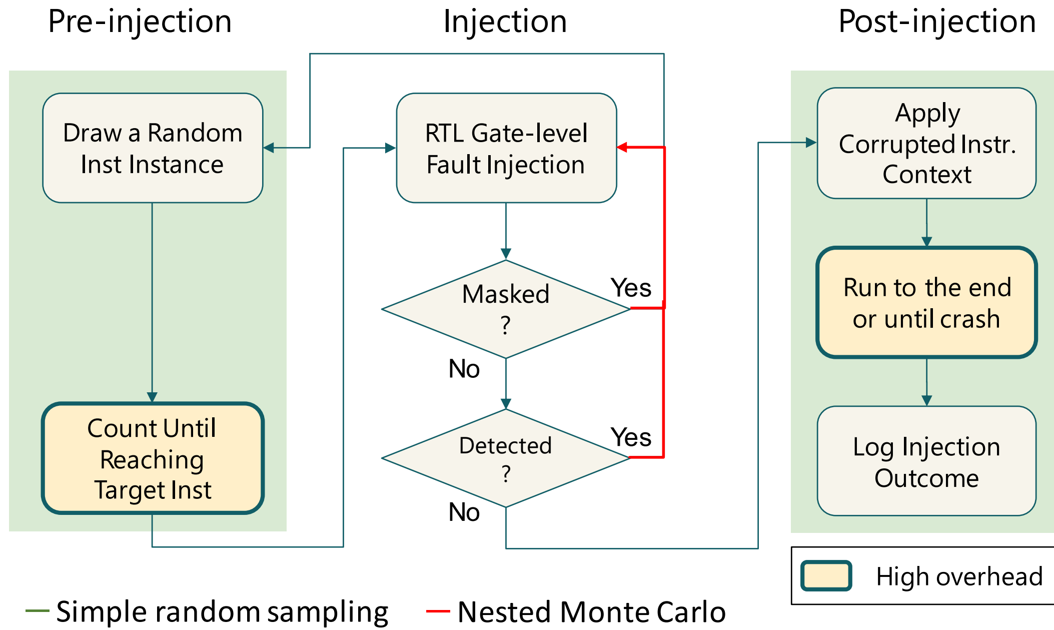

Figure 4.4: Error injection flow: Nested Monte Carlo vs. simple random sampling.

the same as the error-free operation, and thus the fault has no impact at the instruction level. The next section introduces a novel technique to increase the likelihood of generating realistic errors that propagate to the instruction level to accelerate evaluation.

4.3 Nested Monte Carlo

As explained in Section 2.2.3.1, the traditional simple random sampling is not efficient in terms of evaluating high-fidelity errors at the instruction level due to fault masking at lower levels. This section introduces a refined hierarchical injection methodology to boost error injection by orders of magnitude.

The error injection flow for simple random sampling is depicted in Figure 4.4. Note that the pre-injection and post-injection stages account for the majority of overheads (>95%). As a result, the idea is to use a form of nested Monte Carlo methodology. The outer portion follows the traditional instruction-level error injection campaign described in Section 3.1.2. However, within each Monte Carlo trial a nested Monte Carlo is performed at the gate level. In each outer iteration, the correct instruction (arithmetic circuit) output is first logged and then, a single random fault is repeatedly and immediately injected at the gate level until a fault manifests as an error that corrupts the output. By doing so, only an actual error that has not been logically masked is injected into the outer Monte Carlo trial, saving significant time by avoiding complete application runs when the outcome has already been determined. This novel injection flow is also shown in Figure 4.4, which also shows how the methodology is extended to include detectors.

Evaluation using simple random sampling is even worse when error detectors are introduced. Numerous Monte Carlo runs and RTL gate-level fault injections are necessary to generate faults that truly affect the application because good error detectors have high coverage and detect most errors that are not logically masked. As a result, faults are generated iteratively at the RTL level to filtering out detected errors. That is, errors that would be detected should also be pruned since the injection outcome is known at this point (*detected*). Thus, fault injection is repeated until an undetected error is generated to save time. Otherwise, similar to the cases of masked faults at the RTL level,

it is wasteful to restart the application for other injection points.

This filtering notion can be extended to resilience techniques at different abstraction layers by cascading error detectors starting from fine granularity to coarse granularity. For instance, only errors that are not detected by instruction-level hardware detectors are sent to fine-grained software state-recovery mechanisms [63]. Note that the longer the detector chain, the greater the savings that can be achieved.

An important aspect of this nested Monte Carlo methodology is that it requires multiple faults that propagate to actual errors to be identified in order to establish statistical bounds on the logical fault-masking rates for different instructions and applications. In that way one trial at the outer level is indeed equivalent to a flat methodology for the purpose of evaluating the logical masking rate and detector coverage. Note that a limit on injection attempts needs to be set to avoid practically infinite loop when the masking rate of the circuit is high. The selection of the limit can affect the tradeoff between evaluation time and accuracy. While the nested algorithm provides the same statistics as a traditional Monte Carlo, the specific instructions to which actual errors are injected will differ from a traditional injection campaign where each trial randomly selects an instruction. This nested methodology is fully equivalent to a traditional campaign despite this potential difference. The validation results are shown in Section 4.5.1.

4.4 Evaluation Methodology

The evaluation consists of six main parts: (a) verifying the proposed nested Monte Carlo methodology, (b) evaluating the benefits of nested Monte Carlo, (c) evaluating the impact of the error model on the reliability of applications, (d) evaluating the impact of error detection on the reliability and error types of applications, (e) evaluating the impact of the error model on application output quality in the context of HPC scientific applications, and (f) evaluating the impact of the error model on the overhead of an overall resilience scheme (specifically, checkpoint-restart).

Before discussing the evaluation results, this section describes the error models, the detector models, and the applications studied in this work.

4.4.1 Detector Models

To study the impact of hardware detectors on application resilience, arithmetic residue checkers are chosen. They are shown to provide high coverage for errors from execution units with relatively low cost [51, 52], and are also adopted by commodity processors (e.g., POWER6 [64]) and previous work on hardware resilience design [65, 66].

Residue checkers detect errors by comparing output of the arithmetic unit with that of a relatively low-cost datapath. The checking can be described by Equation 4.1 where $\oplus$ denotes integer addition, subtraction, or

multiplication and $|x|_m$ denotes $x \ mod \ m$.

$$|a \oplus b|_m \stackrel{?}{=} |\left(|a|_m \oplus |b|_m\right)|_m \tag{4.1}$$

It is assumed that no errors affect the final equality checker, so these residue checkers have perfect coverage for single-bit datapath errors. Specifically, implementations include a residue checker with modulus 3 and another with two moduli, 3 and 5. The latter provides higher coverage because an error is not detected only when both checks fail.

4.4.2 Error Models

Assume on-chip SRAM and system DRAM are protected by ECC and only inject errors to instructions using arithmetic and logic units. In each experiment, an error is injected into a random instruction's output operand with one of the four error models below.

- **Single-bit flip (RB1)**: randomly flips a single bit, as commonly done in prior work in the HPC community.
- **Double-bit flip (RB2)**: randomly flips two bits.
- **Random (RND)**: replaces the output with a random value, possibly mimicking worst-case errors.
- **RTL gate-level model (RTL)**: generates an error pattern using the methodology described in Section 4.2. *RTL-G* and *RTL-L* are used to

denote injection into gates only and latches only, respectively; latch injection is only done for pipelined floating-point units.

- **model+**: uses *model* on a design with a single-modulus residue checker where *model* is one of the error model described above.
- **model++**: resembles **model+** but with a double-modulus residue checker (i.e., stronger detection).

For the RTL model, gate-level netlists of integer and floating-point execution units are synthesized using Synopsys tools (Design Compiler and DesignWare Library) with the 45nm Nangate Open Cell Library, optimized for performance. Because the DesignWare Library does not include pipelined floating-point units, the register retiming feature of Design Compiler is used to pipeline the circuits. The pipeline stages are tuned to mimic those used by Intel Broadwell processors based on the latency data from [67]. Table 4.1 lists the circuits used in this work. Note that the synthesized circuits can be different from those designed and optimized for commodity processors, but this work has shown that they lead to errors different from single-bit errors at the instruction level.

Table 4.1: Circuits used in the RTL error model, their pipeline stages, and the fault-masking probabilities of injecting a logic gate and a latch, respectively. The reported masking rates are averages of all applications in this work.

Name	Stages	Gate Masking Rate	Latch Masking Rate
INT_add_sub	1	0.17±0.02	n/a
INT_mult	1	0.09±0.05	n/a
Shift	1	0.30±0.08	n/a
FP_add_sub	3	0.28±0.03	0.27±0.04
FP_mult	3	0.41±0.02	0.33±0.02
FP_div	10	0.49±0.09	0.54±0.12
FP_sqrt	1	0.46±0.10	n/a

4.4.3 Benchmarks and Output Quality

Evaluation is based on the serial version of 9 HPC benchmark programs and applications (Table 4.2).[3] For each application, 3000 injection experiments are performed, which ensures a margin of error <2% for a confidence level of 95% [68], on 10 combinations of error models and detector models. Injection outcomes are classified into three categories as in Section 3.2.1. Table 4.3 lists the output quality for each application. The same metric shipped with the application or suggested by previous work are adopted [69, 70].

[3] A more resilient IS is used in which an assertion is inserted at the end of `randlc()` to check if the output falls within $[0, 1]$.

Table 4.2: Benchmark, input, and injection overhead per experiment.

Program	Input	Native Time	Simple Time	RTL Time	RTL+ Time	RTL++ Time
FFT [71]	-m 16	0.01s	1.3s	1.7s	5.2s	19.1s
miniFE [72]	nx=18 ny=16 nz=16	0.04s	2.1s	6.0s	8.1s	12.1s
LU_cb [71]	default	0.04s	1.2s	1.8s	4.8s	11.2s
IS [73]	A	1.2s	3.0s	4.6s	7.4s	21.3s
CG [73]	A	1.2s	7.1s	7.8s	8.8s	10.9s
MG [73]	A	1.5s	11.9s	15.0s	18.0s	51.7s
CoMD [74]	default	5.7s	28.5s	29.1s	32.2s	52.4s
LULESH [75]	default	22.1s	103.7s	109.7s	112.3s	121.9s
XSBench [76]	small	30.9s	91.7s	92.5s	93.4s	96.3s

Table 4.3: Benchmark output quality metric.

Program	Quality Metric	DUE_{test} Criteria
FFT [71]	Rel-L2-Norm	Infinite values
miniFE [72]	Resid-Norm	Resid-Norm > 1
LU_cb [71]	MaxAbsDiff	Infinite values
IS [73]	n/a	Failed verification
CG [73]	Zeta	Infinite values
MG [73]	L2-Norm	L2-Norm > 1
CoMD [74]	Potential energy	Lost atoms, infinite energy, or potential energy > 0
LULESH [75]	Measure of symmetry (MaxAbsDiff)	[69]
XSBench [76]	n/a	n/a

4.5 Experimental Results

4.5.1 Validation of Nested Monte Carlo

To validate the statistical equivalence between the nested Monte Carlo and simple random sampling, the outcome distributions of each methodology are compared (Figure 4.5). Notice that any differences between the two methodologies, for all experiments we conducted with RTL injections are within the 95% confidence intervals of each experiment. In other words, the specific numbers obtained are not identical, but the 95% confidence intervals of the two methodologies overlap. Note that the nested approach has narrower (better) confidence intervals for a given number of trials because each outer Monte Carlo trial identifies multiple non-masked errors in the inner stage.

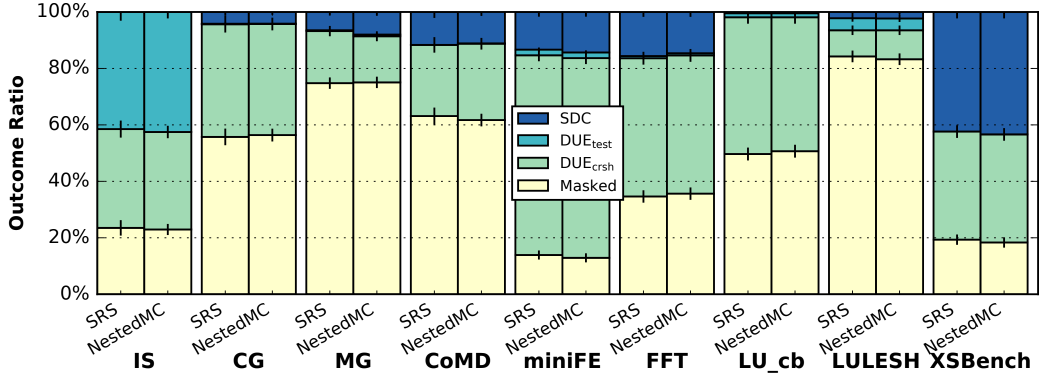

Figure 4.5: Validation of nested Monte Carlo against simple random samping in terms of outcome distributions. Error bars are 95% confidence intervals.

Evaluation of output quality shows similar statistical equivalence.

Note that there is a pitfall when comparing the RTL error injection results of nested Monte Carlo with simple random sampling: *the proportion of each instruction type among the error samples is different between the two methods because each circuit has different logical masking rate.* Nested Monte Carlo collects error samples following the instruction mix of the application since fault injection is repeated until an error is generated for each experiment, while simple random sampling collects more samples from circuits with lower masking rate since the experiments end as soon as the injected fault is masked. Thus, it is necessary to normalize the proportion of each instruction type among the error samples before comparing the injection results. Such normalization is performed in the aforementioned validation campaign.

Table 4.4 illustrates the normalization process. Suppose there are two types of instructions in a program: ADD and MUL. The instruction mixes

Table 4.4: Illustration of the normalization process for comparing simple random sampling (*SRS*) with nested Monte Carlo (*NestedMC*).

		ADD	MUL	Total
instruction type ratios	SRS	0.2	0.8	
I	$NestedMC$	0.6	0.4	
	$NestedMC/SRS$	3.0	0.5	
original outcome ratios	$Masked$	0.2	0.4	
II	DUE	0.5	0.2	
	SDC	0.3	0.4	
normalized outcome ratios	$Masked$	0.6	0.2	**0.23**
III	DUE	1.5	0.1	**0.46**
	SDC	0.9	0.2	**0.31**

obtained using simple random sampling and nested Monte Carlo are shown in region I. The original outcome distributions by instruction types measured with simple random sampling are shown in region II. The normalization process consists of two steps: (1) the outcome ratio of each instruction type is multiplied with the $NestedMC/SRS$ value, and (2) the outcome ratios are summed across instruction types and then normalized. The last column in region III shows the normalized outcome ratios (0.23, 0.46, 0.31 for Masked, DUE, and SDC, respectively).

Observation 1: *the nested Monte Carlo approach is equivalent to traditional injection campaigns, despite significant potential benefits in execution time.*

4.5.2 Injector Overhead

Overhead of various error models in Hamartia: Table 4.2 compares the execution time overheads of using the Hamartia injector with a simple bit-flipping error model, RTL injection, and RTL injection with detectors, to the time each application runs natively. Recall that instrumentation is disabled after the injection point (Section 3.1.4.1), so the injector overhead depends on the injection point (i.e., a dynamic instruction). Therefore, the table reports the average injector overhead, which corresponds to injection at the mid-point of program execution. Evaluation is performed on the testbed system described in Section 3.2.2.

There are three interesting observations. First, very short-running applications, which require a fraction of a second to run natively, incur a significant relative overhead for injection because starting up the Pin injector and invoking the error model have a fixed overhead of $1-2$s. Second, longer running applications incur a reasonable injection overhead with a slowdown of $3-5\times$ without detectors and $5-10\times$ with detectors, even when RTL injection is used. This is because the relative time to bring up the injection infrastructure is small relative to the execution time of these applications. Third, the overhead of RTL injection and detector evaluation varies between applications because the masking factor and detector coverage are data dependent—the higher the masking or coverage, the more iterations are required within the inner Monte Carlo step.

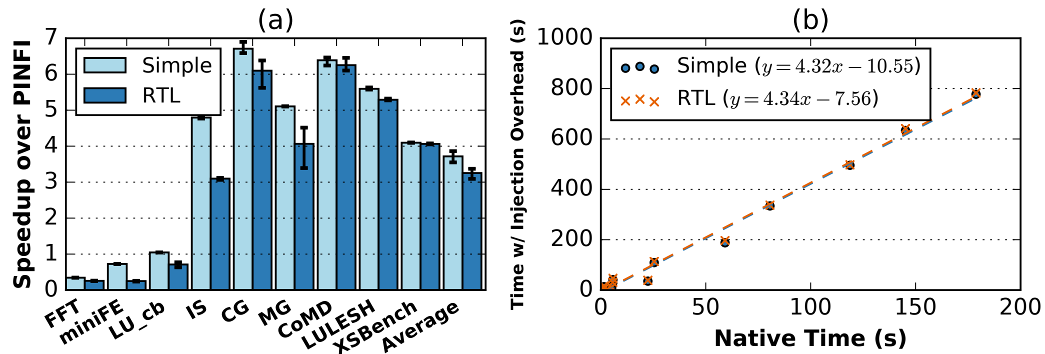

Figure 4.6: (a) Speedup of Hamartia over PINFI with simple error models (RB1, RB2, and RND) and the proposed RTL gate-level model. Error bars denote maximum and minimum speedups. (b) Scaling of per-experiment execution time as native time increases.

Overhead of Hamartia vs. the state-of-the-art: Figure 4.6(a) shows that the overhead of Hamartia, even with RTL injection, is actually lower than PINFI [13], which is one of the fastest injectors currently available. PINFI is chosen because recent work has shown that it has lower overhead than compiler-based injectors [8]. Hamartia outperforms PINFI even with RTL-level injection, except for applications that can finish in under two seconds. The performance loss is due to the additional features for injecting specific binaries and instructions, but the overhead is amortized for larger applications. On average, the injector developed in this research is faster than PINFI by a factor of 3.

Since full-scale HPC applications have much longer execution times, the tool's execution time vs. problem size is shown in Figure 4.6(b). The input size of `LULESH` is increased to mimic native execution time of larger HPC applications. Note that the overhead increases linearly due to instrumentation

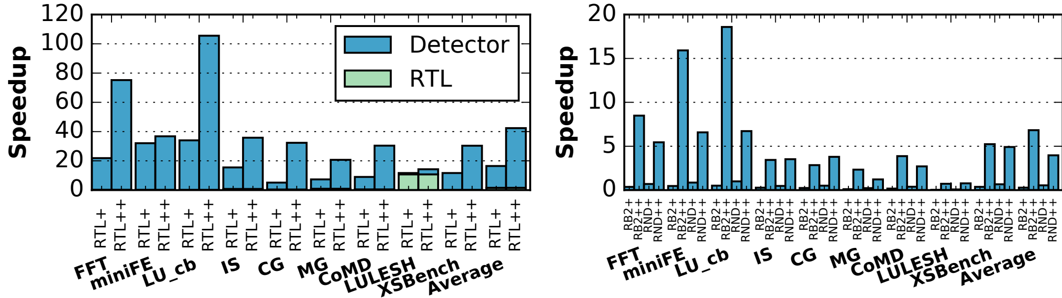

Figure 4.7: Execution savings (relative to actual runs) by nested Monte Carlo for RTL and injection-detection coupling. Left: RTL error model with residue checkers. Right: Simple error models with residue checkers.

overhead of the Pin tool. When the problem size is large enough, the difference in overhead between simple models and the RTL model is negligible (4.32X and 4.34X slowdown for simple and RTL model, respectively). The overhead can be further reduced by taking checkpoints during the profiling phase and starting each injection experiment at the checkpoint closest to the target injection point as in [47].

Benefits of nested Monte Carlo: Next, the execution savings of the error injection experiments are measured for each application. The savings are presented as the ratio of runs saved due to nested Monte Carlo: $\sum_{n=1}^{N} iter_n \times dyninst_n / total_dyninst$, where N is the number of injection experiments, $iter_n$ is the number of local iteration for the nth experiment, $dyninst_n$ is the dynamic instance number of the nth experiment, and $total_dyninst$ is the total dynamic instructions of the application. This work assumes the overhead of actual error injection at the injection site is small relative to the overhead of pre-injection and post-injection period, which are true for all applications that

are not very short. Figure 4.7 shows that the overall savings are larger when the RTL model is used since the injector keeps injecting faults until one manifests as an error. Also the savings increase with the strength of the detector used. Note that `LU_cb` has the highest savings due to most of the instructions can be protected by the residue checkers (Table 4.5). In contrast, `LULESH` has a small portion of instructions protected by residue checker but its savings at the RTL level is significant because of the input data result in high logical masking.

Observation 2: *with the nested approach, the overhead of injection, even when using detailed error models and high-coverage detectors, can be kept low and roughly match that of current injectors with simple error models.*

4.5.3 Reliability Outcome Distributions

RTL vs. simple error models: Figure 4.8 shows the outcome distribution of all combinations of error models without detectors. The most surprising result here is the small difference between RTL and RB1 for all applications. While the small confidence intervals demonstrate that the outcome distribution is sometimes statistically different, the absolute difference is very small. The largest difference in SDC ratios between the two models is only 4% (`XSBench`), and the difference in DUE ratios is also only 4% (`CG`). This shows that even though the two models are quite different at the instruction level (Section 4.1.3), the impact on applications is likely unimportant due to the large proportion of single-bit errors, correlated error patterns, and

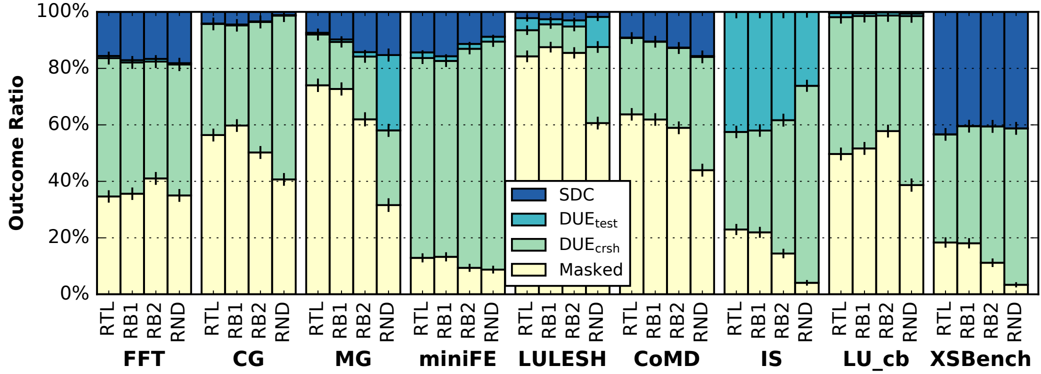

Figure 4.8: Injection outcome distributions (without detectors). Error bars are 95% confidence intervals.

application-level error masking. However, neither RB2 nor RND can closely approximate the RTL model because of aggressive (uncorrelated) bit-flips in these two models.

Gate soft error vs. latch soft error: Next, the impact of gate soft errors (i.e., a logic gate output is flipped) vs. latch soft errors (i.e., the output of a flip-flop is flipped) is studied. Results of six applications are reported here because compiled binaries for the others do not use pipelined circuits. The left of Figure 4.9 compares the distribution of bit-flip count at the instruction level between gate and latch soft errors. Notice that latch soft errors have more multi-bit patterns than gate soft errors. However, similar final impact on application resilience is observed for both error types (right of Figure 4.9) due to application-level error masking.

Observation 3: *without detectors, RB1 is a good approximation of RTL injection, whether experiencing gate or latch faults.*

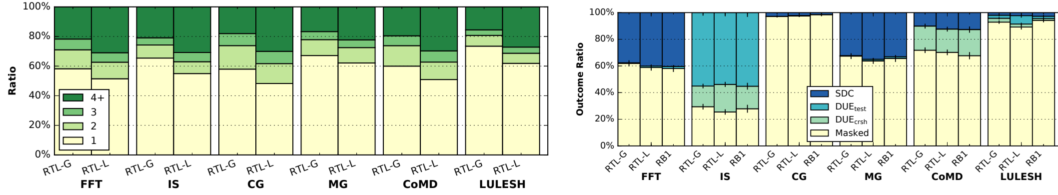

Figure 4.9: Comparison of bit-flip count distribution at the instruction level (left) and injection outcome distribution (right) between gate soft error and latch soft error. Only floating-point instructions are injected here because they are pipelined (i.e., having both gates and latches).

4.5.4 Impact of Hardware Residue Checkers

Coverage: Figure 4.10 shows the coverages (i.e., error detection rates) of the checkers with different error models applied. First, coverage is application-dependent when the RTL error model is used. The residue checker with a single modulus can detect 88% of errors on average, while the coverage increases to 97% with two moduli. Second, when RB2 is used, the coverage of the single-modulo checker is slightly application-independent, while that of the double-modulus checker has larger variation across applications. Finally, when RND is used, coverages of both detectors are nearly the same across applications but significantly lower than RTL model.

Outcome distributions: The impact of residue checkers on final outcome distribution is shown in Figure 4.11. The results only include the distribution of protected instructions (i.e., integer add/sub/mul instructions) so as to decouple protection ratio from the result. To account for the fact that each instruction is injected multiple times until an undetected error is generated, this work assumes a detected error triggers an exception that leads to DUE_{crsh}

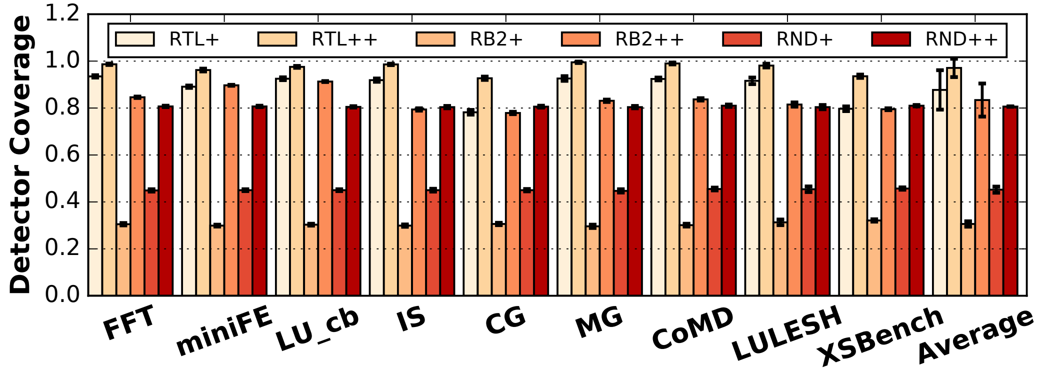

Figure 4.10: Coverage of residue checkers with different error models. Coverage for RB1 (not shown) is always 100%. Error bars are 95% confidence intervals.

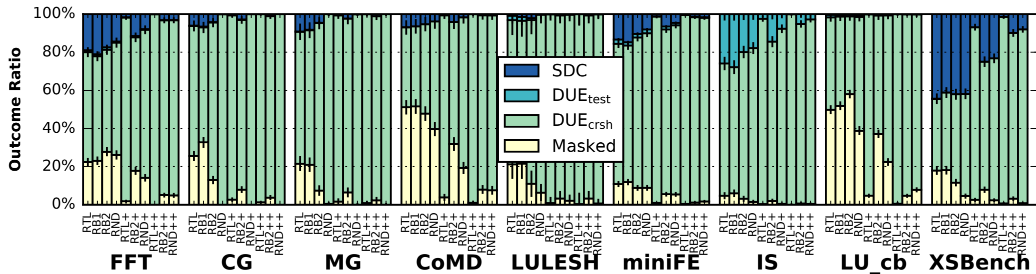

Figure 4.11: Injection outcome distribution for instructions protected by residue checkers (i.e., integer addition, subtraction, and multiplication operations). See Section 4.4.2 for the definitions of error models and detector models.

and thus compute the distribution as following. First, four empty bins corresponding to each outcome category are created. Next, for each of the 3000 injection outcomes, it contributes c_e to the DUE_{crsh} bin and $(1 - c_e)$ to the outcome category of the undetected error where c_e is the averaged detector coverage of error model e in Figure 4.10. In the end, the outcome distribution and confidence intervals are computed with data in the four bins.

Since residue checkers have perfect coverage for RB1 and good coverage

for the RTL model, the RTL model has larger improvement in SDC ratio compared with RB2 and RND. However, with the RTL model, the additional improvement in SDC ratio by adding another modulus to the residue checker is marginal because the coverage of the single-modulus residue checker is already high (except for `XSBench`).

It is important to note that SDC improvement due to residue checkers on *all* instructions is highly dependent on the protection ratio (Table 4.5). For example, although SDC ratio is improved by nearly 9% for protected instructions in `MG`, the actual SDC ratio improvement is only 2% as only 25% of instructions are protected. The injection outcome results can guide the adoption of software error detectors and how they should be tuned. When the hardware detector can detect most critical errors such that the resultant SDC ratio already meets the resilience target, the software detector is not necessary and thus performance would not be degraded. On the other hand, the software detector should be tuned to target those errors left by hardware detectors to minimize impact on performance.

Observation 4: *When detectors are introduced, the impact of the error model is large when the fraction of instructions protected is also large.*

4.5.5 Impact of Error Models and Detector Models on Application Output Quality

Perturbed application output quality due to SDC: Previous work demonstrates that reliability and output quality may be traded off (e.g., [77,

Table 4.5: Ratio of dynamic instructions protected by residue checkers.

Program	Ratio	Program	Ratio	Program	Ratio
FFT	0.78	miniFE	0.88	LU_cb	0.99
IS	0.38	CG	0.59	MG	0.25
CoMD	0.35	LULESH	0.09	XSBench	0.65

70]). With a bounded degradation of quality, the cost of protecting the application against soft errors can be reduced. Thus, the application output qualities of SDC cases between error models are compared. Two methods are used to objectively compare output quality degradation distributions from different error models. First, comparison is based on the quality metric of each SDC sample to the golden value to identify the most significant decimal position of the outcome error. A histogram of these positions is computed for each application and model. Then the histograms between each pairs of error models are compared using the chi-squared test, which is widely used to test the similarity of one set of binned data against another [78].

The second comparison method treats the perturbed quality of each error model as a continuous random variable. Thus, for each error model, the perturbed quality can be uniquely described as a cumulative distribution function (CDF). To measure similarity between two CDFs of continuous data, the Kolmogorov-Smirnov test is used.

For both hypothesis tests, the null hypothesis H_0 is no difference in output quality between a pair of error models; the alternative hypothesis H_1 is there is difference in output quality between error models. Thus, if the null

Table 4.6: p-values of Chi-squared and Kolmogorov-Smirnov tests for application output quality between error models. Bold fonts represent output qualities are significantly different.

Program	p-values (Chi-squared / Kolmogorov-Smirnov)				
	RTL vs. RB1	RTL vs. RB2	RTL vs. RND	RTL+ vs. RB1+	RTL++ vs. RB1++
FFT	0.87 / 0.36	**0.00** / **0.00**	**0.00** / **0.00**	**0.00** / **0.00**	**0.00** / **0.00**
miniFE	0.77 / 0.99	**0.03** / **0.01**	0.06 / 0.09	**0.00** / **0.00**	**0.00** / **0.00**
CG	0.05 / 0.66	0.74 / 0.22	**0.01** / 0.18	**0.00** / 0.16	0.21 / 0.50
MG	0.76 / 0.32	0.23 / 0.51	0.10 / 0.60	0.95 / 0.76	0.74 / 0.89
CoMD	0.74 / 0.69	0.38 / 0.49	**0.00** / 0.45	0.15 / **0.04**	0.91 / 0.33
LULESH	0.67 / 0.86	**0.01** / 0.07	0.05 / **0.00**	0.24 / 0.51	0.69 / 0.78

hypothesis is rejected, it indicates output quality differs significantly between the pair of error models. The significance level (α) is chosen to be 0.05. As a result, if the calculated p-value from a test is less than 0.05, the observed data rejects the null hypothesis. The results of both tests are shown in Table 4.6.

Both tests show that there is no significant difference in output quality between RB1 and RTL when there are no hardware residue checkers.[4] However, neither RB2 nor RND result in similar output quality as RTL. When residue checkers exist, the perturbed output quality of RTL is statistically different from bit-flipping models for applications in which the detectors protect a reasonable fraction of instructions. This is because multi-bit errors generated with RTL model may not be detected by residue checkers and thus affect the output quality.

Effective application output quality: So far this work only shows how

[4]These results do not mean that both RB1 and RTL lead to the same distributions of output quality for these applications.

error models and detector models affect application output quality *given an error always results in SDC*. Nonetheless, the effective output quality actually depends on SDC ratio and the coverage of the detector (if any). The effective output quality can be computed as

$$\begin{aligned} &(1-c) * ((1-P_{SDC}) * Mean_{errFree} \,+\, P_{SDC} * Mean_{SDC}) \\ &\qquad\qquad\qquad\qquad\qquad + c * (Mean_{errFree}) \end{aligned} \tag{4.2}$$

where c is detector coverage, P_{SDC} the SDC ratio of the error model, $Mean_{errFree}$ the expected output without error, and $Mean_{SDC}$ the mean perturbed output due to SDC. Here detected errors contribute to $Mean_{errFree}$ because they can be corrected either by checkpoint-restart, or by restarting the whole application from the beginning.

Using this metric, the findings are that: (1) both RB1 and RTL lead to very similar effective output quality because their SDC ratios and $Mean_{SDC}$ are similar, (2) residue checkers fail to improve effective quality of `miniFE` and `LULESH` because the $Mean_{SDC}$ is significantly different from $Mean_{errFree}$, and (3) the impact of error models on effective quality is negligible for the applications whose effective quality is improved by the residue checkers (e.g., RND++ and RTL++ have similar effective quality for `FFT`, `CG`, `MG`, and `CoMD`).

Observation 5: *While the specific output quality degradation of a specific run with detectors requires a high-fidelity error model, the overall impact on ensemble methods can be reasonably estimated with RB1.*

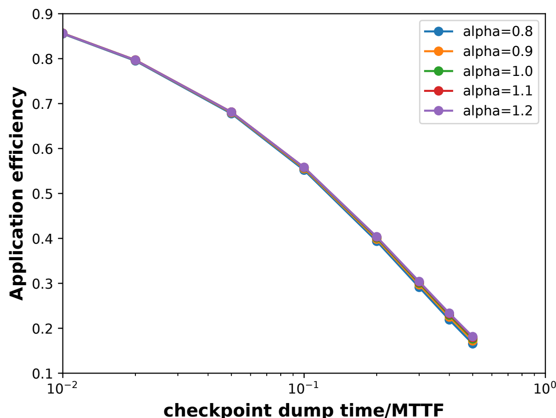

Figure 4.12: Application efficiency due to checkpointing overhead vs. the ratio of dump time to MTTF where *alpha* is the ratio of estimated MTTF to true MTTF. This implies that slight estimation error of DUE ratios has negligible impact on achieving minimal checkpointing overhead.

4.5.6 Impact of Error Models on Resilience Overhead

Recall that the DUE ratios obtained with RB1 and RTL are slightly different (max difference is 4% for `CG` in Figure 4.8). With the DUE ratios from error injection results, researchers can tune the checkpointing interval for minimal overhead incurred by checkpointing and rollback. To quantify the impact of tuning the checkpointing interval based on RB1 instead of the RTL model, the application efficiency is derived (i.e., the fraction of time for real computation as opposed to checkpointing) using the higher-order formula in [79, 80]. Note that the 4% difference of DUE ratios for `CG` (39.4% vs. 35.5%) leads to around 10% estimation error of the mean time to failure (MTTF). However, such difference has negligible impact on application efficiency (Figure 4.12). This is also true when the residue checkers are considered. The maximal difference of the DUE ratios between RB1 and RTL with detectors

is `XSBench` (100% for RB1+ vs. 90.5% for RTL+), which also leads to 10% estimation error for MTTF. Thus, tuning the workload-specific checkpointing interval through single-bit error injection is good enough.

Observation 6: *RB1 is sufficient for evaluating the overall performance efficiency impact of errors (on checkpoint-restart) whether detectors are used or not.*

4.6 Summary

This chapter develops high-fidelity error models for particle strikes along with an acceleration technique called nested Monte Carlo which speeds up evaluation by orders of magnitude. By demonstrating the outcome distributions, application output quality, and the impact on checkpointing overhead, single-bit errors remain a good approximation of realistic soft errors from arithmetic and logic circuits when hardware detectors are not considered. If hardware detectors are known to exist, a more realistic error model should be used to evaluate and tune the software resilience techniques.

Chapter 5

Error Models for Voltage Droops and Acceleration with Injection-Point Overprovisioning[1]

This chapter develops high-fidelity error models for voltage droops along with an acceleration technique called injection-point overprovisioning that speeds up evaluation by an order of magnitude. Section 5.1 describes the background of voltage droops and motivation of developing high-fidelity models. Section 5.2 presents the design of the high-fidelity injector for voltage droops in Hamartia. Section 5.3 introduces the injection-point overprovisioning technique. Section 5.4 and Section 5.5 compare the results of error injection using high-fidelity errors vs. current low-fidelity ones.

5.1 Background and Motivation

Section 5.1.1 describes the propagation of timing errors caused by voltage droops within digital circuits. Section 5.1.2 explains the impact of timing errors on modern computer systems. Section 5.1.3 discusses the capabilities of

[1]Part of this chapter appears in [81]. The author of this dissertation is the main contributor of the idea, implementation, and evaluation. The other coauthors in [81] assist evaluation of the idea.

existing timing error models.

As defined in Section 2.1, *faults* denote physical events that affect hardware components. If a fault eventually changes architectural state, it becomes an *error*.

5.1.1 Timing Errors in Digital Circuits

Timing faults result from variations in supply voltage, temperature, or device characteristics that change circuit timing. As a result, some inputs may take too long to propagate through the circuit and lead to an error when a corresponding output flip-flop latches a not-yet-stable value. In a sequential circuit, a timing fault manifests as a timing error when at least one flip-flop latches an incorrect value and it eventually perturbs architectural state. This process is complicated because it depends on the variation's magnitude and duration, the circuit, its operating condition (including clock frequency, voltage, temperature, etc.), and the sequence of input values to the circuit; both current inputs and previous inputs are important. *This work targets timing errors as a result of voltage droops.*

Although the circuit structure determines the delay of each path, input values control which paths are toggled (or *sensitized*). For example, in Figure 5.1, given the input pair, it is the red path that determines the timing of the endpoint E1, while the other paths are *false paths* because their timing is irrelevant given the inputs. Only variations that are strong enough can significantly delay the toggled paths such that old values or glitches (interme-

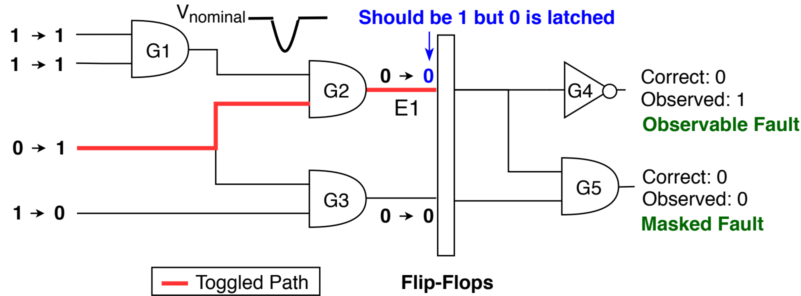

Figure 5.1: An example timing error caused by a voltage droop.

diate values) are latched by flip-flops at the circuit endpoints. However, old values are not always different from the correct values. As a result, whether an instruction is affected by a timing fault depends on the input history of the circuit as well (i.e., circuit inputs at previous cycle).

Whether faults become errors also depends on *logical masking*. For instance, in Figure 5.1, even though the flip-flop of E1 fails to latch the new value produced by gate G2, the fault only affects the output of G4 but not G5 in the subsequent stage. This is because the other input to G5 is a controlling value (in this case, 0). Thus, high-fidelity modeling of timing errors in pipelined circuits needs to consider logical masking as well.

Timing errors may affect multiple instructions and corrupt multiple bits in output operands. For instance, consider variations that persist across multiple cycles or those that impact multiple pipeline stages simultaneously. Note that this is different from radiation-induced errors, which usually affect a single instruction. For a detailed discussion of timing errors, see [10, 12].

5.1.2 Impact of Timing Errors on Computer Systems

Modern systems apply substantial voltage and/or frequency guardbands in hardware to ensure that timing errors never occur, even at the worst operating condition. However, guardbands waste energy and performance because typical use cases do not lead to timing errors. Prior work has shown that such guardbands can account for about 18% of total node power for IBM POWER7 [31] and ARMv8 systems [32].

Researchers have been advocating cross-layer techniques to overcome the power constraints of future systems without expensive guardbanding [82, 12, 83]. Nonetheless, due to dynamic variations (e.g., voltage droops and temperature fluctuations) and shared power delivery networks in many-core processors [84, 85, 12], the likelihood that timing errors occur in large-scale systems may become non-negligible. Technologies such as near-threshold voltage computing are even more susceptible to timing errors [86]. Thus, a methodology to evaluate the resilience of applications against timing errors is highly desirable.

5.1.3 Existing Timing Error Models

Table 5.1 summarizes prior work on modeling timing errors. Features of a high-fidelity error model include whether it is value-dependent, whether it models logical masking for pipelining, and whether the operating condition is tunable. Since no prior work is comprehensive, this research develops high-fidelity timing error models available to the research community.

Table 5.1: Prior work on timing error modeling.

	Publicly Available	Value Dependent	Logical Masking	Frequency Tunable	Voltage Variation	Temperature Variation	Process Variation
SWAT-Sim [19]		✓	✓	✓			
VARIUS-NTV [87]	✓			✓	✓		✓
CrashTest [48]	✓		✓	✓			
b-HiVE [49]	✓	✓			✓		
CLIM [88]		✓		✓			
Constantins et al. [50]		✓		✓	✓		
This work	✓	✓	✓	✓	✓	✓	*

*This is achievable by enhancing the underlying timing analysis tool with the capability of statistical timing analysis.

5.2 A High-Fidelity Error Injector for Voltage Droops

This injector is developed on top of Hamartia which was introduced in Chapter 3. It is based on the same architecture for hierarchical injection, consisting of an instruction-level injector and a gate-level injector to accelerate injection and still generate high-fidelity errors. At the injection point, the instruction-level injector provides the gate-level injector with: (1) the instruction type (used to determine which circuit to simulate) and (2) a pair of input vectors to the circuit for the previous and current cycle. A user-provided error configuration supplies the voltage droop profile (i.e., magnitude and duration). The gate-level timing error injector then returns a potentially corrupted output to the instruction-level injector. Figure 5.2 shows the overall flow of the timing error injector.

5.2.1 The Instruction-Level Injector

As mentioned in Section 5.1.1, the manifestation of a timing error depends on the circuit's computation history. As a result, the role of the

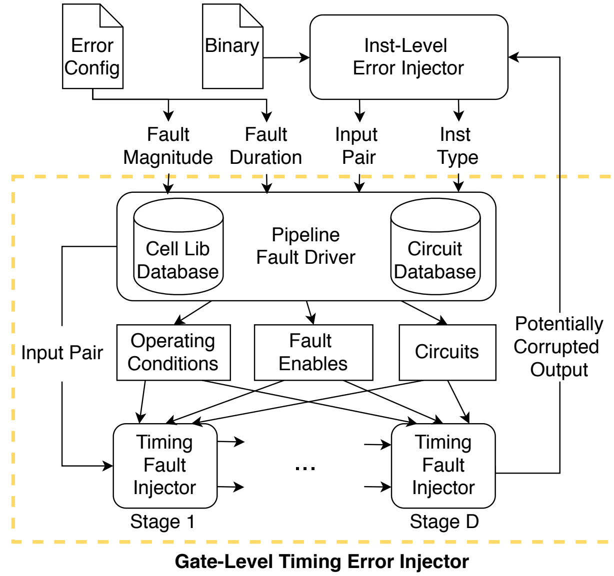

Figure 5.2: Instruction-level timing error injector for a pipeline of depth D. Each timing fault injector is the per-stage injector in Figure 5.4.

instruction-level injector is to collect the data necessary for reproducing circuit state. The workflow of the injector is explained with the example in Figure 5.3. The instruction instances affected by a timing fault are called *victim instructions*. Specifically, the injector logs: (1) input pairs, which are input operands of victim instructions and those of the previous instructions that utilized the same circuit and (2) instruction types of those instructions (e.g., ADD, MUL, etc.). For the example in Figure 5.3, the input pair is `((r1,r2), (r4,r5))` and the instruction type is floating-point addition. These data are used to build the error context which is passed to the gate-level injector along with the error configuration.

```
// binary snippet
fpadd r1, r2, r3
  // r3 = r1 + r2 (previous inst)
fpadd r4, r5, r6
  // r6 = r4 + r5 (victim inst)
```

```
// error config
voltage: "0.85V"
frequency: "400MHz"
duration: "single_cycle"
occupancy: "low"
```

Figure 5.3: Example binary snippet and error configuration.

5.2.2 The Gate-Level Injector

This injector produces the potentially corrupted output operand for each victim instruction. The core consists of a fault driver and multiple timing fault injectors. Each fault injector models temporal masking in one pipeline stage. The fault driver prepares inputs for the fault injectors which are chained to model logical masking across stages.

5.2.2.1 The Fault Driver

Based on the instruction type, it first looks up the circuit database to determine the circuit used by the victim instruction. If the circuit's pipeline depth is D, then D fault injectors are chained together. Next, the driver prepares input for each fault injector. It decides the operating condition of the circuit based on the error configuration. It also generates fault-enable signals to control whether a stage should be injected or not. These fault-enable signals are determined based on the fault duration and pipeline occupancy (explained in Section 5.4.1.1). Once all input to fault injectors are ready, the driver launches simulation of the first stage and waits for the result of the last stage.

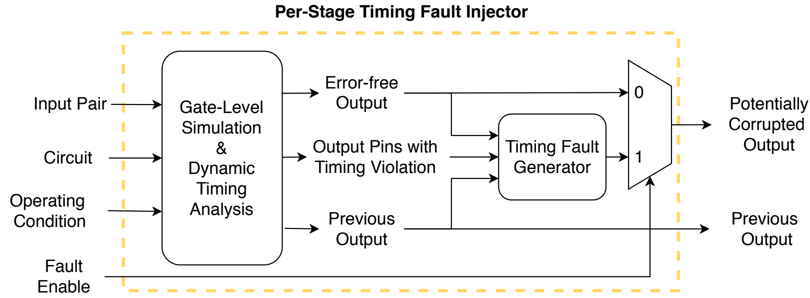

Figure 5.4: Per-stage timing fault injector.

For the example in Figure 5.3, the victim instruction uses the floating-point adder, which has three pipeline stages in the evaluation of this research (see Table 5.2). Therefore, three fault injectors are chained together. Since the user specifies that the droop decreases the voltage to 0.85V, the driver fetches the characterization corner for that voltage level from the cell library database. Assuming the victim instruction is in the third stage when the single-cycle droop occurs, only the fault-enable signal to the third stage should be set (i.e., the results of first two stages are error-free). Finally, the input pair, `((r1,r2), (r4,r5))`, is sent to the first stage and the simulation begins. The output of the first stage becomes the input to the second stage, and so on. In the end, the fault driver sends the potentially corrupted output of the last stage back to the instruction-level injector, which modifies the value of `r6` accordingly.

5.2.2.2 Modeling Temporal Masking

Figure 5.4 shows how to model temporal masking within each pipeline stage. To this end, those endpoints (i.e., input of flip-flops or circuit's output

pin) experiencing a timing violation must be identified given the input pair to the circuit and its operating condition. Dynamic timing analysis is performed to determine which endpoints encounter a timing violation using the method proposed by Cherupalli and Sartori [89]. The idea is to remove false paths before running static timing analysis (STA) such that only paths with gates toggled in the specific cycle determine signal arrival times at the endpoints (e.g., the red path in Figure 5.1). The false paths are derived by parsing the value change dump (VCD) generated by gate-level simulation. Although the original evaluation in [89] uses a proprietary STA tool, this research uses OpenTimer, an open-source STA tool [90]. For gate-level simulation, the open-source Icarus Verilog simulator [54] is used.

Once the output pins encountering a timing violation, the error-free output of the victim instruction, and error-free output of the previous instruction that used the same circuit are identified, the potentially corrupted output operand for the victim instruction (the timing fault generator box in Figure 5.4) can be derived. For example, if the error-free output of the victim instruction is 0010, the output of previous instruction is 1111, and all bits except the least-significant bit encounter a timing violation, then the corrupted output operand is 1110.

5.2.2.3 Modeling Logical Masking

Logical masking is modeled by chaining fault injectors. If a fault is injected at one stage, it may be logically masked by one of its following stages.

Each pipelined circuit is preprocessed by splitting it into individual stages using Pyverilog, a hardware design processing toolkit for Verilog HDL [62]. The functionality of the resultant circuits are verified to ensure that the circuit transformation does not break functionality. This is a one-time procedure and it is completely transparent to the user. At runtime, the fault driver provides the circuit of each pipeline stage to each corresponding fault injector.

5.2.3 Limitations

This research makes two assumptions: (1) the magnitude of variations is constant within each cycle, and (2) the execution order of instructions follows the program order. This is not always true for processors with out-of-order execution, which require detailed modeling of their micro-architecture for full fidelity. For such a study, the gate-level injector can be integrated with injectors at the micro-architectural level [47, 25]. Metastability is not modeled because the estimated mean time between metastability events (using the models in [91]) is larger than 10^{40} years even for the worst operating condition evaluated in this work.[2] Although the evaluation focuses on timing errors as a result of voltage droops, the tool can be used to evaluate timing faults due to overclocking or temperature fluctuations as well.

[2]If the user specifies very small cycle time or very strong droop, the tool reports the mean time between metastability events based on existing models and warns the user if the mean time between metastability is lower than 10^5 years (i.e., metastability rate is higher than 0.1 FIT).

5.3 Injection-Point Overprovisioning

This research assumes that timing faults occur randomly and uniformly during execution because timing errors depend on a large number of factors that include dynamics of the system and variations of the operating environment. To evaluate the resilience of applications against timing errors, the Monte Carlo method is adopted because the error space is enormous. However, as explained in Section 2.2.3.1, it takes a large number of Monte Carlo trials to collect statistics on errors that do affect applications. Although hierarchical injection helps reduce per-trial evaluation time, the overall evaluation is still slow because the number of trials remains large.

The nested Monte Carlo method proposed in Section 4.3 is not applicable for injecting timing errors hierarchically. This is because particle strikes usually affect only one component (or just a few components) in the circuit at a time. If a fault is masked, nested Monte Carlo can inject a fault to another component. On the other hand, timing errors affect many components in the circuit simultaneously (all components that share the same power delivery network in the cases of voltage droops). As a result, if a timing fault is masked, there are no other sites to inject a fault into at the same instruction.

This dissertation introduces another novel sampling technique called Injection-Point Overprovisioning. This technique is based on the observation that masked faults do not change the architectural state (i.e., are invisible to applications). Unlike simple random sampling which provides only one injection point per application run, injection-point overprovisioning provides mul-

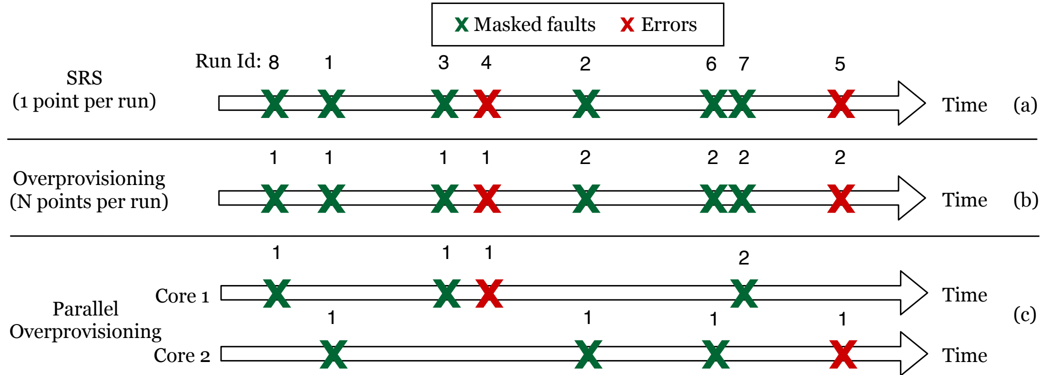

Figure 5.5: The proposed injection-point overprovisioning technique vs. simple random sampling. (a) simple random sampling, (b) injection-point overprovisioning, and (c) an example of applying injection-point overprovisioning on two cores in parallel. Green crosses denote masked faults and red ones are errors that change the architectural state.

tiple potential injection points, sorted by injection time. If a fault is masked at an early injection point, the next point is evaluated until an error is generated or until all injection points for the run are exhausted. Once injection succeeds in manifesting an architecturally-visible error, all remaining points are passed to the next run to ensure that the sampling is unbiased.

Figure 5.5b illustrates the idea of injection-point overprovisioning with the same set of injection points as shown in Figure 5.5a for simple random sampling. Injection-point overprovisioning reduces the number of applications runs from 8 to 2 in this example. Note that injection-point overprovisioning increases the injection overhead per run because it performs multiple injections. However, as long as the per-run injection overhead is negligible compared to the overhead of running the application itself, the overall evaluation time will be improved. An analytical model of the benefit is discussed later in this

chapter.

Injection-point overprovisioning is easily generalized to machines with parallel architectures and still reaches the same level of parallelism as simple random sampling. Figure 5.5c shows an example of parallel overprovisioning on two cores with the same set of injection points. Note that dependencies only exist between application runs that execute back-to-back on the same core, while runs on different cores are independent. A synchronization mechanism among cores is needed to terminate the entire error injection campaign when it collects enough errors to claim statistical significance, but this synchronization is infrequent.

Implementation: This research demonstrates an example implementation of injection-point overprovisioning with the master-worker parallel pattern. The master thread is responsible for generating and updating the batch of injection points associated with each core and for synchronizing worker threads upon termination.

Recall that the overall injection flow of Hamartia consists of three phases: *profiling*, *injection*, and *analysis* (Section 3.1.2). The profiling phase simply determines the range of injection points, while the injection phase does fault/error injection with the injection-point overprovisioning technique.

In the injection phase, the master thread initializes each batch with N points randomly drawn within the range determined in the profiling phase. It then launches a worker thread for each core that performs fault injection

to the specified application. Next, each worker thread injects faults at points specified in its batch until an error is generated or until the batch is exhausted, and then records the remaining points (if any). When a worker thread on core C_i is done, the master thread updates the batch associated with core C_i by either passing the remaining points or generating a new batch of points. Once the total number of collected errors reaches the target, the master thread waits until all worker threads finish before terminating the injection phase.

The analysis phase examines the injection result for each application run and generates statistics such as the injection outcome distributions. Note that the efficiency of the proposed technique depends on the batch size N and other factors.

Analytical Modeling: The goal is to understand when injection-point over-provisioning outperforms simple random sampling. To this end, a first-order analytical model for the cost of generating an error relative to application execution time is derived.

Assume that at each injection point, the fault masking rate is an i.i.d. Bernoulli random variable with masking-probability P. Let α be the injection overhead normalized to the execution time of the application. The cost of using simple random sampling is then formulated as in Equation 5.1. The first term is the cost of injecting a fault for each application run and the second term is the mean number of fault injections (equal to the number of application runs) needed to generate an error. For instance, if P is 50%, it takes two runs on

average to generate one error.

$$C_{SRS} = (1 + \alpha) \times (\frac{1}{1 - P}) \tag{5.1}$$

The cost of injection-point overprovisioning (with N points per run) can be formulated as in Equation 5.2, based on the law of total probability. The first term is the expected cost if the first injection leads to an error. The expected relative cost of running until the injection point is $\frac{1}{N}$ because the injection points are generated uniformly across time. The second term is the cost if the first injection is masked but the second injection results in an error, and so on.

$$\begin{aligned} C_{IPO} &= (1 - P)(\frac{1}{N} + \alpha) + P(1 - P)(\frac{2}{N} + 2\alpha) \\ &\quad + \ldots + P^{N-1}(1 - P)(\frac{N}{N} + N\alpha) \\ &= (\frac{1}{N} + \alpha) \times (\frac{1 - (1 + N - NP)P^N}{1 - P}) \end{aligned} \tag{5.2}$$

The speedup of overprovisioning is defined as the ratio of C_{SRS} to C_{IPO}. If speedup is greater than 1, then injection-point overprovisioning outperforms simple random sampling. Parameters are swept in both equations and speedup is shown in Figure 5.6. When injection overhead is small (e.g., $\alpha \leq 0.1$), injection-point overprovisioning outperforms simple random sampling by at least 4X in most cases, and speedup increases with the batch size (N). However, when the masking ratio (P) is less than 0.2, or when the injection overhead is large (e.g., $\alpha = 1$), injection-point overprovisioning is worse

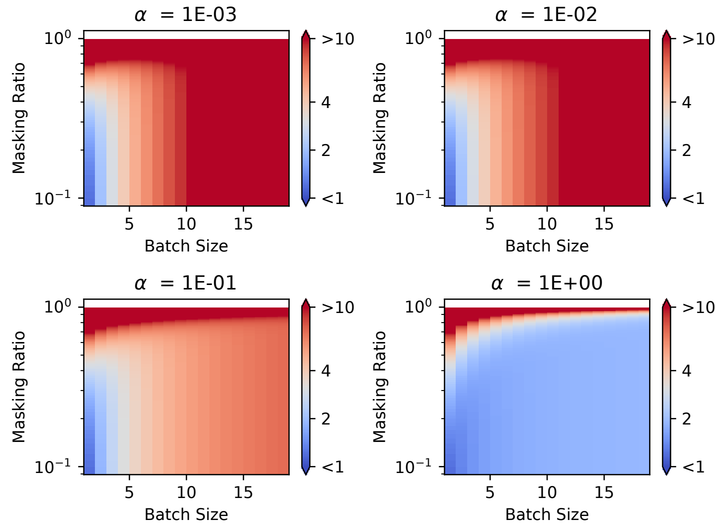

Figure 5.6: Speedup (C_{SRS} / C_{IPO}) of injection-point overprovisioning as a function of the batch size, the masking ratio, and injection overhead. X-axis is the batch size (N), and y-axis the masking ratio (P). α is the injection overhead normalized to execution time of the application.

than simple random sampling.

In reality, the fault masking ratio depends on multiple factors (including circuits, operating condition, input data, etc.). Thus, it is not as simple as a Bernoulli random variable. Experimental results for the benefits of injection-point overprovisioning are presented in Section 5.5.1.

Optimization with Checkpoint-Restart: Injection-point overprovisioning can be further improved with the aid of low-overhead checkpoint-restart. The idea is to reduce the overhead of running applications toward injection points by fastforwarding to the nearest checkpoint before each injection point. Note that this optimization helps only when the overhead of restarting is small

compared to application execution time. The analysis is beyond the scope of this research.

5.4 Evaluation Methodology

The evaluation consists of the following parts: (1) assessing the benefits of injection-point overprovisioning, (2) characterizing the timing error models at the instruction level, (3) evaluating the impact of error models on the resilience of applications, and (4) conducting sensitivity studies of application resilience to droop profiles and pipeline occupancy.

5.4.1 Error Models

This research focuses on timing errors resulting from transient voltage droops (i.e., $\frac{dI}{dt}$ droops), but the tool and framework can be used to study other types of timing errors as well. However, the injector overhead would be too high to evaluate faulty events with very long duration (e.g., temperature variations can last milliseconds or longer, which requires simulating millions of instructions at the gate level). The proposed high-fidelity error models that use just-in-time error generation are compared to two well-known low-fidelity error models.

5.4.1.1 High-Fidelity Timing Error Models

Recall that the error manifestation process of timing errors is very complex and depends on the following factors: (1) the circuit structure, (2)

Table 5.2: Circuits and their fault injection overhead.

	INT-ADD	INT-MUL	FP-ADD	FP-MUL	FP-DIV
Pipeline Depth	1	3	3	3	1
Overhead Mean	0.39s	4.80s	1.59s	3.95s	4.15s
Overhead Std.	0.003	0.112	0.011	0.026	0.029

the operating condition, (3) the variation's profile, and (4) the history of input values to the circuit.

Circuits and Operating Conditions: The tool injects errors to arithmetic units because those circuits are more prone to timing errors according to previous work [92, 93]. Gate-level netlists of integer and floating-point adders and multipliers are synthesized using Synopsys tools (Design Compiler and DesignWare Library) with Synopsys's SAED 32nm technology, optimized for performance. The pipeline depths are tuned to match Intel's Broadwell processors based on the latency data from [67]. OpenTimer is used to obtain the critical path of each circuit and set the clock frequency as 400MHz. The nominal voltage of the cell library is 1.05V and temperature is set to 25°C. Table 5.2 lists the circuits studied in this work. These circuits can be different from those designed and optimized for commodity processors, but the developed tool is generic enough to evaluate other circuits as well.

Droop Profile: Other than the nominal 1.05V voltage, two other voltage levels (0.85V and 0.78V) are used to model droops. Both single-cycle droops

and multi-cycle droops that persist for the maximum pipeline depths (i.e., 3 cycles for the circuits under test) are modeled.

Pipeline Occupancy: The number of instructions affected by a voltage droop depends on the droop's duration and the occupancy of the affected pipeline. In general, for a pipelined circuit of depth D, the maximal number of instructions affected by a T-cycle droop is $D+T-1$. The extreme cases are modeled to understand whether accurate modeling of pipeline occupancy is necessary. A low-occupancy pipeline has only one instruction in the pipeline throughout duration of the droop, while a high-occupancy pipeline is fully utilized with each stage occupied by an instruction.

For each droop magnitude, the combination of droop duration and pipeline occupancy leads to four error groups:

- **Single-cycle Low-occupancy (SL)**: a single-cycle droop affects a low-occupancy pipeline.
- **Single-cycle High-occupancy (SH)**: a single-cycle droop affects a high-occupancy pipeline.
- **Multi-cycle Low-occupancy (ML)**: a multi-cycle droop affects a low-occupancy pipeline.
- **Multi-cycle High-occupancy (MH)**: a multi-cycle droop affects a high-occupancy pipeline.

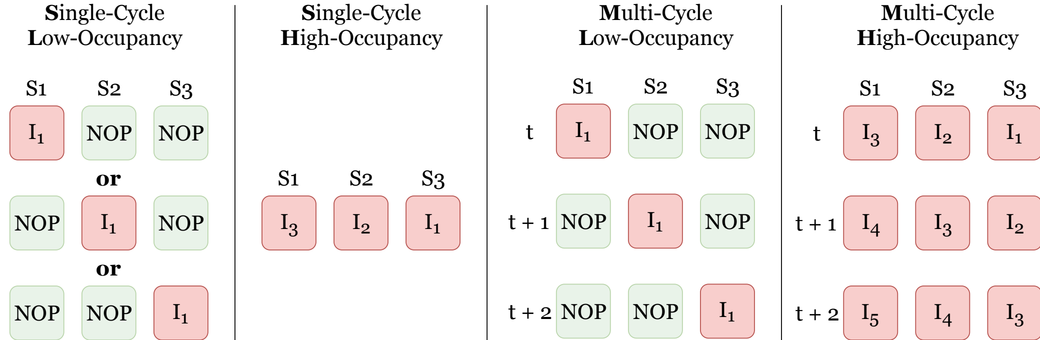

Figure 5.7: Illustration of error groups for a 3-stage pipeline. Data flow from stage 1 (S_1) to stage 3 (S_3).

Figure 5.7 illustrates the four error groups for a 3-stage pipeline. In this example, *SL* affects one instruction at a random stage. *SH* affects three instructions simultaneously in one cycle. *ML* affects the same instruction in three consecutive cycles at different stages. *MH* affects five instructions in total across three cycles. For each instruction, the error group determines which stages are affected by the droop. Recall that in the injector design, fault-enable signals are used to control which stages to inject faults into (Figure 5.2). For instance, when the *ML* error group is specified, I_1 is affected by the multi-cycle droop in three consecutive stages. Thus, fault-enable signals corresponding to those stages are set.

To summarize, the error models are:

- **Single-bit flip (RB1)**: randomly flips a single bit, shown to be a good approximation of high-fidelity arithmetic errors due to particle strikes [1].
- **Previous value (PREV)**: models a severe voltage droop which causes

timing violation at all output pins and thus latches the output value of the previous instruction using the same execution unit [39].

- **Decreasing voltage to 0.85V (0.85V)**: generates a timing error resulting from a droop that decreases the voltage to 0.85V. It consists of four error groups with different combinations of droop duration and pipeline occupancy (Figure 5.7).
- **Decreasing voltage to 0.78V (0.78V)**: generates a timing error resulting from a droop that decreases the voltage to 0.78V. It also consists of four error groups.

5.4.1.2 Experimental Settings

This work assumes that timing faults occur randomly and uniformly (e.g., when hardware error-mitigation techniques fail due to an unexpectedly strong voltage droop). In each experiment, the tool injects an error into a random instruction's output operand with one of the four error models listed above. For each model, injection-point overprovisioning is used to collect 2000 random *errors*. This ensures a margin of measurement error around 2.2% for a confidence level of 95% [68]. Based on Figure 5.6, the batch size (N) is set to 10. For SIMD instructions, the tool injects timing faults to all SIMD lanes because they usually share the same power delivery network.

Table 5.3: Benchmarks, their input, average execution time, and the criteria to classify injection outcomes as SDCs.

	Input	Native Time	Time w/ Injection	SDC Criteria
FT	Class A	3.5s	23.4s	Failed verification
LU	Class A	29.3s	201.8s	Failed verification
MG	Class B	5.8s	46.7s	Failed verification
CG	Class A	1.2s	7.6s	Failed verification
CoMD	default	5.7s	33.8s	Potential energy $> 10^{-10}$ [74]
LULESH	default	22.1s	130.5s	MaxAbsDiff $> 10^{-8}$ [69]

5.4.2 Benchmarks

Six serial scientific kernels and proxy applications are used: `FT` and `LU` and `CG` and `MG` from NPB [73], `CoMD` [74], and `LULESH` [75]. Table 5.3 summarizes the benchmarks, their input, native execution time per experiment, execution time with injection overhead, and how injection outcomes are classified as SDCs. All benchmarks are compiled with gcc 4.8.5 using their original build scripts.

5.5 Experimental Results

5.5.1 Injection-Point Overprovisioning

Figure 5.8 shows the speedup of injection-point overprovisioning for different error groups under different droop magnitudes. The maximum speedup is 7X (for `LULESH` in the setting with 0.85V and the SL error group). Speedup is highest for the SL error group (i.e., voltage droop affects only one instruction at a random stage). In this case, timing errors are more likely to be logically

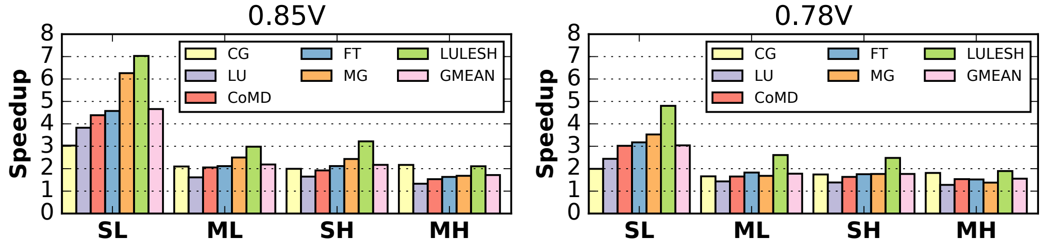

Figure 5.8: Speedup of injection-point overprovisioning for high-fidelity timing error injection. Speedup here is defined as the ratio of evaluation time using simple random sampling to injection-point overprovisioning. Left: 0.85V; Right: 0.78V. X-axis: four error groups (Figure 5.7).

masked at later pipeline stages, and thus the fault-masking rates are higher. Also, speedup is higher when droop magnitude is lower (0.85V) because the fault-masking rate is higher (mostly due to temporal masking). On average, for the weaker droops (0.85V), speedup of injection-point overprovisioning is 4.7X, 2.2X, 2.2X, and 1.7X for the SL, ML, SH, and MH error groups, respectively. For stronger droops (0.78V), speedup is 3.0X, 1.8X, 1.8X, and 1.6X, respectively.

In terms of resource savings due to injection-point overprovisioning, 2,943 core-hours are saved for the entire evaluation. The original resource requirement for simple random sampling was 5,585 core-hours. Notice that the savings can be even higher if one needs higher accuracy (i.e., observing more errors). The accuracy in this work is around 2%. The projected savings for a 1% accuracy target is 13,978 core-hours for the same set of experiments. The resource requirement would be 26,527 core-hours if simple random sampling is used.

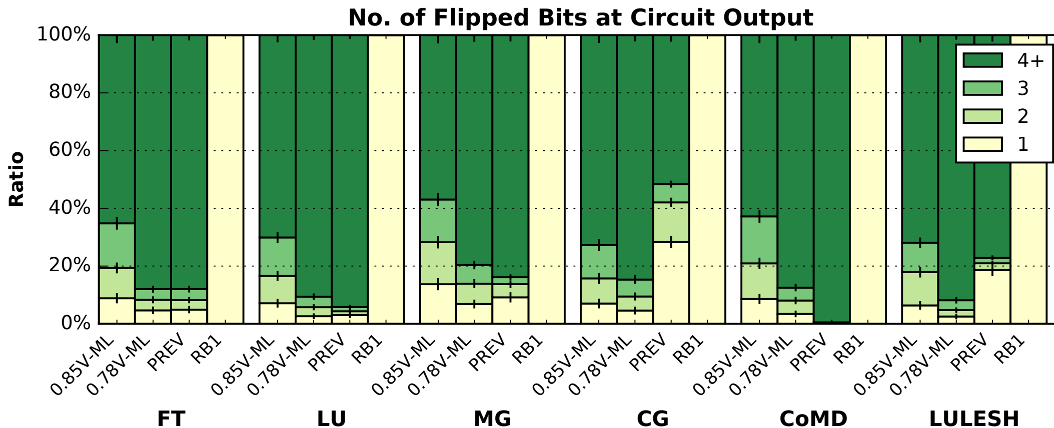

Figure 5.9: Distributions of the number of flipped bits at circuit output (simple error models vs. the high-fidelity ML timing error group).

5.5.2 Injection Outcome and Analysis

In this section, presented are the characteristics of timing errors at the instruction level and the injection outcome at the application level.

5.5.2.1 Instruction-Level Error Patterns

The characteristics of how many bits of the circuit output are flipped (Figure 5.9) and which bits are more likely to be flipped (Figure 5.10) are derived. Note that distributions are application-dependent.

According to Figure 5.9, timing errors flip multiple bits in most cases ($> 80\%$). The fact that timing errors tend to flip lower-significance bits is due to value locality at the higher-significance bits. In other words, the higher-significance bits rarely transition between previous output and current output at the circuit level. Therefore, even though higher-significance bits are more likely to experience timing violation, the latched values are still correct.

Note that PREV in some cases (e.g., `CG` and `LULESH`) flips fewer bits than the high-fidelity model. This is due to the fact that errors generated by PREV exhibit a different instruction mix when compared with the high-fidelity model; PREV essentially injects errors into some instructions that never generate an error when using a higher-fidelity model. Although circuits with lower complexity (e.g., integer adders) are less sensitive to timing errors, PREV can still corrupt results generated by such simple circuits because it lacks timing information.

Observation 1: *timing errors that corrupt arithmetic circuits tend to flip multiple lower-significance bits.*

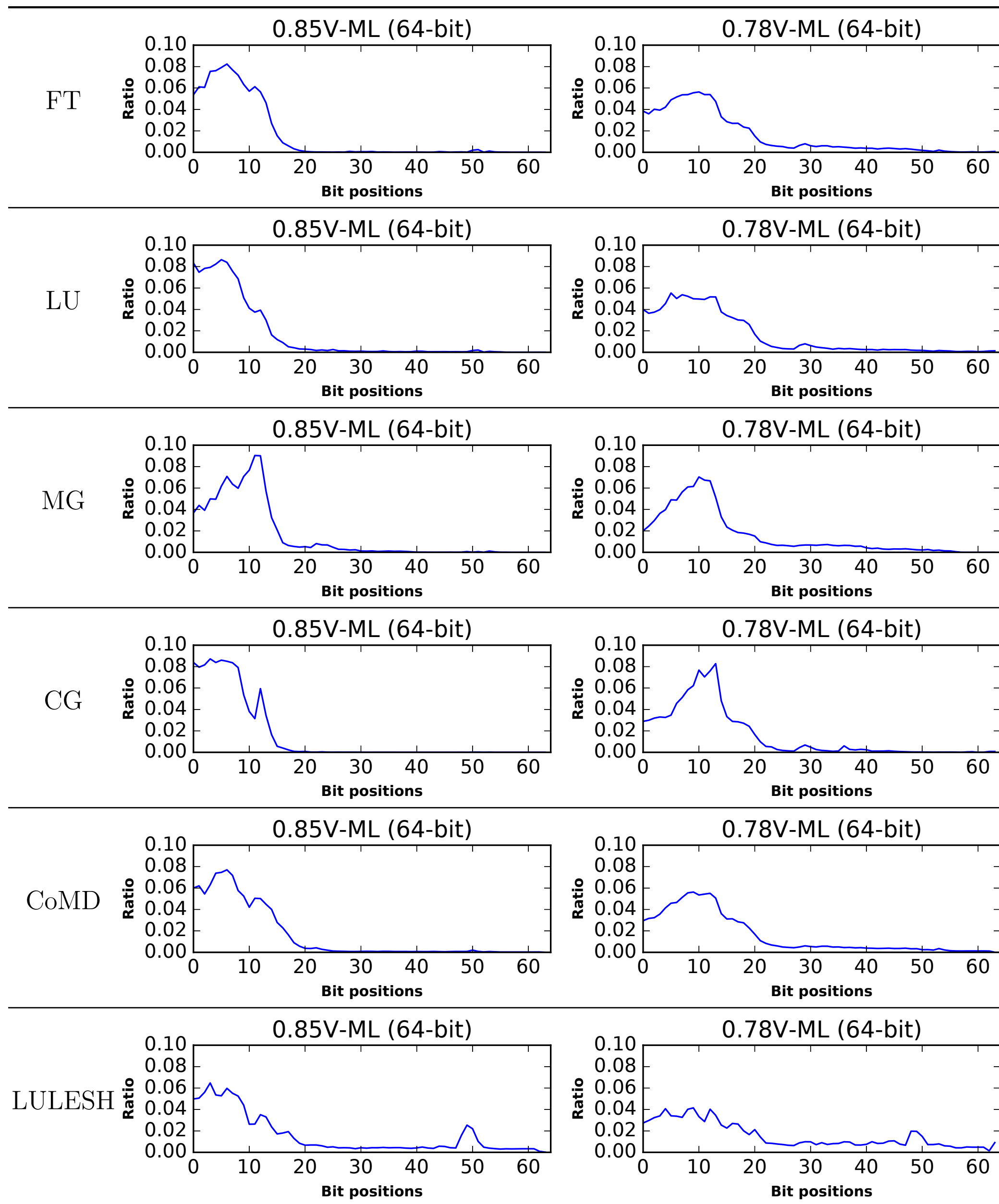

Figure 5.10: Distribution of bit-flip positions at circuit output with input from each benchmark under the ML error group. Note that these are not distributions of positions with timing violation.

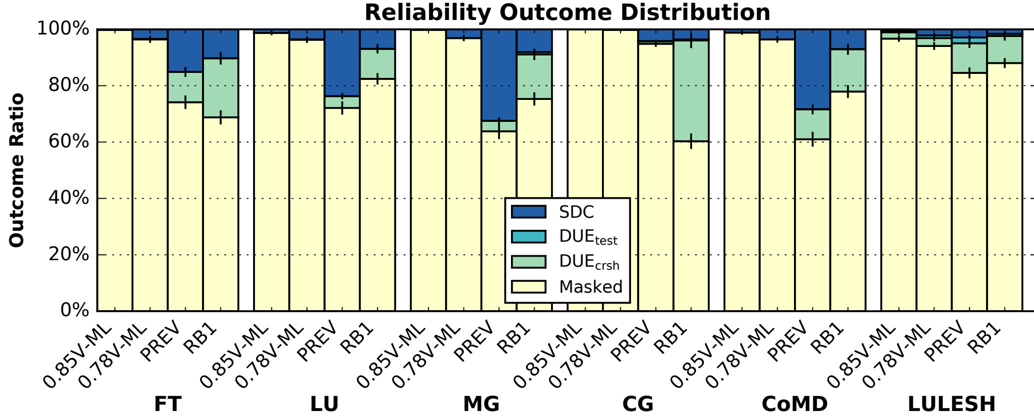

Figure 5.11: Injection outcome distributions (simple error models vs. the high-fidelity ML timing error group). Results of other error groups are in Figure 5.12.

5.5.2.2 Application-Level Impact

Figure 5.11 shows the distributions of error injection outcomes using different error models. Recall that RB1 is shown to be a good approximation of high-fidelity arithmetic errors due to particle strikes [1], and thus it can be used as a proxy to compare timing errors with errors due to particle strikes.

Observation 2: *timing errors rarely lead to DUEs.*

Compared to RB1 and PREV, high-fidelity timing errors rarely lead to DUEs. This observation is related to previous results at the instruction level: lower-significance bits are more likely to be flipped. On the other hand, since RB1 flips each bit with equal likelihood, it tend to flip higher-significance bits and causes more segmentation faults. `LULESH` is the only application in which high-fidelity timing errors result in some DUEs. It turns out that most DUEs are bus errors (i.e., unaligned memory accesses) instead of segmentation faults.

Observation 3: *timing errors result in low SDC ratio.*

High-fidelity errors cause fewer SDCs compared with RB1 and PREV. This is also related to the results at the instruction level. For instance, for floating-point instructions, flipping lower-significance bits leads to changes in the significand field, which has less impact compared to the exponent field. SDC ratios are expected to be even smaller for weaker droops (e.g., droops that decreases the voltage to 0.9V). Comparing the SDC ratios between RB1 and high-fidelity timing errors, the maximum difference occurs in `FT` where the SDC ratio of RB1 is 10% and those of 0.78V and 0.85V are 3% and 0%, respectively.

Observation 2 and observation 3 indicate that it is plausible to save power by reducing guardbands if the user can tolerate some DUEs and SDCs. However, such decisions must be made carefully as timing errors can occur when the processor is in kernel mode as well. The evaluation of kernel mode is beyond the scope of this research.

Observation 4: *neither RB1 nor PREV is a good approximation for high-fidelity timing errors.*

Both RB1 and PREV result in pessimistic results compared with high-fidelity timing errors. RB1 overestimates DUE ratio by at least 7% in all cases and it overestimates SDC ratio by 5.7% and 3.6% for 0.85V and 0.78V, respectively. PREV greatly overestimates both DUE ratio and SDC ratio in all applications.

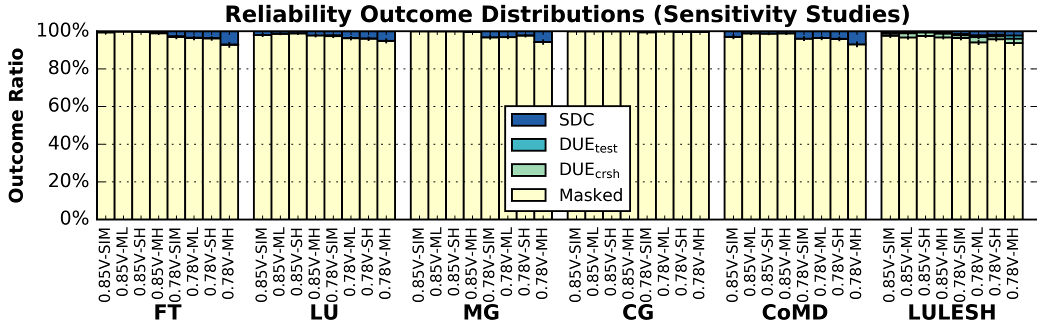

Figure 5.12: Injection outcome distribution for sensitivity studies of droop magnitude and error groups.

5.5.3 Sensitivity Studies

Figure 5.12 shows the sensitivity studies of injection outcome distributions to droop profiles and pipeline occupancy. The first observation is that SDC ratio increases with droop magnitude because stronger droops are more likely to flip higher-significance bits. Second, for the same droop magnitude, three error groups (SL, ML, and SH) lead to similar distributions (error bars are overlapped). This also means that for single-cycle droops (SL and SH), injection outcomes depend only on droop magnitude. Note that MH leads to higher SDC ratio than ML because it not only affects more instructions but also affects them multiple times. The maximum difference of SDC ratio between MH and the other three groups is 3.5% in `MG` with 0.78V. Therefore, for evaluating the impact of multi-cycle droops on applications, more accurate modeling for pipeline occupancy is needed.

Observation 5: *for single-cycle droops, injection outcomes depend only on droop magnitude, while for multi-cycle droops, injection outcomes depend on*

both droop magnitude and pipeline occupancy.

5.6 Summary

This chapter develops high-fidelity error models for voltage droops along with an acceleration technique called injection-point overprovisioning which speeds up evaluation by an order of magnitude. Injection results show that voltage droops tend to flip multiple lower-significance bits at the instruction level and rarely lead to DUEs and SDCs at the application level. It is shown that droop duration affects injection outcomes. For single-cycle droops, injection outcomes depend only on droop magnitude. For multi-cycle droops, injection outcomes depend on both droop magnitude and pipeline state. The developed high-fidelity error models for voltage droops are valuable for the research community because existing low-fidelity models, single-bit flips and previous values, do not represent realistic voltage droops that affect arithmetic units.

Chapter 6

Effectiveness of Software-Based Error Detectors on High-Fidelity Errors[1]

This chapter evaluates software-based error detectors using the high-fidelity error models introduced in previous chapters. Section 6.1 motivates the trend of detecting errors in the software stack. Two categories of software-based detectors are evaluated: instruction duplication at the level of the compiler intermediate representation (IR) (Section 6.2) and application-level error detection (Section 6.3).

6.1 Motivation

Although hardware-based fault detection techniques such as dual modular redundancy provide high coverage for SDCs, they are costly in terms of area and power, such that they are only adopted by mission-critical systems. As power constraints pose challenges for future systems, researchers have been advocating software-based alternatives to harden systems against hardware errors. Software detectors are attractive because they are shown to provide high

[1]Part of this chapter appears in [94]. The author of this dissertation is the main contributor of the idea, implementation, and evaluation. The other coauthors in [94] assist development of the idea and implementation.

coverage for hardware errors and can be selectively employed to only protect critical sections [95, 96, 58, 77, 97].

Furthermore, software-based detectors are both flexible and efficient. In terms of flexibility, they are hardware-agnostic and can be enabled only for target applications. In terms of efficiency, software detectors can focus on critical parts of the applications to maximize error coverage given a fixed cost of performance overhead.

Specifically, this research focuses on two types of software detectors: instruction duplication and application-level error detection. The former detects errors by inserting redundant instructions and checking instructions at compile time [98, 95, 96, 97], while the latter detects errors with characteristics at the application level [99, 100, 58, 101]. Unlike previous work that evaluates these detectors using single-bit flips, this dissertation uses the high-fidelity error models developed in this research.

6.2 Compiler IR-Level Instruction Duplication

Instruction duplication (Figure 6.1) is a technique that detects errors by inserting redundant operations and checking instructions at compile time. The naive method, known as full duplication, replicates and protects all instructions in the program but incurs significant performance overhead. On the other hand, selective instruction duplication protects a subset of instructions to maximize error coverage at some reasonable performance overhead. This approach works well because not all instructions in a program are equally

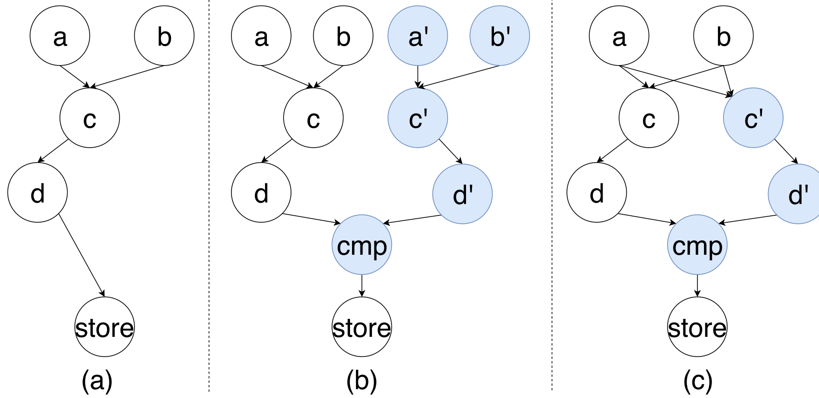

Figure 6.1: An example of instruction duplication methods. (a) original dataflow graph. (b) full instruction duplication. (c) selective instruction duplication.

vulnerable. In fact, only a small number of instructions in a program are responsible for the majority of SDCs. Hence, protecting these highly vulnerable instructions with priority gives developers a reasonable trade-off between the coverage and performance overhead.

6.2.1 LLVM IR-Level Instruction Duplication

Researchers have relied on compiler IR, such as LLVM IR, for resilience studies and selective instruction duplication [13, 102, 59, 103, 104]. This is because LLVM is platform-independent and is a well-supported open-source tool.

Since errors originate in hardware and the IR is a high-level abstraction of a program, debates over the accuracy of IR-level injection and the effectiveness of IR-level protection remain. Specifically, researchers have made two assumptions when using IR for selective instruction duplication, which

have yet to be validated: (1) IR-level injection is accurate enough to project the vulnerability of its lower layer counterparts (i.e., the generated binary), and (2) instruction duplication at the IR level captures high-fidelity hardware errors. Although researchers have investigated the accuracy of IR-based fault injection [8, 13, 105], they focused on the programs without protection and did not use realistic errors in their evaluation.

6.2.2 Evaluation Methodology

The evaluation consists of the following parts: (1) the effectiveness of IR-level selective instruction duplication at reducing SDCs resulting from high-fidelity error models, (2) the effectiveness of full IR instruction duplication at mitigating SDCs, and (3) the impact of different error models on the effectiveness of selective instruction duplication.

Implementation of selective instruction duplication: The implementation consists of four steps: (1) characterization, (2) selection, (3) duplication, and (4) code generation. The characterization step identifies which IR instructions are responsible for SDCs. Since the instruction-level injector of Hamartia is at the binary level (Section 3.1.4), LLFI [13], an LLVM IR-level injector, is used instead. LLFI characterizes each IR instruction's contribution to SDCs using the RB1 error model.

Based on the characterization results, the selection step determines which instructions to protect by solving a 0-1 knapsack problem, which is the

same methodology used in [58, 97]. The output of this step is a list of IR instructions to duplicate.

The duplication step transforms the IR by adding redundant operations and checking instructions. The instructions provided by the selection step are duplicated and a checker is placed right before their following branch, store, function call, or function return.

The code generation step converts the transformed IR into a x86 binary using LLVM's code generation tools.

Error models: The focus here is errors due to particle strikes since instruction duplication assumes that the error affects only a single instruction.[2] The baseline is single-bit flips at the LLVM IR level, used in most prior studies on IR-based selective instruction duplication [8, 59, 13, 104]. It is compared with error models at the binary level in Hamartia. To summarize, the error models include:

- **Single-bit flip at the LLVM IR level (RB1-LLVM)**: randomly flips a single bit in the destination register of a random dynamic LLVM IR instruction.
- **Single-bit flip at binary level (RB1-BIN)**: randomly flips a single bit in the destination operand of a random dynamic instruction at the

[2]Recall that voltage droops are likely to affect multiple instructions including the instructions inserted for error detection.

binary level.

- **RTL-level model (RTL)**: the high-fidelity error model for particle strikes introduced in Chapter 4.

Metric: To quantify cost-effectiveness, the effectiveness of SDC reduction is examined as the fraction of instructions being duplicated is increased. For each benchmark and each error model, a protection curve is derived, a graphical representation that helps researchers trade-off resilience and performance overhead [58, 70]. The x-axis is the protection level (i.e., the fraction of dynamic instructions duplicated). For instance, a protection level of 50% means that at most half of the total dynamic instructions are duplicated for protection. The y-axis is the SDC coverage, the reduction of SDC probability after protection divided by the SDC probability before protection. For example, if the initial SDC probability is 20% and the resultant SDC probability is 10% for some protection level, then the SDC coverage at the protection level is 50%.

Benchmarks: This research evaluates the serial version of programs from common benchmark suites (including Parboil [106], Rodinia [107], Parsec [108], and SPEC [109]), an earthquake simulation application, `hercules`, from Carnegie Mellon University [110], and the molecular dynamics application, `puremd`, from Purdue University [111]. Table 6.1 summarizes the benchmarks and their inputs. All programs are compiled using LLVM 3.4 with -O2.

Table 6.1: Benchmarks and their input.

	Suite/Author	Input
bfs	Parboil	graph4096.txt
blackscholes	Parsec	1 in_4.txt
cutcp	Parboil	watbox.sl40.pqr
hercules	Carnegie Mellon University	simple_case.e
hotspot	Rodinia	64 64 1 1 temp_64 power_64
libquantum	SPEC	33 5
nw	Rodinia	2048 10 1
puremd	Purdue University	geo ffield control
sad	Parboil	reference.bin frame.bin

6.2.3 Experimental Results

The effectiveness of IR-level selective instruction duplication: Figure 6.2 shows the protection curves measured under different error models. The ground truth is the results under the *RTL* error model (red triangles). The observation is that most SDCs can be detected by duplicating a fraction of IR instructions in a program. In most cases, protecting 20% of instructions is able to cover more than 50% of SDCs. Note that the knee of the protection curves (i.e., the protection level after which the protection curve plateaus out) varies across applications. Prior work has observed similar results [58, 70, 104].

Next, consider the curves of *RTL* (red triangles) and *RB1-LLVM* (dashed curves). In each benchmark, IR-based evaluation usually leads to pessimistic results at low protection levels but optimistic results at high protection levels. However, both curves follow similar trends across benchmarks. The mean absolute difference between the two curves across benchmarks is

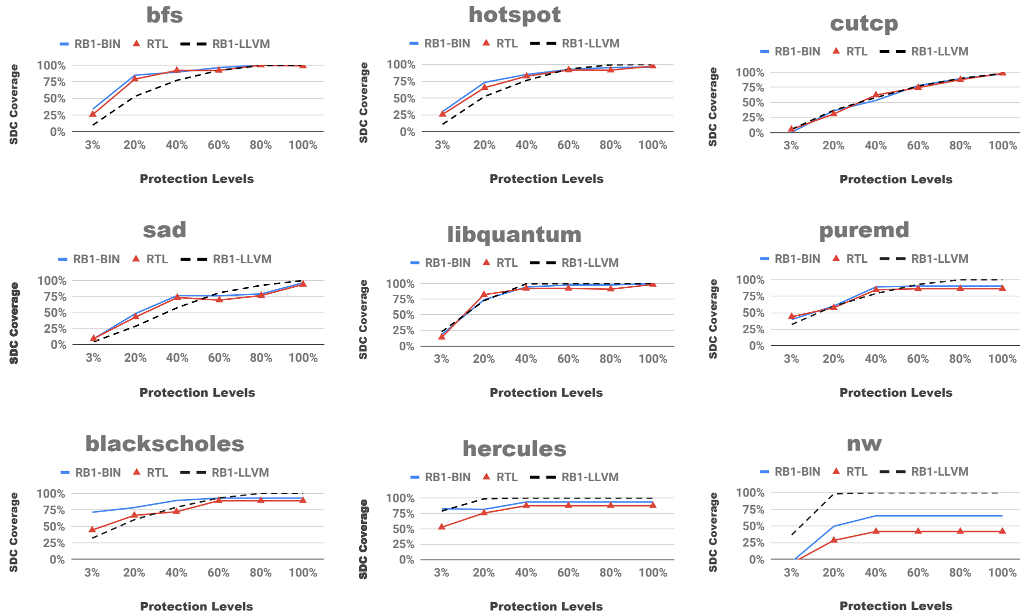

Figure 6.2: Protection curves (x-axis: protection level; y-axis: SDC coverage).

9.32% (excluding the outlier, `nw`, which is explained below). These results indicate that IR-based selective instruction duplication can approach reasonable trade-offs between SDC coverage and performance overhead.

The exception is `nw` where the LLVM curve is consistently higher than that of *RTL*. The difference is not statistically meaningful because the SDC ratio even without any protection is already low (1.58%).

Observation 1: *IR-based selective instruction duplication is able to provide cost-effective protection to mitigate SDCs under high-fidelity errors.*

The effectiveness of full duplication at mitigating SDCs: Table 6.2 shows that under *RTL* and *RB1-BIN*, there are a small number of errors that

Table 6.2: SDC ratios under full IR-level instruction duplication.

	bfs	blackscholes	cutcp	hercules	hotspot	libquantum	nw	puremd	sad	**gmean**
RTL	0.21%	0.10%	0.38%	0.75%	0.33%	0.05%	0.92%	0.08%	1.52%	0.28%
RB1-BIN	0.13%	0.10%	0.71%	0.25%	0.62%	0.05%	0.46%	0.08%	0.84%	0.24%

escape full duplication of IR instructions and eventually lead to SDCs. On average, the SDC ratios of *RTL* and *RB1-BIN* are 0.28% and 0.24%, respectively. Since the duplication is at the IR level, injection using *RB1-LLVM* leads to zero SDC. The reason why errors injected at lower layers escape IR instruction duplication is because the code generation step inserts additional instructions that are not visible at the IR level (e.g., instructions that set up or tear down stack frames). Hence, they are not protected by IR duplication.

Observation 2: *Only a small fraction (0.28%) of high-fidelity errors escape full IR instruction duplication and lead to SDCs.*

The impact of error models: As shown in Figure 6.2, the protection curves measured using *RTL* (red triangles) and *RB1-BIN* (blue solid curves) have very similar shapes. The mean absolute difference across benchmarks is 7.11% (5.55% if `nw` is excluded). The exceptions are `nw` and `blackscholes` whose SDC ratios are low (around 1%) even without any protection. Notice that for `blackscholes` when protection level is at 3%, there is a 27% difference between the two curves. This is because the SDC ratios without protection are 1.4% and 0.9% for *RB1-BIN* and *RTL* respectively. Although they are close in terms of absolute values, their difference is large relatively.

Observation 3: *Protection curves measured by* RTL *and* RB1-BIN *are mostly similar (5% mean absolute difference). Hence,* RB1-BIN *can also be used as a fast proxy to evaluate the effectiveness of selective instruction duplication on high-fidelity errors.*

6.3 Application-Level Error Detection

Application-level error detection refers to an approach that detects errors using application-specific characteristics. Unlike compiler-based techniques, the implementations of application-level detectors usually require modification to the source code. This section evaluates two applications with application-level detectors: CLAMR that contains a detector that checks invariants in the algorithm [101], and HeatDist that can be protected by value-prediction [112, 113].

6.3.1 CLAMR

CLAMR is a cell-based adaptive mesh refinement mini-app for hydrodynamic simulation [101]. The developers design an application-level detector that takes advantage of the conservation of mass. If the mass of the water deviates beyond an allowable difference, the program raises an exception indicating an error has been detected.

Evaluation: The goal is to understand the impact of different error models on the final output of CLAMR. Two versions of CLAMR are evaluated: one

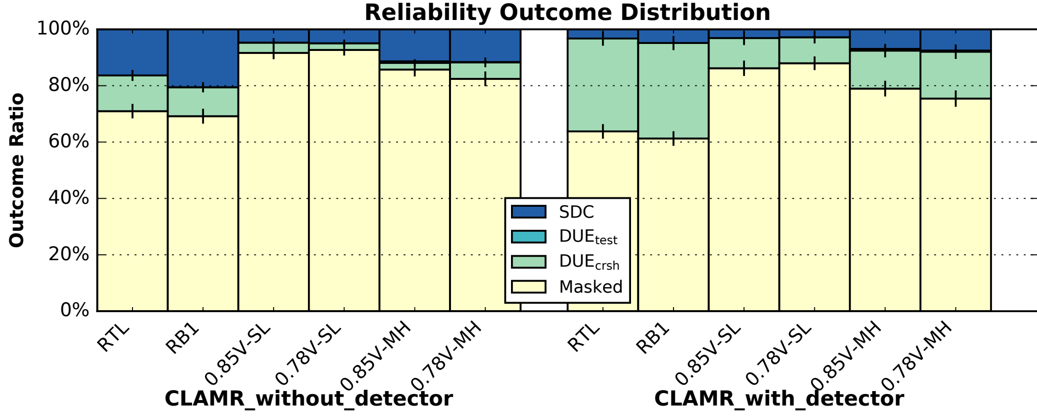

Figure 6.3: Injection outcome distributions of CLAMR without protection and CLAMR with the mass-conservation detector. Error models from left to right are high-fidelity errors due to particle strikes, single-bit flip, and four high-fidelity error models due to voltage droops (see Section 5.4).

with the mass-conservation detector and the other without it. In addition to single-bit flips, high-fidelity errors due to particle strikes and errors due to voltage droops are also evaluated.

Results: Figure 6.3 shows the outcome distributions of different error models. The first observation is that the mass-conservation detector can significantly reduce the SDCs due to particle strikes (RTL) and single-bit flips (RB1). For RTL, the SDC ratio reduces to 3.28% from 16.36%; as for RB1, the SDC ratio reduces to 4.86% from 20.55%. Also, RB1 leads to distributions similar to RTL regardless of the presence of the detector.

Observation 1: *The mass-conservation detector can effectively capture SDCs due to particle strikes.*

However, the effectiveness of the detector at mitigating SDCs due to

voltage droops is limited. Voltage droops that corrupt only one instruction (the SL models) lead to low SDC ratios whether a detector is present or not. This is because voltage droops tend to flip the low-significance bits of the output operand as found in Section 5.5. The mass-conservation detector is only able to reduce the SDC ratios by about 2% for both 0.85V-SL and 0.78V-SL. On the other hand, voltage droops that corrupt multiple instructions (the MH models) result in higher SDC ratios. The mass-conservation detector can reduce the SDC ratios by about 4% for both 0.85V-MH and 0.78V-MH. The results indicate that other detectors are needed to address the impact of voltage droops on CLAMR.

Observation 2: *The effectiveness of the mass-conservation detector at capturing SDCs due to voltage droops is limited, indicating the need of additional detectors to handle voltage droops.*

6.3.2 HeatDist

HeatDist computes the steady-state heat distribution with Laplace's equation using the Jacobi iterative method. The developers observe that in general data of HPC applications change smoothly over time, and thus they propose two types of prediction-based detectors. The first type predicts data values using data from previous timestamps (*temporal prediction* [112]), and the second type predicts values using neighboring data points in the data structure (*spatial prediction* [113]). If data deviate beyond an allowable difference from the predicted value, the detector reports an error.

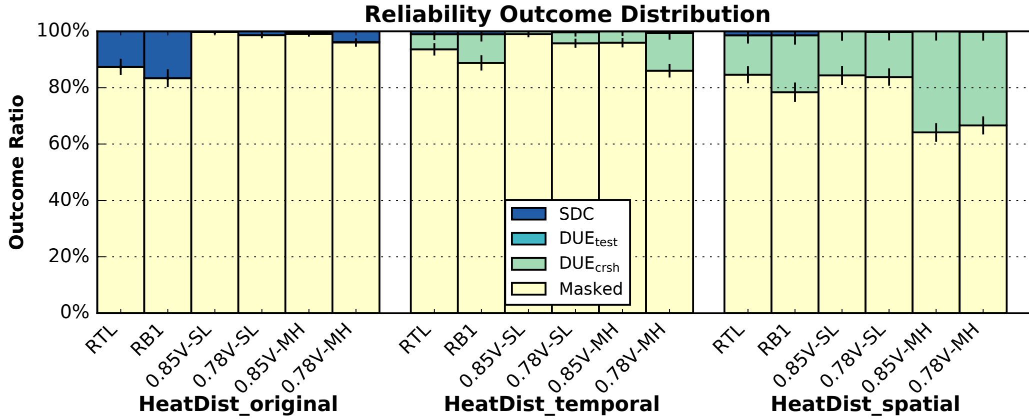

Figure 6.4: Injection outcome distributions of HeatDist without protection, with the detector using temporal prediction, and with the detector using spatial prediction. Error models from left to right are high-fidelity errors due to particle strikes, single-bit flip, and four high-fidelity error models due to voltage droops (see Section 5.4).

Evaluation: The goal is to understand the impact of different error models on HeatDist. Three versions of HeatDist are evaluated: one without protection, another with a detector based on temporal prediction, and the other with a detector based on spatial prediction. Note that both prediction-based detectors incur similar performance overhead for HeatDist. In addition to single-bit flips, high-fidelity errors due to particle strikes and errors due to voltage droops are evaluated.

Results: Figure 6.4 shows the outcome distributions of different error models and the detailed SDC ratios are shown in Table 6.3. Both types of detectors can effectively reduce SDCs due to particle-strike error models. For both RTL and RB1, the SDC ratio reduces to about 1% with either detector. However,

the detector using spatial prediction leads to higher DUE ratios. It turns out that 97% of errors detected as DUEs by spatial prediction are false positives.[3] In contrast, the detector using temporal prediction has a low false-positive ratio (3%). As a result, temporal prediction should be used instead since both prediction-based detectors incur similar performance overhead for HeatDist.

Observation 1: *Although both prediction-based detectors can effectively capture SDCs due to particle strikes, the detector using temporal prediction is more favorable due to its low false-positive ratio.*

As for comparison between RB1 and RTL, it is observed that, without detection, RB1 estimates higher SDC ratios compared with RTL (16% vs. 12%). With either detector, RB1 also estimates higher DUE ratios vs. RTL (10% vs. 5% for temporal prediction and 20% vs. 14% for spatial prediction).

For HeatDist without detectors, RTL and RB1 estimate higher SDC ratios than the high-fidelity voltage-droop error models ($12-16\%$ vs. $0-4\%$) because voltage droops tend to flip the low-significance bits of the output operand. Adding either type of detector removes all SDCs caused by voltage droops that decrease voltage to 0.85V. However, stronger droops can still cause SDCs even with a detector.

Observation 2: *Prediction-based detectors can eliminate SDCs caused by*

[3]False positives are errors reported as DUEs by the detector but their impact on the application is insignificant and should be classified as Masked without detection. To count the percentage of false positives, the detector does not terminate the application upon detection.

Table 6.3: SDC ratios of HeatDist without protection, with the detector using temporal prediction, and with the detector using spatial prediction.

	RTL	RB1	0.85V-SL	0.78V-SL	0.85-MH	0.78-MH
HeatDist_original	12.60%	16.60%	0.12%	1.34%	0.34%	3.79%
HeatDist_temporal	1.00%	1.00%	0.00%	0.30%	0.00%	0.61%
HeatDist_spatial	1.40%	1.40%	0.00%	0.19%	0.00%	0.14%

voltage droops of lower magnitude. Other detectors are needed to handle SDCs caused by stronger droops.

6.4 Summary

This chapter evaluates software-based detectors using high-fidelity error models. Evaluation shows that IR-based selective instruction duplication is able to cost-effectively reduce SDCs resulting from particle strikes. Application-level detectors, on the other hand, can effectively detect errors caused by particle strikes but not voltage droops, indicating the need of additional detectors to handle voltage droops. Single-bit flips remain a good approximation of particle strikes even when applications are protected by software-based detectors studied in this research.

Chapter 7

Summary and Concluding Remarks

To summarize, the research objectives are: (1) increasing error modeling fidelity, (2) reducing the number of application runs while keeping sampling quality equal, and (3) evaluating the effects of modeling fidelity on experimental results. This dissertation presents Hamartia, an open-source hardware error analysis suite with high fidelity and low overhead. Hamartia increases error modeling fidelity for particle strikes and voltage droops with hierarchical injection. Two novel acceleration techniques, nested Monte Carlo and injection-point overprovisioning, are included in Hamartia to speed up error injection by $1-2$ orders of magnitude while keeping sampling quality equal. This high-fidelity and low-overhead error injection methodology discovers new insights in terms of the impact of errors on applications, the effectiveness of detectors, and the effects of modeling fidelity on experimental results. Key insights from evaluation include:

- For estimating injection outcome distributions, single-bit flips are a good approximation of particle strikes.
- For estimating application output quality, high-fidelity error models are required.

- Existing low-fidelity error models (single-bit flips and previous values) do not represent errors caused by voltage droops.

- Software-based detectors can effectively detect errors caused by particle strikes but not voltage droops, indicating the need of additional detectors for handling voltage droops.

7.1 Broad Applicability

Although this research focuses on applications in the high-performance computing (HPC) domain, Hamartia opens a new chapter of resilience studies in other domains as well. For instance, hardware errors pose reliability and availability problems for data centers due to their massive scale [114, 115, 116]. Another applicable domain is automotive systems where multiple electronic control units are expected to be consolidated into a single unit for lower costs. The usage of advanced manufacturing technologies implies that systems would be more sensitive to transient errors. As a result, the design ought to handle hardware problems to guarantee functional safety [117, 118].

7.2 Hardware Errors Beyond This Research

As this research focuses on arithmetic errors, there exist hardware errors not modeled in the current implementation of Hamartia. However, some can be addressed by extending Hamartia, and the acceleration techniques proposed in this dissertation are equally useful for evaluating other error types.

Memory errors: These include errors in register files, caches, various buffers at the micro-architectural level, and errors in DRAM. Although memory structures are usually protected by parity or ECC, some errors can still escape detection and impact applications. In addition to memory array structures, the peripheral circuits are also vulnerable. Errors from emerging non-volatile memory should also be studied as well.

Uncore errors: These include errors in the cache controllers, interconnection networks, memory controllers, and I/O controllers. Addressing these errors is important as the uncore part of modern CPUs occupies a significant portion of die area. Prior work studies the impact of particle strikes on the uncore of the OpenSPARC T2 processor and designs hardware-based techniques to recover failures in the cache controllers [119]. Resilient cache coherence protocols are also proposed to address transient errors [120].

Hamartia can be extended to evaluate the impact of various errors on applications, the effectiveness of detectors, and the effects of modeling fidelity on experimental results. Some errors (e.g., DRAM errors) can be evaluated by implementing new instruction-level error models in Hamartia, while some (e.g., errors in micro-architectural components) require a new interface in addition to the original error context API. To address error masking, both nested Monte Carlo and injection-point overprovisioning can be applied to reduce the number of application runs while keeping sampling quality equal.

Bibliography

[1] Chun-Kai Chang, Sangkug Lym, Nicholas Kelly, Michael B. Sullivan, and Mattan Erez. Evaluating and accelerating high-fidelity error injection for hpc. In *Proceedings of the ACM/IEEE International Conference on High-Performance Computing, Networking, Storage, and Analysis (SC)*, Dallas, TX, November 2018.

[2] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Annual Design Automation Conference*, page 101. ACM, 2013.

[3] Ghani Kanawati, Nasser Kanawati, and Jacob Abraham. Emax: An automatic extractor of high-level error models. In *9th Computing in Aerospace Conference*, page 4698, 1993.

[4] Giacinto P Saggese, Nicholas J Wang, Zbigniew T Kalbarczyk, Sanjay J Patel, and Ravishankar K Iyer. An experimental study of soft errors in microprocessors. *IEEE MICRO*, 25(6):30–39, 2005.

[5] Dong Li, Jeffrey S Vetter, and Weikuan Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Proceedings of the International Conference on*

High Performance Computing, Networking, Storage and Analysis (SC), page 57. IEEE Computer Society Press, 2012.

[6] Qiang Guan, Nathan Debardeleben, Sean Blanchard, and Song Fu. F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1245–1254. IEEE, 2014.

[7] Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F DeMara, Chen-Yong Cher, and Pradip Bose. Understanding the propagation of transient errors in hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. IEEE, 2015.

[8] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S. Nikolopoulos, and Martin Schulz. Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed. In *Proceedings of International Conference on High-Performance Computing, Networking, Storage, and Analysis (SC)*, Denver, CO, November 2017. IEEE.

[9] Daniel Oliveira, Laércio Pilla, Nathan DeBardeleben, Sean Blanchard, Heather Quinn, Israel Koren, Philippe Navaux, and Paolo Rech. Experimental and analytical study of xeon phi reliability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 28. ACM, 2017.

[10] Vijay Janapa Reddi and Meeta Sharma Gupta. Resilient architecture design for voltage variation. *Synthesis Lectures on Computer Architecture*, 8(2):1–138, 2013.

[11] Josep Torrellas. Extreme-scale computer architecture: Energy efficiency from the ground up. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 200. European Design and Automation Association, 2014.

[12] Abbas Rahimi, Luca Benini, and Rajesh K Gupta. Variability mitigation in nanometer cmos integrated systems: A survey of techniques from circuits to software. *Proceedings of the IEEE*, 104(7):1410–1448, 2016.

[13] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 375–382. IEEE, 2014.

[14] Vishal Chandra Sharma, Arvind Haran, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Towards formal approaches to system resilience. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 41–50. IEEE, 2013.

[15] Radha Venkatagiri, Khalique Ahmed, Abdulrahman Mahmoud, Sasa Misailovic, Darko Marinov, Christopher W Fletcher, and Sarita V Adve. gem5-approxilyzer: An open-source tool for application-level soft error

analysis. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 214–221. IEEE, 2019.

[16] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 245–255. IEEE, 2015.

[17] Raghuraman Balasubramanian and Karthikeyan Sankaralingam. Understanding the impact of gate-level physical reliability effects on whole program execution. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 60–71. IEEE, 2014.

[18] Schuyler Eldridge, Alper Buyuktosunoglu, and Pradip Bose. Chiffre: A configurable hardware fault injection framework for risc-v systems. In *2nd Workshop on Computer Architecture Research with RISC-V (CARRV '18)*, 2018.

[19] Man-Lap Li, Pradeep Ramachandran, Ulya R Karpuzcu, Siva Kumar Sastry Hari, and Sarita V Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, pages 105–116. IEEE, 2009.

[20] Shahrzad Mirkhani, Meisam Lavasani, and Zainalabedin Navabi. Hierarchical fault simulation using behavioral and gate level hardware models. In *Proceedings of the 11th Asian Test Symposium (ATS)*, pages 374–379. IEEE, 2002.

[21] Zbigniew Kalbarczyk, Ravishankar K Iyer, Gregory L Ries, Jaqdish U Patel, Myeong S Lee, and Yuxiao Xiao. Hierarchical simulation approach to accurate fault modeling for system dependability evaluation. *IEEE Transactions on Software Engineering*, 25(5):619–632, 1999.

[22] Hungse Cha, Elizabeth M Rudnick, Janak H Patel, Ravishankar K Iyer, and Gwan S Choi. A gate-level simulation environment for alpha-particle-induced transient faults. *IEEE Transactions on Computers*, 45(11):1248–1256, 1996.

[23] 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 *ACM SIGPLAN Notices*, volume 47, pages 123–134. ACM, 2012.

[24] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V Adve, and Helia Naeimi. Ganges: Gang error simulation for hardware resiliency evaluation. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 61–72. IEEE, 2014.

[25] Manolis Kaliorakis, Dimitris Gizopoulos, Ramon Canal, and Antonio Gonzalez. Merlin: Exploiting dynamic instruction behavior for fast

and accurate microarchitecture level reliability assessment. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 241–254. ACM, 2017.

[26] Taiki Uemura, Soonyoung Lee, Udit Monga, Jaehee Choi, Seungbae Lee, and Sangwoo Pae. Technology scaling trend of soft error rate in flip-flops in 1xnm bulk finfet technology. *IEEE Transactions on Nuclear Science*, 65(6):1255–1263, 2018.

[27] Norbert Seifert, Shah Jahinuzzaman, Jyothi Velamala, Ricardo Ascazubi, Nikunj Patel, Balkaran Gill, Joseph Basile, and Jeffrey Hicks. Soft error rate improvements in 14-nm technology featuring second-generation 3d tri-gate transistors. *IEEE Transactions on Nuclear Science*, 62(6):2570–2577, 2015.

[28] Vilas Sridharan and David R Kaeli. Using hardware vulnerability factors to enhance avf analysis. *ACM SIGARCH Computer Architecture News*, 38(3):461–472, 2010.

[29] Michail Maniatakos, Maria Michael, Chandra Tirumurti, and Yiorgos Makris. Revisiting vulnerability analysis in modern microprocessors. *IEEE Transactions on Computers*, 64(9):2664–2674, 2015.

[30] Robert Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, 2005.

[31] Charles R Lefurgy, Alan J Drake, Michael S Floyd, Malcolm S Allen-Ware, Bishop Brock, Jose A Tierno, and John B Carter. Active management of timing guardband to save energy in power7. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–11. IEEE, 2011.

[32] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, Peter Lawthers, and Shidhartha Das. Harnessing voltage margins for energy efficiency in multicore cpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 503–516. ACM, 2017.

[33] Todd Austin, Valeria Bertacco, David Blaauw, and Trevor Mudge. Opportunities and challenges for better than worst-case design. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 2–7. ACM, 2005.

[34] Vijay Janapa Reddi, Meeta S Gupta, Glenn Holloway, Gu-Yeon Wei, Michael D Smith, and David Brooks. Voltage emergency prediction: Using signatures to reduce operating margins. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 18–29. IEEE, 2009.

[35] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, et al. Razor: A low-power pipeline based on circuit-level

timing speculation. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 7. IEEE Computer Society, 2003.

[36] Shidhartha Das, Carlos Tokunaga, Sanjay Pant, Wei-Hsiang Ma, Sudherssen Kalaiselvan, Kevin Lai, David M Bull, and David T Blaauw. Razorii: In situ error detection and correction for pvt and ser tolerance. *IEEE Journal of Solid-State Circuits*, 44(1):32–48, 2008.

[37] Meeta S Gupta, Krishna K Rangan, Michael D Smith, Gu-Yeon Wei, and David Brooks. Decor: A delayed commit and rollback mechanism for handling inductive noise in processors. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 381–392. IEEE, 2008.

[38] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA 10, page 497508, New York, NY, USA, 2010. Association for Computing Machinery.

[39] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[40] Chen-Yong Cher, Meeta S Gupta, Pradip Bose, and K Paul Muller. Understanding soft error resiliency of blue gene/q compute chip through hardware proton irradiation and software fault injection. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 587–596. IEEE, 2014.

[41] Anys Bacha and Radu Teodorescu. Dynamic reduction of voltage margins by leveraging on-chip ecc in itanium ii processors. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 297–307. ACM, 2013.

[42] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran, Pradip Bose, and Vijay Janapa Reddi. Safe limits on voltage reduction efficiency in gpus: a direct measurement approach. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 294–307. IEEE, 2015.

[43] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, Peter Lawthers, and Shidhartha Das. Harnessing voltage margins for energy efficiency in multicore cpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 503–516. ACM, 2017.

[44] Konstantinos Parasyris, Panos Koutsovasilis, Vassilis Vassiliadis, Christos D Antonopoulos, Nikolaos Bellas, and Spyros Lalis. A framework

for evaluating software on reduced margins hardware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 330–337. IEEE, 2018.

[45] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on computers*, 44(2):248–260, 1995.

[46] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. Fail*: Towards a versatile fault-injection experiment framework. In *2012 25th International Conference on Architecture of Computing Systems*, pages 1–5. IEEE, 2012.

[47] Athanasios Chatzidimitriou and Dimitris Gizopoulos. Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 69–78. IEEE, 2016.

[48] Andrea Pellegrini, Kypros Constantinides, Dan Zhang, Shobana Sudhakar, Valeria Bertacco, and Todd Austin. Crashtest: A fast high-fidelity fpga-based resiliency analysis framework. In *2008 IEEE International Conference on Computer Design*, pages 363–370. IEEE, 2008.

[49] G Tziantzioulis, AM Gok, SM Faisal, Nikolaos Hardavellas, S Ogrenci-Memik, and Srinivasan Parthasarathy. b-hive: A bit-level history-based error model with value correlation for voltage-scaled integer and floating

point units. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, page 105. ACM, 2015.

[50] Jeremy Constantin, Zheng Wang, Georgios Karakonstantis, Anupam Chattopadhyay, and Andreas Burg. Statistical fault injection for impact-evaluation of timing errors on application performance. In *Proceedings of the 53rd Annual Design Automation Conference (DAC)*, page 13. ACM, 2016.

[51] Abhisek Pan, James W Tschanz, and Sandip Kundu. A low cost scheme for reducing silent data corruption in large arithmetic circuits. In *IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 343–351. IEEE, 2008.

[52] Keith A Campbell, Pranay Vissa, David Z Pan, and Deming Chen. High-level synthesis of error detecting cores through low-cost modulo-3 shadow datapaths. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, page 161. ACM, 2015.

[53] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[54] Stephen Williams. Icarus verilog, 2006. `http://iverilog.icarus.com/`.

[55] Sriram Krishnamoorthy Vishal Chandra Sharma, Ganesh Gopalakrishnan. Towards reseiliency evaluation of vector programs. In *21st IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, 2016.

[56] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 249–258. IEEE, 2017.

[57] Chun-Kai Chang, Sangkug Lym, Nicholas Kelly, Michael B Sullivan, and Mattan Erez. Hamartia: A fast and accurate error injection framework. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 101–108. IEEE, 2018.

[58] Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2012.

[59] Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F DeMara, Chen-Yong Cher, and Pradip Bose. Understanding the propagation of transient errors in hpc applications. In *Proceedings of the International*

Conference for High Performance Computing, Networking, Storage and Analysis (SC), page 72. ACM, 2015.

[60] Qiang Guan, Nathan BeBardeleben, Panruo Wu, Stephan Eidenbenz, Sean Blanchard, Laura Monroe, Elisabeth Baseman, and Li Tan. Design, use and evaluation of p-fsefi: A parallel soft error fault injection framework for emulating soft errors in parallel applications. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, pages 9–17. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

[61] Daniel Oliveira, Vinicius Frattin, Philippe Navaux, Israel Koren, and Paolo Rech. Carol-fi: an efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators. In *Proceedings of the Computing Frontiers Conference*, pages 295–298. ACM, 2017.

[62] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*, pages 451–460. Springer, 2015.

[63] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 58:1–11, Salt Lake City, UT, November 2012.

[64] Pia N Sanda, Jeffrey W Kellington, Prabhakar Kudva, Ronald Kalla, Ryan B McBeth, Jerry Ackaret, Ryan Lockwood, John Schumann, and Christopher R Jones. Soft-error resilience of the ibm power6 processor. *IBM Journal of Research and Development*, 52(3):275–284, 2008.

[65] Albert Meixner, Michael E Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 210–222. IEEE, 2007.

[66] Javier Carretero, Pedro Chaparro, Xavier Vera, Jaume Abella, and Antonio González. End-to-end register data-flow continuous self-test. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 105–115. ACM, 2009.

[67] Agner Fog. Optimization manual 4 instruction tables. *Copenhagen University College of Engineering, Software Optimization Resources, http://www.agner.org/optimize,(1996-2017)*, pages 215–230.

[68] Régis Leveugle, A Calvez, Paolo Maistri, and Pierre Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 502–506. European Design and Automation Association, 2009.

[69] Bo Fang, Panruo Wu, Qiang Guan, Nathan DeBardeleben, Laura Monroe, Sean Blanchard, Zhizong Chen, Karthik Pattabiraman, and Matei

Ripeanu. Sdc is in the eye of the beholder: A survey and preliminary study. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop*, pages 72–76. IEEE, 2016.

[70] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE, 2016.

[71] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36. IEEE, 1995.

[72] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

[73] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[74] ExMatEx. Comd: Classical molecular dynamics proxy application. `https://github.com/ECP-copa/CoMD`.

[75] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[76] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *2014 The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, Kyoto.

[77] Daya Shanker Khudia and Scott Mahlke. Harnessing soft computations for low-budget fault tolerance. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 319–330. IEEE, 2014.

[78] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C*, volume 2. Cambridge university press Cambridge, 1996.

[79] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future generation computer systems*, 22(3):303–312, 2006.

[80] William M Jones, John T Daly, and Nathan DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *Proceedings of the 19th ACM International Sym-*

posium on High Performance Distributed Computing, pages 276–279. ACM, 2010.

[81] Chun-Kai Chang, Wenqi Yin, and Mattan Erez. Assessing the impact of timing errors on hpc applications. In *Proceedings of the ACM/IEEE International Conference on High-Performance Computing, Networking, Storage, and Analysis (SC)*, Denver, CO, November 2019.

[82] Puneet Gupta, Yuvraj Agarwal, Lara Dolecek, Nikil Dutt, Rajesh K Gupta, Rakesh Kumar, Subhasish Mitra, Alexandru Nicolau, Tajana Simunic Rosing, Mani B Srivastava, et al. Underdesigned and opportunistic computing in presence of hardware variability. *IEEE Transactions on Computer-Aided Design of integrated circuits and systems*, 32(1):8–23, 2013.

[83] Eric Cheng, Shahrzad Mirkhani, Lukasz G Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R Stan, Klas Lilja, Jacob A Abraham, Pradip Bose, et al. Clear: Cross-layer exploration for architecting resilience - combining hardware and software techniques to tolerate soft errors in processor cores. In *Proceedings of the 53rd Annual Design Automation Conference (DAC)*, page 68. ACM, 2016.

[84] Vijay Janapa Reddi, Svilen Kanev, Wonyoung Kim, Simone Campanoni, Michael D Smith, Gu-Yeon Wei, and David Brooks. Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling. In *Proceedings of the 43rd Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 77–88. IEEE Computer Society, 2010.

[85] Ramon Bertran, Alper Buyuktosunoglu, Pradip Bose, Timothy J Slegel, Gerard Salem, Sean Carey, Richard F Rizzolo, and Thomas Strach. Voltage noise in multi-core processors: Empirical characterization and optimization opportunities. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 368–380. IEEE, 2014.

[86] Ronald G Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010.

[87] Ulya R Karpuzcu, Krishna B Kolluru, Nam Sung Kim, and Josep Torrellas. Varius-ntv: A microarchitectural model to capture the increased sensitivity of manycores to process variations at near-threshold voltages. In *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–11. IEEE, 2012.

[88] Xun Jiao, Abbas Rahimi, Yu Jiang, Jianguo Wang, Hamed Fatemi, Jose Pineda De Gyvez, and Rajesh K Gupta. Clim: A cross-level workload-aware timing error prediction model for functional units. *IEEE Transactions on Computers*, 67(6):771–783, 2018.

[89] Hari Cherupalli and John Sartori. Graph-based dynamic analysis: Efficient characterization of dynamic timing and activity distributions. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 729–735. IEEE, 2015.

[90] Tsung-Wei Huang and Martin DF Wong. Opentimer: A high-performance timing analysis tool. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 895–902. IEEE Press, 2015.

[91] David Harris and N Weste. CMOS VLSI Design. *Pearson Education, Inc*, 2010.

[92] Abbas Rahimi, Luca Benini, and Rajesh K Gupta. Analysis of instruction-level vulnerability to dynamic voltage and temperature variations. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1102–1105. IEEE, 2012.

[93] Youngtaek Kim, Lizy Kurian John, Indrani Paul, Srilatha Manne, and Michael Schulte. Performance boosting under reliability and power constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 334–341. IEEE, 2013.

[94] Chun-Kai Chang, Guanpeng Li, and Mattan Erez. Evaluating compiler ir-level selective instruction duplication with realistic hardware errors. In *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, pages 41–49. IEEE, 2019.

[95] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. Swift: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 243–254. IEEE Computer Society, 2005.

[96] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 385–396. ACM, 2010.

[97] Qining Lu, Guanpeng Li, Karthik Pattabiraman, Meeta S Gupta, and Jude A Rivers. Configurable detection of sdc-causing errors in programs. In *ACM Transactions on Embedded Computing Systems (TECS)*, volume 16, page 88. ACM, 2017.

[98] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[99] Kuang-Hua Huang and Jacob A Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

[100] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2012.

[101] D Nicholaeff, N Davis, D Trujillo, and RW Robey. Cell-based adaptive mesh refinement implemented with general purpose graphics processing units. *Tech. Rep. LA-UR-11-07127*, 2012.

[102] Guanpeng Li, Karthik Pattabiraman, Chen-Yang Cher, and Pradip Bose. Understanding error propagation in gpgpu applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 240–251. IEEE, 2016.

[103] Anna Thomas and Karthik Pattabiraman. Error detector placement for soft computation. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[104] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. Modeling soft-error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 27–38. IEEE, 2018.

[105] Lucas Palazzi, Guanpeng Li, Bo Fang, and Karthik Pattabiraman. A Tale of Two Injectors: End-to-End Comparison of IR-level and Assembly-Level Fault Injection. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2019.

[106] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W

Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[107] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[108] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[109] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[110] Ricardo Taborda and Jacobo Bielak. Large-scale earthquake simulation: computational seismology and complex engineering systems. *Computing in Science & Engineering*, 13(4):14–27, 2011.

[111] Hasan Metin Aktulga, Joseph C Fogarty, Sagar A Pandit, and Ananth Y Grama. Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques. *Parallel Computing*, 38(4-5):245–259, 2012.

[112] Sheng Di and Franck Cappello. Adaptive impact-driven detection of

silent data corruption for hpc applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2809–2823, 2016.

[113] Omer Subasi, Sheng Di, Leonardo Bautista-Gomez, Prasanna Balaprakash, Osman Unsal, Jesus Labarta, Adrian Cristal, and Franck Cappello. Spatial support vector regression to detect silent errors in the exascale era. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 413–424. IEEE, 2016.

[114] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: a large-scale field study. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):193–204, 2009.

[115] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 415–426. IEEE, 2015.

[116] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018.

[117] Subho S Banerjee, Saurabh Jha, James Cyriac, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 586–597. IEEE, 2018.

[118] Saurabh Jha, Subho Banerjee, Timothy Tsai, Siva KS Hari, Michael B Sullivan, Zbigniew T Kalbarczyk, Stephen W Keckler, and Ravishankar K Iyer. Ml-based fault injection for autonomous vehicles: a case for bayesian fault injection. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 112–124. IEEE, 2019.

[119] Hyungmin Cho, Eric Cheng, Thomas Shepherd, Chen-Yong Cher, and Subhasish Mitra. System-level effects of soft errors in uncore components. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(9):1497–1510, 2017.

[120] Konstantinos Aisopos and Li-Shiuan Peh. A systematic methodology to develop resilient cache coherence protocols. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 47–58. IEEE, 2011.