# Understanding and Modeling Error Propagation in Programs

by

Guanpeng Li

Bachelor of Applied Science, University of British Columbia, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

**Doctor of Philosophy**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

February 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

## Understanding and Modeling Error Propagation in Programs

submitted by **Guanpeng Li** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Electrical and Computer Engineering**.

**Examining Committee:**

Karthik Pattabiraman, Electrical and Computer Engineering
*Supervisor*

Matei Ripeanu, Electrical and Computer Engineering
*Supervisory Committee Member*

Ali Mesbah, Electrical and Computer Engineering
*Supervisory Committee Member*

Ronald Garcia, Computer Science
*University Examiner*

Sudip Shekhar, Electrical and Computer Engineering
*University Examiner*

# Abstract

Hardware errors are projected to increase in modern computer systems due to shrinking feature sizes and increasing manufacturing variations. The impact of hardware faults on programs can be catastrophic, and can lead to substantial financial and societal consequences. Error propagation is often the leading cause of catastrophic system failures, and hence must be mitigated. Traditional hardware-only techniques to avoid error propagation are energy hungry, and hence not suitable for modern computer systems (i.e., commodity systems). Researchers have proposed selective software-based protection techniques to prevent error propagation at lower costs. However, these techniques use expensive fault injection simulations to determine which parts of a program must be protected. Fault injection simulation artificially introduces a fault to program execution and observe failures (if any) upon the completion of the program execution. Thousands of such simulations need to be performed in order to achieve statistical significance. It is time-consuming as even a single program execution of a common application may take a long time. In this dissertation, I first characterize error propagation in programs that lead to different types of failures, proposed both empirical and analytical approaches to identify and mitigate error propagation without expensive fault injections. The key observation is that only a small fraction of states are responsible for almost all error propagation in programs, and the propagation falls into identifiable patterns which can be modeled efficiently. The proposed techniques are nearly as close as fault injection approaches in measuring failure rates of programs, and orders of magnitude faster than fault injections. This allows developers to build low-cost fault-tolerant applications in an extremely efficient manner.

# Lay Summary

Transient hardware faults become more and more prevalent nowadays. They often lead to error propagation in programs, which may cause serious social and financial impact. Protection techniques such as hardware duplications were used in the past but they incur huge overheads in performance and energy consumption. Researchers have expected modern software to tolerate hardware faults in a low-cost and flexible manner. Studies have found there is only a small amount of program states that are responsible for almost all the propagations. If those states can be identified and protected, we can improve the reliability of computer systems at low cost.

In this thesis, we characterize error propagation in programs, and propose models to identify the vulnerability of program states using static and dynamic analysis techniques.

# Preface

This thesis is the result of work carried out by me, in collaboration with my advisor (Dr. Karthik Pattabiraman), my colleague (Qining Lu), and other research scientists from IBM and NVIDIA. Chapters 4, 5, 6, 7, and 8 are based on work published in the conferences of DSN and SC. In each work, I was responsible for writing the paper, designing approaches (where applicable), conducting experiments, analyzing data, and evaluating the results. Other collaborators were responsible for editing and writing portion of the manuscripts, providing guidance and feedback.

Below are publication details for each chapter:

- Chapter 4

  — Guanpeng Li, Qining Lu and Karthik Pattabiraman, "Fined-grained Characterization of Long Latency Causing Crashes in Programs", *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015, 250-261. [88]

- Chapter 5

  — Guanpeng Li, Karthik Pattabiraman, Chen-Yong Cher and Pradip Bose, "Understanding Error Propagation in GPGPU Applications", *IEEE International Conference for High-Performance Computing, Storage and Networking (SC)*, 2016, 240-251. [90]

- Chapter 6

  — Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan and Timothy Tsai, "Modeling Soft-Error Propagation in Programs",

*IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, 27-38, [59]

- Chapter 7

  — Guanpeng Li and Karthik Pattabiraman, "Modeling Input-Dependent Error Propagation in Programs", *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, 279-290. [87]

- Chapter 8

  — Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler, "Understanding Error Propagation in Deep-Learning Neural Networks (DNN) Accelerators and Applications", *ACM International Conference for High-Performance Computing, Storage and Networking (SC)*, 2017, 8:1-8:12. [91]

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgments

First of all, I would like to express my sincere gratitude to my doctoral advisor Dr. Karthik Pattabiraman for his support in the past years. If it was not for his continuous investment in me, I would not have been able to complete this thesis. Karthik's advice on my career has been invaluable and helped me grow as a researcher.

Besides my advisor, I would like to thank my thesis committee members and examiners, who gave me valuable suggestions and helped me improve my thesis. I would also like to give my special thanks to Prof. Matei Ripeanu, who has provided me feedback and advice over the past years, and sat through many practice talks I have given.

In the end, thanks to all my friends and lab mates for making these years as joyful as possible. I also want to thank my families, who have supported me unconditionally in my study and career.

# Chapter 1

# Introduction

Transient hardware faults (i.e., soft errors) are predicted to increase in future computer systems [20, 40]. These faults are caused by high-energy particles passing through transistors, causing them to accumulate charge or discharge, eventually leading to single bit flips in logic values. As a result, such faults are also known as Single Event Upset (SEU). Due to the growing system scale, progressive technology scaling, and lowering operating voltages, even a small amount of charge accumulated or lost in the circuit can cause a SEU in modern processors [20, 40]. Consequently, computer systems have become more and more vulnerable to hardware faults. In the past, only highly reliable systems (e.g., aerospace applications) required protections from SEUs, but nowadays even commodity systems (e.g., consumer products) need to be protected due to the increasing error rates [21, 95, 120].

SEUs in hardware often result in error propagation in programs, which may lead to catastrophic outcomes. For example, Amazon's S3 server once went down for a few hours in 2008, causing a financial loss of millions of dollar for the company [10]. As reported by Amazon, it was very likely to be a SEU in their hardware, causing error propagation in the software, and finally leading to the incident [10]. Until a few years ago, error propagation was mostly masked through hardware-only solutions such as dual or triple modular redundancy and circuit hardening. However, these techniques are becoming increasingly challenging to deploy as they consume significant amounts of energy, and as energy is becoming a first-class constraint in processor design, especially in commodity systems [21]. On

the other hand, software-implemented protection techniques are more flexible and cost-effective as they can selectively protect the program states of interest and do not require any expensive hardware modifications [95, 102, 120]. Therefore, researchers have postulated that future processors will expose hardware faults to the software and expect the software to tolerate the faults at low costs [95, 102, 120].

Studies have shown that only a small fraction of program states are responsible for almost all the error propagations leading to catastrophic failures [52, 124], and so one can selectively protect these states to meet the reliability target of the application while incurring lower energy and performance overheads than full duplication techniques [64, 97]. Therefore, in the development of fault-tolerant applications, it is important to understand what states in the program are vulnerable to what kinds of failures caused by SEUs, and identify those vulnerable states so that they can be protected with low costs.

Fault Injection (FI) has been commonly employed to characterize error propagation, and identify which program states are vulnerable to failures. FI involves perturbing the program state to emulate the effect of a hardware fault and executing the program to completion to determine if the fault caused a certain failure [64, 147]. However, real-world programs may consist of billions of dynamic instructions, and even a single execution of the program may take a long time. Performing thousands of FIs to get statistically meaningful results for each instruction takes too much time to be practical [65, 124]. As a result, FI is mostly performed offline for the characterizations of application resilience. It is too slow to be deployed as an online evaluation method when developing fault-tolerant applications in a fast software development cycle. In this dissertation, we leverage FI as a basic approach to characterize error propagation in programs, and use the results to understand how different errors propagate in the program and why. Based on the empirical observations, we propose models and heuristics that analyze error propagation and guide the protection of programs with few to no FIs.

As Figure 1.1 shows, once a fault occurs and is read by the program that is executing, the fault is activated and becomes an error. Consequently, the error starts propagating in the program, leading to one of the three outcomes: (1) Benign: The error can be masked during the propagation in the program, hence the program finishes its execution without any anomaly. (2) Crash: The error triggers a hardware

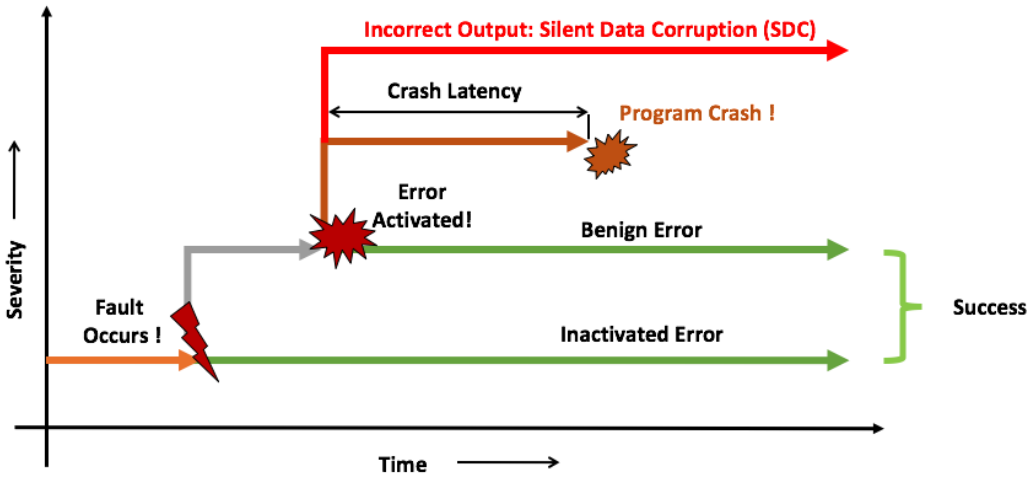



**Figure 1.1:** Types of Failures

exception (i.e., illegal memory access) and causes the program to terminate before it finishes the execution. (3) Silent Data Corruption (SDC): The error propagates to the program output, modifying it from the correct output. Among the three outcomes, benign, as its name suggests, is not considered to be harmful as the program successfully completes the execution without any anomaly. Crash and SDC are both regarded as failures, depending on the reliability target and applications. Therefore, they are the failure types we will focus on in this dissertation.

Crashes are usually considered detectable failures because of the early termination of the program execution and the warnings issued by hardware exceptions. The user can then simply restart the program for a correct execution. As a result, researchers do not really pay much attention to crashes. However, their underlying assumption is that failures are reported as soon as faults occur, which is also known as the fail-stop assumption [150]. While this assumption holds most of the time, we find that there is a small amount of errors that can actually propagate for a long time before causing crashes. Hence, such errors violate the fail-stop assumption. We call them long latency crashes, or LLCs. Consequently, the error may propagate to the program's state elements such as checkpoints and corrupt them

before causing any crash, leading to failures in recovery. Studies show that LLCs may drastically reduce the availability of systems, and hence, must be identified and mitigated in systems that require high availability [150]. The main challenge of identifying LLCs is that LLC are typically confined to a small fraction of the program's data, and finding the LLC-causing data through FI often causes search space explosion. The key observation in this study is that we find that most LLCs are caused by faults occurring in a few code structures that can be identified by static and dynamic program analysis. Based on this observation, we proposed heuristics to identify those code structures in order to protect the program from LLCs.

SDCs, on the other hand, are considered as the most insidious type of failure because there are no indications of the incorrect program outputs. Studies show that the program SDC probabilities are essentially application-dependent, and vary from less than 1% to more than 50% depending on the program [97, 147]. Therefore, given different reliability targets and applications, SDCs must be evaluated and mitigated on a per-program and per-target basis. As a result, it is critical to accurately estimate the SDC probability of a program as well as its individual instructions for evaluation and mitigation. Unfortunately, current approaches for identifying SDCs such as FIs are too slow to integrate into fast software development cycles. Other existing approaches for resilience estimation such as analytical models, suffer from serious inaccuracies because error propagation is complicated and has not been comprehensively understood [51, 100]. The dynamic nature of program execution makes building an accurate model extremely challenging. In this dissertation, we find that almost all the error propagations that lead to SDCs can be abstracted into three levels. We use this insight to build TRIDENT, which is a compiler-based model that is able to predict the SDC probabilities of both program and individual instructions without any FIs.

While the first part of this dissertation aims to investigate error propagation in general applications (e.g., CPU programs), the second part discusses the error resilience of applications that run on hardware accelerators. Hardware accelerators such as graphic processing units (GPUs) and deep learning neural network (DNN) accelerators have found wide adoptions nowadays due to their massive parallel capability and efficient data-flow. Recently, they have been deployed for running

reliability- and safety-critical applications (e.g., scientific applications and self-driving cars etc), which require strict correctness. Therefore, it is important to understand the resilience of applications running on these accelerators. Unfortunately, most studies focus on the performance aspects of the accelerators, and hence their resilience is not well investigated. While we explored the error propagation in general applications, we are interested in error propagation in the applications running on those accelerators, which have a very different structure from general applications. One of the challenges in such applications is the lack of capable tools to perform fault injection experiments and analysis. Therefore, we first built tools that are able to perform fault injections for the applications, and used them to characterize their error propagations. We find that the characteristics of their error propagation are very different from what we have seen in CPU programs. This is because of the different architectures and program models used in accelerators. Finally, we quantify the resilience of the applications and propose efficient protection techniques based on their unique error propagation characteristics.

## 1.1 Contributions

Our research contributions are as follows:

- Characterized LLCs in programs, and identified the code patterns that lead to LLCs. Based on the characterization, we developed an automated compiler-based technique, CRASHFINDER, to identify the code patterns in programs. We showed CrashFinder is able to identify more than 90% of the locations that cause LLCs without extensive FIs, and by protecting those identified locations, most of the LLCs can be eliminated in programs.

- Built an automated model, TRIDENT, that analyzes the SDC probabilities of both whole programs and individual instructions in the program. Given a program, the model requires no fault injections to estimate the SDC probabilities. We showed that most of the error propagations that lead to SDCs can be described by a combination of three probabilistic modules, which synergistically build on top of each other to model error propagation. We evaluated TRIDENT experimentally, and found that it is able to predict the

SDC probabilities almost as accurately as FI, but takes only a fraction of the time that FI does, thereby making it possible to integrate it into the software development cycle. We also extended the model to support multiple program inputs. We showed that the extended model is able to bound SDC probabilities of programs given multiple program inputs.

- Investigated the characteristics of error propagation in the applications running on GPUs and DNN accelerators. We first built FI tools and simulators for the applications, and evaluated their error resilience. Based on the observations, we proposed mitigation techniques that are cost-effective for the applications.

The investigation of error propagation in this dissertation allows developers to design more cost-effective error detectors and protection techniques based on the characteristics of different errors, programming models, and hardware platforms. The studies also explain why certain vulnerabilities are observed in certain program structures. Our proposed analytical models allow developers to estimate programs' resilience and selectively protect programs in a fast and accurate way, which represents a fundamental advance in the way fault-tolerant applications are designed. Moreover, the analytical nature of our proposed models explains the relations between program and resilience, providing insights to researchers for designing better fault-tolerant applications in the future.

# Chapter 2

# Related Work

There have been many papers that investigate error propagation in programs and tolerate hardware faults through software techniques. We organize this chapter based on the structure shown in Figure 2.1. First, we discuss the related work that studied error propagation in CPU programs. More specifically, we are interested in the ones that investigate LLCs (Chapter 2.1.1), and SDCs (Chapter 2.1.2). Secondly, we talk about the studies that characterized the resilience of GPU applications (Chapter 2.2.1), and DNN accelerator applications (Chapter 2.2.2).

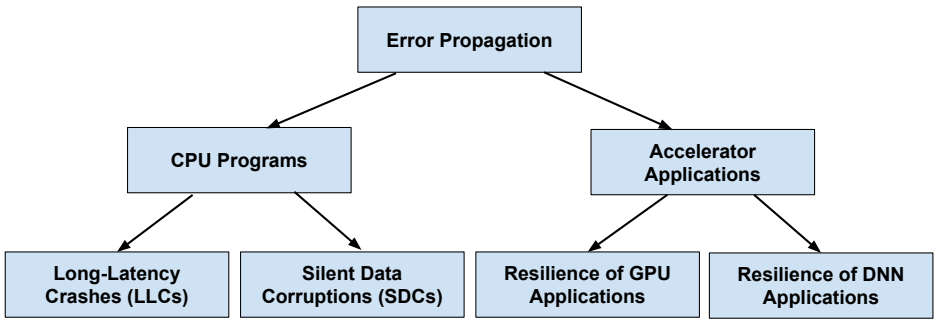



**Figure 2.1:** High-Level Organization of Chapter 2

## 2.1 Error Propagation in CPU Programs

### 2.1.1 Long-Latency Crash (LLC)

In Chapter 4, we develop a technique to automatically find program locations where LLC causing faults originate so that the locations can be protected to bound the program's crash latency. In the past, there have been a few studies that characterized the latency of error propagation in programs. We consider the related work below.

The first one by Gu et. al. [9] injected faults into the Linux kernel and found that less than 1% of the errors resulted in LLCs. They further found that many of the severe failures that result in extended downtimes in the system were caused by these LLCs, due to error propagation. The authors give examples of faults that resulted in the LLCs, but they do not attempt to categorize the code patterns that were responsible for the LLCs. The second study is by Yim et al. [150], who studied the correlation between LLC-causing errors and the fault location in the program. However, they perform a coarse-grained categorization of the fault locations based on where the data resides (e.g., stack, heap etc.). Such a coarse-grained categorization is unfortunately not very useful when one wants to protect specific variables or program locations, as protecting the entire stack/heap segment is too expensive. Although they provide some insights on the characteristics of possible LLC-causing errors, they do not develop an automated way to predict which faults would lead to an LLC and which would not. It is also worth noting that neither of the above papers achieves exhaustive characterization of the LLC-causing faults. Rashid et. al. [20] have built an analytical trace-based model to predict the propagation of intermittent hardware errors in a program. The model can be used to predict the latency of crash causing faults in the program, and hence find the LLC locations. They find that less than 0.5% of faults cause LLCs using the model. While useful, their model requires building the program's Dynamic Dependence Graph (DDG), which can be prohibitively expensive for large programs as it is directly proportional to the number of instructions executed by it. Further, they make many simplifying assumptions in their model which may not hold in the real world. Similarly, Lanzaro et. al. [84] built an automated tool which is able to analyze arbitrary memory

corruptions based on execution trace when faults present in system. While their technique is useful in terms of analyzing error propagation, it incurs prohibitive overheads as it requires the entire trace to be captured at runtime. Further, they focus on software faults as opposed to hardware faults. Finally, they do not make any attempt to identify LLC-causing faults. Chandra et. al. [29] study program errors that violate the fail-stop model and result in corrupting the data written to permanent storage, or communicated to other processes. They find that between 2% to 7% of faults cause such violations, and propose using a transaction-based mechanism to prevent the propagation of these faults. While transaction-based techniques are useful, they require significant software engineering effort, as the application needs to be rewritten to use transactions. This is very difficult for most commodity systems. In contrast to the above techniques, our technique identifies specific program locations that result in LLCs, and can hence support fine-grained protection. Further, it uses predominantly static analysis coupled with dynamic analysis and a selective fault injection experiment, making it highly scalable and efficient compared to the above approaches. Finally, our technique, CRASHFINDER, does not require any programmer intervention or application rewriting and is hence practical to deploy on existing software.

### 2.1.2 Silent Data Corruption (SDC)

In Chapter 5, we construct a three-level model, TRIDENT, that captures error propagation at the static data dependency, control-flow and memory levels, based on the observations of error propagations in programs. TRIDENT is implemented as part of a compiler, and can predict both the overall SDC probability of a given program, and the SDC probabilities of individual instructions of programs, without fault injections.

There is a significant body of work on identifying error propagation that leads to SDC either through FI [42, 58, 65, 66, 90, 147], or through modeling error propagation in programs [51, 52, 130]. The main advantage of FI is that it is simple, but it has limited predictive power. Further, its long running time often limits the FI approach from deriving program vulnerabilities at finer granularity (e.g., SDC probabilities of individual instructions). The main advantage of modeling tech-

niques is that they have predictive power and are significantly faster, but existing techniques suffer from poor accuracy due to important gaps in the models. The main question we answer in the TRIDENT project is that whether we can combine the advantages of the two approaches by constructing a model that is both accurate and scalable. Shoestring [52] was one of the first papers to attempt to model the resilience of instructions without using fault injection. Shoestring stops tracing error propagations after control-flow divergence, and assumes that any fault that propagates to a store instruction leads to an SDC. Hence, it is similar to removing fc and fm in TRIDENT and considering only fs , which we show is not very accurate. Further, Shoestring does not quantify SDC probabilities of programs and instructions, unlike TRIDENT. Gupta et al. [61] investigate the resilience characteristics of different failures in large-scale systems. However, they do not propose automated techniques to predict failure rates. Lu et al. [97], Li et al. [88] identify vulnerable instructions by characterizing different features of instructions in programs. While they develop efficient heuristics in finding vulnerable instructions in programs, their techniques do not quantify error propagation, and hence cannot accurately pinpoint SDC probabilities of individual instructions. Sridharan et al. [100] introduce PVF, an analytical model which eliminates microarchitectural dependency from architectural vulnerability to approximate SDC probabilities of programs. While the model requires no FIs and is hence fast, it has poor accuracy in determining SDC probabilities as it does not distinguish between crashes and SDCs. Fang et al. [51] introduce ePVF, which derives tighter bounds on SDC probabilities than PVF, by omitting crash-causing faults from the prediction of SDCs. However, both techniques focus on modeling the static data dependency of instructions, and do not consider error propagation beyond control-flow divergence, which leads to large gaps in the predictions of SDCs (as we showed in Chapter 5).

Furthermore, we study the variation of SDC probabilities across different inputs of a program, and identify the reasons for the variations in Chapter 6 . Based on the observations, we propose a model, vTRIDENT, which predicts the variations in programs' SDC probabilities without any FIs, for a given set of inputs.

There has been little work investigating error propagation behaviours across different inputs of a program. Czek et al. [43] were among the first to model the variability of failure rates across program inputs. They decompose program

executions into smaller unit blocks (i.e., instruction mixes), and use the volatility of their dynamic footprints to predict the variation of failure rates, treating the error propagation probabilities as constants in their unit blocks across different inputs. Their assumption is that similar executions (of the unit blocks) result in similar error propagations, so the propagation probabilities within the unit blocks do not change across inputs. Thus, their model is equivalent to considering just the execution volatility of the program (Section 6.2), which is not very accurate as we show in Section 6.3.

Folkesson et al. [54] investigate the variability of the failure rates of a single program (*Quicksort*) with its different inputs. They decompose the variability into the execution profile, and its data usage profile. The latter requires the identification of critical data and its usage within the program - it is not clear how this is done. They consider limited propagation of errors across basic blocks, but not within a single block. This results in their model significantly underpredicting the variation of error propagation. Finally, it is difficult to generalize their results as they consider only one (small) program.

Di Leo et al. [46] investigate the distribution of failure types under hardware faults when the program is executed with different inputs. However, their study focuses on the measurement of the volatility in SDC probabilities, rather than on predicting it. They also attempt to cluster the variations and correlate the clusters with the program's execution profile. However, they do not propose a model to predict the variations, nor do they consider sources of variation beyond the execution profile - again, this is similar to using only the execution volatility to explain the variation of SDC probabilities. Tao et al. [139] propose efficient detection and recovery mechanisms for iterative methods across different inputs. Mahmoud et al. [98] leverage software testing techniques to explore input dependence for approximate computing. However, neither of them focus on hardware faults in generic programs. Gupta et al. [61] measure the failure rate in large-scale systems with multiple program inputs during a long period, but they do not propose techniques to bound the failure rates. In contrast, our work investigates the root causes behind the SDC volatility under hardware faults, and proposes a model to bound it in an accurate and scalable fashion.

Other papers that investigate error propagation confine their studies to a single

input of each program. For example, Hari et al. [65, 66] group similar executions and choose the representative ones for FI to predict SDC probabilities given a single input of each program. Li et al. [88] find patterns of executions to prune the FI space when computing the probability of long-latency propagating crashes. Lu et al. [97] characterize error resilience of different code patterns in applications, and provide configurable protection based on the evaluation of instruction SDC probabilities. Feng et al. [52] propose a modeling technique to identify likely SDC-causing instructions. Our prior work. TRIDENT [59], which vTRIDENT is based on, also restricts itself to single inputs. These papers all investigate program error resilience characteristics based on static and dynamic analysis, without large-scale FI. However, their characterizations are based on the observations derived from a single input of each program, and hence their results may be inaccurate for other inputs.

## 2.2 Error Propagation in Hardware Accelerator Applications

### 2.2.1 GPU Applications

In Chapter 7, we perform an empirical study to understand and characterize error propagation in GPU applications. We build a compiler-based fault-injection tool for GPU applications to track error propagation, and define metrics to characterize propagation in GPU applications. We consider the following related work in the area of GPU fault injection and fault-tolerance techniques.

Yim et al. [151] built one of the first fault injectors for GPGPU applications. However, their injector operates at the source code level, and only considers faults that are visible at the source level. Fang et. al. [50] designed a GPGPU fault injector, GPU-Qin, that operates on the CUDA assembly code (SASS). They use the CUDA debugger (CUDA-gdb) to inject faults, which takes significantly longer than performing fault injections by code instrumentation (Section III). In follow up work, Hari et. al. [63] built SASSIFI, a GPGPU fault injector that transforms the SASS code of the program to inject faults similar to LLFI-GPU. Both GPU-Qin and SASSIFI operate on the SASS assembly code level, which makes it difficult

to map back the faults to the source code. In contrast, LLFI-GPU operates at the LLVM IR level, which is much closer to the program's source code. This makes it possible to perform program analysis on the program and map the fault injection results back to the source code, thereby helping programmers make their code error resilient.

There have been a few papers on building fault tolerance techniques on GPGPU platforms. Jeon et. al. [76] duplicated kernel threads that have the same input to detect errors. Dimitrov et. al. [48] leverage both instruction level and thread-level parallelism to duplicate application code. Tan et. al. [137] proposed an analytical method to evaluate the error-resilience of GPU platforms. Pena et. at. [110] explored low-cost data protection and recovery mechanisms for GPGPU platforms based on API interception. Finally, Yim et al. [151] proposed a technique to detect errors by duplicating code at within the loop bodies of GPGPU programs. While these are useful, none of the above papers study error propagation in GPGPU programs, which is our focus.

### 2.2.2 DNN Accelerators and Applications

In Chapter 8, we experimentally evaluate the resilience characteristics of DNN systems (i.e., DNN software running on specialized accelerators). We find that the error resilience of a DNN system depends on the data types, values, data reuses, and types of layers in the design. Based on our observations, we propose two efficient protection techniques for DNN systems. We consider the following related work.

There were a few studies years ago on the fault tolerance of neural networks [9, 17, 111]. While these studies performed fault injection experiments on neural networks and analyzed the results, the networks they considered had very different topologies and many fewer operations than today's DNNs. Further, these studies were neither in the context of the safety critical platforms such as self-driving cars, nor did they consider DNN accelerators for executing the applications. Hence, they do not provide much insight about error propagation in modern DNN systems. Reis et al. [115] and Oh et al. [106] proposed software techniques that duplicate all instructions to detect soft errors. Due to the high overheads of these approaches, researchers have investigated selectively targeting the errors that are

important to an application. For example, Hari et al. analyzed popular benchmarks and designed application-specific error detectors. Feng et al. [52] proposed a static analysis method to identify vulnerable parts of the applications for protection. Lu et al. [97] identified SDC-prone instructions to protect using a decision tree classifier, while Laguna et al. [32] used a machine learning approach to identify sensitive instructions in scientific applications. While these techniques are useful to mitigate soft errors in general purpose applications, they cannot be easily applied to DNN systems which implement a different instruction set and operate on a specialized microarchitecture. Many studies [34, 35, 62] proposed novel microarchitectures for accelerating DNN applications. These studies only consider the performance and energy overheads of their designs and do not consider reliability. In recent work, Fernades et. al. [53] evaluated the resilience of histogram of oriented gradient applications for self-driving cars, but they did not consider DNNs. Reagen et.al. [113] and Chatterjee et.al. [32] explored energy-reliability limits of DNN systems, but they focused on different networks and fault models. The closest related work is by Temam et. al. [140], which investigated the error resilience of neural network accelerators at the transistor-level. Our work differs in three aspects: (1) Their work assumed a simple neural network prototype, whereas we investigate modern DNN topologies. (2) Their work does not include any sensitivity study and is limited to the hardware datapath of a single neural network accelerator. Our work explores the resilience sensitivity to different hardware and software parameters. (3) Their work focused on permanent faults, rather than soft errors. Soft errors are separately regulated by ISO 26262, and hence we focus on them.

# Chapter 3

# Background

In this chapter, we first introduce our fault model for studying error propagation in CPU programs. The fault model applies to Chapter 4, 5, 6 and 7. For Chapter 8, we present the fault model in the chapter. We then define the terms we have in this thesis and introduce LLVM compiler that we use to study error propagation in both CPU and GPU programs. Finally, we summarize GPU architecture and programming model before discussing DNN accelerators and applications.

## 3.1 Fault Model

We consider transient hardware faults that occur in the computational elements of the processor, including pipeline registers and functional units. We do not consider faults in the memory or caches, as we assume that these are protected with error correction code (ECC). Likewise, we do not consider faults in the processor's control logic as we assume that it is protected. Neither do we consider faults in the instructions' encodings. Finally, we assume that the program does not jump to arbitrary illegal addresses due to faults during the execution, as this can be detected by control-flow checking techniques [105]. However, the program may take a faulty legal branch (the execution path is legal but the branch direction can be wrong due to faults propagating to it). Our fault model is in line with other work in the area [42, 52, 65, 97].

## 3.2 Terms and Definitions

We use the following terms in this thesis.

- **Fault occurrence**: The event corresponding to the occurrence of the hardware fault. The fault may or may not result in an error.
- **Fault activation**: The event corresponding to the manifestation of the fault to the software, i.e., the fault becomes an error and corrupts some portion of the software state (e.g., register, memory location). The error may or may not result in a crash.
- **Crash:** The raising of a hardware trap or exception due to the error, because the program attempted to perform an action it should not have (e.g., read outside its memory segments).
- **Crash latency**: The number of dynamic instructions executed by the program from fault activation to crash. This definition is slightly different from prior work which has used CPU cycles to measure the crash latency. The main reason we use dynamic instructions rather than CPU cycles is that we wish to obtain a platform independent characterization of long latency crashes.
- **Long latency crashes (LLCs)**: Crashes that have crash latency of greater than 1,000 dynamic instructions. Prior work has used a wide range of values for long latency crashes, ranging from 10,000 CPU cycles [112] to as many as 10 million CPU cycles [150]. We use 1,000 instructions as our threshold as (1) each instruction corresponds to multiple CPU cycles in our system, and (2) we found that in our benchmarks, the length of the static data dependency sequences are far smaller, and hence setting 1,000 instructions as the threshold already filters out 99% of the crash-causing faults (Section 4.3), showing that 1000 instructions is a reasonable threshold.
- **Silent Data Corruption (SDC):** A mismatch between the output of a faulty program run and that of an error-free execution of the program.

- **Benign Faults:** Program output matches that of the error-free execution even though a fault occurred during its execution. This means either the fault was masked or overwritten by the program.
- **Error propagation:** Error propagation means that the fault was activated, and has affected some other portion of the program state, say 'X'. In this case, we say the fault has propagated to state X. We focus on the faults that affect the program state and therefore consider error propagation at the application level.
- **SDC Probability:** We define the SDC probability as the probability of an SDC given that the fault was activated – other work uses a similar definition [52, 66, 88, 147].

## 3.3 LLVM Compiler

There are many FI frameworks in the literature [7, 26, 78, 132], which focus on different platforms, components, and faults. We use the LLVM compiler [85] to perform our program analysis and FI experiments and to implement our model. Our choice of LLVM is motivated by three reasons. First, LLVM uses a typed intermediate representation (IR) that can easily represent source-level constructs. In particular, it preserves the names of variables and functions, which makes source mapping feasible. This allows us to perform a fine-grained analysis of which program locations cause certain failures and map them to the source code. Second, LLVM IR is a platform-neutral representation that abstracts out many low-level details of the hardware and assembly language. This greatly aids in portability of our analysis to different architectures and simplifies the handling of the special cases in different assembly language formats. Finally, LLVM IR has been shown to be accurate for doing FI studies [147], and there are many fault injectors developed for LLVM [12, 90, 121, 147]. Many of the papers we compare our technique with in this chapter also use LLVM infrastructure [51, 52]. Therefore, when we say instruction in CPU or GPU programs, we mean an instruction at the LLVM IR level.

## 3.4 GPU Architecture and Programming Model

We focus on CUDA (Compute Unified Device Architecture) GPU programs in this thesis as CUDA the most popular programming model used by GPU developers [2]. CUDA is a parallel computing platform and application programming interface model created by Nvidia [2]. CUDA kernels, which are the parts of the code that run on GPU hardware, adopt the single instruction multiple threads (SIMT) model to exploit the massive parallelism of GPU devices. From a software perspective, the CUDA programming model abstracts the SIMT model into a hierarchy of kernels, blocks and threads. The CUDA kernels consist of blocks, which consist of threads. This hierarchy allows various levels of parallelism such as fine-grained data parallelism, coarse-grained data parallelism and task parallelism. From a hardware perspective, blocks of threads run on streaming mutiprocessors (SMs) which have on-chip shared memory for threads inside the same block. Within a block, threads are launched in fixed groups of 32 threads, which are called warps. Threads in a warp execute the same sequence of instructions but with different data values.

GPU has its own memory space that is distinct from and not synchronized with the host CPU's memory. In the CUDA programming model, there are four kinds of memory: (1) global, (2) constant, (3) texture, and (4) shared. Global, constant, and texture memory accesses are generally slower from large device memory. Shared memory space, which can be software managed, is much smaller and built on chip, hence it is much faster to access. CUDA applications need to be aware of the memory hierarchy to access GPU memory efficiently.

## 3.5 DNN Accelerators and Applications

### 3.5.1 Deep Learning Neural Networks

A deep learning neural network is a directed acyclic graph consisting of multiple computation layers [86]. A higher level abstraction of the input data or a feature map (fmap) is extracted to preserve the information that are unique and important in each layer. There is a very deep hierarchy of layers in modern DNNs, and hence their name.

We consider convolutional neural networks of DNNs, as they are used in a

broad range of DNN applications and deployed in self-driving cars, which are our focus. In such DNNs, the primary computation occurs in the convolutional layers (CONV) that perform multi-dimensional convolution calculations. The number of convolutional layers can range from three to a few tens of layers [67, 81]. Each convolutional layer applies a kernel (or filter) on the input fmaps (ifmaps) to extract underlying visual characteristics and generate the corresponding output fmaps (ofmaps).

Each computation result is saved in an activation (ACT) after being processed by an activation function (e.g., ReLU), which is in turn, the input of the next layer. An activation is also known as a neuron or a synapse - in this work, we use ACT to represent an activation. ACTs are connected based on the topology of the network. For example, if two ACTs, A and B, are both connected to an ACT C in the next layer, then ACT C is calculated using ACT A and B as the inputs. In convolutional layers, ACTs in each fmap are usually fully connected to each other, whereas the connection of ACTs between each fmap in other layers are usually sparse. In some DNNs, a small number (usually less than 3) of fully-connected layers (FC) are typically stacked behind the convolutional layers for classification purposes. In between the convolutional and fully-connected layers, additional layers can be added, such as the pooling (POOL) and normalization (NORM) layers. POOL selects the ACT of the local maximum in an area to be forwarded to the next layer and discards the rest in the area, so the size of fmaps will become smaller after each POOL. NORM averages ACT values based on the surrounding ACTs. Thus ACT values will also be modified after the NORM layer.

Once a DNN topology is constructed, the network can be fed with training input data, and the associated weights, abstracted as connections between ACTs, will be learned through a back-propagation process. This is referred to as the *training* phase of the network. The training is usually done once, as it is very time-consuming, and then the DNN is ready for image classification with testing input data. This is referred to as the *inferencing* phase of the network and is carried out many times for each input data set. The input of the inferencing phase is often a digitized image, and the output is a list of output candidates of possible matches such as *car, pedestrian, animal*, each with a confidence score. Self-driving cars deploy DNN applications for inferencing, and hence, we focus on the inferencing

phase of DNNs.

### 3.5.2 DNN Accelerator

Many specialized accelerators [34, 35, 62] have been proposed for DNN inferencing, each with different features to cater to DNN algorithms. However, there are two properties common to all DNN algorithms that are used in the design of all DNN accelerators: (1) MAC operations in each feature map have very sparse dependencies, which can be computed in parallel, and (2) there are strong temporal and spatial localities in data within and across each feature map, which allow the data to be strategically cached and reused. To leverage the first property, DNN accelerators adopt spatial architectures [143], which consist of massively parallel processing engines (PEs), each of which computes MACs. Figure 3.1 shows the architecture of a general DNN accelerator. A DNN accelerator consists of a global buffer and an array of PEs. The accelerator is connected to DRAM where data is transferred from. A CPU is usually used to off-load tasks to the accelerator. The overall architecture is shown in Figure 3.1A. The ALU of each PE consists of a multiplier and an adder as execution units to perform MACs — this is where the majority of computations happen in DNNs. A general structure of the ALU in each PE is shown in Figure 3.1B.

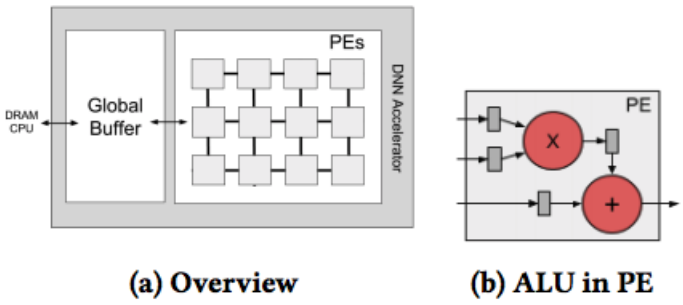



**Figure 3.1:** Architecture of general DNN accelerator

To leverage the second property of DNN algorithms, special buffers are added on each PE as local scratchpad to cache data for reuse. Each DNN accelerator may implement its own dataflow to explore data localities. We classify the localities in

DNNs into three major categories:

- **Weight Reuse:** Weight data of each kernel can be reused in each fmap as the convolutions involving the kernal data are used many times on the same ifmap.
- **Image Reuse:** Image data of each fmap can be reused in all convolutions where the ifmap is involved, because different kernels operate on the same sets of ifmap in each layer.
- **Output Reuse:** Computation results of MACs can be buffered and consumed on-PE without transferring off the PEs.

Table 3.1 illustrates nine different DNN accelerators that have been proposed in prior work and the corresponding data localities they exploit in their dataflows. As can be seen, each accelerator exploits one of more localities in its dataflow. Eyeriss [35] considers all of the three data localities in its dataflow.

We separate faults that originate in the datapath (i.e., latches in execution units) from those that originate in buffers (both on- and off-PEs), because they propagate differently: faults in the datapath will be only read once, whereas faults in buffers may be read multiple times due to reuse and hence the same fault can be spread to multiple locations within short time windows.

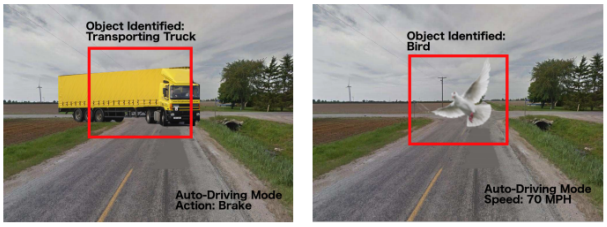
### 3.5.3 Consequences of Soft Errors

The consequences of soft errors that occur in DNN systems can be catastrophic as many of them are safety-critical, and error mitigation is required to meet certain reliability targets. For example, in self-driving cars, a soft error can lead to misclassification of objects, resulting in a wrong action taken by the car. In our fault injection experiments, we found many cases where a truck can be misclassified under a soft error. We illustrate this in Figure 3.2. The DNN in the car should classify the coming object as a transporting truck in a fault-free execution and apply the brakes in time to avoid a collision (Figure 3.2A). However, due to a soft error in the DNN, the truck is misclassified as a bird (Figure 3.2B), and the braking action may not be applied in time to avoid the collision, especially when

**Table 3.1:** Data Reuses in DNN Accelerators

| | **Weight Reuse** | **Image Reuse** | **Output Reuse** |
|---|---|---|---|
| Zhang et al. [153], Diannao [34], Dadiannao [36] | N | N | N |
| Chakradhar et al. [28], Sriram et al. [131], Sankaradas et al. [122], nn-X [57], K-Brain [107], Origami [27] | Y | N | N |
| Gupta et al. [60], Shidiannao [49], Peemen et al. [109] | N | N | Y |
| Eyeriss [35] | Y | Y | Y |

the car is operating at high speed. This is an important concern as it often results in the violation of standards such as ISO 26262 dealing with the functional safety of road vehicles [119], which requires the System on Chip (SoC) carrying DNN inferencing hardware in self-driving cars to have a soft error FIT rate less than 10 FIT [13], regardless of the underlying DNN algorithm and accuracy. Since a DNN accelerator is only a fraction of the total area of the SoC, the FIT allowance of a DNN accelerator should be much lower than 10 in self-driving cars. However, we find that a DNN accelerator alone may exceed the total required FIT rate of the SoC without protection (Section 8.4.2).




**(a) Fault-free execution: A truck is identified by the DNN and brakes are applied**

**(b) SDC: Truck is incorrectly identified as bird and brakes may be not applied**

**Figure 3.2:** Example of SDC that could lead to collision in self-driving cars due to soft errors

# Chapter 4

# Fine-Grained Characterization of Faults Causing Long Latency Crashes in Programs

In this chapter, we investigate into an important but neglected problem in the design of dependable software systems, namely identifying faults that propagate for a long time before causing crashes, or long-latency crashes (LLCs). We first define the problem, then characterize the code patterns that lead to LLCs through an empirical fault injection study. Based on the observations, we propose efficient heuristics to identify these code patterns for protection in order to eliminate LLCs in programs. The identification is through static and dynamic analyses of a given program without requiring to perform extensive fault injections. Hence, the proposed technique is much faster than traditional fault injection methods. Finally, we evaluate the proposed technique and present the results.

## 4.1 Introduction

A hardware fault can have many possible effects on a program. First, it may be masked or be benign. In other words, the fault may have no effect on the program's final output. Second, it may cause a crash (i.e., hardware exception) or hang (i.e., program time out). Finally, it may cause Silent Data Corruptions (SDCs), or the

program producing incorrect outputs. Of the above outcomes, SDCs are considered the most severe, as there is no visible indication that the application has done something wrong. Therefore, a number of prior studies have focused on detecting SDC-causing program errors, by selectively identifying and protecting elements of program state that are likely to cause SDCs [52, 64, 97, 141].

Compared to SDCs, crashes have received relatively less attention from the perspective of error detection. This is because crashes are considered to be the detection themselves, as the program can be recovered from a checkpoint (if one exists) or restarted after a crash. However, all of these studies make an important assumption, namely that the crash occurs soon after the fault is manifested in the program. This is important to ensure that the program is prevented from writing corrupted state to the file system (e.g., checkpoint), or sending wrong messages to other processes [15]. While this assumption is true for a large number of faults, studies have shown that a small but non-negligible fraction of faults persist for a long time in the program before causing a crash, and that these faults can cause significant reliability problems such as extended downtimes [58, 146, 150]. We call these long-latency crashes (LLCs). Therefore, there is a compelling need to develop techniques for protecting programs from LLC causing faults.

Prior work has experimentally assessed LLCs through fault injection experiments [58]. However, they do not provide much insight into why some faults cause LLCs. This is important because (1) fault injection experiments require a lot of computation time, especially to identify relatively rare events such as LLCs, and (2) fault injection cannot guarantee completeness in identifying all or even most LLC causing locations. The latter is important in order to ensure that crash latency is bounded in the program by protecting LLC causing program locations. Yim et al. [150] analyze error propagation latency in the program, and develop a coarse-grained categorization of program locations based on whether a fault in the location can cause LLCs. The categorization is based on where the program data resides, such as text segment, stack segment or heap segment. While this is useful, it does not help programmers decide which parts of the program need to be protected, as protecting all parts of the program that manipulate the heap data or stack data can lead to prohibitive performance overheads.

In contrast to the above work, we present a technique to perform fine grained

classification of program's data at the level of individual variables and program statements, based on whether a fault in the data item causes an LLC. The main insight underlying our work is that very few program locations are responsible for LLCs, and that these locations conform to a few dominant code patterns. Our technique performs static analysis of the program to identify the LLC causing code patterns. However, not every instance of the LLC-causing code pattern leads to an LLC. Our technique further uses dynamic analysis coupled with a very selective fault injection experiment, to filter the false positives and isolate the few instances of the patterns that lead to LLCs. We have implemented our technique in a completely automated tool called CRASHFINDER, which is integrated with the LLVM compiler infrastructure [85]. *To the best of our knowledge, we are the first to propose an automated and efficient method to systematically identify LLC causing program locations for protection in a fine-grained fashion*.

We make the following contributions in this work.

- Identify the dominant code patterns that can cause LLCs in programs through a large-scale fault injection experiment we conducted on a total of ten benchmark applications,
- Develop an automated static analysis technique to identify the LLC-causing code patterns in programs, based on the fault injection study,
- Propose a dynamic analysis and selective fault injection-based approach to filter out the false-positives identified by the static analysis technique, and identify LLCs.
- Implement the static and dynamic analysis techniques in an automated tool. We call this CRASHFINDER.
- Evaluate CRASHFINDER on benchmark applications from the SPEC [68], PARBOIL [133], PARSEC [18] and SPLASH-2 [148] benchmark suites. We find that CRASHFINDER can accurately identify over 90% of the LLC causing locations in the program, with no false-positives, and is about nine orders of magnitude faster than performing exhaustive fault injections to identify all LLCs in a program.

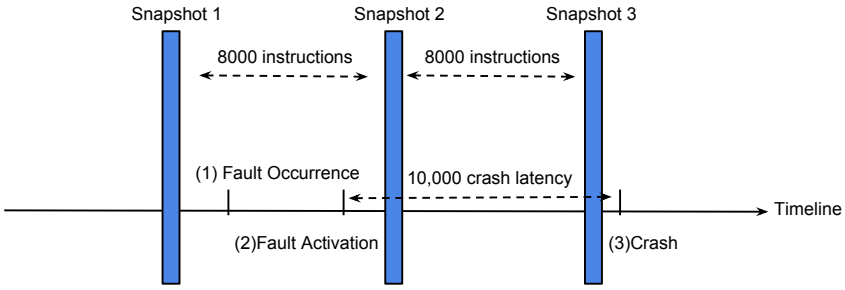



**Figure 4.1:** Long Latency Crash and Checkpointing

## 4.2 Why bound the crash latency?

We now explain our rationale for studying LLCs and why it is important to bound the crash latency in programs. We note that similar observations have been made in prior work [58, 150], and that studies have shown that having unbounded crash latency can result in severe failures. We consider one example.

Assume that the program is being checkpointed every 8,000 instructions so that it can be recovered in the case of a failure (we set aside the practicality of performing such fine grained checkpointing for now). We assume that the checkpoints are gathered in an application independent manner, i.e., the entire state of the program is captured in the checkpoint. If the program encounters an LLC of more than 10,000 instructions, it is highly likely that one or more checkpoints will be corrupted (by the fault causing the LLC). This situation is shown in Figure 4.1. However, if the crash latency is bounded to 1,000 instructions (say), then it is highly unlikely for the fault to corrupt more than one checkpoint. Note that the latency between the fault activation and the fault occurrence does not matter in this case, as the checkpoint is corrupted only when the fault actually gets activated. Therefore, we focus on the crash latency in this chapter, i.e., the number of dynamic instructions from the fault activation to the crash.

Identifying program locations that are prone to LLC is critical to improve system reliability so that one can bound crash latency by selectively protecting LLC-prone locations with minimal performance overheads. For example, one can duplicate the backward slices of the LLC-prone locations, or use low-cost detectors

for these locations like what prior work has done [123]. In this chapter, we focus on identifying such LLC-causing program locations, and defer the problem of protecting the locations to future work.

## 4.3 Initial Fault Injection Study

In this section, we perform an initial fault injection study for characterizing the LLC causing locations in a program. The goal of this study is to measure the frequency of LLCs, and understand the reasons for them in terms of the program's code. In turn, this will allow us to formulate heuristics for identifying the LLC-causing code patterns in Section 4.4. We first explain our experimental setup for this study, and then discuss the results.

### 4.3.1 Fault Injection Experiment

To perform the fault injection study, we use LLFI [147], an open-source fault injector that operates at the LLVM compiler's IR level. We inject faults into the destination registers of LLVM IR instructions, as per our fault model in Section 3. We first profile each program to get the total number of dynamic instructions. We then inject a single bit flip in the destination register of a single dynamic instruction chosen at random from the set of all dynamic instructions executed by the program. Recent study [31] has shown that the fault injection method is representative. Our benchmarks are chosen from the SPEC [68], PARBOIL [133], PARSEC [18] and SPLASH-2 suites [148]. We choose ten programs at random from these suites, and inject a total of $1,000$ faults in each, for a total of $10,000$ fault injection experiments. The details of the benchmarks are explained in Section 4.6.1.

Note that our way of injecting faults using LLFI ensures that the fault is activated right away as it directly corrupts the program's state during the injection. Therefore, we do not measure activation as the set of activated faults is the same as the set of injected faults. We categorize the results into Crashes, SDCs, Hangs and Benign faults in our experiment. Because our focus in this chapter is on LLCs, we record the crash latency for crash-causing faults in terms of the number of dynamic LLVM IR instructions between the fault injection and the crash. However, when the program crashes, its state will be lost, and hence we periodically write

to permanent storage the number of dynamic instructions executed by the program after the fault injection. The counting of the dynamic instructions is done using the tracing feature of LLFI, which we have enabled in our experiments.

### 4.3.2 Fault Injection Results

We classify the results of the fault injection experiments into SDC, crash and benign. Hangs were negligible in our experiment and are not reported. Figure 4.2 shows the aggregated fault injection results across the benchmarks. We find that on average, crashes constitute about 35% of the faults, SDC constitute 4.2%, and the remaining are benign faults (about 60%). We focus on crashes in the rest of this chapter, as our focus is on LLCs.

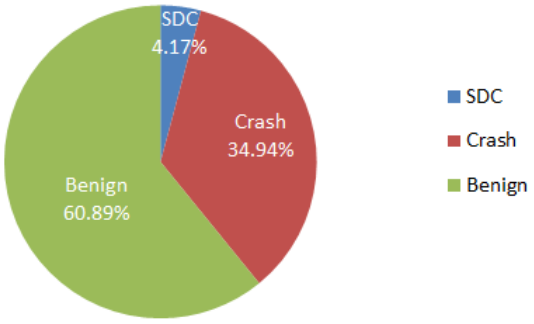



**Figure 4.2:** Aggregate Fault Injection Results across Benchmarks

Figure 4.3 shows the distribution of crash latencies for all the faults that led to crashes in the injections. On average, the percentage of LLCs is about 0.38% across the ten benchmarks. Recall that we set 1,000 dynamic instructions as the threshold for determining whether a crash is an LLC. Therefore, LLCs constitute a relatively small fraction of the total crashes in programs. This is why it is important to devise fine-grained techniques to identify them, as even a relatively large fault injection experiment such as ours exposes very few LLCs in the program (38 in absolute numbers). The percentages of LLCs among all the crash causing faults, vary from 0% to 3.6% across programs due to benchmark specific characteristics. The reasons for these variations are discussed in Section 4.7.

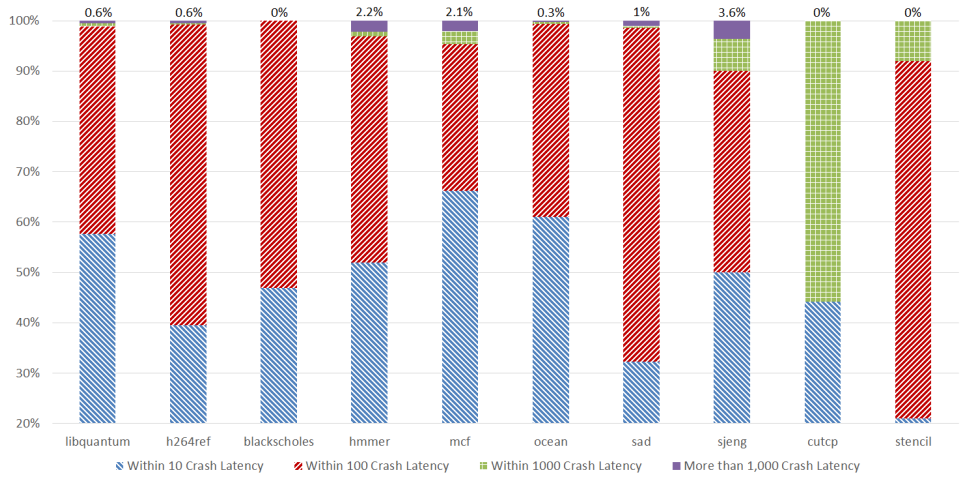



**Figure 4.3:** Latency Distribution of Crash-Causing Errors in Programs: The purple bars represent the LLCs as they have a crash latency of more than 1000 instructions. The number shown at the top of each bar shows the percentage of crashes that resulted in LLCs. The error bars for LLCs range from 0%(cutcp) to 1.85%(sjeng).

We also categorized the LLCs based on the code patterns in which the LLC locations occurred. In other words, we study the kinds of program constructs which when fault injected, are likely to cause LLCs. We choose the largest five applications from the ten benchmarks for studying the code characteristics since the larger the programs, the more code patterns they may reveal. Thus we choose sjeng, hmmer, href, libquantum and mcf for our detailed investigation.

Figure 4.4 shows the distribution of the LLC-causing code patterns we found in our experiments. The patterns themselves are explained in Section 4.3.3. *We find that about 95% of the LLC causing code falls into three dominant patterns, namely (1) Pointer Corruption (20%), (2) Loop Corruption (56%), and (3) State Corruption (19%).* Therefore we focus on these three patterns in the rest of this chapter.

### 4.3.3 Code Patterns that Lead to LLCs

As mentioned in the previous section, we find that LLCs fall into three dominant patterns namely, *pointer related LLC*, *loop related LLC* and *state related LLC*. We

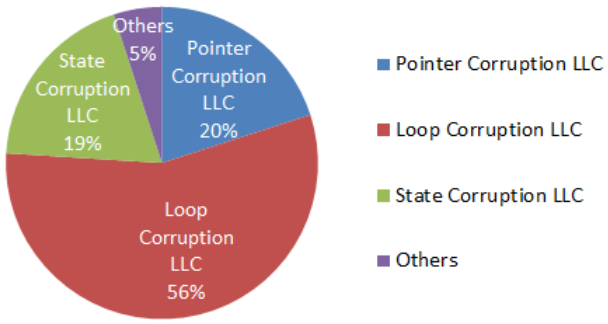



**Figure 4.4:** Distribution of LLC Categories across 5 Benchmarks (sjeng, libquantum, hmmer, h264ref and mcf). Three dominant categories account for 95% of the LLCs.

explain each category with code examples in the following subsections. Although these observations were made at the LLVM IR level, we use C code for clarity to explain them.

**Pointer Corruption LLC** occurs when a fault is injected into pointers that are written to memory. An erroneous pointer value is stored in the memory, and this value can be used as a memory operation later on to cause crash. Because the pointer may not be read for a long time, this pattern has the potential to cause an LLC. Figure 4.5A shows the case we observed in sjeng from our fault injection experiment. In the function *reloadMT*, **p0* and *next* are assigned to a global static variable, *state*, at line 7 and line 8 respectively. The fault is injected on the pointer, **p0*, at line 10. As a result, an erroneous pointer value is saved in the memory and it is used as a memory operation in the function *randomMT* at line 18 after a long time. This leads to an LLC.

**Loop Corruption LLC** When faults are injected into loop conditions or array indices inside the loop, the array manipulated by the loop (if any) may aggressively corrupt the stack, and cause LLC. We categorize this as *Loop Corruption LLC*. There are two cases in which this LLC can occur.

The first case is when a fault occurs in the array index of an array written within the loop. This fault can corrupt a large area of stack since an erroneous array index

```
1  static unsigned int state[N+1];
2  static unsigned int *next;
3  ...
4  unsigned int reloadMT(void)
5  {
6    ...
7    register unsigned int *p0 = state;
8    next = state+1;
9    ...
10   *p0++ = *pM++ ^ ... ;
11   ...
12 }
13 ...
14 unsigned int randomMI(void)
15 {
16   unsigned int y;
17   ...
18   y  = *next++;
19   ...
20 }
21 ...
```

(A) Pointer corruption (sjeng)

```
1  ...
2  arc_t *primal_bea_mpp(){
3    ...
4    for(...){
5      next++;
6      perm[next]->a = arc;
7    }
8    ...
9  }
10 ...
```

(B) Loop corruption on array index (mcf)

```
1  ...
2  void reset_piece_square (void) {
3    piece_count = 32;
4    ...
5    for (i = 1; i <= piece_count; i++){
6      ...
7      promoted_board[pieces[i]] = 1;
8      ...
9    }
10 }
11 ...
```

(C) Loop corruption on termination condition (sjeng)

```
1  int opstatus = 0;
2  unsigned char *objcode = 0;
3  ...
4  void quantum_objcode_stop()
5  {
6    ...
7    opstatus = 0;
8    free(objcode);
9    ...
10 }
11 ...
12 void quantum_objcode_start()
13 {
14   ...
15   opstatus = 1;
16   objcode = malloc(OBJCODE_PAGE * sizeof(char));
17   ...
18 }
19 ...
20 int quantum_objcode_put(unsigned char operation,
       ...)
21 {
22   ...
23   if(!opstatus)
24     return 0;
25   ...
26   for(i=0; i<size; i++)
27   {
28     objcode[position] = buf[i];
29     position++;
30   }
31 }
32 ...
```

(D) State corruption (libquantum)

**Figure 4.5:** Code examples showing the three kinds of LLCs that occurred in our experiments.

is used for array address offset calculations in every iteration of the loop. This large-scale corruption to the stack significantly increases the chance of corrupting address values (i.e., pointers, return address etc) on the stack, which in turn can result in a crash much later. For example, in Figure 4.5B, when a fault is injected into *next* making a corrupted value saved back to it at the line 5, the struct array *perm[]* at line 9 corrupts values on the stack. When the corrupted value is used for memory operations later in the program, an LLC is observed.

The second case occurs when faults are injected into termination conditions of the loop, causing a stack overflow to occur. This is shown in Figure 4.5C. Assume that a fault is injected into *piece_count* at line 3, and makes it a large value. This will cause the *for* loop at line 5 to execute for a much larger number of iterations, thereby corrupting the stack and eventually leading to a LLC.

**State Corruption LLC** occurs when faults are injected into state variables or lock (synchronization) variables in state machine structures. These variables are declared as static or global variables and are used to allocate or deallocate particular pieces of memory. If these states are corrupted, crashes may happen between states, thus causing LLC. For the code shown in Figure 4.5D, when we inject a fault in *opstatus* at line 7, the variable *opstatus* becomes a nonzero value (from zero) when the state goes to *quantum_objcode_stop*. Later in the function *quantum_objcode_put* when the state is updated to *quantum_objcode_stop*, the *opstatus* variable is examined to decide whether a particular memory area should be accessed (line 23). Due to the fault injected, we observed that *objcode* is accessed at line 28 while in the state *quantum_objcode_stop*. This leads to a LLC as it accesses the unallocated memory area *objcode*, which is illegal.

## 4.4 Approach

In this section, we describe our proposed technique CRASHFINDER, to find *all the LLCs* in a program. CRASHFINDER consists of three phases, as Figure 4.6 shows

In the first phase, it performs a static analysis of the program's source code to determine the potential locations that can cause LLCs. The analysis is done based on the code patterns in Section 4.3.3. We refer to this phase of CRASHFINDER as CRASHFINDER STATIC. In the second phase, it performs dynamic analysis of the program (under a given set of inputs) to determine which dynamic instances of the static locations are likely to result in LLCs. We call this phase CRASHFINDER DYNAMIC. In the last phase, it injects a selected few faults to the dynamic instances chosen by CRASHFINDER DYNAMIC. We refer to this phase of CRASHFINDER as *selective fault injection*. We describe the three phases in the three subsections.

### 4.4.1 Phase 1: Static Analysis (CRASHFINDER STATIC)

CRASHFINDER STATIC is the static analysis portion of our technique that statically searches the program's code for the three patterns corresponding to those identified in Section 4.3.3. We found that these three patterns are responsible for almost all the LLCs in the program, and hence it suffices to look for these patterns to cover

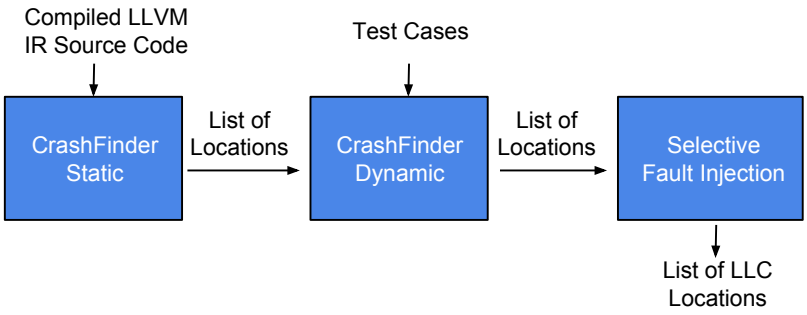



**Figure 4.6:** Workflow of CRASHFINDER

the LLCs. However, not every instance of these patterns may lead to an LLC, and hence we may get false-positives in this phase. False-positives are those locations that conform to the LLC causing patterns but do not lead to an LLC, and will be addressed in the next phase.

The algorithm of CRASHFINDER STATIC takes the program's source code compiled to the LLVM IR as an input and outputs the list of potential LLC causing locations. Specifically, CRASHFINDER STATIC looks for the following patterns in the program:

### Pointer Corruption LLC

CRASHFINDER STATIC finds pointers that are written to memory in the program. More specifically, it examines static data dependency sequences of all pointers, and only consider the ones that end with *store* instruction.

### Loop Corruption LLC

In this category, CRASHFINDER STATIC finds loop termination variables in loop headers and array index assignment operations. For loop termination variable(s), it looks for the variable(s) that is used for comparison with the loop index variable in loop headers. For array index assignment, CRASHFINDER STATIC first locates binary operations with a variable and a constant as operands, then checks if the result being stored is used as offset in array address calculation. If yes, then we can infer that the variable being updated will be used as the address offset of an array.

In LLVM, offset calculations are done through a special instruction and are hence easy to identify statically.

#### State Corruption LLC

CRASHFINDER STATIC finds static and global variables used to store state or locks. Because these may depend on the application's semantics, we devise a heuristic to find such variables. If a static variable is loaded and directly used in comparison and branches, we assume that it is likely to be a state variable or a lock variable. We find that this heuristic allow us to cover most of these cases without semantic knowledge of the application.

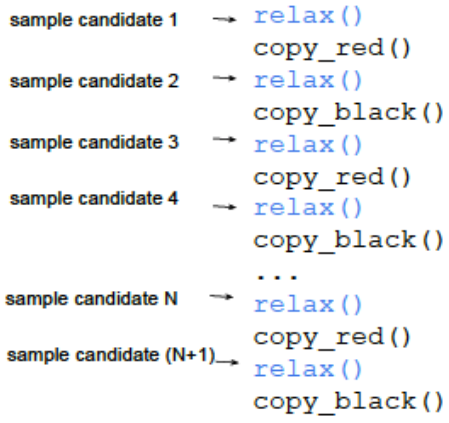
### 4.4.2 Phase 2: Dynamic Analysis (CRASHFINDER DYNAMIC)

In this phase, our technique attempts to eliminate the false positives from the static locations identified in phase 1. One straw man approach for doing so is to inject faults in every dynamic instance of the static locations to determine if it leads to an LLC. However, a single static instruction may correspond to hundreds of thousands of dynamic instances in a typical program, especially if it is within a loop. Further, each of these dynamic instances needs to be fault injected multiple times to determine if it will lead to an LLC, and hence a large number of fault injections will need to be performed. All this adds up to considerable performance overheads, and hence the above straw man approach does not scale.

We propose an alternate approach to cut down the number of fault injection locations to filter out the false positives. Our approach uses dynamic analysis to identify a few dynamic instances to consider for injection among the set of all the identified static locations. The main insight we leverage is that there are repeated control-flow sequences in which the dynamic instances occur, and it is sufficient to sample dynamic instances in each *unique* control-flow sequence to obtain a representative set of dynamic instances for fault injection. This is because the crash latency predominantly depends on the control-flow sequence executed by the program after the injection at a given program location. Therefore, it suffices to obtain one sample from each unique control flow pattern in which the dynamic instance occurs. We determine the control flow sequences at the level of function calls. That

```
1  void multig(long my_id){
2    ...
3    while ((!flag1) && (!flag2)) {
4      ...
5      relax();
6      copy_red();
7      relax();
8      copy_black();
9      ...
10     }
11 }
12
13 void relax(){
14   ...
15   for (...) {
16     ...
17     t1a = (double *) t2a[i];
18     ...
19   }
20 }
```

(a)




(b)

**Figure 4.7:** Dynamic sampling heuristic. (a) Example source code (ocean program), (b) Execution trace and sample candidates.

is we sample the dynamic instances with different function call sequences, and ignore the ones that have the same function call sequences. We show in Section 4.7 that this sampling heuristic works well in practice.

We consider an example to illustrate the sampling heuristic to determine which dynamic instances to choose. Figure 4.7(b) shows the dynamic execution trace generated by the code in Figure 4.7(a). For example, we want to sample the dynamic instances corresponding to the variable *t1a* at line 17 in Figure 4.7(a). Firstly, because *t1a* is within a loop in function *relax*, it corresponds to multiple dynamic instances in the trace. We only consider one of them as candidate for choosing samples (we call it a sample candidate), since they have same function call sequences (no function calls) in between. Secondly, function *relax* is called within a loop in function *multig* at lines 5 and 7. As can be seen in the Figure 4.7, there are two recurring function call sequences circumscribing the execution of the static location corresponding to the sample candidates, namely *relax() copy_red()* and *relax() copy_black()*. We collect one sample of each sequence regardless of how many times they occur. In this case, only sample candidate 1 and 2 are selected for later fault injections. We find that this dramatically reduces the fault injection space thereby saving considerable time.

### 4.4.3 Phase 3: Selective Fault Injections

The goal of this phase is to filter out all the false-positives identified in the previous phase through fault injections. Once we have isolated a set of dynamic instances from CRASHFINDER DYNAMIC to inject for the static location, we configure our fault injector to inject two faults into each dynamic instance, one fault at a time. We choose one high-order bit and and one low-order bit at random to inject into, as we found experimentally that LLCs predominantly occur either in the high-order bits or the low-order bits, and hence one needs to sample both.

We then classify the location as an LLC location (i.e., not a false positive) if any one of the injected faults results in an LLC. Otherwise, we consider it a false-positive, and remove it from the list of LLC locations. Note that this approach is conservative as performing more injections can potentially increase the likelihood of finding an LLC, and hence it is possible that we miss some LLCs. However, as

we show in Section 4.7, our approach finds most LLCs even with only two fault injections per each dynamic instance. We also show that increasing the number of fault injections beyond 2 for each dynamic instance does not yield substantial benefits, and hence we stick to 2 injections per instance.

## 4.5 Implementation

We implemented CRASHFINDER STATIC as a pass in the LLVM compiler [85] to analyze the IR code and extract the code patterns. We implemented the CRASHFINDER DYNAMIC also as an LLVM pass that instruments the program to obtain its control-flow. CRASHFINDER DYNAMIC then analyzes the control-flow patterns and determines what instances to choose for selective fault injection. We use the LLFI fault injection framework [147] to perform the fault injections. Finally, we used our crash latency measurement library to determine the crash latencies after injection.

To use CRASHFINDER [1], all the user needs to do is to compile the application code with the LLVM compiler using our module. No annotations are needed. The user also needs to provide us with representative inputs so that CRASHFINDER can execute the application, collect the control-flow patterns and choose the dynamic instances to inject faults.

## 4.6 Experimental Setup

We empirically evaluate CRASHFINDER in terms of accuracy and performance. We use a fault injection experiment to measure the accuracy, and use execution time of the technique to measure its performance. We evaluate both CRASHFINDER and CRASHFINDER STATIC separately to understand the effect of different parts of the technique (CRASHFINDER includes CRASHFINDER STATIC, CRASHFINDER DYNAMIC and the selective fault injection). We compare both the accuracy and the performance of both techniques to those of exhaustive fault injections that are needed to find all the LLCs in a program [2]. Our experiments are all carried out on an Intel Xeon E5 machine, with 32 GB RAM running Ubuntu Linux 12.04.

[1] CRASHFINDER and its source code can be freely downloaded from https://github.com/DependableSystemsLab/Crashfinder

[2] Our goal is to find *all* LLC causing locations in the program so that we can selectively protect them and bound the crash latency.

We first present the benchmarks used (Section 4.6.1, followed by the research questions (Section 4.6.2). We then present an overview of the methodology we used to answer each of the research questions (Section 4.6.3).

### 4.6.1 Benchmarks

We choose a total of ten benchmarks from various domains for evaluating CRASHFINDER. The benchmark applications are from SPEC [68], PARBOIL [133], PARSEC [18] and SPLASH-2 [148]. All the benchmark application are compiled and linked into native executables using LLVM, with standard optimizations enabled. We show the detailed information of the benchmarks in Table 7.1.

**Table 4.1:** Characteristics of Benchmark Programs

| **Benchmark** | **Benchmark Suite** | **Description** |
|---|---|---|
| libquantum | SPEC | A library for the simulation of a quantum computer |
| h264ref | SPEC | A reference implementation of H.264/AVC (Advanced Video Coding) |
| blackscholes | PARSEC | Option pricing with Black-Scholes Partial Differential Equation (PDE) |
| hmmer | SPEC | Uses statistical description of a sequence family's consensus to do sensitive database searching |
| mcf | SPEC | Solves single-depot vehicle scheduling problems planning transportation |
| ocean | SPLASH-2 | Large-scale ocean movements simulation based on eddy and boundary currents |
| sad | PARBOIL | Sum of absolute differences kernel, used in MPEG video encoders |
| sjeng | SPEC | A program that plays chess and several chess variants |
| cutcp | PARBOIL | Computes the short-range component of Coulombic potential at each grid point |
| stencil | PARBOIL | An iterative Jacobi stencil operation on a regular 3-D grid |

### 4.6.2 Research Questions

We answer the following research questions(RQs) in our experiments.

**RQ1:** *How much speedup do* CRASHFINDER STATIC *and* CRASHFINDER *achieve over exhaustive injection ?*

**RQ2:** *What is the precision of* CRASHFINDER STATIC *and* CRASHFINDER*?*

**RQ3:** *What is the recall of* CRASHFINDER STATIC *and* CRASHFINDER*?*

**RQ4:** *How well do the sampling heuristics used in* CRASHFINDER *work in practice ?*

### 4.6.3 Experimental Methodology

We describe our methodology for answering each of the RQs below. We perform fault injections using the LLFI fault injector [147] as described earlier.

#### Performance

In order to answer RQ1, we measure the total time taken for executing CRASHFINDER STATIC, CRASHFINDER and the exhaustive fault injections. More specifically, for each benchmark, we measure the total time used for (1) CRASHFINDER STATIC, (2) CRASHFINDER, which includes CRASHFINDER STATIC, CRASHFINDER DYNAMIC and selective fault injections to identify LLCs and, (3) exhaustive fault injections to find LLCs.

#### Precision

The precision is an indication of the false-positives produced by CRASHFINDER STATIC and CRASHFINDER. To measure the precision, we inject 200 faults randomly at each static program location identified by CRASHFINDER STATIC or CRASHFINDER, and measure the latency. If none of the injections at the location result in an LLC, we declare it to be a false positive. Note that we choose 200 fault injections per location to balance time and comprehensiveness. If we increase the number of faults, we may find more LLC causing locations, thus decreasing the false positives. Thus, this method gives us a conservative upper bound on the false-positives of the technique.

#### Recall

The recall is an indication of the false-negatives produced by CRASHFINDER STATIC and CRASHFINDER. To measure the recall of CRASHFINDER STATIC and CRASHFINDER, we randomly inject 3,000 faults for each benchmark and calculate the fraction of the observed LLCs that were covered by CRASHFINDER STATIC and CRASHFINDER

respectively. Thus 30,000 faults in total are injected over ten benchmark applications for this experiment. Note that this is in addition to the 1,000 fault injection experiments performed in the initial study, which were used to develop the two techniques. We do not include the initial injections in the recall measurement to avoid biasing the results.

#### Heuristics for Sampling

As mentioned in Section 4.4, there are two heuristics used by CRASHFINDER to reduce the space of fault injections it has to perform. The first is to limit the chosen instances to unique dynamic instances of control-flow patterns in which the static instructions appear, and the second is to limit the number of faults injected in the dynamic instances to two faults per instance. These heuristics may lead to loss in coverage. We investigate the efficacy of these heuristics by varying the parameters used in them and measure the resulting recall.

## 4.7 Results

This section presents the results of our experiments for evaluating CRASHFINDER STATIC and CRASHFINDER. Each subsection corresponds to a research question (RQ).

### 4.7.1 Performance (RQ1)

We first present the results of running CRASHFINDER and CRASHFINDER STATIC in terms of the number of instructions in each benchmark, and then examine how much speedup can CRASHFINDER achieve over exhaustive fault injections.

Table 4.2 shows the numbers of instructions for each benchmark. In the table, columns *Total S.I* and *Total D.I* show the total numbers of static instructions and dynamic instructions of each benchmark. Columns CRASHFINDER STATIC *S.I* and CRASHFINDER STATIC *D.I* indicate the numbers of static instructions and dynamic instructions corresponding to the static instructions that were found by CRASHFINDER STATIC as LLC causing locations. Columns CRASHFINDER *S.I* and CRASHFINDER *D.I* show the numbers of static instructions and dynamic instances of the static locations that CRASHFINDER identified as LLC causing lo-

**Table 4.2:** Comparison of Instructions Given by CRASHFINDER and CRASHFINDER STATIC

| | Total S.I. | Total D.I. (in million) | CF Static S.I (%) | CF Static D.I (%) | CF S.I (%) | CF D.I (%) |
|---|---|---|---|---|---|---|
| libquantum | 15319 | 870 | 1.85% | 9.27% | 0.18% | 0.011% |
| h264ref | 189157 | 116 | 0.85% | 3.92% | 0.14% | 0.150% |
| blackscholes | 758 | 0.13 | 3.29% | 1.81% | 0.66% | 0.004% |
| hmmer | 92287 | 4774 | 0.51% | 3.53% | 0.13% | 0.437% |
| mcf | 4086 | 6737 | 6.29% | 8.75% | 2.62% | 1.383% |
| ocean | 21300 | 1061 | 3.46% | 3.11% | 0.53% | 0.003% |
| sad | 3176 | 1982 | 4.47% | 5.56% | 0.69% | 0.473% |
| sjeng | 33931 | 137 | 1.70% | 15.55% | 0.16% | 0.567% |
| cutcp | 3868 | 11389 | 3.13% | 6.35% | 0.39% | 0.001% |
| stencil | 2193 | 7168 | 4.38% | 0.84% | 0.41% | 0.819% |
| **Average** | **36608** | **3423** | **2.99%** | **5.87%** | **0.89%** | **0.385%** |

cations. As can be seen from the table, on average, CRASHFINDER STATIC identified 2.99% of static instructions as LLC causing, which corresponds to about 5.87% of dynamic instructions. In comparison, CRASHFINDER further winnowed the number of static LLC-causing locations to 0.89%, and the number of dynamic instructions to just 0.385%, thereby achieving a significant reduction in the dynamic instructions. The implications of this reduction are further investigated in Section 4.8.

Figure 4.8 shows the orders of magnitude of time reduction achieved by using CRASHFINDER STATIC and CRASHFINDER to find LLCs, compared to exhaustive fault injections, for each benchmark. In the figure, CRASHFINDER STATIC refers to the time taken to run CRASHFINDER STATIC. CRASHFINDER refers to the time taken to run all three components of CRASHFINDER, namely CRASHFINDER STATIC, CRASHFINDER DYNAMIC and the selective fault injection phase. Note that the exhaustive fault injection times are an estimate based on the number of dynamic instructions that need to be injected, and the time taken to perform a single injection. We emphasize that the numbers shown represent the orders of magnitude in terms of speedup. For example, a value of 12 in the graph, means

that the corresponding technique was $10^{12}$ times faster than performing exhaustive fault injections. In summary, on average CRASHFINDER STATIC achieves a total of 13.47 orders of magnitude of time reduction whereas CRASHFINDER achieves 9.29 orders of magnitude of time reduction over exhaustive fault injecton to find LLCs.

We also measured the wall clock time of the different phases of CRASHFINDER. The geometric means of time taken for CRASHFINDER STATIC are 23 seconds, for CRASHFINDER DYNAMIC the time taken is 3.1 hours, while it takes about 3.9 days for the selective fault injection phase. Overall, it takes about 4 days on average for CRASHFINDER to complete the entire process. While this may seem large, note that both the CRASHFINDER DYNAMIC and selective fault injection phases can be parallelized to reduce the time. We did not however do this in our experiments.

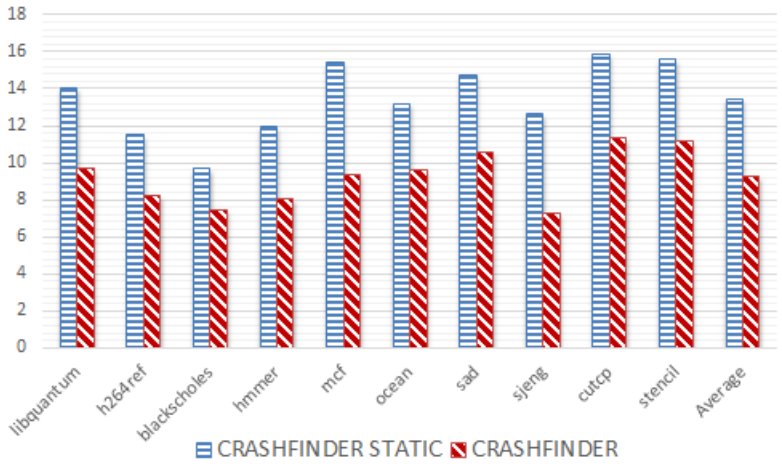



**Figure 4.8:** Orders of Magnitude of Time Reduction by CRASHFINDER STATIC and CRASHFINDER compared to exhaustive fault injections

### 4.7.2 Precision (RQ2)

Figure 4.9 shows the precision of CRASHFINDER STATIC and CRASHFINDER for each benchmark. The average precision of CRASHFINDER STATIC and CRASHFINDER are 25.42% and 100% respectively. The reason CRASHFINDER has a precision of 100% is that all the false-positives produced by the static analysis phase (CRASHFINDER

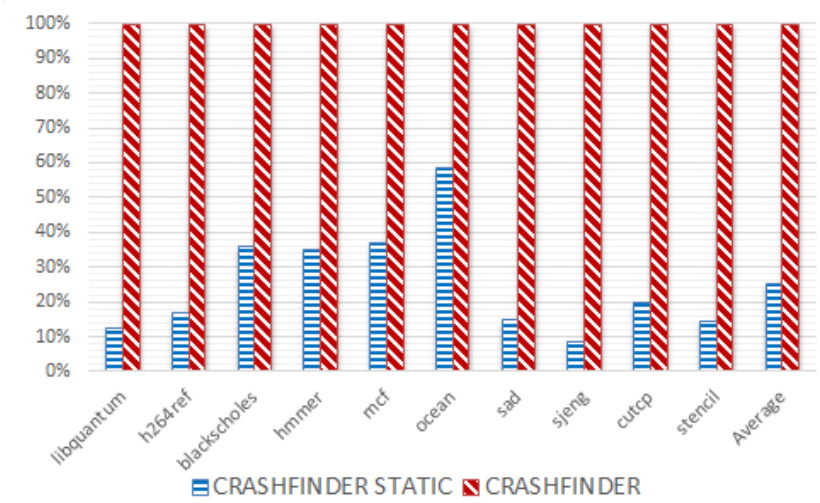



**Figure 4.9:** Precision of CRASHFINDER STATIC and CRASHFINDER for finding LLCs in the program

STATIC) are filtered out by the latter two phases, namely CRASHFINDER DYNAMIC, and selective fault injections. The main reason why CRASHFINDER STATIC has low precision is because it cannot statically determine the exact runtime behavior of variables. For example, a pointer can be saved and loaded to/from memory in very short intervals, and would not result in an LLC. This behavior is determined by its runtime control flow, and cannot be determined at compile time, thus resulting in false positives by CRASHFINDER STATIC. However, CRASHFINDER does not have this problem as it uses dynamic analysis and selective fault injection.

### 4.7.3 Recall (RQ3)

Figure 4.10 shows the recall of CRASHFINDER STATIC and CRASHFINDER. The average recall of CRASHFINDER STATIC and CRASHFINDER are 92.47% and 90.14% respectively. Based on the results, we can conclude that (1) CRASHFINDER STATIC is able to find most of the LLC causing locations showing that the code patterns we identified are comprehensive and, (2) our heuristics used in CRASHFINDER DYNAMIC and selective fault injections do not filter out many legitimate LLC locations since there is only a 2.33% difference between the recalls of CRASHFINDER

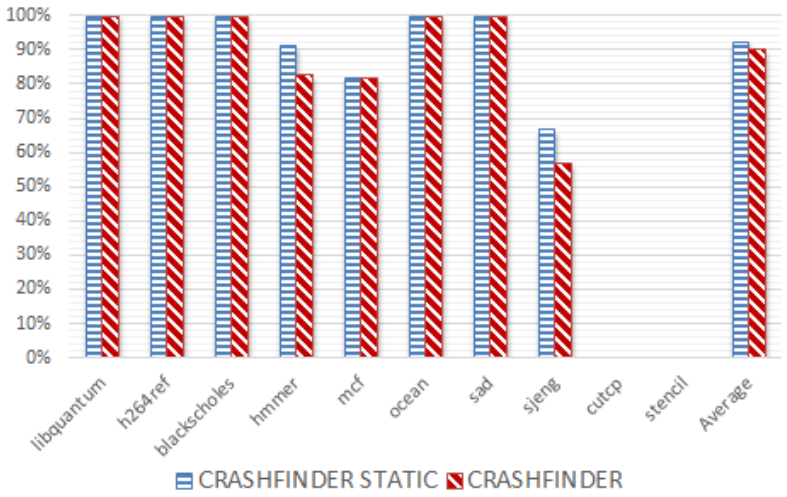



**Figure 4.10:** Recall of CRASHFINDER STATIC and CRASHFINDER

STATIC and CRASHFINDER (however, they filter out most of the false positives as evidenced by the high precision of CRASHFINDER compared to CRASHFINDER STATIC). We will discuss this further in the next subsection.

There are two reasons why CRASHFINDER STATIC does not achieve 100% recall: (1)There are a few cases as mentioned in Section 4.3 that do not fall into the three dominant patterns. (2) While CRASHFINDER STATIC is able to find most of the common cases of LLCs, it does not find some cases where the dependency chain spans multiple function calls. For example, the return value of an array index calculation can be propagated through complex function calls and finally used in the address offset operations in a loop. This makes the pointer analysis in LLVM return too many candidates for the pointer target, and so we truncate the dependence chain. However, there is no fundamental reason why we cannot handle these cases. Even without handling the cases, CRASHFINDER finds 92.47% of the cases leading to LLCs in the program.

Note that we did not observe any LLCs in the two benchmark programs *stencil* and *cutcp*. This may be because they have fewer LLC causing locations, and/or they have a small range of bits which may result in LLCs. This was also the case in the initial study 4.3.

### 4.7.4 Efficacy of Heuristics (RQ4)

As mentioned earlier, there are two heuristics used by CRASHFINDER DYNAMIC to speed up the injections. First, in the dynamic analysis phase (CRASHFINDER DYNAMIC), only a few instruction instances are chosen for injection. Second, in the selective fault injection phase, only a few bits in each of the chosen locations are injected. We examine the effectiveness of these heuristics in practice.

In order to understand the LLC-causing errors that are covered by CRASHFINDER STATIC but not CRASHFINDER, we manually inspected these injections. We found that all of the missed errors are due to the second heuristic used by the selective fault injection phase. None of the missed errors were due to the first heuristic employed by CRASHFINDER DYNAMIC.

The heuristic for choosing bit positions for selective injections picks two random positions in the word to inject faults into, one from the high-level bits and one from the low-level bits. Unfortunately, this may miss other positions that lead to LLCs. We evaluated the effect of increasing the number of sampled bits to 3 and 5, but even this did not considerably increase the number of LLCs found by CRASHFINDER. This is because most of the missed errors can only be reproduced by injecting into very specific bit positions, and finding these positions will require near exhaustive injections on the words found by CRASHFINDER DYNAMIC, which will prohibitively increase the time taken to complete the selective fault injection phase. Therefore, we choose to retain the heuristic as it is, especially because the difference between CRASHFINDER STATIC and CRASHFINDER is only 2.33%.

With the above being said, the heuristic-based approach used here is an approximation. Hence, there may be multiple sources of inaccuracy in these heuristics. We will further quantify the limits of the heuristic based approach in future work.

## 4.8 Discussion

In this section, we discuss some of the implications of CRASHFINDER on selective protection and checkpointing. We also discuss some of the limitations of CRASHFINDER and improvements.

### 4.8.1 Implication for Selective Protection

One of the main results from evaluating CRASHFINDER is that we find that a very small number of instructions are responsible for most of the LLCs in the program. As per Table 4.2, only 0.89% of static instructions are responsible for more than 90% of the LLC causing errors in the program (based on the recall of CRASHFINDER). Further, CRASHFINDER is able to precisely pinpoint these instructions, thereby allowing these to be selectively protected.

An example of a selective protection technique is value range checking in software [64]. A range check is typically inserted after the instruction that produces the data item to be checked. For example, the *assertion(ptr_address<0x001b, true)* inserted after the static instruction producing *ptr_address* will check the value of the variable whenever the instruction is executed. Since the total number of executions of all such LLC causing instructions is only 0.385% (Table 4.2), the overhead of these checks is likely to be extremely low. We will explore this direction in future work.

### 4.8.2 Implication for Checkpointing Techniques

Our study also establishes the feasibility of fine-grained checkpointing techniques for programs, as such checkpointing techniques would incur frequent state corruptions in the presence of LLCs. For example, Chandra et al. [30] found that the frequency of checkpoint corruption when using a fine-grained checkpointing technique ranges between 25 and 40% due to LLCs. They therefore conclude that one should not use such fine-grained checkpointing techniques, and instead use application-specific coarse-grained checkpointing in which the corresponding probability of checkpoint corruption is 1% to 19%. However, by deploying our technique and selectively protecting the LLC causing locations in the program, one could restrict the crash latency, thus minimizing the chances of checkpoint corruption. Based on the 90% recall of CRASHFINDER, we can achieve a 10-fold reduction in the number of LLC causing locations, thus bringing the checkpoint corruption probability of fine-grained checkpointing down. This would make fine-grained checkpointing feasible, thus allowing faster recovery from errors. This is also a direction we plan to explore in the future.

### 4.8.3 Limitations and Improvements

One of the main limitations of CRASHFINDER is that it takes a long time (on average 4 days) to find the LLC causing errors in the program. The bulk of this time is taken by the selective fault injection phase, which has to inject faults into thousands of dynamic instances found by CRASHFINDER DYNAMIC to determine if they are LLCs. While this is still orders of magnitude faster than performing exhaustive fault injections, it is still a relatively high one-time cost to protect the program. One way to speed this up would be to parallelize it, but that comes at the cost of increased computation resources.

An alternate way to speed up the technique is to improve the precision of CRASHFINDER STATIC. As it stands, CRASHFINDER STATIC takes only a few seconds to analyze even large programs and find LLC causing locations in them. The main problem however is that CRASHFINDER STATIC has a very low precision (of 25.4%). However, this may be acceptable in some cases, where we can protect a few more locations and incur higher overheads in doing so. Even with this overprotection, we still only protect less than 6% of the program's dynamic instructions (Table 4.2). However, one can improve the precision further by finding all possible aliases and control flow paths at compile time [117], and filtering out the patterns that are unlikely to cause LLCs.

Another limitation is that the recall of CRASHFINDER is only about 90%. Although this is still a significant recall, one can improve it further by (1) building a more comprehensive static analyzer to cover the uncovered cases that do not belong to the dominant LLC-causing patterns, and (2) improving the heuristic used in the selective fault injection phase, by increasing the number of fault injections in the selective fault injection phase, albeit at the cost of increased performance overheads (as we found in RQ4, this heuristic was responsible for most of the difference in recall between CRASHFINDER and CRASHFINDER STATIC).

Finally, though the benchmark applications are chosen from a variety of domains such as scientific computing, multimedia, statistics and games, there are other domains that are not covered such as database programs, or system software applications. Further, they are all single-node applications. We defer the extension of CRASHFINDER for distributed applications to our future work.

## 4.9 Summary

In this chapter, we identify an important but neglected problem in the design of dependable software systems, namely identifying faults that propagate for a long time before causing crashes, or LLCs. Unlike prior work which has only performed a coarse grained analysis of such faults, we perform a fine grained characterization of LLCs. Interestingly, we find that there are only three code patterns in the program that are responsible for almost all LLCs, and that these patterns can be identified efficiently through static analysis. We build a static analysis technique to find these patterns, and augment it with a dynamic analysis and selective fault-injection based technique to filter out the false positives. We implement our technique in a completely automated tool called CRASHFINDER. We find that CRASHFINDER is able to achieve 9 orders of magnitude speedup over exhaustive fault injections to identify LLCs, has no false-positives, and successfully identifies over 90% of the LLC causing locations in ten benchmark programs.

# Chapter 5

# Modeling Soft-Error Propagation in Programs

This chapter describes a fast and accurate modeling technique that quantitively estimates the Silent Data Corruptions (SDCs) probabilities of a given program and its individual instructions without any fault injections. We name our model TRIDENT. Different from the heuristic-based technique we have discussed in Chapter 4, the technique proposed in this chapter is an analytical model which systematically tracks error propagation in the entire propagation space of program executions. We first describe the challenges in identifying SDCs in programs before presenting the details of the model. We then design experiments to evaluate the accuracy and performance of the model. Finally, we discuss an use-case where developers can use the model to guide the selective protection in programs.

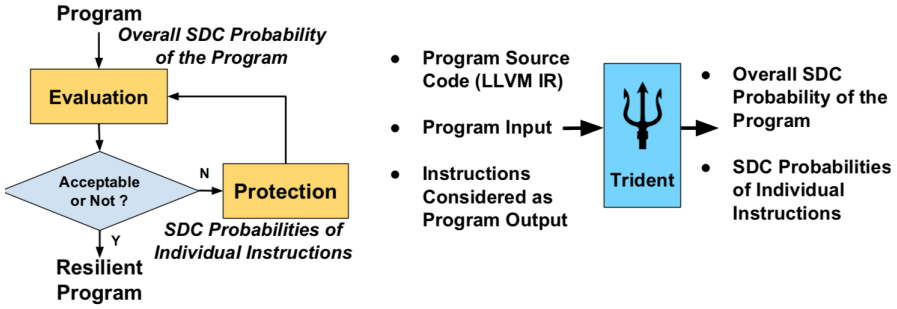
## 5.1 Introduction

One consequence of such hardware errors is incorrect program output, or silent data corruptions (SDCs), which are very difficult to detect and can hence have severe consequences [129]. Studies have shown that a small fraction of the program states are responsible for almost all the error propagations resulting in SDCs, and so one can selectively protect these states to meet the target SDC probability while incurring lower energy and performance costs than full duplication techniques [52,

130]. Therefore, in the development of fault-tolerant applications (Figure 5.1A), it is important to estimate the SDC probability of a program – both in the aggregate, and on an individual instruction basis - to decide whether protection is required, and if so, to selectively protect the SDC-causing states of the program. This is the goal of our work.

Fault Injection (FI) has been commonly employed to estimate the SDC probabilities of programs. FI involves perturbing the program state to emulate the effect of a hardware fault and executing the program to completion to determine if the fault caused an SDC. However, real-world programs may consist of billions of dynamic instructions, and even a single execution of the program may take a long time. Performing thousands of FIs to get statistically meaningful results for each instruction takes too much time to be practical [65, 66]. As a result, researchers have attempted to analytically model error propagation to identify vulnerable instructions [52, 97, 130]. The main advantage of these analytical models is scalability, as the models usually do not require FIs, and they are fast to execute. However, most existing models suffer from a lack of accuracy, as they are limited to modeling faults in the normal (i.e., fault-free) control-flow path of the program. Since program execution is dynamic in nature, a fault can propagate to not only the data-dependencies of an instruction, but also to the subsequent branches (i.e., control flow) and memory locations that are dependent on it. This causes deviation from the predicted propagation, leading to inaccuracies. Unfortunately, tracking the deviation in control-flow and memory locations due to a fault often leads to state space explosion.

This chapter proposes a model, TRIDENT, for tracking error propagation in programs that addresses the above two challenges. The key insight in TRIDENT is that error propagation in dynamic execution can be decomposed into a combination of individual modules, each of which can be abstracted into probabilistic events. TRIDENT can predict both the overall SDC probability of a program and the SDC probability of individual instructions based on dynamic and static analysis of the program without performing FI. We implement TRIDENT in the LLVM compiler [85] and evaluate its accuracy and scalability vis-a-vis FI. *To the best of our knowledge, we are the first to propose a model to estimate the SDC probability of individual instructions and the entire program without performing any FIs.*




(a) Development Cycle (b) Workflow of TRIDENT

**Figure 5.1:** Development of Fault-Tolerant Applications

Our main contributions in this chapter are as follows:

- Propose TRIDENT, a three-level model for tracking error propagation in programs. The levels are static-instruction, control-flow and memory levels, and they build on each other. The three-level model abstracts the data-flow of programs in the presence of faults.
- Compare the accuracy and scalability of TRIDENT with FI, to predict the SDC probability of individual instructions and that of the entire program.
- Demonstrate the use of TRIDENT to guide selective instruction duplication for configurable protection of programs from SDCs under a performance overhead.

The results of our experimental evaluation are as follows:

- The predictions of SDC probabilities using TRIDENT are statistically indistinguishable from those obtained through FI, both for the overall program and for individual instructions. On average, the overall SDC probability predicted by TRIDENT is 14.83% while the FI measured value is 13.59% across 11 programs.
- We also create two simpler models to show the importance of modeling control-flow divergence and memory dependencies - the first model considers neither, while the second considers control-flow divergence but not

memory dependencies. The two simpler models predict the average SDC probabilities across programs as 33.85% and 23.76% respectively, which is much higher than the FI results.

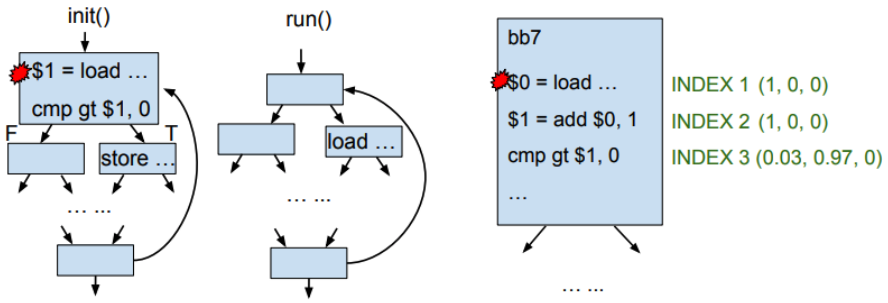
- Compared to FI, whose cost is proportional to the number of injections, TRIDENT incurs a fixed cost, and a small incremental cost for each instruction sampled in the program. For example, TRIDENT takes about 16 minutes to calculate the *individual* SDC probabilities of about 1,000 static instructions, which is significantly faster than the corresponding FI experiments (which often take hours or even days).

- Using TRIDENT to guide selective instruction duplication reduces the overall SDC probability by 65% and 90% at 11.78% and 23.31% performance overheads, respectively (these represent $1/3$rd and $2/3$rd of the full-duplication overhead for the programs respectively). These reductions are higher than the corresponding ones obtained using the simpler models.

## 5.2 The Challenge

We use the code example in Figure 5.2A to explain the main challenge of modeling error propagation in programs. The code is from *Pathfinder* [33], though we make minor modifications for clarity and remove some irrelevant parts. The figure shows the control-flow graphs (CFGs) of two functions: *init()* and *run()*. There is a loop in each function: the one in *init()* updates an array, and the one in *run()* reads the array for processing. The two functions *init()* and *run()* are called in order at runtime. In the CFGs, each box is a basic block and each arrow indicates a possible execution path. In each basic block, there is a sequence of statically data-dependent instructions, or a static data-dependent instruction sequence.

Assume that a fault occurs at the instruction writing to *$1* in the first basic block in *init()*. The fault propagates along its static data-dependent instruction sequence (from *load* to *cmp*). At the end of the sequence, if the fault propagates to the result of the comparison instruction, it will go beyond the static data dependency and cause the control-flow of the program to deviate from the fault-free execution. For example, in the fault-free execution, the *T* branch is supposed to be taken, but




(a) Example of Propagation (b) Propagation in $f_s$

**Figure 5.2:** Running Example

due to the fault, the *F* branch is taken. Consequently, the basic blocks under the *T* branch including the store instruction will not be executed, whereas subsequent basic blocks dominated by the *F* branch will be executed. This will load the wrong value in *run()*, and hence the fault will continue to propagate and it may reach the program's output resulting in an SDC.

We identify the following three challenges in modeling error propagation: (1) Statically modeling error propagation in dynamic program execution requires a model that abstracts the program data-flow in the presence of faults. (2) Due to the random nature of soft errors, a fault may be activated at any dynamic branch and cause control-flow divergence in execution from the fault-free execution. In any divergence, there are numerous possible execution paths the program may take, and tracking all of these paths is challenging. One can emulate all possible paths among the dynamic executions at every dynamic branch and figure out which fault propagates where in each case. However, this rapidly leads to state space explosion. (3) Faults may corrupt memory locations and hence continue to propagate through memory operations. Faulty memory values can be read by (multiple) load instructions at runtime and written to other memory locations as execution progresses. There are enormous numbers of store and load instructions in a typical program execution, and tracing error propagations among these memory dependencies requires constructing a huge data dependency graph, which is very expensive.

As we can see in the above example, if we do not track error propagations be-

yond the static data dependencies and instead stop at the comparison instruction, we may not identify all the cases that could lead to SDCs. Moreover, if control-flow divergence is ignored when modeling, tracking errors in memory is almost impossible, as memory corruptions often hide behind control-flow divergence, as shown in the above example. Existing modeling techniques capture neither of these important cases, and their SDC prediction accuracies suffer accordingly. In contrast, TRIDENT captures both the control-flow divergences and the memory corruptions that potentially arise as a result of the divergence.

## 5.3 TRIDENT

In this section, we first introduce the inputs and outputs of our proposed model, TRIDENT, and then present the overall structure of the model and the key insights it leverages. Finally we present the details of TRIDENT using the running example.

### 5.3.1 Inputs and Outputs

The workflow of TRIDENT is shown in Figure 5.1B. We require the user to supply three inputs: (1) The program code compiled to the LLVM IR, (2) a program input to execute the program and obtain its execution profile (similar to FI methods, we also require a single input to obtain runtime information), and (3) the output instruction(s) in the program that are used for determining if a fault resulted in an SDC. For example, the user can specify *printf* instructions that are responsible for the program's output and used to determine SDCs. On the other hand, *printf*s that log debugging information or statistics about the program execution can be excluded as they do not typically determine SDCs. Without this information, all the output instructions are assumed to determine SDCs by default.

TRIDENT consists of two phases: (1) Profiling and (2) inferencing. In the profiling phase, TRIDENT executes the program, performing dynamic analysis of the program to gather information such as the count and data dependency of instructions. After collecting all the information, TRIDENT starts the inferencing phase which is based on static analysis of the program. In this phase, TRIDENT automatically computes (1) the SDC probabilities of individual instructions, and

(2) the overall SDC probability of the program. In the latter case, the user needs to specify the number of sampled instructions when calculating the overall SDC probability of the program, in order to balance the time for analysis with accuracy.

### 5.3.2 Overview and Insights

Because error propagation follows program data-flow at runtime, we need to model program data-flow in the presence of faults at three levels: (1) Static-instruction level, which corresponds to the execution of a static data-dependent instruction sequence and the transfer of results between registers. (2) Control-flow level, when execution jumps to another program location. (3) Memory level, when the results need to be transferred back to memory. TRIDENT is divided into three sub-models to abstract the three levels, respectively, and we use $f_s$ , $f_c$ and $f_m$ to represent them. The main algorithm of TRIDENT tracking error propagation from a given location to the program output is summarized in Algorithm 1.

*Static-Instruction Sub-Model ( $f_s$ )*: First, $f_s$ is used to trace error propagation of an arbitrary fault activated on a static data-dependent instruction sequence. It determines the propagation probability of the fault from where it was activated to the end of the sequence. For example, in Figure 5.2B, the model computes the probability of the fault propagating to the result of the comparison instruction given that the fault is activated at the load instruction (Line 4 in Algorithm 1). Previous models trace error propagation in data dependant instructions based on the dynamic data dependency graph (DDG) which records the output and operand values of each dynamic instruction in the sequence [51, 130]. However, such detailed DDGs are very expensive to generate and process, and hence the models do not scale. $f_s$ avoids generating detailed dynamic traces and instead computes the propagation probability of each static instruction based on its average case at runtime to determine the error propagation in a static data-dependent instruction sequence. Since each static instruction is designed to manipulate target bits in a pre-defined way, the propagation probability of each static instruction can be derived. We can then aggregate the probabilities to calculate the probability of a fault propagating from a given instruction to another instruction within the same static data-dependent instruction sequence.

*Control-Flow Sub-Model ($f_c$ )*: As explained, a fault may propagate to branches and cause the execution path of the program to diverge from its fault-free execution. We divide the propagation into two phases after divergence: The first phase, modeled by $f_c$ , attempts to figure out which dynamic store instructions will be corrupted at what probabilities if a conditional branch is corrupted (Lines 3-5 in Algorithm 1). The second phase traces what happens if the fault propagates to memory, and is modeled by $f_m$ . The key observation is that error propagation to memory through a conditional branch that leads to control-flow divergence can be abstracted into a few probabilistic events based on branch directions. This is because the probabilities of the incorrect executions of store instructions are decided by their execution paths and the corresponding branch probabilities. For example, in the function *init()* in Figure 5.2A, if the comparison instruction takes the *F* branch, the store instruction is not supposed to be executed, but if a fault modifies the direction of the branch to the *T* branch, then it will be executed and lead to memory corruption. A similar case occurs where the comparison instruction is supposed to take the *T* branch. Thus, the store instruction is corrupted in either case.

*Memory Sub-Model ($f_m$ )*: $f_m$ tracks the propagation from corrupted store instructions to the program output, by tracking memory dependencies of erroneous values until the output of the program is reached. During the tracking, other sub-models are recursively invoked where appropriate. $f_m$ then computes the propagation probability from the corrupted store instruction to the program output (Lines 7-9 in Algorithm 1). A memory data-dependency graph needs to be generated for tracing propagations at the memory level because we have to know which dynamic load instruction reloads the faulty data previously written by an erroneous store instruction (if any). This graph can be expensive to construct and traverse due to the huge number of the dynamic store and load instructions in the program. However, we find that the graph can be pruned by removing redundant dependencies between symmetric loops, if there are any. Consider as an example the two loops in *init()* and *run()* in Figure 5.2A. The first loop updates an array, and the second one reads from the same array. Thus, there is a memory dependence between every pair of iterations of the two loops. In this case, instead of tracking every dependency between dynamic instructions, we only track the aggregate dependencies between

the two loops. As a result, the memory dependence graph needs only two nodes to project the dependencies between the stores and loads in their iterations.

**Algorithm 1:** The Core Algorithm in TRIDENT

```
1 sub-models f_s , f_c , and f_m ;
  Input  : I: Instruction where the fault occurs
  Output: P_SDC: SDC probability
2 p_s = f_s (I);
3 if inst. sequence containing I ends with branch I_b then
4 |   // Get the list of stores corrupted and their prob.
5 |   [¡I_c, p_c¿, ...] = f_c (I_b);
6 |   // Maximum propagation prob. is 1
7 |   Foreach(¡I_c, p_c¿): P_SDC += p_s * p_c * f_m (I_c);
8 else if inst. sequence containing I ends with store I_s then
9 |   P_SDC = p_s*f_m (I_s);
```

### 5.3.3 Details: Static-Instruction Sub-Model ($f_s$ )

Once a fault is activated at an executed instruction, it starts propagating on its static data-dependent instruction sequence. Each sequence ends with a store, a comparison or an instruction of program output. In these sequences, the probability that each instruction masks the fault during the propagation can be determined by analyzing the mechanism and operand values of the instruction. This is because instructions often manipulate target bits in predefined ways.

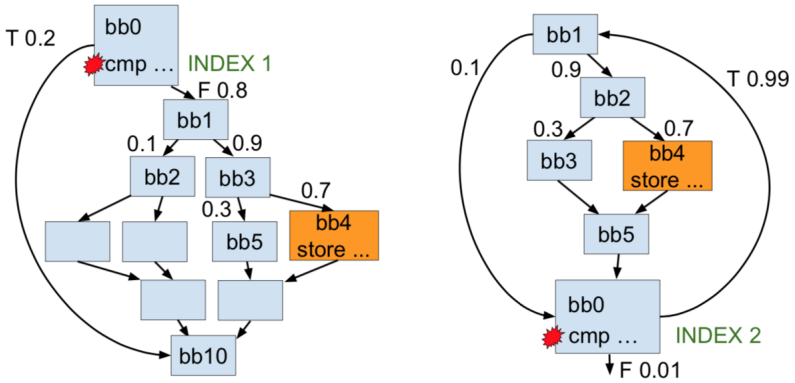
Given a fault that occurs and is activated on an instruction, $f_s$ computes the probability of error propagation when the execution reaches the end of the static computation sequence of the instruction. We use a code example in Figure 5.2B to explain the idea. The code is from *Pathfinder* [33], and shows a counter being incremented until a positive value is reached. In Figure 5.2B, *INDEX 1-3* form a static data-dependent instruction sequence, which an error may propagate along. Assuming a fault is activated at *INDEX 1* and affects *$1*, the goal of $f_s$ is to tell the probabilities of propagation, masking and crash after the execution of *INDEX 3*, which is the last instruction on the sequence. $f_s$ traces the error propagation from *INDEX 1* to *INDEX 3* by aggregating the propagation probability of each instruction on the sequence. We use a tuple for each instruction to represent its probabilities which are shown in the brackets on the right of each instruction in

Figure 5.2B. There are three numbers in each tuple, which are the probabilities of propagation, masking and crash respectively, given that an operand of the instruction is erroneous (we explain how to compute these later). For example, for *INDEX 3*, *(0.03, 0.97, 0)* means that the probability of the error continuing to propagate when *INDEX 3* is corrupted is 0.03, whereas 0.97 is the probability that the error will be masked and not propagate beyond *INDEX 3*. Finally, the probability of a crash at *INDEX 3*, in this case, is 0. Note that the probabilities in each tuple should sum to 1.

After calculating the individual probabilities, $f_s$ aggregates the propagation probability in each tuple of *INDEX 1, 2* and *3* to calculate the propagation probability from *INDEX 1* to *INDEX 3*. That is given by *1*1*0.03=3%* for the probability of propagation, and the probabilities of masking and crash are *97%* and *0%* respectively. Thus, if a fault is activated at *INDEX 1*, there is a 3% of probability that the branch controlled by *INDEX 3* will be flipped, causing a control-flow divergence.

We now explain how to obtain the tuple for each instruction. Each tuple is approximated based on the mechanism of the instruction and/or the profiled values of the instruction's operands. We observe that there are only a few types of instructions that have non-negligible masking probabilities: they are comparisons (e.g., *CMP*), logic operators (e.g., *XOR*) and casts (e.g., *TRUNC*). We assume the rest of instructions neither move nor discard corrupted bits - this is a heuristic we use for simplicity (we discuss its pros and cons in Section 6.7.1).

In the example in Figure 5.2B, the branch direction will be modified based on whether *INDEX 3* computes a positive or negative value. In either case, only a flip of the sign bit of *$1* will modify the branch direction. Hence, the error propagation probability in the tuple of *INDEX 3* is $1/32 = 0.03$, assuming a 32-bit data width. We derive crash probabilities in the tuples for instructions accessing memory (i.e., load and store instructions). We consider crashes that are caused by program reading or writing out-of-bound memory addresses. Their probabilities can be approximated by profiling memory size allocated for the program (this is found in the */proc/* filesystem in Linux). Prior work [51] has shown that these are the dominant causes of crashes in programs due to soft errors.




(a) Example of NLT (b) Example of LT

**Figure 5.3:** NLT and LT Examples of the CFG

### 5.3.4 Details: Control-Flow Sub-Model ($f_c$ )

Recall that the goal of $f_c$ is to figure out which dynamic store instructions will be corrupted and at what probabilities, if a conditional branch is corrupted. We classify all comparison instructions that are used in branch conditions into two types based on whether they terminate a loop. The two types are (1) *Non-Loop-Terminating cmp* (NLT), and (2) *Loop-Terminating cmp* (LT). Figure 5.3 shows two Control Flow Graphs (CFGs), one for each case. We also profile the branch probability of each branch and mark it beside each corresponding branch for our analysis purpose. For example, if a branch probability is 0.2, it means during the execution there is 20% probability the branch is taken. We will use the two examples in Figure 5.3 to explain $f_c$ in each case.

#### Non-Loop-Terminating CMP (NLT)

If a comparison instruction does not control the termination of a loop, it is NLT. In Figure 5.3A, *INDEX 1* is a NLT, dominating a store instruction in *bb4*. There are two cases for the store considered as being corrupted in $f_c$ : (1) The store is not executed while it should be executed in a fault-free execution. (2) The store is executed while it should *not* be executed in a fault-free execution. Combining these cases, the probability of the store instruction being corrupted can be represented by

Equation 5.1.

$$P_c = P_e / P_d \tag{5.1}$$

In the equation, $P_c$ is the probability of the store being corrupted, $P_e$ is the execution probability of the store instruction in fault-free execution, and $P_d$ is the branch probability of which direction dominates the store.

We illustrate how to derive the above equation using the example in Figure 5.3A. There are two legal directions a branch can take. In the first case, the branch of *INDEX 1* is supposed to take the *T* branch at the fault-free execution (20% probability), but the *F* branch is taken instead due to the corrupted *INDEX 1*. The store instruction in *bb4* will be executed when it is not supposed to be executed and will hence be corrupted. The probability that the store instruction is executed in this case is calculated as $0.2 * 0.9 * 0.7 = 0.126$ based on the probabilities on its execution path (*bb0-bb1-bb3-bb4*). In the second case, if the *F* branch is supposed to be taken in a fault-free execution (80% probability), but the *T* branch is taken instead due to the fault, the store instruction in *bb4* will not be executed, while it is supposed to have been executed in some execution path in the fault-free execution under the *F* branch. For example, in the fault-free execution, path *bb0-bb1-bb3-bb4* will trigger the execution of the store. Therefore, the probability of the store instruction being corrupted in this case is $0.8 * 0.9 * 0.7 = 0.504$. Therefore, adding the two cases together, we get $f_c$ in this example as $0.126 + 0.504 = 0.63$. The Equation 5.1 is simplified by integrating the terms in the calculations. In this example, in Equation 5.1, $P_e$ is $0.8 * 0.9 * 0.7$ (*bb0-bb1-bb3-bb4*), $P_d$ is 0.8 (*bb0-bb1*), thus $P_c$ is $0.8 * 0.9 * 0.7/0.8 = 0.63$. Note that if the branch immediately dominates the store instruction, then the probability of the store being corrupted is 1, as shown by the example in Figure 5.2.

**Loop-Terminating CMP (LT)**

If a comparison instruction controls the termination of a loop, it is LT. For example, in Figure 5.3B, the back-edge of *bb0* forms a loop, which can be terminated by the condition computed by *INDEX 2*. Hence, *INDEX 2* is a LT. We find that the probability of the store instruction being corrupted can be represented by Equation. 5.2.

$$P_c = P_b * P_e \tag{5.2}$$

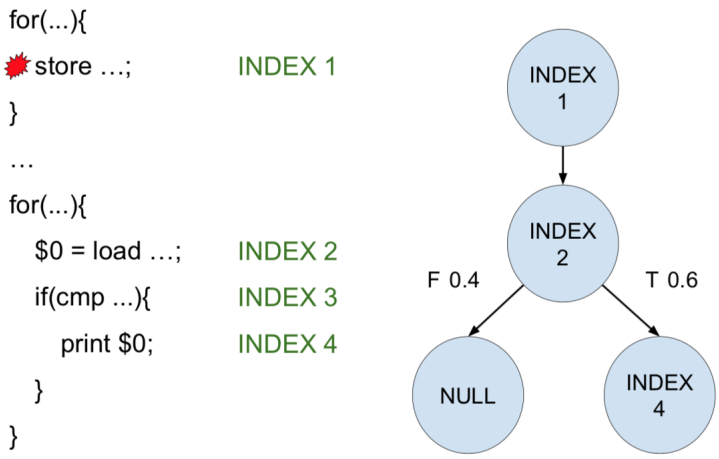
$P_c$ is the probability that a dynamic store instruction is corrupted if the branch is modified, $P_b$ is the execution probability of the back-edge of the branch, and $P_e$ is the execution probability of the store instruction dominated by the back-edge.

We show the derivation of the above equation using the example in Figure 5.3B. In the first case, if the *T* branch (the loop back-edge) is supposed to be taken in a fault-free execution (99% probability), the store instruction in *bb4* may or may not execute, depending on the branch in *bb2*. But if a fault modifies the branch of *INDEX 2*, the store will certainly not execute. So we need to omit the probabilities that the store is not executed in the fault-free execution to calculate the corruption probability of the store. They are $0.99 * 0.9 * 0.3 = 0.27$ for the path *bb0-bb1-bb2-bb3* and $0.99 * 0.1 = 0.099$ for *bb0-bb1-bb0*. Hence, the probability of a corrupted store in this case is $0.99 - 0.27 - 0.099 = 0.62$. In the second case where the *F* branch should be taken in a fault-free execution (1% probability), if the fault modifies the branch, the probability of a corrupted store instruction is $0.01 * 0.9 * 0.7 = 0.0063$. Note that this is usually a very small value which can be ignored. This is because the branch probabilities of a loop-terminating branch are usually highly biased due to the multiple iterations of the loop. So the total probability in this example is approximated to be 0.62, which is what we calculated above. Equation 5.2 is simplified by integrating and cancelling out the terms in the calculations. In this example, $P_b$ is 0.99 (*bb0-bb1*), $P_e$ is 0.7*0.9 (*bb1-bb2-bb4*), and thus $P_c$ is $0.99 * 0.7 * 0.9 = 0.62$.

### 5.3.5 Details: Memory Sub-Model ($f_m$ )

Recap that $f_m$ reports the probability for the error to propagate from the corrupted memory locations to the program output. The idea is to represent memory data dependencies between the load and store instructions in an execution, so that the model can trace the error propagation in the memory.

We use the code example in Figure 5.4A to show how we prune the size of the memory dependency graph in $f_m$ by removing redundant dependencies (if any). There are two inner loops in the program. The first one executes first, storing data to an array in memory (*INDEX 1*). The second loop executes later, loading the data from the memory (*INDEX 2*). Then the program makes some decision (*INDEX 3*) and decides whether the data should be printed (*INDEX 4*) to the program output.




(a) Code Example (b) Dependency Graph

**Figure 5.4:** Examples for Memory Sub-model

Note that the iterations between loops are symmetric in the example, as both manipulate the same array (one updates, and the other one reloads). This is often seen in programs because they tend to manipulate data in blocks due to spatial locality. In this example, if one of the dynamic instructions of *INDEX 1* is corrupted, one of the dynamic instructions of *INDEX 2* must be corrupted too. Therefore, instead of having one node for every dynamic load and store in the iterations of the loop executions, we need only two nodes in the graph to represent the dependencies. The rest of the dependencies in the iterations are redundant, and hence can be removed from the graph as they share the same propagation. The dependencies between dynamic loads and stores are tracked at runtime with their static indices and operand memory addresses recorded. The redundant dependencies are pruned when repeated static load and store pairs are detected.

We show the memory data dependency graph of $f_m$ for the code example in Figure 5.4B. Assume each loop is invoked once with many iterations. We create a node for the store (*INDEX 1*), load (*INDEX 2*) and printf (*INDEX 3*, as program output) in the graph. We draw an edge between nodes to present their dependencies. Because *INDEX 3* may cause divergence of the dependencies and hence error propagation, we weight the propagation probability based on its ex-

ecution probability. We place a *NULL* node as a placeholder indicating masking if *F* branch is taken in *INDEX 3*. Note that an edge between nodes may also represent a static data-dependent instruction sequence, e.g., the edge between *INDEX 2* and *INDEX 4*. Therefore, $f_s$ is recursively called every time a static data-dependent instruction sequence is encountered. We then aggregate the propagation probabilities starting from the node of *INDEX 1* to each leaf node in the graph. Each edge may have different propagation probabilities to aggregate – it depends on what $f_s$ outputs if a static data-dependent instruction sequence is present on the edge. In this example, assume that $f_s$ always outputs 1 as the propagation probability for each edge. Then, the propagation probability to the program output (*INDEX 4*), if one of the store (*INDEX 1*) in the loop is corrupted, is $1*1*1*0.6/(0.4+0.6)+1*1*0*0.4/(0.4+0.6)=0.6$. The zero in the second term represents the masking of the *NULL* node. As an optimization, we memoize the propagation results calculated for store instructions to speed up the algorithm. For example, if later the algorithm encounters *INDEX 1*, we can use the memoized results, instead of recomputing them. We will evaluate the effectiveness of the pruning in Section 5.4.3.

**Floating Point:** When we encounter any floating point data type, we apply an additional masking probability based on the output format of the floating point data. For example, in benchmarks such as *Hotspot*, the *float* data type is used. By default, *Float* carries 7-digit precision, but in (many) programs' output, a "%g" parameter is specified in *printf* which prints numbers with only 2-digit precision. Based on the specification of IEEE-754 [6], we assume that only the mantissa bits (23 bits in *Float*) may affect the 5 digits that are cut off in the precision. This is because bit-flips in exponential bits likely cause large deviations in values, and so cutting-off the 5 digits in the precision is unlikely to mask the errors in the exponent. We also assume that each mantissa bit has equal probability to affect the missing 5 digits of precision. In that way, we approximate the propagation probability to be *((32-23)+23*(2/7))/32 = 48.66%*. We apply this masking probability on top the propagation probabilities, for *Float* data types used with the non-default format of *printf*.

## 5.4 Evaluation

In this section, we evaluate TRIDENT in terms of its accuracy and scalability. To evaluate accuracy, we use TRIDENT to predict overall SDC probabilities of programs as well as the SDC probabilities for individual instructions, and compare them with those obtained using FI and the simpler models. To evaluate scalability, we measure the time for executing TRIDENT, and compare it with the time taken by FI. We first present the experimental setup and then the results. We also make TRIDENT and the experimental data publicly available[1].

### 5.4.1 Experimental Setup

#### Benchmarks

We choose eleven benchmarks from common benchmark suites [18, 33, 68], and publicly available scientific programs [8, 79, 136] — they are listed in Table 7.1. Our benchmark selection is based on three criteria: (1) Diversity of domains and benchmark suites, (2) whether we can compile with our LLVM infrastructure, and (3) whether fault injection experiments of the programs can finish within a reasonable amount of time. We compiled each benchmark with LLVM with standard optimizations (-O2).

#### FI Method

We use LLFI [147] which is a publicly available open-source fault injector to perform FIs at the LLVM IR level on these benchmarks. LLFI has been shown to be accurate in evaluating SDC probabilities of programs compared to assembly code level injections [147]. We inject faults into the destination registers of the executed instructions to simulate faults in the computational elements of the processor as per our fault model. Further, we inject single bit flips as these are the de-facto model for emulating soft errors at the program level, and have been found to be accurate for SDCs [121]. There is only one fault injected in each run, as soft errors are rare events with respect to the time of execution of a program. Our FI method ensures that all faults are activated, i.e., read by an instruction of the program, as we define

[1] https://github.com/DependableSystemsLab/Trident

**Table 5.1:** Characteristics of Benchmarks

| Benchmark | Suite/Author | Area | Program Input |
|---|---|---|---|
| Libquantum | SPEC | Quantum computing | 33 5 |
| Blackscholes | Parsec | Finance | in_4.txt |
| Sad | Parboil | Video encoding. | reference.bin frame.bin |
| Bfs | Parboil | Graph traversal | graph_input.dat |
| Hercules | Carnegie Mellon University | Earthquake simulation | scan simple_case.e |
| Lulesh | Lawrence Livermore National Laboratory | Hydrodynamics modeling | -s 1 -p |
| PuReMD | Purdue University | Reactive molecular dynamics simulation | geo ffield control |
| Nw | Rodinia | DNA sequence optimization | 2048 10 1 |
| Pathfinder | Rodinia | Dynamic programming | 1000 10 |
| Hotspot | Rodinia | Temperature and power simulation | 64 64 1 1 temp_64 power_64 |
| Bfs | Rodinia | Graph traversal | graph4096.txt |

SDC probabilities based on the activated instructions (Section 3.2). The FI method is in line with other papers in the area [12, 12, 51, 64, 82].

### 5.4.2 Accuracy

We design two experiments to evaluate the accuracy of TRIDENT. The first experiment examines the prediction of overall SDC probabilities of programs, and the second examines predicted SDC probabilities of individual instructions. In the experiments, we compare the results derived from TRIDENT with those from the two simpler models and FI. As described earlier, TRIDENT consists of three sub-models in order: $f_s$ , $f_c$ and $f_m$ . We create two simpler models to (1) understand the accuracy gained by enabling each sub-model and (2) as a proxy to investigate

other models, which often lack modeling beyond static data dependencies (Section 5.6.3 performs a more detailed comparison with prior work). We first disable $f_m$ in TRIDENT, leaving the two sub-models $f_s$ and $f_c$ enabled, to create a model: $f_s + f_c$ . We then further remove $f_c$ to create the second simplified model which only has $f_s$ enabled, which we represent as $f_s$ .

### Overall SDC probability

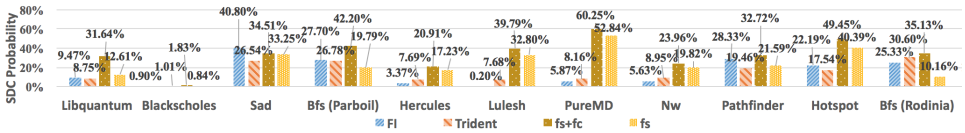



**Figure 5.5:** Overall SDC Probabilities Measured by FI and Predicted by the Three Models (Margin of Error for FI: $\pm$0.07% to $\pm$1.76% at 95% Confidence)

To evaluate the overall SDC probability of a given program, we use statistical FI. We measure error bars for statistical significance at the 95% confidence level. We randomly sample 3,000 dynamic instructions for FIs (one fault per run) as these yield tight error bars at the 95% confidence level ($\pm$0.07% to $\pm$1.76%) - this is in line with other work that uses FI. We calculate SDC probability of each program based on how many injected faults result in SDC. We then use TRIDENT, as well as the two simpler models, to predict the SDC probability of each program, and compare the results with those from FI. To ensure fair comparison, we sample 3,000 instructions in our models as well (Section 5.3.1).

The results are shown in Figure 5.5. We use *FI* to represent the FI method, TRIDENT for our three-level model, and *fs+fc* and *fs* for the two simpler models. We find TRIDENT prediction matches the overall SDC probabilities obtained through FI, with a maximum difference of 14.26% in *Sad*, and a minimum difference of 0.11% in *Blackscholes*, both in percentage points. This gives a *mean absolute error* of 4.75% in overall SDC prediction. On the other hand, $f_s + f_c$ and $f_s$ have a *mean absolute error* of 19.56% and 15.13% respectively compared to FI – *more than 4 and 3 times higher than those obtained using the complete three-level model.* On average, $f_s + f_c$ and $f_s$ predict the overall SDC probability as 33.85%

and 23.76% across the different programs, whereas TRIDENT predicts it to be 14.83%. The SDC probability obtained from FI is 13.59%, which is much more in line with the predictions of TRIDENT.

We observe that in *Sad, Lulesh* and *Pathfinder*, TRIDENT encounters relatively larger differences between the prediction and the FI results (14.26%, 7.48% and 8.87% respectively). The inaccuracies are due to a combination of gaps in the implementation, assumptions, and heuristics we used in TRIDENT. We discuss them in Section 5.6.1.

To compare the results more rigorously, we use a paired T-test experiment [134] to determine how similar the predictions of the overall SDC probabilities by TRIDENT are to the FI results.[2] Since we have 11 benchmarks, we have 11 sets of paired data with one side being FI results and the other side being the prediction values of TRIDENT. The null hypothesis is that there is no statistically significant difference between the results from FIs and the predicted SDC probabilities by TRIDENT in the 11 benchmarks. We calculate the p-value in the T-test as 0.764. By the conventional criteria (p-value$>$0.05), we fail to reject the null hypothesis, indicating that the predicted overall SDC probabilities by TRIDENT are not statistically different from those obtained by FI.

We find that the model $f_s + f_c$ always over-predicts SDCs compared with TRIDENT. This is because an SDC is assumed once an error propagates to store instructions, which is not always the case, as it may not propagate to the program output. On the other hand, $f_s$ may either over-predict SDCs (e.g., Libquantum, Hercules) because an SDC is assumed once an error directly hits any static data-dependent instruction sequence ending with a store, or under-predict them (e.g., Bfs, Blackscholes) because error propagation is not tracked after control-flow divergence.

### SDC Probability of Individual Instructions

We now examine the SDC probabilities of individual instructions predicted by TRIDENT and compare them to the FI results. The number of static instruc-

[2] We have verified visually that the differences between the two sides of every pair are approximately normally distributed in all the T-test experiments we conduct, which is the requirement for validity of the T-test.

tions per benchmark varies from 76 to 4,704, with an average of 944 instructions. Because performing FIs into each individual instruction is very time-consuming, we choose to inject 100 random faults per instruction to bound our experimental time. We then input each static instruction to TRIDENT, as well as the two simpler models ($f_s$ +$f_c$ and $f_s$ ), to compare their predictions with the FI results. As before, we conduct paired T-test experiments [134] to measure the similarity (or not) of the predictions to the FI results. The null hypothesis for each of the three models in each benchmark is that there is no difference between the FI results and the predicted SDC probability values in each instruction.

**Table 5.2:** p-values of T-test Experiments in the Prediction of Individual Instruction SDC Probability Values ($p > 0.05$ indicates that we are not able to reject our null hypothesis – the counter-cases are shown in bold)

| Benchmark | TRIDENT | fs+fc | fs |
|---|---|---|---|
| Libquantum | 0.602 | **0.000** | **0.000** |
| Blackscholes | 0.392 | 0.173 | 0.832 |
| Sad | **0.000** | **0.003** | **0.000** |
| Bfs (Parboil) | 0.893 | **0.000** | 0.261 |
| Hercules | 0.163 | **0.000** | **0.003** |
| Lulesh | **0.000** | **0.000** | **0.000** |
| PureMD | 0.277 | **0.000** | **0.000** |
| Nw | 0.059 | **0.000** | **0.000** |
| Pathfinder | **0.033** | 0.130 | 0.178 |
| Hotspot | 0.166 | **0.000** | **0.000** |
| Bfs (Rodinia) | 0.497 | **0.001** | 0.126 |
| No. of rejections | 3/11 | 9/11 | 7/11 |

The p-values of the experiments are listed in the Table 5.2. At the 95% confidence level, using the standard criteria ($p > 0.05$), we are not able to reject the null hypothesis in 8 out of the 11 benchmarks using TRIDENT in the predictions. This indicates that the predictions of TRIDENT are shown to be statistically indistinguishable from the FI results in most of the benchmarks we used. The three outliers for TRIDENT again are *Sad, Lulesh* and *Pathfinder*. Again, even though the individual instructions' SDC probabilities predicted are statistically distinguishable from the FI results, these predicted values are still reasonably close to the FI results.

In contrast, when using $f_s + f_c$ and $f_s$ to predict SDC probabilities for each individual instruction, there are only 2 and 4 out of the 11 benchmarks having p-values greater than 0.05, indicating that the null hypotheses cannot be rejected for most of the benchmarks. *In other words, the predictions from the simpler models for individual instructions are (statistically) significantly different from the FI results.*

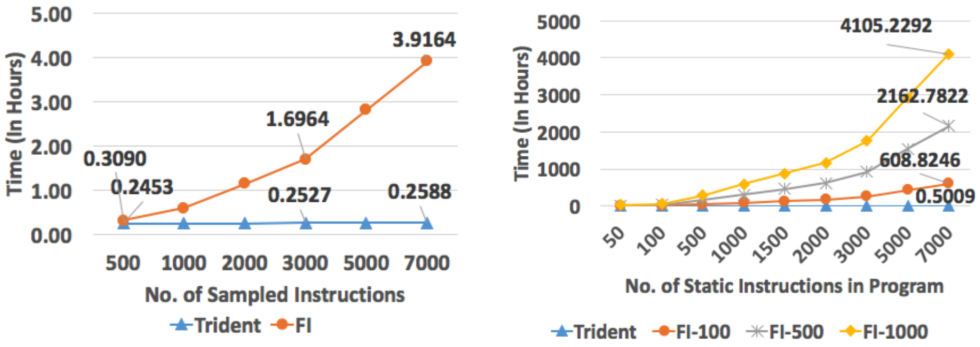
### 5.4.3 Scalability

In this section, we evaluate the scalability of TRIDENT to predict the overall SDC probabilities of programs and the SDC probabilities of individual instructions, and compare it to FI. By scalability, we mean the ability of the model to handle large numbers of instruction samples in order to obtain tighter bounds on the SDC probabilities. In general, the higher the number of sampled instructions, the higher the accuracy and hence the tighter are the bounds on SDC probabilities for a given confidence level (e.g., 95% confidence). This is true for both TRIDENT and for FI. The number of instructions sampled for FI in prior work varies from 1,000 [147] to a few thousands [51, 52, 90]. We vary the number of samples from 500 to $7,000$. The number of samples is equal to the number of FI trials as one fault is injected per trial.

Note that the total computation is proportional to both the time and power required to run each approach. Parallelization will reduce the time spent, but not the power consumed. We assume there is no parallelization for the purpose of comparison in the case of TRIDENT and FI, though both TRIDENT and FI can be parallelized. Therefore, the computation can be measured by the wall-clock time.

#### Overall SDC Probability

The results of the time spent to predict the overall SDC probability of program are shown in Figure 5.6A. The time taken in the figure is projected based on the measurement of one FI trial (averaged over 30 FI runs). As seen, the curve of FI time versus number of samples is much steeper than that of TRIDENT, which is almost flat. TRIDENT is 2.37 times faster than the FI method at 1,000 samples, it is 6.7 times faster at 3,000 samples and 15.13 times faster at 7,000 samples.




(a) Overall SDC Probability (b) Instruction SDC Probability

**Figure 5.6:** Computation Spent to Predict SDC Probability

From 500 to 7,000 samples, the time taken by TRIDENT increases only 1.06 times (0.2453 to 0.2588), whereas it increases 14 times (0.2453 to 3.9164) for FI - an exact linear increase. The profiling phase of TRIDENT takes 0.24 hours (or about 15 minutes) on average. This is a fixed cost incurred by TRIDENT regardless of the number of sampled instructions. However, once the model is built, the incremental cost of calculating the SDC probability of a new instruction is minimal (we only calculate the SDC probabilities on demand to save time). FI does not incur a noticeable fixed cost, but its time rapidly increases as the number of sampled instructions increase. This is because FI has to run the application from scratch on each trial, and hence ends up being much slower than TRIDENT as the number of samples increase.

### Individual Instructions

Figure 5.6B compares the average time taken by TRIDENT to predict SDC probabilities of individual instructions with FI, for different numbers of static instructions. We consider different numbers of samples for each static instruction chosen for FI: 100, 500 and 1,000 (as mentioned in Section 5.3.1, TRIDENT does not need samples for individual instructions' SDC probabilities). We denote the number of samples as a suffix for the FI technique. For example, *FI-100* indicates 100 samples are chosen for performing FI on individual instructions. We also vary the

number of static instructions from 50 to 7,000 (this is the X-axis). As seen from the curves, the time taken by TRIDENT as the number of static instructions vary remains almost flat. On average, it takes 0.2416 hours at 50 static instructions, and 0.5009 hours at 7,000 static instructions, which is only about a 2X increase. In comparison, the corresponding increases for *FI-100* is 140X, which is linear with the number of instructions. Other FI curves experience even steeper increases as they gather more samples per instruction.

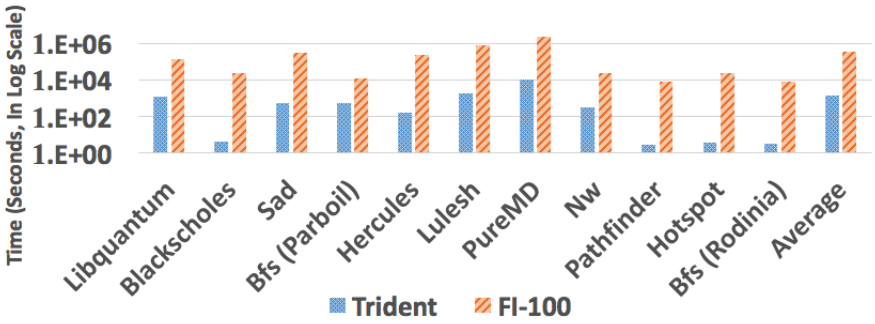



**Figure 5.7:** Time Taken to Derive the SDC Probabilities of Individual Instructions in Each Benchmark

Figure 5.7 shows the time taken by TRIDENT and *FI-100* to derive the SDC probabilities of individual instructions in each benchmark (due to space constraints, we do not show the other FI values, but the trends were similar). As can be seen, there is wide variation in the times taken by TRIDENT depending on the benchmark program. For example, the time taken in *PureMD* is 2.893 hours, whereas it is 2.8 seconds in *Pathfinder*. This is because the time taken by TRIDENT depends on factors such as (1) the total number of static instructions, (2) the length of static data-dependent instruction sequence, (3) the number of dynamic branches that require profiling, and (4) the number of redundant dependencies that can be pruned. The main reason for the drastic difference between *PureMD* and *Pathfinder* is that we can prune only 0.08% of the redundant dependencies in the former, while we can prune 99.83% of the dependencies in the latter. On average, 61.87% of dynamic load and store instructions are redundant and hence removed from the memory dependency graph.

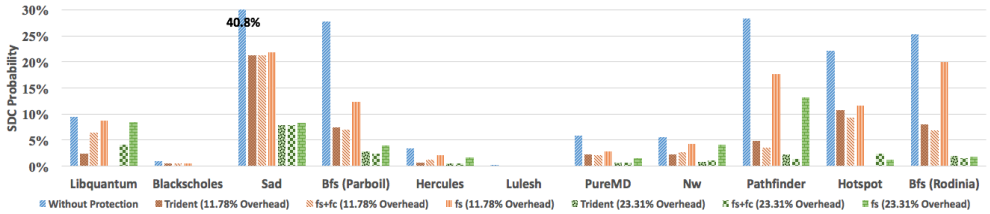



**Figure 5.8:** SDC Probability Reduction with Selective Instruction Duplication at 11.78% and 23.31% Overhead Bounds (Margin of Error: $\pm$0.07% to $\pm$1.76% at 95% Confidence)

## 5.5 Use Case: Selective Instruction Duplication

In this section, we demonstrate the utility of TRIDENT by considering a use-case of selectively protecting a program from SDC causing errors. The idea is to protect only the most SDC-prone instructions in a program so as to achieve high coverage while bounding performance costs. We consider instruction duplication as the protection technique, as it has been used in prior work [51, 52, 97]. The problem setting is as follows: given a certain performance overhead $P$, what static instructions should be duplicated in order to maximize the coverage for SDCs while keeping the overhead below $P$.

Solving the above problem involves finding the SDC probability of each instruction in the program in order to decide which set of instructions should be duplicated. It also involves calculating the performance overhead of duplicating the instructions. We use TRIDENT for the former, namely, to estimate the SDC probability of each instruction, without using FI. For the latter, we use the dynamic execution count of each instruction as a proxy for the performance overhead incurred by it. We then formulate the problem as a classical 0-1 knapsack problem [99], where the objects are the instructions and the knapsack capacity is represented by $P$, the maximum allowable performance overhead. Further, object profits are represented by the estimated SDC probability (and hence selecting the instruction means obtaining the coverage), and object costs are represented by the dynamic execution count of the instruction. Note that we assume that the SDC probability estimates of the instructions are independent of each other – while this is not necessarily true in practice, it keeps the model tractable, and in the worst

case leads to conservative protection (i.e., over-protection). We use the dynamic programming algorithm for the 0-1 knapsack problem - this is similar to what prior work did [97].

For the maximum performance overhead $P$, we first measure the overhead of duplicating all the instructions in the program (i.e., full duplication) and set this as the baseline as it represents the worst-case overhead. The overheads are measured based on the wall-clock time of the actual execution of the duplicated programs (averaged on 3 executions each). We find that full duplication incurs an overhead of 36.18% across benchmarks. We consider 2 overhead bound levels, namely the $1/3$rd and $2/3$rd of the full duplication overheads, which are (1) 11.78% and (2) 23.31% respectively.

For each overhead level, our algorithm chooses the instructions to protect using the knapsack algorithm. The chosen instructions are then duplicated using a special pass in LLVM we wrote, and the duplication occurs at the LLVM IR level. Our pass also places a comparison instruction after each instruction protected to detect any deviations of the original computations and duplicated computations. If protected instructions are data dependent on the same static data-dependent instruction sequence, we only place one comparison instruction at the latter protected instruction to reduce performance overhead. This is similar to what other related work did [51, 97]. For comparison purposes, we repeat the above process using the two simpler models ($f_s + f_c$ and $f_s$ ). We then use FI to obtain the SDC probabilities of the programs protected using the different models at different overhead levels. Note that FI is used only for the evaluation and not for any of the models.

Figure 5.8 shows the results of the SDC probability reduction at different protection levels. Without protection, the average SDC probability of the programs is 13.59%. At the 11.78% overhead level, after protection based on TRIDENT, $f_s + f_c$ and $f_s$ the corresponding SDC probabilities are 5.50%, 5.53%, 9.29% respectively. On average, the protections provided by the three models reduce the SDC probabilities by 64%, 64% and 40% respectively. At the 23.31% overhead level, after the protections based on TRIDENT, $f_s + f_c$ and $f_s$ respectively, the average SDC probabilities are 1.55%, 2.00% and 4.04%. This corresponds to a reduction of 90%, 87% and 74% of the SDC probability in the baseline respectively. Thus, on average, TRIDENT provides a higher SDC probability reduction for the same

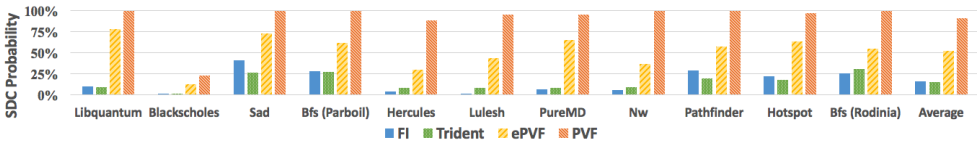



**Figure 5.9:** Overall SDC Probabilities Measured by FI and Predicted by TRIDENT, ePVF and PVF (Margin of Error: $\pm$0.07% to $\pm$1.76% at 95% Confidence)

overhead level compared with the two simpler models.

Taking a closer look, the protection based on $f_s + f_c$ achieves comparable SDC probability reductions with TRIDENT. This is because the relative ranking of SDC probabilities between instructions plays a more dominant role in the selective protection than the absolute SDC probabilities. The ranking of the SDC probabilities of individual instructions derived by $f_s + f_c$ is similar to that derived by TRIDENT. Adding $f_m$ boosts the overall accuracy of the model in predicting the absolute SDC probabilities (Figure 5.5), but not the relative SDC probabilities – the only exception is *Libquantum*. This shows the importance of modeling control-flow divergence, which is missing in other existing techniques [51, 52, 130].

## 5.6 Discussion

We first investigate the sources of inaccuracy in TRIDENT based on the experimental results (Section 5.4). We then examine some of the threats to the validity of our evaluation. Finally, we compare TRIDENT with two closely related prior techniques, namely PVF and ePVF.

### 5.6.1 Sources of Inaccuracy

**Errors in Store Address:** If a fault modifies the address of a store instruction, in most cases, an immediate crash would occur because the instruction accesses memory that is out of bounds. However, if the fault does not cause a crash, it can corrupt an arbitrary memory location, and may eventually lead to SDC. It is difficult to analyze which memory locations may be corrupted as a result of such faults, leading to inaccuracy in the case. In our fault injection experiments, we observe that on average about 5.05% of faults affect addresses in store instructions

and survive from crashes.

**Memory Copy:** Another source of inaccuracy in TRIDENT is that we do not handle bulk memory operations such as memmove and memcpy, which are represented by special instructions in the LLVM IR. We find such operations in benchmark such as *Sad, Lulesh, Hercules* and *PureMD*, which makes our technique somewhat inaccurate for these programs.

**Manipulation of Corrupted Bits:** As mentioned in Section 5.3.3, we assume only instructions such as comparisons, logical operators and casts have masking effects to simplify our calculations, and that none of the other instructions mask the corrupted bits. However, this is not always the case as other instructions may also cause masking. For example, division operations such as *fdiv* may also average out corrupted bits in the mantissa of floating point numbers, and hence mask errors. We find that 1% of the faults affect *fdiv* in program such as *Lulesh*, thereby leading to inaccuracies.

**Conservatism in Determining Memory Corruption:** Recall that when control-flow divergence happens, we assume all the store instructions that are dominated by the faulty branch are corrupted (Section 5.3). This is a conservative assumption, as some stores may end up being coincidentally correct. For example, if a store instruction is supposed to write a zero to its memory location, but is not executed due to the faulty branch, the location will still be correct if there was a zero already in that location. These are called *lucky loads* [42, 51].

### 5.6.2 Threats to Validity

**Benchmarks:** As mentioned in Section 5.4.1, we choose 11 programs to encompass a wide variety of domains rather than sticking to just one benchmark suite (unlike performance evaluation, there is no standard benchmark suite for reliability evaluation). Our results may be specific to our choice of benchmarks, though we have not observed this to be the case. Other work in this domain makes similar decisions [51, 97].

**Platforms:** In this work, we focus on CPU programs for TRIDENT. Graphic Processing Units (GPU) are another important platform for reliability studies. We have attempted to run TRIDENT on GPU programs, but were crippled by the lack

of automated tools for code analysis and fault injection on GPUs. Our preliminary results in this domain using small CUDA kernels (instrumented manually) confirm the accuracy of TRIDENT. However, more rigorous evaluation is needed.

**Program Input:** As the high-fidelity fault injection experiments take a long time (Section 5.4.3), we run each program only under 1 input. This is also the case for almost all other studies we are aware of in this space [51, 52]. Di Leo et at. [46] have found SDC probabilities of programs may change under different program inputs. We plan to consider multiple inputs in our future work.

**Fault Injection Methodology:** We use LLFI, a fault injector that works at the LLVM IR level, to inject single bit flips. While this method is accurate for estimating SDC probabilities of programs [121, 147], it remains an open question as to how accurate it is for other failure types. That said, our focus in this chapter is SDCs, and so this is an appropriate choice for us.

### 5.6.3 Comparison with ePVF and PVF

ePVF (enhanced PVF) is a recent modeling technique for error propagation in programs [51]. It shares the same goal with TRIDENT in predicting the SDC probability of a program, both at the aggregate level and instruction level. ePVF is based on PVF [130], which stands for *Program Vulnerability Factor*. The main difference is that PVF does not distinguish between crash-causing faults and SDCs, and hence its accuracy of SDC prediction is poor [51]. ePVF improves the accuracy of PVF by removing most crashes from the SDC prediction. Unfortunately, ePVF cannot distinguish between benign faults and SDCs, and hence its accuracy suffers accordingly [51]. This is because ePVF only models error propagation in static data-dependent instruction sequence and in memory if the static data-dependent instruction sequence ends with a store instruction, ignoring error propagation to control-flow and other parts of memory. Both ePVF and PVF, like TRIDENT, require no FI in their prediction of SDC, and can be implemented at the LLVM IR level[3]. We implement both techniques using LLVM, and compare their results with TRIDENT's results.

Since crashes and SDCs are mutually exclusive, by removing the crash-causing

[3]ePVF was originally implemented using LLVM, but not PVF.

faults, ePVF computes a relatively closer result to SDC probability than PVF [51]. However, the *crash propagation model* proposed by ePVF in identifying crashes requires a detailed DDG of the entire program's execution, which is extremely time-consuming and resource hungry. As a result, ePVF can be only executed in programs with a maximum of a million dynamic instructions in practice [51]. To address this issue and reproduce ePVF on our benchmarks and workloads (average 109 million dynamic instructions), we modify ePVF by replacing its *crash propagation model* with the measured results from FI. In other words, we assume ePVF identifies 100% of the crashes accurately, which is higher than the accuracy of the ePVF model. Hence, this comparison is conservative as it overestimates the accuracy of ePVF.

We use TRIDENT, ePVF and PVF to compute the SDC probabilities of the same benchmarks and workloads, and then compare them with FI which serves as our ground truth. The number of randomly sampled faults are 3,000. The results are shown in Figure 5.9. As shown, ePVF consistently overestimates the SDC probabilities of the programs with a *mean absolute error* of 36.78% whereas it is 4.75% in TRIDENT. PVF results in an even larger *mean absolute error* of 75.19% as it does not identify crashes. The observations are consistent with those reported by Fang et al. [51]. The average SDC probability measured by FI is 13.59%. ePVF and PVF predict it as 52.55% and 90.62% respectively, while TRIDENT predicts it as 14.83% and is significantly more accurate as a result.

## 5.7 Summary

In this chapter, we proposed TRIDENT, a three-level model for soft error propagation in programs. TRIDENT abstracts error propagation at static instruction level, control-flow level and memory level, and does not need any fault injection (FI). We implemented TRIDENT in the LLVM compiler, and evaluated it on 11 programs. We found that TRIDENT achieves comparable accuracy as FI, but is much faster and scalable both for predicting the overall SDC probabilities of programs, and the SDC probabilities of individual instructions in a program. We also demonstrated that TRIDENT can be used to guide selective instruction duplication techniques, and is significantly more accurate than simpler models.

# Chapter 6

# Modeling Input-Dependent Error Propagation in Programs

In this chapter, we discuss how program inputs could affect error propagation in programs as real-world applications are executed with arbitrarily different inputs in production environment. We first explain the common misunderstandings found in the literature about error propagation versus program inputs. Then we identify the predominant components that need to be considered when modeling input-dependent error propagation. We find that it is possible to extend the analytical model discussed in Chapter 5 to support multiple inputs while achieving a reasonably high accuracy and speed. Finally, we show the evaluation results of the extended model in bounding the SDC probabilities of a given program with multiple inputs.

## 6.1 Introduction

Fault injections (FIs) are commonly used for evaluating and characterizing programs' resilience, and to obtain the overall SDC probability of a program. In each FI campaign, a single fault is injected into a randomly sampled instruction, and the program is executed till it crashes or finishes. FI therefore requires that the program is executed with a specific input. In practice, a large number of FI campaigns are usually required to achieve statistical significance, which can be extremely time-

consuming. As a result, most prior work limits itself to a single program input or at most a small number of inputs. Unfortunately, the number of possible inputs can be large, and there is often significant variance in SDC probabilities across program inputs. For example, in our experiments, we find that the overall SDC probabilities of the same program (Lulesh) can vary by more than 42 times under different inputs. This seriously compromises the correctness of the results from FI. Therefore, there is a need to characterize the variation in SDC probabilities across multiple inputs, without expensive FIs.

We find that there are two factors determining the variation of the SDC probabilities of the program across its inputs (we call this the *SDC volatility*): (1) Dynamic execution footprint of each instruction, and (2) SDC probability of each instruction (i.e., error propagation behaviour of instructions). Almost all existing techniques [43, 46, 54] on quantifying programs' failure variability across inputs consider only the execution footprint of instructions. However, we find that the error propagation behavior of individual instructions often plays as important a role in influencing the SDC volatility (Section 6.2). Therefore, all existing techniques experience significant inaccuracy in determining a program's SDC volatility.

In this chapter, we propose an automated technique to determine the SDC volatility of a program across different inputs, that takes into account both the execution footprint of individual instructions, and their error propagation probabilities. Our approach consists of three steps. First, we perform experimental studies using FI to analyze the properties of SDC volatility, and identify the sources of the volatility. We then build a model, vTRIDENT, which predicts the SDC volatility of programs automatically without any FIs. vTRIDENT is built on our prior model, TRIDENT (Chapter 5) for predicting error propagation, but sacrifices some accuracy for speed of execution. Because we need to run vTRIDENT for multiple inputs, execution speed is much more important than in the case of TRIDENT. The intuition is that for identifying the SDC volatility, it is more important to predict the relative SDC probabilities among inputs than the absolute probabilities. Finally, we use vTRIDENT to bound the SDC probabilities of a program across multiple inputs, while performing FI on only a single input. *To the best of our knowledge, we are the first to systematically study and model the variation of SDC probabilities in programs across inputs.*

The main contributions are as follows:

- We identify two sources of SDC volatility in programs, namely INSTRUCTION-EXECUTION-VOLATILITY that captures the variation of dynamic execution footprint of instructions, and INSTRUCTION-SDC-VOLATILITY that captures the variability of error propagation in instructions, and mathematically derive their relationship (Section 6.2).

- To understand how SDC probabilities vary across inputs, we conduct a FI study using nine benchmarks with ten different program inputs for each benchmark, and quantify the relative contribution of INSTRUCTION-EXECUTION-VOLATILITY and INSTRUCTION-SDC-VOLATILITY (Section 6.3) to the overall SDC volatility.

- Based on the understanding, we build a model, vTRIDENT[1], on top of our prior framework for modeling error propagation in programs TRIDENT (Section 6.4.2). vTRIDENT predicts the SDC volatility of instructions without any FIs, and also bounds the SDC probabilities across a given set of inputs.

- Finally, we evaluate the accuracy and scalability of vTRIDENT in identifying the SDC volatility of instructions (Section 6.5), and in bounding SDC probabilities of program across inputs (Section 6.6).

Our main results are as follows:

- Volatility of overall SDC probabilities is due to both the INSTRUCTION-EXECUTION-VOLATILITY and INSTRUCTION-SDC-VOLATILITY. Using only INSTRUCTION-EXECUTION-VOLATILITY to predict the overall SDC volatility of the program results in significant inaccuracies, i.e., an average of 7.65x difference with FI results (up to 24x in the worst case).

- We find that the accuracy of vTRIDENT is 87.81% when predicting the SDC volatility of individual instructions in the program. The average difference between the variability predicted by vTRIDENT and that by FI is only 1.26x (worst case is 1.29x).

[1] vTRIDENT stands for "Volatility Prediction for TRIDENT".

- With VTRIDENT 78.89% of the given program inputs' overall SDC probabilities fall within the predicted bounds. With INSTRUCTION-EXECUTION-VOLATILITY alone, only 32.22% of the probabilities fall within the predicted bounds.

- Finally, the average execution time for VTRIDENT is about 15 minutes on an input of nearly 500 million dynamic instructions. *This constitutes a speedup of more than 8x compared with the* TRIDENT *model to bound the SDC probabilities, which is itself an order of magnitude faster than FI [59].*

## 6.2 Volatilities and SDC

In this section, we explain how we calculate the overall SDC probability of a program under multiple inputs. Statistical FI is the most common way to evaluate the overall SDC probability of a program and has been used in other related work in the area [43, 54, 65, 66, 88]. It randomly injects a large number (usually thousands) of faults under a given program input, one fault per program execution, by uniformly choosing program instruction for injection from the set of all executed instructions.

Equation 6.1 shows the calculation of the overall SDC probability of the program, $P_{overall}$, from statistical FI. $N_{SDC}$ is the number of FI campaigns that result in SDCs among all the FI campaigns. $N_{total}$ is the total number of FI campaigns. Equation 6.1 can be expanded to the equivalent equations shown in Equation 6.2. $P_i$ is the SDC probability of each (static) instruction that is chosen for FI, $N_i$ is the amount of times that the static instruction is chosen for injection over all FI campaigns. $i$ to $n$ indicates all the distinct static instructions that are chosen for injection.

$$P_{overall} = N_{SDC}/N_{total} \tag{6.1}$$

$$= (\sum_{i=1}^{n} P_i * N_i)/N_{total} = \sum_{i=1}^{n} P_i * (N_i/N_{total}) \tag{6.2}$$

In Equation 6.2, we can see that $N_i/N_{total}$ and $P_i$ are the two relevant factors in the calculation of the overall SDC probability of the program. $N_i/N_{total}$ can

be interpreted as the probability of the static instruction being sampled during the program execution. Because the faults are uniformly sampled during the program execution, $N_i/N_{total}$ is statistically equivalent to the ratio between the number of dynamic executions of the chosen static instruction, and the total number of dynamic instructions in the program execution. We call this ratio the dynamic execution footprint of the static instruction. The larger the dynamic execution footprint of a static instruction, the higher the chance that it is chosen for FI.

Therefore, we identify two kinds of volatilities that affect the variation of $P_{overall}$ when program inputs are changed from Equation 6.2: (1) INSTRUCTION-SDC-VOLATILITY, and (2) INSTRUCTION-EXECUTION-VOLATILITY. INSTRUCTION-SDC-VOLATILITY represents the variation of $P_i$ across the program inputs, INSTRUCTION-EXECUTION-VOLATILITY is equal to the variation of dynamic execution footprints, $N_i/N_{total}$, across the program inputs. We also define the variation of $P_{overall}$ as OVERALL-SDC-VOLATILITY. As explained above, INSTRUCTION-EXECUTION-VOLATILITY can be calculated by profiling the number of dynamic instructions when inputs are changed, which is straight-forward to derive. However, INSTRUCTION-SDC-VOLATILITY is difficult to identify as $P_i$ requires a large number of FI campaigns on every such instruction $i$ with different inputs, which becomes impractical when the program size and the number of inputs become large. As mentioned earlier, prior work investigating OVERALL-SDC-VOLATILITY considers only the INSTRUCTION-EXECUTION-VOLATILITY, and ignores INSTRUCTION-SDC-VOLATILITY [43, 54]. However, as we show in the next section, this can lead to significant inaccuracy in the estimates. Therefore, we focus on deriving INSTRUCTION-SDC-VOLATILITY efficiently in this paper.

## 6.3 Initial FI Study

In this section, we design experiments to show how INSTRUCTION-SDC-VOLATILITY and INSTRUCTION-EXECUTION-VOLATILITY contribute to OVERALL-SDC-VOLATILITY, then explain the variation of INSTRUCTION-SDC-VOLATILITY across programs.

**Table 6.1:** Characteristics of Benchmarks

| Benchmark | Suite/Author | Description | Total Dynamic Instructions (Millions) |
|---|---|---|---|
| Libquantum | SPEC | Simulation of quantum computing | 6238.55 |
| Nw | Rodinia | A nonlinear global optimization method for DNA sequence alignments | 564.63 |
| Pathfinder | Rodinia | Use dynamic programming to find a path on a 2-D grid | 6.71 |
| Streamcluster | Rodinia | Dense Linear Algebra | 3907.70 |
| Lulesh | Lawrence Livermore National Laboratory | Science and engineering problems that use modeling hydrodynamics | 3382.79 |
| Clomp | Lawrence Livermore National Laboratory | Measurement of HPC performance impacts | 11324.17 |
| CoMD | Lawrence Livermore National Laboratory | Molecular dynamics algorithms and workloads | 17136.62 |
| FFT | Open Source | 2D fast Fourier transform | 6.37 |
| Graph | Open Source | Graph traversal in operational research | 0.15 |

### 6.3.1 Experiment Setup

#### Benchmarks

We choose nine applications in total for our experiments. These are drawn from standard benchmark suites, as well as from real world applications. Note that there are very few inputs provided with the benchmark applications, and hence we had to generate them ourselves. We search the entire benchmark suites of Rodinia [33], SPLASH-2 [148], PARSEC [18] and SPEC [68], and choose applications based on two criteria: (1) Compatibility with our toolset (i.e., we could compile them to LLVM IR and work with LLFI), and (2) Ability to generate diverse inputs for our experiments. For the latter criteria, we choose applications that take numeric values as their program inputs, rather than binary files or files of unknown formats, since we cannot easily generate different inputs in these applications. As a result,

there are only three applications in Rodinia and one application in SPEC meeting the criteria. To include more benchmarks, we pick three HPC applications (Lulesh, Clomp, and CoMD) from Lawrence Livermore National Laboratory [70], and two open-source projects (FFT [72] and Graph [71]) from online repositories. The nine benchmarks span a wide range of application domains from simulation to measurement, and are listed in Table 7.1.

### Input Generation

Since all the benchmarks we choose take numerical values as their inputs, we randomly generate numbers for their inputs. The inputs generated are chosen based on two criteria: (1) The input should not lead to any reported errors or exceptions that halt the execution of the program, as such inputs may not be representative of the application's behavior in production, And (2) The number of dynamic executed instructions for the inputs should not exceed 50 billion to keep our experimental time reasonable. We report the total number of dynamic instructions generated from the 10 inputs of each benchmark in Table 7.1. The average number of dynamic instructions per input is 472.95 million, which is significantly larger than what have been used in most other prior work [52, 88, 97, 150, 151]. We consider large inputs to stress vTRIDENT and evaluate its scalability.

### FI methodology

As mentioned before, we use LLFI [147] to perform the FI experiments. For each application, we inject 100 random faults for each static instruction of the application – this yields error bars ranging from 0.03% to 0.55% depending on the application for the 95% confidence intervals. Because we need to derive SDC probabilities of every static instruction, we have to perform multiple FIs on every static instruction in each benchmark. Therefore, to balance the experimental time with accuracy, we choose to inject 100 faults on each static instruction. This adds up to a total number of injections ranging from 26,000 to 2,251,800 in each benchmark, depending on the number of static instructions in the program.

### 6.3.2 Results

#### INSTRUCTION-EXECUTION-VOLATILITY and OVERALL-SDC-VOLATILITY

We first investigate the relationship between INSTRUCTION-EXECUTION-VOLATILITY and OVERALL-SDC-VOLATILITY. As mentioned in Section 6.2, INSTRUCTION-EXECUTION-VOLATILITY is straight-forward to derive based on the execution profile alone, and does not require performing any FIs. If it is indeed possible to estimate OVERALL-SDC-VOLATILITY on the basis of INSTRUCTION-EXECUTION-VOLATILITY alone, we can directly plug in INSTRUCTION-EXECUTION-VOLATILITY to $N_i$ and $N_{total}$ in Equation 6.2 when different inputs are used and treat $P_i$ as a constant (derived based on a single input) to calculate the overall SDC probabilities of the program with the inputs.

We profiled INSTRUCTION-EXECUTION-VOLATILITY in each benchmark and use it to calculate the overall SDC probabilities of each benchmark across all its inputs. To show OVERALL-SDC-VOLATILITY, we calculate the differences between the highest and the lowest overall SDC probabilities of each benchmark, and plot them in Figure 6.1. In the figure, *Exec. Vol.* represents the calculation with the variation of INSTRUCTION-EXECUTION-VOLATILITY alone in Equation 6.2, treating $P_i$ as a constant, which are derived by performing FI on only one input. *FI* indicates the results derived from FI experiment with the set of all inputs of each benchmark. As can be observed, the results for individual benchmark with OVERALL-SDC-VOLATILITY estimated from *Exec. Vol.* alone are significantly lower than the FI results (up to 24x in *Pathfinder*). The average difference is 7.65x. *This shows that* INSTRUCTION-EXECUTION-VOLATILITY *alone is not sufficient to capture* OVERALL-SDC-VOLATILITY*, motivating the need for accurate estimation of* INSTRUCTION-SDC-VOLATILITY. This is the focus of our work.

#### Code Patterns Leading to INSTRUCTION-SDC-VOLATILITY

To figure out the root causes of INSTRUCTION-SDC-VOLATILITY, we analyze the FI results and their error propagation based on the methodology proposed in our prior work [59]. We identify three cases leading to INSTRUCTION-SDC-

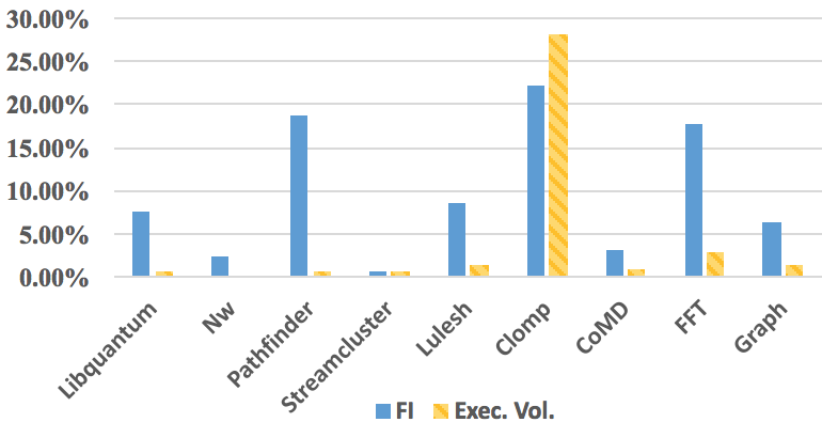



**Figure 6.1:** OVERALL-SDC-VOLATILITY Calculated by INSTRUCTION-EXECUTION-VOLATILITY Alone (Y-axis: OVERALL-SDC-VOLATILITY, Error Bar: 0.03% to 0.55% at 95% Confidence)

VOLATILITY.

*Case 1: Value Ranges of Operands of Instructions*

Different program inputs change the values that individual instructions operate with. For example, in Figure 6.2A, there are three instructions (LOAD, CMP and BR) on a straight-line code sequence. Assume that under some INPUT A, R1 is 16 and R0 is 512, leading the result of the CMP (R3) to be FALSE. Since the highest bit of 512 is the 9th bit, any bit-flip at the bit positions that are higher than 9 in R1 will modify R1 to a value that is greater than R0. This may in turn cause the result of the CMP instruction (R3) to be TRUE. In this case, the probability for the fault that occurred at R1 of the LOAD instruction to propagate to R3 is *(32-9)/32=71.88%* (assuming a 32-bit data width of R1). In another INPUT B, assume R1 is still 16, but R0 becomes 64 of which the highest bit is the 6th bit. In this case, the probability for the same fault to propagate to R3 becomes *(32-6)/32=81.25%*. In this example, the propagation probability increases by almost 10% for the same fault for a different input. In other words, the SDC volatility of the LOAD instruction in the example is changed by about 10%. We find that in the nine benchmarks, the proportion of instructions that fall into this pattern varies from 3.07% (*FFT*) to 15.23% (*Nw*) - the average is 6.98%. The instructions exhibit

different error propagation even if the control flow does not change.

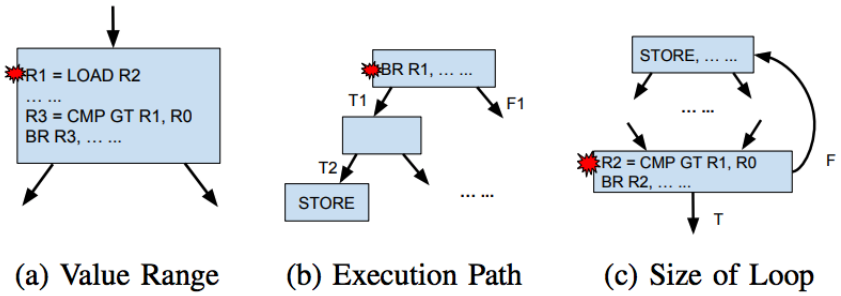



**Figure 6.2:** Patterns Leading to INSTRUCTION-SDC-VOLATILITY

*Case 2: Execution Paths and Branches*

Different program inputs may exercise different execution paths of programs. For example, in Figure 6.2B, there are three branch directions labeled with T1, F1 and T2. Each direction may lead to a different execution path. Assume that the execution probabilities of T1, F1 and T2 are 60%, 70% and 80% for some INPUT A. If a fault occurs at the BR instruction and modifies the direction of the branch from F1 to T1, the probability of this event is 70% as the execution probability of F1 is 70%. In this case, the probability for the fault to propagate to the STORE instruction under T2 is *70%*80%=56%*. Assuming there is another INPUT B which makes the execution probabilities of T1, F1 and T2, 10%, 90% and 30% respectively. The probability for the same fault to propagate to the STORE instruction becomes *90%*30%=27%*. Thus, the propagation probability of the fault decreases by 29% from INPUT A to INPUT B, and thus the SDC volatility of the BR instruction is 29%. In the nine benchmarks, we find that 43.28% of the branches on average exhibit variations of branch probabilities across inputs, leading to variation of SDC probability in instructions.

*Case 3: Number of Iterations of Loops*

The number of loop iterations can change when program inputs are changed, causing volatility of error propagation. For example, in Figure 6.2C, there is a loop whose termination is controlled by the value of R2. The CMP instruction compares R1 against R0 and stores it in R2. If the F branch is taken, the loop will continue, whereas if T branch is taken, the loop will terminate. Assume that under some

INPUT A the value of R0 is 4, and that in the second iteration of the loop, a fault occurs at the CMP instruction and modifies R2 to TRUE from FALSE, causing the loop to terminate early. In this case, the STORE instruction is only executed twice whereas it should be executed 4 times in a correct execution. Because of the early termination of the loop, there are 2 STORE executions missing. Assume there is another INPUT B that makes R0 8, indicating there are 8 iterations of the loop in a correct execution. Now for the same fault in the second iteration, the loop terminates resulting in only 2 executions of the STORE whereas it should execute 8 times. 6 STORE executions are missing with INPUT B (*8-2=6*). If the SDC probability of the STORE instruction stays the same with the two inputs, INPUT B triples (6/2=3) the probability for the fault to propagate through the missing STORE instruction, causing the SDC volatility. In the nine benchmarks, we find that 90.21% of the loops execute different numbers of iterations when the input is changed.

## 6.4 Modeling INSTRUCTION-SDC-VOLATILITY

We first discuss the drawback of TRIDENT which is proposed in Chapter 5. We then describe vTRIDENT, an extension of TRIDENT to predict INSTRUCTION-SDC-VOLATILITY. The main difference between the two models is that vTRIDENT simplifies the modeling in TRIDENT to improve running time, which is essential for processing multiple inputs.

### 6.4.1 Drawbacks of TRIDENT

Even though TRIDENT is orders of magnitude faster than FI and other models in measuring SDC probabilities, it can sometimes take a long time to execute depending on the program input. Further, when we want to calculate the variation in SDC probabilities across inputs, we need to execute TRIDENT once for each input, which can be very time-consuming. For example, if TRIDENT takes 30 minutes on average per input for a given application (which is still considerably faster than FI), it would take more than 2 days (50 hours) to process 100 inputs. This is often unacceptable in practice. Further, because TRIDENT tracks memory error propagation in a fine-grained manner, it needs to collect detailed memory

traces. In a few cases, these traces are too big to fit into memory, and hence we cannot run TRIDENT at all. This motivates vTRIDENT, which does not need detailed memory traces, and is hence much faster.

### 6.4.2 vTRIDENT

As mentioned above, the majority of time spent in executing TRIDENT is in profiling and traversing memory dependencies of the program, which is the bottleneck in scalability. vTRIDENT extends TRIDENT by pruning any repeating memory dependencies from the profiling, and keeping only distinct memory dependencies for tracing error propagation. The intuition is that if we equally apply the same pruning to all inputs in each program, similar scales of losses in accuracy will be experienced across the inputs. Therefore, the relative SDC probabilities across inputs are preserved. Since volatility depends only on the relative SDC probabilities across inputs, the volatilities will also be preserved under pruning.

#### Workflow

Figure 6.3 shows the workflow of vTRIDENT. It is implemented as a set of LLVM compiler passes which take the code of the program (compiled into LLVM IR) and a set of inputs of the program. The output of vTRIDENT is the INSTRUCTION-SDC-VOLATILITY and INSTRUCTION-EXECUTION-VOLATILITY of the program across all the inputs provided, both at the aggregate level and per-instruction level. Based on Equation 6.2, OVERALL-SDC-VOLATILITY can be computed using INSTRUCTION-SDC-VOLATILITY and INSTRUCTION-EXECUTION-VOLATILITY.

vTRIDENT executes the program with each input provided, and records the differences of SDC probabilities predicted between inputs to generate INSTRUCTION-SDC-VOLATILITY. During each execution, the program's dynamic footprint is also recorded for the calculation of INSTRUCTION-EXECUTION-VOLATILITY. The entire process is fully automated and requires no intervention of the user. Further, no FIs are needed in any part of the process.

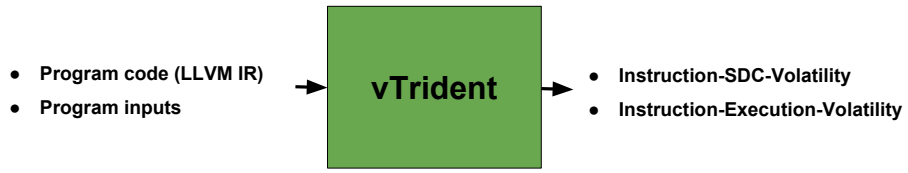



**Figure 6.3:** Workflow of vTRIDENT

### Example

We use an example from *Graph* in Figure 6.4A to illustrate the idea of vTRIDENT and its differences from TRIDENT. We make minor modifications for clarity and remove some irrelevant parts in the example. Although vTRIDENT works at the level of LLVM IR, we show the corresponding C code for clarity. We first explain how TRIDENT works for the example, and then explain the differences with vTRIDENT.

In Figure 6.4A, the C code consists of three functions, each of which contains a loop. In each loop, the same array is manipulated symmetrically in iterations of the loops and transferred between memory back and forth. So the load and store instructions in the loops (*LOOP 1, 2* and *3*) are all memory data-dependent. Therefore, if a fault contaminates any of them, it may propagate through the memory dependencies of the program. *init()* is called once at the beginning, then *Parcour()* and *Recher()* are invoked respectively in *LOOP 4* and *5*. *printf (INDEX 6)* at the end is the program's output. In the example, we assume *LOOP 4* and *5* execute two iterations each for simplicity. Therefore, the fault leads to an SDC if the fault propagates to the instruction.

To model error propagation via memory dependencies of the program, a similar memory dependency graph is created in Figure 6.5. Each node represents either a dynamic load or store instruction of which indices and loop positions of their static instructions are marked on their right. In the figure, each column of nodes indicates data-dependent executions of the instructions - there is no data flowing between columns as the array of data are manipulated by *LOOP 1, 2* and *3* symmetrically. In this case, TRIDENT finds the opportunity to prune the repeated columns of nodes to speed up its modeling time as error propagations are similar in the columns. The pruned columns are drawn with dashed border in the fig-

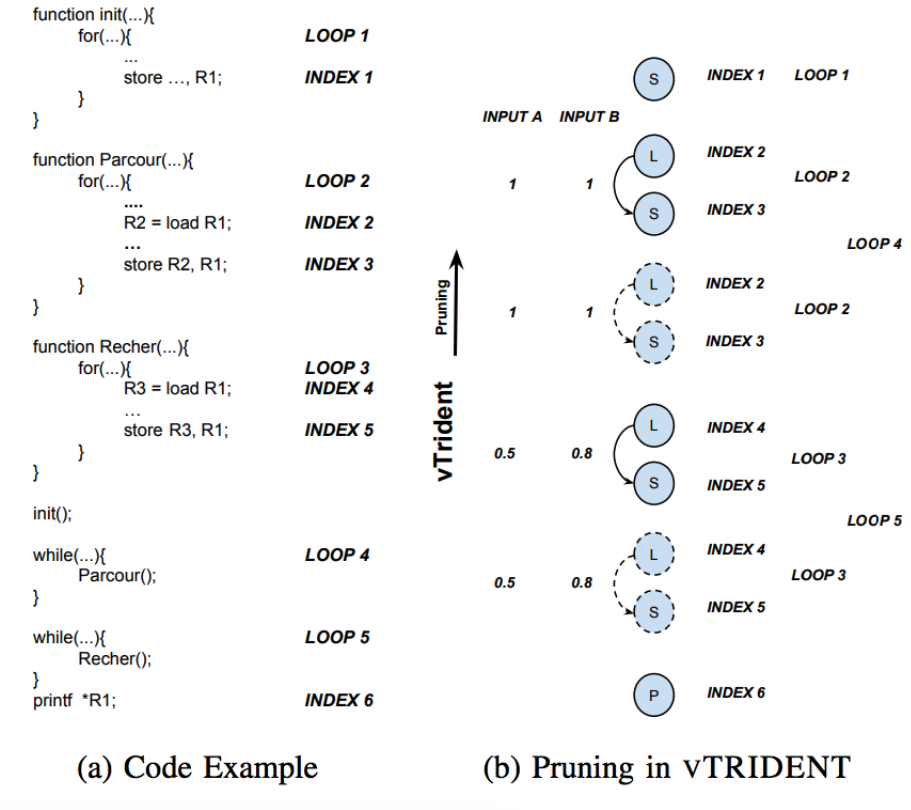



**Figure 6.4:** Example of Memory Pruning

ure, and they indicate the pruning of the inner-most loops. TRIDENT applies this optimization for memory-level modeling, resulting in significant acceleration compared with previous modeling techniques [59]. However, as mentioned, the graph can still take significant time to construct and process.

To address this issue, vTRIDENT further prunes memory dependency by tracking error propagations only in distinct dependencies to speed up the modeling. Figure 6.4B shows the idea: The graph shown in the figure is pruned to the one by TRIDENT in Figure 6.5. Arrows between nodes indicate propagation probabilities in the straight-line code. Because there could be instructions leading to crashes and error masking in straight-line code, the propagation probabilities are not 1. The propagation probabilities marked beside the arrows are aggregated

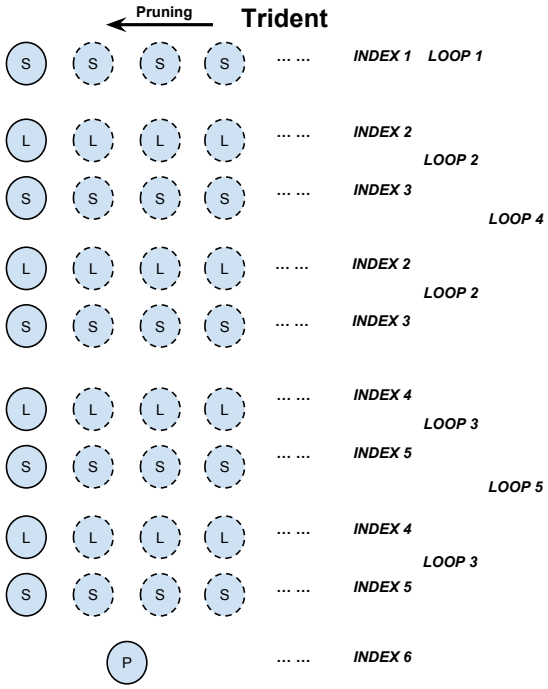



**Figure 6.5:** Memory Dependency Pruning in TRIDENT

to compute SDC probabilities for INPUT A and INPUT B respectively. For example, if a fault occurs at *INDEX 1*, the SDC probability for the fault to reach program output (*INDEX 6*) is calculated as $1*1*0.5*0.5 = 25\%$ for INPUT A, and $1*1*0.8*0.8 = 64\%$ for INPUT B. Thus, the variation of the SDC probability is 39% for these two inputs. vTRIDENT prunes the propagation by removing repeated dependencies (their nodes are drawn in dashed border in Figure 6.4B). The calculation of SDC probability for the fault that occurred at *INDEX 1* to *INDEX 6* becomes *1*0.5 = 50%* with INPUT A, and *1*0.8 = 80%* with INPUT B. The variation between the two inputs thus becomes 30%, which is 9% lower than that computed by TRIDENT (i.e., without any pruning).

We make two observations from the above discussion: (1) If the propagation probabilities are 1 or 0, the pruning does not result in loss of accuracy (e.g., *LOOP 4 in Figure 6.4B*). (2) The difference with and without pruning will be higher if the numbers of iterations become very large in the loops that contain non-1 or

non-0 propagation probabilities (i.e., *LOOP 5 in Figure 6.4B*). This is because more terms will be removed from the calculation by vTRIDENT. We find that about half (55.39%) of all faults propagating in the straight-line code have either *all* 1s or at least one 0 as the propagation probabilities, and thus there is no loss in accuracy for these faults. Further, the second case is rare because large iterations of aggregation on non-1 or non-0 numbers will result in an extremely small value of the overall SDC probability. This is not the case as the average SDC probability is 10.74% across benchmarks. Therefore, the pruning does not result in significant accuracy loss in vTRIDENT.

## 6.5 Evaluation of vTRIDENT

In this section, we evaluate the accuracy and performance of vTRIDENT in predicting INSTRUCTION-SDC-VOLATILITY across multiple inputs. We use the same benchmarks and experimental procedure as before in Section 6.3. The code of vTRIDENT can be found in our GitHub repository.[2]

### 6.5.1 Accuracy

To evaluate the ability of vTRIDENT in identifying INSTRUCTION-SDC-VOLATILITY, we first classify all the instructions based on their INSTRUCTION-SDC-VOLATILITY derived by FI and show their distributions – this serves as the ground truth. We classify the differences of the SDC probabilities of each measured instruction between inputs into three categories based on their ranges of variance ($<$10%, 10%$-$20% and $>$20%), and calculate their distribution based on their dynamic footprints. The results are shown in Figure 6.6. As can be seen in the figure, on average, only 3.53% of instructions across benchmarks exhibit variance of more than 20% in the SDC probabilities. Another 3.51% exhibit a variance between 10% and 20%. The remaining 92.93% of the instructions exhibit within 10% variance across inputs.

We then use vTRIDENT to predict the INSTRUCTION-SDC-VOLATILITY for each instruction, and then compare the predictions with ground truth. These results are also shown in Figure 6.6. As can be seen, for instructions that have INSTRUCTION-SDC-VOLATILITY less than 10%, vTRIDENT gives relatively

[2] https://github.com/DependableSystemsLab/Trident

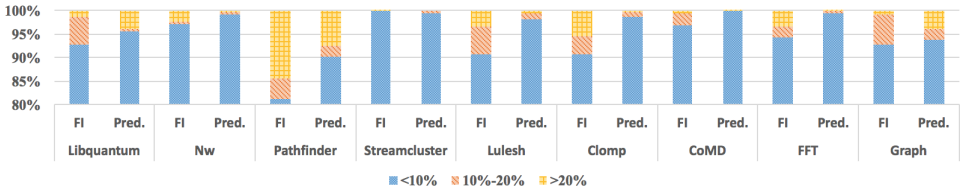



**Figure 6.6:** Distribution of INSTRUCTION-SDC-VOLATILITY predictions by vTrident Versus Fault Injection Results (Y-axis: Percentage of instructions, Error Bar: 0.03% to 0.55% at 95% Confidence)

accurate predictions across benchmarks. On average, 97.11% of the instructions are predicted to fall into this category by VTRIDENT, whereas FI measures it as 92.93%. Since these constitute the vast majority of instructions, VTRIDENT has high accuracy overall.

On the other hand, instructions that have INSTRUCTION-SDC-VOLATILITY of more than 20% are significantly underestimated by VTRIDENT, as VTRIDENT predicts the proportion of such instructions as 1.84% whereas FI measures it as 3.53% (which is almost 2x more). With that said, for individual benchmarks, VTRIDENT is able to distinguish the sensitivities of INSTRUCTION-SDC-VOLATILITY in most of them. For example, in *Pathfinder* which has the largest proportion of instructions that have INSTRUCTION-SDC-VOLATILITY greater than 20%, VTRIDENT is able to accurately identify that this benchmark has the highest proportion of such instructions relative to the other programs. However, we find VTRIDENT is not able to well identify the variations that are greater than 20% as mentioned above. This case can be found in *Nw, Lulesh, Clomp and FFT*. We discuss the sources of inaccuracy in Section 6.7.1. Since these instructions are relatively few in terms of dynamic instructions in the programs, this underprediction does not significantly affect the accuracy of VTRIDENT.

We then measure the overall *accuracy* of VTRIDENT in identifying INSTRUCTION-SDC-VOLATILITY. The accuracy is defined as the number of correctly predicted variation categories of instructions over the total number of instructions being predicted. We show the accuracy of VTRIDENT in Figure 6.7. As can be seen, the highest accuracy is achieved in Streamcluster (99.17%), while the lowest accuracy is achieved in Clomp (67.55%). The average accuracy across nine benchmarks is

87.81%, indicating that vTRIDENT is able to identify most of the INSTRUCTION-SDC-VOLATILITY.

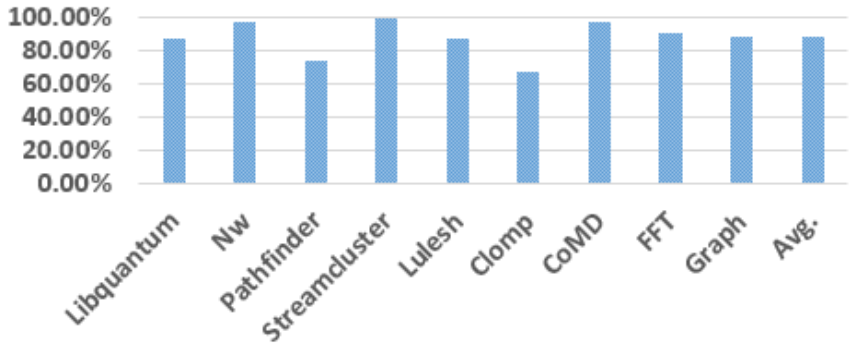



**Figure 6.7:** Accuracy of vTRIDENT in Predicting INSTRUCTION-SDC-VOLATILITY Versus FI (Y-axis: Accuracy)

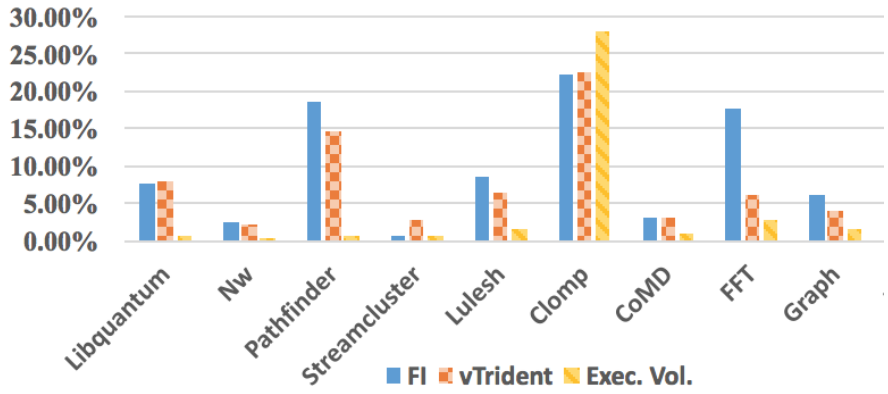



**Figure 6.8:** OVERALL-SDC-VOLATILITY Measured by FI and Predicted by vTRIDENT, and INSTRUCTION-EXECUTION-VOLATILITY alone (Y-axis: OVERALL-SDC-VOLATILITY, Error Bar: 0.03% to 0.55% at 95% Confidence)

Finally, we show the accuracy of predicting OVERALL-SDC-VOLATILITY using vTRIDENT, and using INSTRUCTION-EXECUTION-VOLATILITY alone (as before) in Figure 6.8. As can be seen, the average difference between vTRIDENT and FI is only 1.26x. Recall that the prediction using INSTRUCTION-EXECUTION-VOLATILITY alone (*Exec. Vol.*) gives an average difference of 7.65x (Section 6.3).

The worst case difference when considering only *Exec. Vol.* was 24.54x, while it is 1.29x (in *Pathfinder*) when INSTRUCTION-SDC-VOLATILITY is taken into account. Similar trends are observed in all other benchmarks. This indicates that the accuracy of OVERALL-SDC-VOLATILITY prediction is significantly higher when considering both INSTRUCTION-SDC-VOLATILITY and INSTRUCTION-EXECUTION-VOLATILITY rather than just using INSTRUCTION-EXECUTION-VOLATILITY.

### 6.5.2 Performance

We evaluate the performance of VTRIDENT based on its execution time, and compare it with that of TRIDENT. We do not consider FI in this comparison as FI is orders of magnitude slower than TRIDENT [59]. We measure the time taken by executing VTRIDENT and TRIDENT in each benchmark, and compare the speedup achieved by VTRIDENT over TRIDENT. The total computation is proportional to both the time and power required to run each approach. Parallelization will reduce the time spent, but not the power consumed. We assume that there is no parallelization for the purpose of comparison in the case of TRIDENT and VTRIDENT, though both TRIDENT and VTRIDENT can be parallelized. Therefore, the speedup can be computed by measuring their wall-clock time.

We also measure the time per input as both TRIDENT and VTRIDENT experience similar slowdowns as the number of inputs increase (we confirmed this experimentally). The average execution time of VTRIDENT is 944 seconds per benchmark per input (a little more than 15 minutes). Again, we emphasize that this is due to the considerably large input sizes we have considered in this study (Section 6.3).

The results of the speedup by VTRIDENT over TRIDENT are shown in Figure 6.9. We find that on average VTRIDENT is 8.05x faster than TRIDENT. The speedup in individual cases varies from 1.09x in *Graph* (85.16 seconds versus. 78.38 seconds) to 33.56x in *Streamcluster* (3960 seconds versus. 118 seconds). The variation in speedup is because applications have different degrees of memory-boundedness: the more memory bounded an application is, the slower it is with TRIDENT, and hence the larger the speedup obtained by VTRIDENT (as it does not need detailed memory dependency traces). For example, *Streamcluster* is

more memory-bound than computation-bound than *Graph*, and hence experiences much higher speedups.

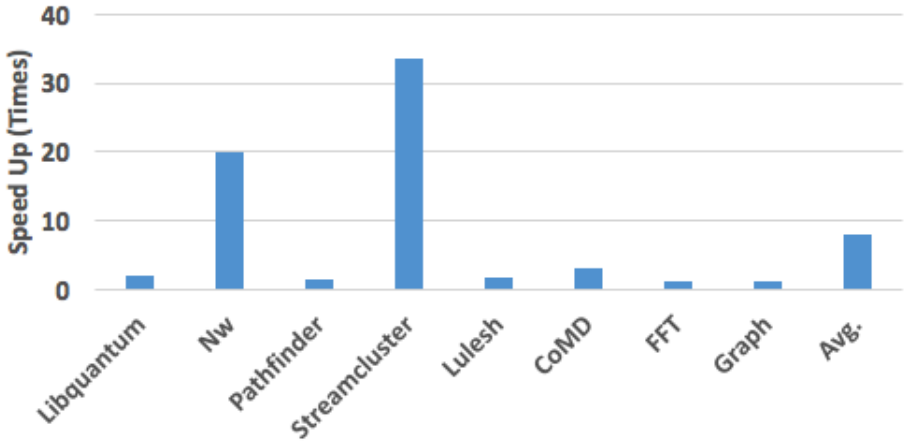



**Figure 6.9:** Speedup Achieved by vTRIDENT over TRIDENT. Higher numbers are better.

Note that we omit *Clomp* from the comparison since *Clomp* consumes more than 32GB memory in TRIDENT, and hence crashes on our machine. This is because *Clomp* generates a huge memory-dependency trace in TRIDENT, which exceeds the memory of our 32GB-memory machine (in reality, it experiences significant slowdown due to thrashing, and is terminated by the OS after a long time). On the other hand, vTRIDENT prunes the memory dependency and incurs only 21.29MB memory overhead when processing *Clomp*.

## 6.6 Bounding Overall SDC Probabilities with vTRIDENT

In this section, we describe how to use vTRIDENT to bound the overall SDC probabilities of programs across given inputs by performing FI with only one selected input. We need FI because the goal of vTRIDENT is to predict the variation in SDC probabilities, rather than the absolute SDC probability which is much more time-consuming to predict (Section 6.4.2). Therefore, FI gives us the absolute SDC probability for a given input. However, we only need to perform FI on

a single input to bound the SDC probabilities of any number of given inputs using vTRIDENT, which is a significant savings as FI tends to be very time-consuming to get statistically significant results.

For a given benchmark, we first use vTRIDENT to predict the OVERALL-SDC-VOLATILITY across all given inputs. Recall that OVERALL-SDC-VOLATILITY is the difference between the highest and the lowest overall SDC probabilities of the program across its inputs. We denote this range by $R$. We then use vTRIDENT to find the input that results in the median of the overall SDC probabilities predicted among all the given inputs. This is because we need to locate the center of the range in order to know the absolute values of the bounds. Using inputs other than the median will result in a shifting of the reference position, but will not change the boundaries being identified, which are more important. Although vTRIDENT loses some accuracy in predicting SDC probabilities as we mentioned earlier, most of the rankings of the predictions are preserved by vTRIDENT. Finally, we perform FI on the selected input to measure the true SDC probability of the program, denoted by $S$. Note that it is possible to use other methods for this estimation (e.g., TRIDENT [59]). The estimated lower and upper bounds of the overall SDC probability of the program across all its given inputs is derived based on the median SDC probability measured by FI, as shown below.

$$[(S-R/2),(S+R/2)] \tag{6.3}$$

We bound the SDC probability of each program across its inputs using the above method. We also use INSTRUCTION-EXECUTION-VOLATILITY alone for the bounding as a point of comparison. The results are shown in Figure 6.10. In the figure, the triangles indicate the overall SDC probabilities with the ten inputs of each benchmark measured by FI. The overall SDC probability variations range from 1.54x (*Graph*) to 42.01x (*Lulesh*) across different inputs. The solid lines in the figure bound the overall SDC probabilities predicted by vTRIDENT. The dashed lines bound the overall SDC probabilities projected by considering only the INSTRUCTION-EXECUTION-VOLATILITY.

*On average, 78.89% of the overall SDC probabilities of the inputs measured by FI are within the bounds predicted by* vTRIDENT. *For the inputs that are outside*

*the bounds, almost all of them are very close to the bounds.* The worst case is *FFT*, where the overall SDC probabilities of two inputs are far above the upper bounds predicted by vTRIDENT. The best cases are *Streamcluster* and *CoMD* where almost every input's SDC probability falls within the bounds predicted by vTRIDENT (Section 6.7.1 explains why).

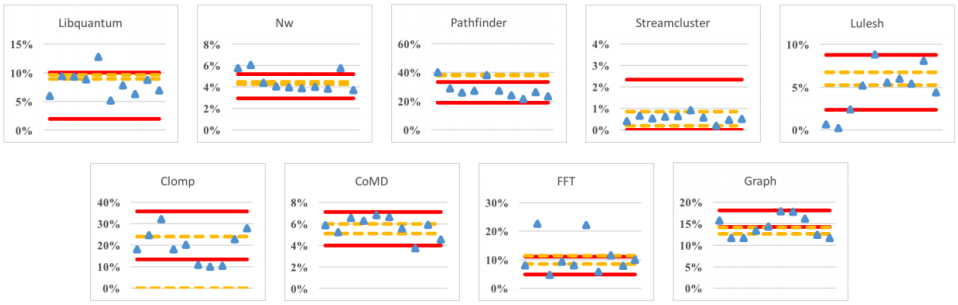



**Figure 6.10:** Bounds of the Overall SDC Probabilities of Programs (Y-axis: SDC Probability; X-axis: Program Input; Solid Lines: Bounds derived by vTRIDENT; Dashed Lines: Bounds derived by INSTRUCTION-EXECUTION-VOLATILITY alone, Error Bars: 0.03% to 0.55% at the 95% Confidence). Triangles represent FI results.

On the other hand, INSTRUCTION-EXECUTION-VOLATILITY alone bounds only 32.22% SDC probabilities on average. This is a sharp decrease in the coverage of the bounds compared with vTRIDENT, indicating the importance of considering INSTRUCTION-SDC-VOLATILITY when bounding overall SDC probabilities. The only exception is Streamcluster where considering INSTRUCTION-EXECUTION-VOLATILITY alone is sufficient in bounding SDC probabilities. This is because Streamcluster exhibits very little SDC volatility across inputs (Figure 6.6).

In addition to coverage, tight bounds are an important requirement, as a loose bounding (i.e., a large $R$ in Equation 6.3) trivially increases the coverage of the bounding. To investigate the tightness of the bounding, we examine the results shown in Figure 6.8. Recall that OVERALL-SDC-VOLATILITY is represented by $R$, so the figure shows the accuracy of $R$. As we can see, vTRIDENT computes bounds that are comparable to the ones derived by *FI* (ground truth), indicating

that the bounds obtained are tight.

## 6.7 Discussion

In this section, we first summarize the sources of inaccuracy in vTRIDENT, and then we discuss the implications of vTRIDENT for error mitigation techniques.

### 6.7.1 Sources of Inaccuracy

Other than the loss of accuracy from the coarse-grain tracking in memory dependency (Section 6.4.2), we identify three potential sources of inaccuracy in identifying INSTRUCTION-SDC-VOLATILITY by vTRIDENT. They are also the sources of inaccuracy in TRIDENT, which vTRIDENT is based on. We explain how they affect identifying INSTRUCTION-SDC-VOLATILITY here.

*Source 1: Manipulation of Corrupted Bits*

We assume only instructions such as comparisons, logical operators and casts have masking effects, and that none of the other instructions mask the corrupted bits. However, this is not always the case as other instructions may also cause masking. For example, repeated division operations such as *fdiv* may also average out corrupted bits in the mantissa of floating point numbers, and hence mask errors. The dynamic footprints of such instructions may be different across inputs hence causing them to have different masking probabilities, so vTRIDENT does not capture the volatility from such cases. For instance, in *Lulesh*, we observe that the number of *fdiv* may differ by as much as 9.5x between inputs.

*Source 2: Memory Copy*

vTRIDENT does not handle bulk memory operations such as memmove and memcpy. Hence, we may lose track of error propagation in the memory dependencies built via such operations. Since different inputs may diversify memory dependencies, the diversified dependencies via the bulk memory operations may not be identified either. Therefore, vTRIDENT may not be able to identify INSTRUCTION-SDC-VOLATILITY in these cases.

*Source 3: Conservatism in Determining Memory Corruption*

We assume all the store instructions that are dominated by the faulty branch are corrupted when control-flow is corrupted, similar to the examples in Figure 6.2B

and Figure 6.2C. This is a conservative assumption, as some stores may end up being coincidentally correct. For example, if a store instruction is supposed to write a zero to its memory location, but is not executed due to the faulty branch, the location will still be correct if there was a zero already in that location. These are called *lucky loads* in prior work [42]. When inputs change, the number of *lucky loads* may also change due to the changes of the distributions of such zeros in memory, possibly causing volatility in SDC. vTRIDENT does not identify *lucky loads*, so it may not capture the volatility from such occasions.

### 6.7.2 Implication for Mitigation Techniques

Selective instruction duplication is an emerging mitigation technique that provides configurable fault coverage based on performance overhead budget [52, 88, 89, 97]. The idea is to protect only the most SDC-prone instructions in a program so as to achieve high fault coverage while bounding performance overheads. The problem setting is as follows: Given a certain performance overhead $C$, what static instructions should be duplicated in order to maximize the coverage for SDCs, $F$, while keeping the performance overhead below $C$. Solving the above problem involves finding two factors: (1) $P_i$: The SDC probability of each instruction in the program, to decide which set of instructions should be duplicated, and (2) $O_i$: The performance overhead incurred by duplicating the instructions. Then the problem can be formulated as a classical 0-1 knapsack problem [99], where the objects are the instructions and the knapsack capacity is represented by $C$, the maximum allowable performance overhead. Further, object profits are represented by the estimated SDC probability (and hence selecting the instruction means obtaining the coverage $F$), and object costs are represented by the performance overhead of duplicating the instructions.

Almost all prior work investigating selective duplication confines their study to a single input of each program in evaluating $P_i$ and $O_i$ [52, 88, 89, 97]. Hence, the protection is only optimal with respect to the input used in the evaluation. Because of the INSTRUCTION-SDC-VOLATILITY and INSTRUCTION-EXECUTION-VOLATILITY incurred when the protected program executes with different inputs, there is no guarantee on the fault coverage $F$ the protection aims to provide, com-

promising the effectiveness of the selective duplication. To address this issue, we argue that the selective duplication should take both INSTRUCTION-SDC-VOLATILITY and INSTRUCTION-EXECUTION-VOLATILITY into consideration. One way to do this is solving the knapsack problem based on the average cases of each $P_i$ and $O_i$ across inputs, so that the protection outcomes, $C$ and $F$, are optimal with respect to the average case of the executions with the inputs. This is a subject of future work.

## 6.8 Summary

Programs can experience Silent Data Corruptions (SDCs) due to soft errors, and hence we need fault injection (FI) to evaluate the resilience of programs to SDCs. Unfortunately, most FI studies only evaluate a program's resilience under a single input or a small set of inputs as FI is very time consuming. In practice however, programs can exhibit significant variations in SDC probabilities under different inputs, which can make the FI results inaccurate.

In this chapter, we investigate the root causes of variations in SDCs under different inputs, and we find that they can occur due to differences in the execution of instructions as well as differences in error propagation. Most prior work has only considered the former factor, which leads to significant inaccuracies in their estimations. We propose a model vTRIDENT to incorporate differences in both execution and error propagation across inputs. We find that vTRIDENT is able to obtain achieve higher accuracy and closer bounds on the variation of SDC probabilities of programs across inputs compared to prior work that only consider the differences in execution of instructions. We also find vTRIDENT is significantly faster than other state of the art approaches for modeling error propagation in programs, and is able to obtain relatively tight bounds on SDC probabilities of programs across multiple inputs, while performing FI with only a single program input.

# Chapter 7

# Understanding Error Propagation in GPGPU Applications

Previous chapters have discussed error propagation in CPU programs. In this chapter, we aim to develop techniques for analyzing error propagation in GPU programs in order to improve their error resilience. Unlike in CPU programs where a variety of tools are available for fault injections, few options can be found in studying program-level error propagation in GPU programs. To overcome the difficulties, we first design a LLVM-based fault injector, LLFI-GPU, which is the first fault injector operating on the LLVM Intermediate Representation (IR) of GPU programs. By injecting faults at the IR level, LLFI-GPU allows the user to gain program-level insights of error propagation. We desmonstrate that LLFI-GPU can be used to conduct fault injection experiments and study error propagation in GPU programs. In the experiments, we observe the error propagation patterns that specific to GPU programs and discuss their implications of improving error resilience.

## 7.1 Introduction

Graphic Processing Units (GPUs) have found wide adoption as accelerators for scientific and high-performance computing (HPC) applications due to their mass

availability and low cost. For example, two of the largest supercomputing clusters in use today, namely Blue-Waters [47] and Titan [144] both use GPUs. GPUs were originally designed for graphics and gaming. However, their use in HPC applications has necessitated the systematic study of their reliability. This is because unlike graphics or gaming applications which are error-tolerant, HPC applications have strict correctness requirements and even a single error can lead to significant deviations in their outcomes. The problem is exacerbated by the lack of standard error detection and correction mechanisms for GPUs, compared to CPUs (Central Processing Units, or the main processor). Recent studies of GPU reliability in the HPC context have found that GPUs can experience significantly higher fault rates compared to CPUs [47, 144], and that GPU applications often experience higher rates of Silent Data Corruption (SDCs), i.e., incorrect outcomes, compared to CPU applications [50, 63].

HPC applications typically run for long periods of time, and hence need to be resilient to faults [114]. Further, in supercomputers, hardware faults have become more and more prevalent due to the shrinking feature sizes and power constraints [55]. One of the most common hardware fault types are transient faults [20, 40], which arise due to cosmic rays or electro-magnetic radiation striking computational and/or memory elements, causing the values computed or stored to be incorrect. To mitigate the effect of transient hardware faults, HPC applications use techniques such as checkpointing and recovery. However, these techniques make an important assumption, namely that faults do not propagate for long periods of time and corrupt the checkpointed state as this would make the checkpoint unrecoverable [58, 88, 150]. Unfortunately, this assumption does not always hold as errors often propagate in real applications [89]. More importantly, unmitigated error propagation can also lead to SDCs, which seriously compromise the applications' correctness.

In this chapter, we investigate the error propagation characteristics of general-purpose GPU (GPGPU) applications with the goal of trying to mitigate error propagation. Prior work has investigated error propagation in CPU applications [11, 88], and has developed techniques to mitigate error propagation based on their results [89]. However, it is not clear how applicable are these results to GPGPU applications, which have a very different programming model. Other work has in-

vestigated the aggregate error resilience of GPGPU applications [50, 63]. While these are valuable, they do not provide insights into how errors propagate within the GPGPU application. Such insights are necessary for driving the design of low-overhead error detection mechanisms for these applications, which is our long-term goal. Such mechanisms have been demonstrated in the CPU space [11, 89]. *To the best of our knowledge, this is the first study of error propagation in GPGPU applications*.

There are two main challenges with performing studies of error propagation on GPGPU applications. First, because GPGPU applications execute on both the CPU and the GPU (as current GPUs do not provide many of the capabilities needed by applications), it is important to track error propagation across the two entities. Further, GPUs are often invoked multiple times in an application (each such invocation is known as a kernel call), so one needs to track error propagation across these invocations. Second, unlike in the CPU space where there are freely available fault injection tools and frameworks to study error propagation [25, 69, 127, 147], there is a paucity of such tools in the GPU space.

We address the first challenge by defining the kernel call as the unit of error propagation, and study error propagation both within kernel calls and across multiple calls. We address the second challenge by building a robust LLVM-based fault injection tool for GPGPU applications. LLVM is a widely-used optimizing compiler [85], and our fault injection tool is written as a module in the LLVM framework. As a result, we are able to leverage the program analysis capabilities provided by LLVM to track error propagation in programs, and correlate it with the program's code.

We make the following contributions in this paper.

- Develop LLFI-GPU, a GPGPU fault injection tool that can operate on the LLVM intermediate representation (IR) of a program and track error propagation in GPGPU programs.
- Define the metrics for tracking and measuring error propagation in GPGPU programs,
- Conduct a comprehensive fault-injection study on how errors propagate in

twelve GPGPU applications (including both benchmarks and real-world applications), and how long and how fast such errors propagate and spread in the application,

- Discuss how the results may be leveraged by dependability techniques to provide targeted mitigation of error propagation for GPGPU applications at low cost.

Our main results from the fault-injection study are:

- Only a small fraction of the crash-causing faults that occur in GPUs propagate to the CPU, and only a minuscule fraction of crash-causing faults propagate to other GPU kernels. Thus, it is sufficient to consider checkpointing and recovery techniques at the GPU-CPU boundary.

- Errors do propagate to multiple memory locations, but this behavior is highly application specific. For example, a single fault can contaminate anywhere between 0.0006% locations to more than 60% of total memory locations, depending on the application.

- Unlike CPU programs, most of the memory corruptions in GPU programs lead to data corruptions in program output. Faults in memory that propagate to output data likely do so within the kernel where faults occurred. This allows error detection techniques to operate at the granularity of the GPU kernel call.

- More than 50% of the faults that occur in the GPU are masked within a single kernel execution. Thus, it may be counterproductive to deploy techniques such as Dual Modular Redundancy (DMR) or Error Detection by Duplicated Instructions (EDDI) [106] within the GPU kernel program as they will end up detecting many faults that are eventually masked.

## 7.2 GPU Fault Injector

We build a fault injector for GPUs based on the open-source LLFI fault injector [147] , which has been extensively used for error propagation studies on the

CPU [11, 88]. However, LLFI does not inherently support GPUs. Furthermore, performing error propagation analysis (EPA) for GPUs is much more intricate than on CPUs. We therefore extended LLFI to perform both fault injection and EPA on GPUs. We refer to the extended version of LLFI as LLFI-GPU[1] to distinguish it from the existing LLFI infrastructure.

Fault injection can be done at different levels of the system such as at the gate-level, circuit-level, architecture level and application level. Prior work [38] has found that there may be significant differences in the raw rates of faults exposed to the software layer when fault injections are performed in the hardware. However, we are interested in faults that are not masked by the hardware and make their way to the application. Therefore, we inject faults directly at the application level.

### 7.2.1 Design Overview

LLFI is a compiler-based fault injection framework, and uses the LLVM compiler to instrument the program to inject faults. The CUDA Nvidia compiler NVCC is also based on LLVM, and compiles LLVM IR to a PTX representation, which then gets compiled to the SAS machine code by Nvidia's backend compiler. So at first glance, it seems trivial to integrate LLFI and NVCC to build a GPU-based fault injector. However, there are two challenges that arise in practice. First, NVCC does not expose the LLVM IR code and directly transforms it to the PTX code. LLFI relies on the IR code to perform instrumentation for fault injection, and hence cannot inject faults into the IR used by NVCC. Second, GPU programs are multi-threaded, often consisting of hundreds of threads, and hence we need to inject faults into a random thread at runtime. However, LLFI does not support injecting faults into multi-threaded programs.

We address the first challenge by attaching a dynamic library to NVCC which can intercept its call to the LLVM compilation module [5]. At that point, we invoke the instrumentation passes of LLFI to perform the instrumentation of the program. We then return the instrumented LLVM IR to NVCC, which proceeds with the rest of the compilation process to transform it to PTX code. We address the second challenge by adding a *threadID* field to the profiling data collected by LLFI to

[1] Available at: *https://github.com/DependableSystemsLab/LLFI-GPU*

identify each thread uniquely. We then choose a thread at random to inject into at runtime from the set of all threads in the program. We also add information on the kernel call executed and the total number of kernel calls to the profiling data. These are used to choose kernel calls to inject faults into.

LLFI-GPU works as follows. First, LLFI-GPU profiles the program and obtains the total number of kernel calls, the number of threads per kernel call, and the total number of instructions executed by each kernel thread. It then creates an instrumented version of the program with the fault injection functions inserted into the CUDA portion of the program's code (this is similar to what LLFI does, except that we restrict the instrumentation to the CUDA portion of the program). LLFI-GPU then chooses a random thread in a random kernel call, and a random dynamic instruction executed by it, based on the profiling data gathered (the instruction is chosen uniformly from the set of all instructions executed). For the chosen instruction, LLFI-GPU overwrites the result value of the instruction with a faulty version of the result (e.g., by flipping a single bit in it), and continues the application.

Thus, LLFI-GPU directly executes the program on the GPU hardware after instrumenting it, unlike prior approaches such as GPU-Qin [50] which use debuggers for fault injection. Debugger-based fault injection has the advantage that it offers more control over the program, but is often significantly slower. As a point of comparison, we ran both GPU-Qin[2] and LLFI-GPU on a simple matrix multiplication benchmark, MAT from NVIDIA SDK Sample [4]. Similar to LLFI-GPU, GPU-Qin operates in two phases: profiling and fault injection. We measured the average time taken by GPU-Qin for these two phases to be 2 hours (=7200 seconds), and 82 seconds per run respectively. In contrast, LLFI-GPU takes only 6 seconds for profiling this benchmark and 2 seconds per run for the fault injection phase for the same set of inputs. The significant speedup of over 1000x obtained by LLFI-GPU is because GPUs execute significantly slower in debug mode [2], and GPU-Qin single steps through every instruction in the program in the profiling phase. We have confirmed that the above execution times are fairly typical depending on the number of dynamic instructions of the programs executed using GPU-Qin[3], and hence

[2]The only other GPU fault injector that we know of, SASSIFI [63], was not publicly available at the time the paper was written, and hence could not be used for comparison.

[3]Based on personal communication with the developers of GPU-Qin.

we did not run any of the other benchmarks with GPU-Qin. Therefore, LLFI-GPU is significantly faster and more scalable than prior techniques, making it feasible for studying realistic HPC workloads.

### 7.2.2 Error Propagation Analysis (EPA)

After injecting a fault, LLFI-GPU tracks memory data at every kernel boundary for the analysis of error propagation. This is because we are interested in error propagation at the GPU kernel boundary, rather than within a kernel. This is different from what LLFI does as it tracks the error propagation using memory and registers after each LLVM instruction. Although we can leverage the existing EPA mechanism in LLFI for tracking error propagation at the kernel boundaries, we found that this incurs very high overheads, and often results in the kernel running substantially slower. Therefore, we decided to build our own error tracking mechanism in LLFI-GPU that is optimized for our use case.

Figure 7.1 illustrates how the EPA mechanism works for a simple GPU kernel. The code fragment is from *bfs*. It allocates memory on device through *cudaMalloc()* before launching kernels, and it deallocates the memory on device through *cudaFree()* at the end of the program. After each kernel invocation, LLFI-GPU saves all memory data allocated on the GPU to disk. This step corresponds to line 6-13. Later, we compare the saved data after each kernel call with that from a golden run and mark any differences as a result of the error propagation. Because we perform this comparison at the kernel boundaries, we do not need to worry about non-determinism introduced by thread interleaving within the GPU.

### 7.2.3 Limitations

Our fault injections are performed at the LLVM IR level rather than at the SASS or PTX code levels. One potential drawback of this approach is that downstream compiler optimizations may change both the number and order of instructions, or even remove the fault injection code we inserted. To mitigate this effect, we made sure that our fault injection pass is applied after various optimization passes in the LLVM IR code. Further, LLFI-GPU gathers all executed instructions in the profiling phase as described above, and we made sure that all the target instructions as

```
1  cudaMalloc(gpu_graph_visited, int*512);
2  cudaMalloc(gpu_graph_edge, int*512);
3  ...
4  kernel1()
5  // Track device data
6  foreach(gpu_graph_visited){
7    // Save to disk
8    dump(each_visited);
9  }
10 foreach(gpu_graph_edge){
11   // Save to disk
12   dump(each_edge);
13 }
14 ...
15 kernel2()
16 // Track device data agin, same as above.
17 ...
18 cudaFree(gpu_graph_visited);
19 cudaFree(gpu_graph_edge);
```

**Figure 7.1:** Example of the error propagation code inserted by LLFI-GPU

fault injection candidates are gathered and injects faults only into these instructions - this ensures that faults are not injected into dead instructions that are optimized out by the backend compiler. Due to backend optimizations after the IR is generated, the mapping of instructions may be changed at the machine assembly levels (e.g., SASS level). This may result in different absolute values of the SDC rate for fault injections performed at different levels. However, as we said before, we are interested in obtaining insights into error propagation intrinsic to applications instead of deriving derated SDC rates. Finally, a previous study on CPU applications showed that there is negligible difference in SDC rates between fault injections performed at the LLVM IR level and the assembly code level [147].

## 7.3 Metrics for Error Propagation

In this section, we define the metrics for measuring error propagation in our experiments. We measure error propagations along two axes, namely (1) execution time, which captures the temporal nature of the propagation, and (2) memory states, which captures the spatial nature of the propagation. We examine these in further detail below.

### 7.3.1 Execution time

A fault can propagate in the program corrupting data values until it either causes program termination (e.g., by a crash), or it is masked. The former happens if the fault crashes the program, or the program finishes execution successfully (program hangs are handled through a watchdog timer). The latter happens if the faulty data is overwritten, or if the values to which the fault propagates are discarded by the program. The execution time metric measures the time between the fault's occurrence and the masking or termination events.

We use kernel invocations to measure the propagation time of an error. For example, if an error occurs in a certain kernel invocation K1, and the program crashes after two more invocations of the kernel, say K2 and K3, we label the execution time of this fault to be 2. There are three reasons for using kernel invocations as the unit of propagation. First, we are often interested in knowing if an error propagates across the CPU-GPU boundary or across multiple kernel invocations on the GPU. The number of kernel invocation captures this value. The second reason is that unlike other metrics such as wall-clock time (e.g., seconds), or the number of executed instructions which are platform dependent, the number of kernel invocations depends only on the GPU application. Thus, it captures the application-level semantics of error propagation without being affected by platform-specific details. This is important as our goal is to design application level error-resilience mechanisms. Finally, unlike CPU applications which have a few long-living threads, GPU applications typically have a large number of short-lived threads executing on the GPU as they focus on throughput. When these threads terminate, the control is passed back to the CPU, thereby resulting in frequent CPU-GPU boundary crossings. We found that the average kernel invocation time in our benchmarks is usually less than one minute - this is in line with prior work [138].

In addition to kernel invocations, GPU applications also need to transfer data from CPU memory to GPU memory and back. Typical GPU applications perform multiple kernel invocations in between transfers to amortize the latency of transfers. We define a *kernel cycle* as a sequence of kernel invocations by the application that is prefixed by memory transfer from the CPU to the GPU, and suffixed by memory transfer from the GPU back to the CPU. All applications in our study

```
1  // Allocate for all being used in device
2  cudaMalloc(); ...
3  // Total Memory
4  cudaMemcpy(gpu_graph_nodes, cpu_graph_nodes);
5  // Total Memory
6  cudaMemcpy(gpu_graph_visited, cpu_graph_visited);
7  // Total Memory
8  cudaMemcpy(gpu_graph_edges, cpu_graph_edges);
9  // Result Memory
10 cudaMemcpy(gpu_result_cost, cpu_result_cost);
11 ...
12 // kernel invocations
13 kernel <<<...>>>, kernel2 <<<...>>>, ... ...
14 ...
15 // Result Memory
16 cudaMemcpy(cpu_graph_nodes, gpu_graph_nodes);
17 // Output Memory
18 foreach(cpu_result_cost):
19 {if(cost<0} dumpCostForResult();}
20 // Deallocate memory used in device
21 cudaFree() ...
```

**Figure 7.2:** Code Example of a Kernel Cycle from Benchmark *bfs*

except LULESH and NMF had only a single kernel cycle. However, all of them have multiple kernel invocations within a single kernel cycle.

Figure 7.2 shows an example of a kernel cycle in the *bfs* application. The first phase of the kernel cycle (lines 1-10) consists of memory allocations, and data movement from the host (CPU) to the device (GPU) memory. The second phase consists of kernel invocations (line 13), which perform computations on the data copied to the GPU memory. Finally, in the third phase, after the kernels finish their work, the CPU collects data from the GPU and processes it (lines 15-20).

### 7.3.2 Memory States

To better understand error propagation, we examine which parts of a GPU program's memory have been affected by an error after fault injection. We divide memory into three categories, namely Total Memory(TM), Result Memory(RM) and Output Memory(OM). TM is a superset of RM, which in turn is a superset of OM. This is shown in Figure 7.3. TM refers to the entire memory space allocated for the program on device. The allocations are usually done through *cudaMalloc* calls, and through global variables declared by kernels. RM refers to the memory

locations containing the computation results that the CPU transfers from the GPU at the end of a kernel cycle. OM refers to the memory locations containing the data that the CPU actually processes for computing the program output.

In the example in Figure 7.2, the transfer occurs at line 16. So *gpu_result_cost* is the pointer to the RM in this example. Further, the processing phase occurs in lines 18-19. So the OM consists of the results of the *dumpCostForResult* function. Note that applications may choose only certain parts of the RM to copy into their output. For example, a floating point application may use only the two most significant digits from the result in RM to compute the output, in which case the OM consists of only these two digits.

We use SDC to refer to corruption of the above three categories of memory, as all of these pertain to data corruptions. We use the memory type as a subscript to denote corruptions of different memory categories, e.g., $SDC_{RM}$. Note that $SDC_{OM}$ is what is typically defined as an SDC in prior work [50, 63, 151] on GPUs, as they only study the effect of faults on the final output of the application. However, our aim is to study error propagation and hence we study data corruption in the memory states of the application. We also refer to corruption of the memory that is in TM but not RM as (TM-RM), and that in RM but not OM as (RM-OM).

For example, in Figure 7.3, assume that an error occurs during a kernel invocation (K1) and affects the memory location (L1) in the TM. However, it does not propagate to the RM. In the next kernel invocation (K2), the faulty value in L1 is read and affects a value at another location L2 in RM. Hence we say the error propagates from the TM to RM during K2. In this case, the error causes an $SDC_{TM}$ after kernel K1, and an $SDC_{RM}$ after kernel K2. However, the error does not propagate to the OM, and hence does not result in an $SDC_{OM}$.

We also measure what fraction of the memory is contaminated by error propagation. We define this as the *spread* of an error. For example, in Figure 7.3, at K3, the faulty value in location L2 is assigned to different memory locations (L3, L4 and L5), and hence propagates to these locations. In this case, the fault value has propagated to a total of 5 locations at the end of K3. Assuming TM consists of 100 memory locations, the error spread after K3 in this case is 5/100=5%.

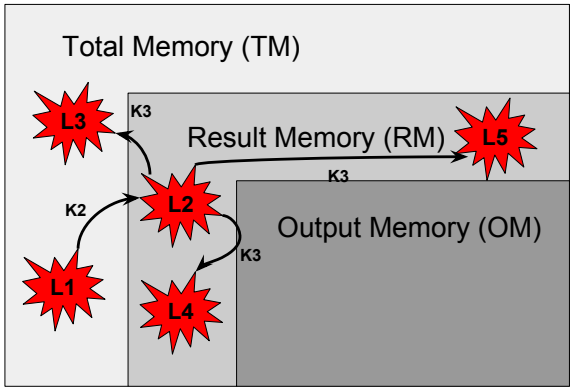



**Figure 7.3:** Memory State Layout for CUDA Programming Model (K2 and K3 are kernel invocations)

## 7.4 Experimental Setup

We describe the benchmarks used, and the fault injection procedure. We also provide details of the hardware and software platform used for the measurements, followed by the research questions.

### 7.4.1 Benchmarks Used

We choose twelve GPGPU applications in total for our experiments. These are drawn from standard GPU benchmark suites such as Rodinia [33] and Parboil [133], as well as real world applications. We choose five programs from Rodinia, and two programs from Parboil. The applications from Rodinia were chosen based on two criteria: (1) compatibility with our toolset (i.e., we could compile them with NVCC), and (2) suitability for our experiments. For the latter criteria, we discard small applications that had too few kernel invocations (as it is uninteresting to measure error propagation in such applications), and applications in which the outputs were non-deterministic (as it is difficult to classify the results of an injection or error propagation in such applications). For Parboil, we randomly choose two applications from the suite to balance time with representativeness.

In addition to the standard benchmarks, we pick five real-world HPC GPGPU applications, Lulesh [80], Barnes-Hut [116], Fiber [152], Circuit [1] and NMF [16]. These applications perform n-body simulation, hydrodynamics modeling, fiber scattering simulation, circuit solving and audio source processing respectively.

Table 7.1 shows the details of the applications used in our study and the input used. The number of kernel invocations ranges from 4 to 8567 in these applications. The lines of C code of these applications ranges from 222 to 5684. We configured our benchmarks to run on a single GPU as our goal is to study error propagation between kernels. Multi-GPU programs also transfer data and synchronize at kernel boundaries [3], and we hypothesize that our results generalize to such programs - validating this hypothesis is a subject of future work.

Similar to prior work in the area [11, 50, 63, 65, 150], we run each benchmark application with a single input. However, questions remain on whether multiple inputs may affect error propagation behaviors. We hypothesize that different inputs have limited effect on error propagation as the propagation is primarily dominated by the application's algorithm, rather than problem size. This is because different inputs likely only scale the execution times of certain code sections, rather than change the underlying program structure. We will further validate this hypothesis in future work.

**Table 7.1:** Characteristics of GPGPU Programs in our Study

| Benchmark | Benchmark Suite/Author | Description | Kernel Invocations | LOC | Input |
|---|---|---|---|---|---|
| BFS | Rodinia (v2.1) | An algorithm for traversing or searching tree or graph data structures | 15 | 342 | 4096 |
| LUD | Rodinia (v2.1) | An algorithm to calculate the solutions of a set of linear equations | 9 | 564 | 64 |
| PathFinder | Rodinia (v2.1) | Use dynamic programming to find a path on a 2-D grid | 4 | 236 | 100000 100 20 |
| Gaussian | Rodinia (v2.1) | Compute result row by row, solving for all of the variables in a linear system | 29 | 394 | 16 |
| HotSpot | Rodinia (v2.1) | Estimate processor temperature based on an architectural floorplan and simulated power measurements | 15 | 328 | 512 2 32 |
| cutcp | PARBOIL (v0.2) | Computes the short-range component of Coulombic potential at each grid point | 10 | 1540 | watbox. sl40.pqr |
| stencil | PARBOIL (v0.2) | An iterative Jacobi stencil operation on a regular 3-D grid | 99 | 1584 | 128 128 32 100 |
| Barnes-Hut | Texas State Univ. San Marcos (v2.1) | An approximation algorithm for performing an n-body simulation developed by Texas State Univ. San Marcos | 20 | 965 | 4 4 |
| Lulesh | Lawrence Livermore National Laboratory (v1.0) | Science and engineering problems that use modelling hydrodynamics | 8567 | 5684 | edgeNodes =2 |
| Fiber | Northeastern University (v1.5) | High Performance Computing of Fiber Scattering Simulation application | 2881 | 1437 | 480 4 20 |
| Circuit | Rice University | Parallel circuit solver for solving 2D circuit grid using Jacobi method | 450 | 222 | 0.00001 1 |
| NMF | UC Berkley | Audio analysis and source separation. | 409 | 2398 | default |

### 7.4.2 Fault Injection Method

As mentioned before, we use CRASHFINDER to perform the fault injection experiments. We consider only one fault per run as hardware transient faults are rare events relative to the program execution times. For each application, we inject 10,000 faults in total - this yields error bars ranging from 0.22% to 1.11% depending on the application for the 98% confidence intervals. Further, we use the single bit-flip model for injecting faults as it is the de-facto fault model used in studies of transient faults. Although recent work [38] has found that hardware faults may manifest as both single and multiple bit flips at the software level, other studies have shown that there is very little difference in failure rates due to single and multiple bit flips [14, 37, 96]. Therefore, we stick to the single bit flip model in this study.

To obtain a golden run for error propagation analysis, we first run the program without any fault injections. We then gather the output of the program and memory data stored after each kernel invocation as described in Section 7.2.2. We measure error propagation and error spreading by comparing data from fault injection runs with the golden run. This comparison is done on a bit-wise basis, except for floating point numbers, which are compared using 40 digits of precision. As we omit benchmarks that have random values in program output, the golden runs of the chosen benchmarks are deterministic. We manually verified that this was the case for our benchmarks.

There are three kinds of failures that can occur due to an injected fault: Crashes, SDCs and Hangs. Crashes are found by using the CUDA API call *cudaGetLastError()* after every kernel invocation. SDCs are found by comparing the program's output with the golden run for each memory type ($TM$, $RM$, and $OM$). Hangs are found by setting a watchdog timer for 5000 seconds when the program starts - this is much larger than the time taken by each application run.

### 7.4.3 Hardware and Software Platform

Fault injection experiments are performed on host PCs with an Intel Xeon CPU and 32GB DDR3 memory. We use two GPU platforms both from Nvidia, namely Tesla K20 the GTX960, for running our experiments. The results were similar on

both platforms - this is not surprising as our experiments were at the application level. We therefore report only the results on the K20 platform in this paper. We will further detail our comparison in RQ7. The operating system running on the host is Ubuntu Linux 12.04 64bit, and the CUDA driver and Toolkit used is V6.0.1.

### 7.4.4 Research questions (RQs)

We answer the following questions in our study.

RQ1: What is the percentage of SDCs in different memory states?

RQ2: How long do errors take to propagate to the RM?

RQ3: Do errors spread into different memory states and why?

RQ4: How many faults are masked within the GPU kernel and not allowed to propagate?

RQ5: Do crash-causing faults propagate to the host CPU before they cause crashes?

RQ6: Do crash-causing faults propagate across kernels before they cause crashes?

RQ7: Are resilience characteristics of applications different on different GPU platforms?

## 7.5 Results

The results are organized by the research questions asked in Section 7.4. We first present the aggregate results of the fault injections across all the benchmarks.

### 7.5.1 Aggregate Fault Injections

Figure 7.4 shows the aggregate results for our fault injection experiments. SDCs and Benign are measured by comparing the programs' final outputs with the golden run. This corresponds to data recored in OM after the programs finish their executions. So SDCs here correspond to $SDC_{OM}$s, and benign to $Benign_{OM}$. On average, crashes constitute 17.52%, SDCs constitute 18.98% and Benign faults constitute 63.35% of the injections. In our experiments, hangs are negligible, and are hence not reported.

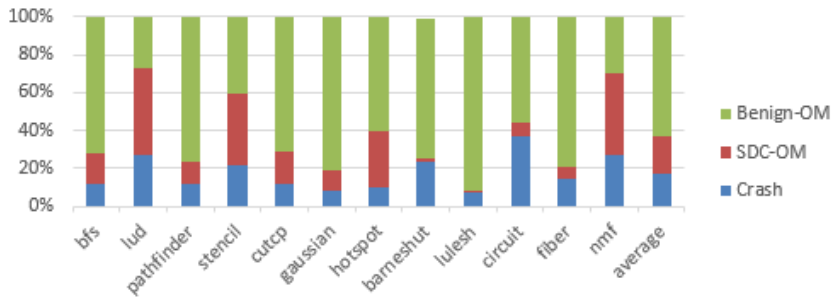



**Figure 7.4:** Aggregate Fault Injection Results across the 12 Programs

**Table 7.2:** SDCs that occur in the different memory types

| | bfs | lud | pathfinder | stencil | cutcp | gaussian | hotspot | barneshut | lulesh | circuit | fiber | nmf | average |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $SDC_{OM}$ | 17.01% | 45.58% | 20.60% | 37.00% | 16.50% | 10.90% | 29.40% | 1.30% | 0.97% | 7.50% | 7.01% | 43.20% | 19.67% |
| $SDC_{(RM-OM)}$ | 0.00% | 0.00% | 1.30% | 0.00% | 28.90% | 0.00% | 6.00% | 2.30% | 0.10% | 14.80% | 12.69% | 10.10% | 6.35% |
| $SDC_{RM}$ | 17.01% | 45.58% | 21.90% | 37.00% | 45.40% | 10.90% | 31.50% | 3.60% | 1.07% | 22.30% | 19.70% | 53.30% | 25.70% |
| $SDC_{(TM-RM)}$ | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% | 0.80% | 0.00% | 0.20% | 0.00% | 0.10% | 0.00% | 0.00% | 0.09% |
| $SDC_{TM}$ | 17.01% | 45.58% | 21.90% | 37.00% | 45.40% | 11.70% | 31.50% | 3.80% | 1.07% | 22.40% | 19.70% | 53.30% | 25.79% |

### 7.5.2 Error Propagation

**RQ1: What's the percentage of SDCs in different memory states?**

We analyze the memory data in each memory type after the last kernel invocation in kernel cycle, and compare with the golden run. Table 7.2 shows SDCs measured at different memory locations at the end of the kernel cycle. On average, $SDC_{OM}$, $SDC_{RM}$ and $SDC_{TM}$ are 19.67%, 25.70% and 25.79%. As can be seen, the values of $SDC_{RM}$ and $SDC_{TM}$ are very similar across applications. In other words, most faults in the TM propagate to the RM. This is surprising as it suggests that there is little to no masking of errors in the TM. On the other hand, CPU applications are known to exhibit significant error masking in memory [11]. One possible reason is that unlike CPUs, GPUs perform highly specialized computations, and hence all the results produced are important to the application and hence they propagate to the output.

However, there is a difference of about 6% on average between $SDC_{OM}$ and $SDC_{RM}$ (see in $SDC_{(RM-OM)}$). This is due to the application masking the error either through type-casting or selective truncation of floating point data. An example

of this is *cutcp*, which exhibits a difference of nearly 30% between the $SDC_{OM}$ and the $SDC_{RM}$. Overall, about 76% of the faults propagate from the RM to the OM, which is again much higher than observed in CPU applications [11].

We note that the *lulesh* application comes equipped with application-level algorithm correctness checks (i.e., residual check). For example, the documentation for *lulesh* states that the correct output consists of the correct number of iterations, and six most-significant digits in the *final origin energy* variable [79]. Therefore, $SDC_{OM}$ here is measured based on these correctness checks. None of the other applications however come with such checks, and hence we consider the entire output in these cases for comparison with the fault-injected run.

**RQ2: How long do errors take to propagate to the RM?**

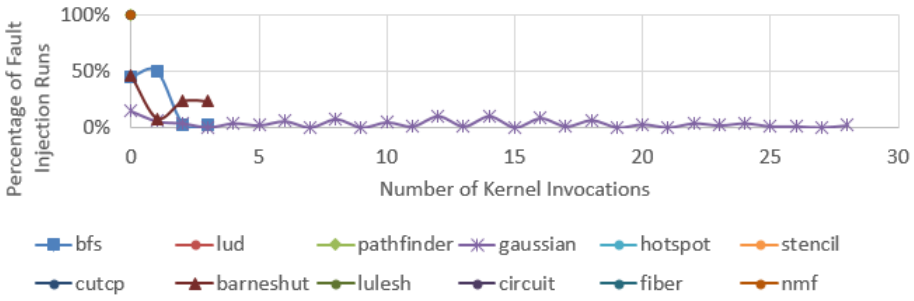



**Figure 7.5:** Detection Latency of faults that result in $SDC_{RM}$

Once a fault is injected in a kernel, we measure the number of kernel calls after which it propagates to RM (if it does). Figure 7.5 shows the results. As shown in the figure, most faults that affect the RM do so within a single kernel invocation after injection. This means that errors propagate relatively soon to the RM after their occurrence. The exceptions are *bfs*, *gaussian* and *barneshut*.

Figure 7.6 shows the percentage of RM updated after each kernel invocation. For the sake of space, we only show the results for two applications, namely *lud* and *gaussian*. As we can see, *lud* updates the RM in every kernel invocation, whereas *gaussian* does not. This explains why the propagation occurred within one kernel execution for *lud*, but not *gaussian*.

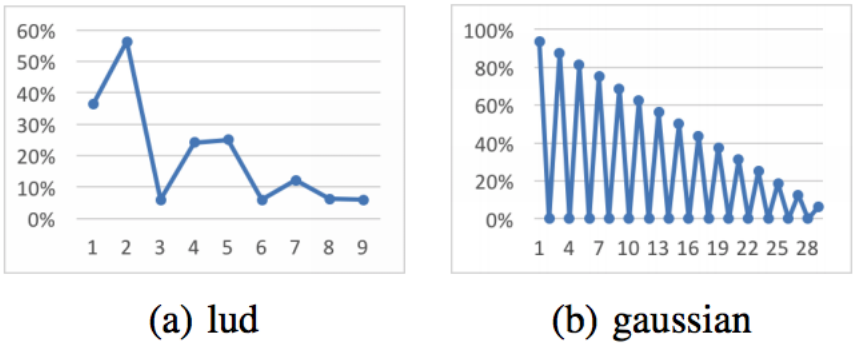



**Figure 7.6:** Percentage of RM Updated by Each Kernel Invocation. Y-axis is the percentage of RM locations that are updated during each kernel invocation. X-axis represents timeline in terms of kernel invocations.

### 7.5.3 Error Spreading

We defined error spreading as the percentage of memory locations contaminated due to an error (Section 7.3). Unlike the previous section where we considered any difference from the golden run as an SDC for that memory type, here we only consider the amount of memory that is different.

**RQ3: Do errors spread into different memory states and why?**

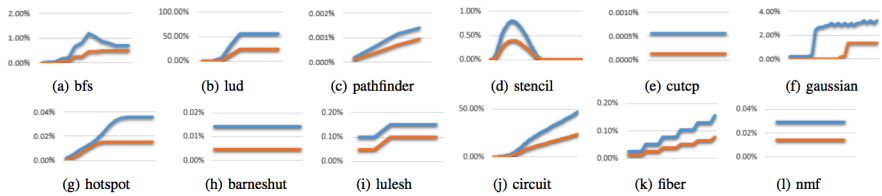



**Figure 7.7:** Percentage of TM and RM Contaminated at Each Kernel Invocation. Y-axis is the percentage of contaminated memory locations, X-axis is timeline in terms of kernel invocations. Blue lines indicate TM, and red lines represent RM.

From our fault injection experiments, we examine how errors spread as a function of time (i.e., kernel invocations). Because we performed 10,000 injections per application, we cannot show all the data. Therefore, we only show represen-

tative injections for each application. Figure 7.7 shows the percentage of memory locations in TM and RM that are contaminated by the injected faults at the first dynamic kernel invocation. The spread is calculated as (*Contaminated TM or RM Locations* / *Total TM Locations*) * 100.

Our main findings are: (1) error-spreading is very application-specific. For example, a single fault can contaminate nearly 60% of TM memory locations in *lud*, whereas the number is as low as 0.0006% in *cutcp*. (2) only a very small amount of memory locations are affected in the same kernel where the fault was injected. Rather, most faults propagate into memory locations in later kernel invocations. (3) trends of error spreading between RM and TM of the same application are rather similar, though the absolute values may be different. In other words, applications that have extensive error spread in the TM have extensive error spread in the RM as well.

Finally, in almost all applications the error spreading either increases or remains constant as the number of kernel calls increase. The exception to this is the *stencil* application in which the error spreading decreases significantly as the number of kernel calls increase. This is because the algorithm of *stencil* takes neighbours' values and keeps averaging them. The errors may be finally masked as the averaging process progresses. We also observed that there is a small decrease in error spreading in the *bfs* application after it reaches its peak in TM. This is because some of locations in TM are reassigned during program execution, and faulty data in these locations can be overwritten with correct data, thereby masking the errors.

Because *lud* exhibits the highest error spread, we study its code structure to understand the reasons. Figure 7.8 shows the code structure leading to extensive error spread in *lud*. The code exhibits a cyclic data flow from global memory to shared memory, and then back to global memory. Note that shared memory is used to transfer data between threads only in the same block. In the example, *dia* is a pointer to shared memory, and *m* points to global memory. At line 5, a portion of shared memory in *dia* is initialized by global memory *m*. This portion of data in *dia* is shared by other threads in its block. After the shared data is consumed, it writes data back to global memory *m* at line 9 again. If a fault occurs in *m* at line 5, *dia* will be first compromised and the data processed by other threads in the same

```
1  ...
2  kernel lud_perimeter(*m){sv
3    __shared__ dia[BLOCK_SIZE][BLOCK_SIZE];
4    ...
5    dia[i][j] = m[array_offset+idx];
6    ...
7    for(...){
8      ...
9      m[array_offset+(blockId.x+1)+...] = dia[i][idx];
10     ...
11   }
12   ...
13 }
14 ...
```

**Figure 7.8:** Code Structure Leading to Extensive Error Spread in *lud*

block may be affected after reading the corrupted data from *dia*. And then at the end of the kernel invocation, a different part of *m* may be corrupted by reading data from *dia*(line 9). In the next invocation of this kernel, faulty values in *m* may be used in the initialization of *dia* again at line 5. But this time, it may be initialized to a different portion of *dia* that are used by threads in a different block, because there are different parts of *m* that were corrupted in the previous kernel invocation. This leads to extensive error spread for this application.

### 7.5.4 Fault Masking

In the previous two sections, we examined how faults propagate across different memory locations and kernel executions. We now ask the complementary question: how many faults are masked within a single kernel invocation of their occurrence.

**RQ4: How many faults are masked within the GPU kernel and not allowed to propagate?**

Table 7.3 shows the percentage of benign faults measured at the first kernel invocation after the fault injection ($Benign_{TM}$). We measure this value by comparing all memory locations in TM with the golden run right after the first kernel invocation where the fault is injected. If there is no difference between the two TMs, we count it as a benign fault. We consider TM here as it is a superset of both RM and OM. Therefore if a fault is masked in the TM, it is also masked in the other two types of memory (even in future kernel executions).

**Table 7.3:** Percentage of Benign Faults Measured at the First Kernel Invocation after Fault Injection

| bfs | lud | pathfinder | stencil | cutcp | gaussian | hotspot | barneshut | lulesh | circuit | fiber | nmf | average |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 63.5% | 28.5% | 69.9% | 25.0% | 42.7% | 81.1% | 57.2% | 76.4% | 92.5% | 25.6% | 62.9% | 20.0% | 53.4% |

As we can see from the table, more than 50% of the faults injected are masked within the same kernel they are injected in, and do not propagate. The maximum masking is achieved in the case of *lulesh* in which nearly 92.5% of the faults are masked. The minimum masking is achieved for *nmf* in which only 20% of the faults are masked. Such variations across applications are because programs contain different amount of code structures leading to masking effects.

We found there are two prevalent patterns leading to error masking in our benchmark programs, namely (1) Comparison, and (2) Truncation. An example of Truncation is shown in Figure 7.9, on the left. R0 and R1 are initialized at lines 2 and 3, and R2 holds the result of comparing R0 and R1 at line 3. Consider a fault that flips the first bit of R1 - R1 erroneously becomes 1110 from 1111. However, the result of R2 will not be affected since R1 is still greater than R0. An example of Truncation is shown in the right part of Figure 7.9. At line 2, R0 is initialized. At line 3, value of R1 is truncated from 0001 to 01. Consider a fault that occurs at line 2 and flips either of the left-most 2 bits of R0 - it will not affect the value of R1 at line 3 due to the truncation. Hence the fault will be masked.

```
1 ...
2 R0 = 0000
3 R1 = 1111
4 R2 = cmp R0, R1
5 ...
```

```
1 ...
2 R0 = 0001
3 R1 = cast R0, 2
4 // R1 -> 01
5 ...
```

**Figure 7.9:** Examples of Fault Masking. (a) Comparison, (b) Truncation

### 7.5.5 Crash-causing Faults

The next two research questions have to do with crash-causing faults and their propagation to the CPU (host) or other GPU kernels. We focus on crash-causing faults as prior work has found that long-latency crashes can lead to checkpoint

corruptions, and cause unrecoverable failures [89, 150].

**RQ5: Do faults propagate to the host CPU and cause crashes?**

From our experiments, we can observe that there is a very small chance (0.02% on average) for a fault that occurs in a kernel to contaminate the CPU states and lead to a crash eventually. This is because memory address spaces are separate in the CPU and GPU, and hence faulty pointers produced by the GPU are unlikely to be used in the CPU to access memory. Because faulty pointers are responsible for the majority of crash causing errors on the CPU [89], these errors do not lead to crashes.

**RQ6: Do crash-causing faults propagate across kernels before they cause crashes?**

In our experiments, we find that there is no fault that propagates across kernels and causes a crash. In other words, cash-causing faults typically cause crashes within the kernel in which they occur. This is because pointers or memory address offset variables are usually passed between kernels in the constant memory space, which is read-only. Note however that it is possible for faults to propagate across kernels if address offsets of pointers are passed through global variables (we have empirically verified this observation through carefully constructed code samples - we do not present these due to space constraints). However, typical GPGPU programs do not exhibit this behavior, as each thread is responsible for its own memory locations, and hence multiple threads do not read the same offset to calculate the same memory address.

### 7.5.6 Platform Differences

**RQ7: Are resilience characteristics of applications different on different GPU platforms?**

We use the two platforms, K20 and GTX960 for this experiment. To answer this question, we performed 1,000 fault injections for each benchmark application on the two platforms. We focus on SDCs for this experiment as these are often the most important concern in practice. Note that we have omitted two programs, *fiber* and *nmf*, as we encountered errors when compiling them on GTX960, probably because they use features that are not supported on that platform. To compare

the distributions of the SDC values, we ran a t-test between the values obtained on the two platforms. We found that the p-value was 0.991. Thus, our results show that we fail to reject the null hypothesis, indicating that the values are statistically indistinguishable from each other. Therefore, we can conclude that the resilience characteristics of the applications do not vary significantly between the two platforms.

## 7.6 Implications

In this section, we consider the implications of the results on error detection and recovery techniques. These are organized by the RQs.

**Table 7.4:** Size of OM, RM and TM

| | bfs | lud | pathfinder | stencil | cutcp | gaussian | hotspot | barneshut | lulesh | circuit | fiber | nmf | geo mean |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OM | 14.29% | 7.50% | 0.99% | 7.50% | 7.50% | 1.67% | 5.00% | 18.75% | 12.50% | 15.00% | 49.98% | 0.03% | 5.02% |
| RM | 14.29% | 50.00% | 1.98% | 50.00% | 50.00% | 3.21% | 66.67% | 37.50% | 18.75% | 50.00% | 49.98% | 0.03% | 13.56% |
| TM | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |

**RQ1: Percentages of SDCs in different memory states** In RQ1, we found that most SDCs in the TM propagate to the RM, and more than 76% of the SDCs in the RM propagate to the OM. Table 7.4 shows the size of data in OM, RM and TM in each application. As we can see, the RM is only a small fraction of the TM. This suggests that checking the RM for consistency (e.g., using detectors [108]) may be much more efficient than checking the entire TM. Another possibility is checking the OM which is even smaller than the RM. However, the OM may not be updated until the end of the program and hence checking the OM may incur high detection latency.

**RQ2: Detection Latency of Errors in RM** Error detection latency is critical when designing checkpoint intervals. If the detection latency is too long, errors may propagate to checkpoints before they are detected, thereby corrupting checkpoints [89]. Our results show that errors propagate to the RM relatively soon after their occurrence (i.e., within one kernel call in most cases). Therefore, placing detectors on RM will ensure low-latency error detection.

**RQ3: Error spreading to different memory states** Error spreading is highly application specific in GPGPU applications. Further, only certain code structures in GPGPU programs may lead to extensive error spread. Therefore, one can statically

analyze the program to identify such structures to protect. Further, the code can be restructured to avoid error spreading in some circumstances. In some applications (e.g., *Stencil*), there may be natural mechanisms in the code to dilute the effect of error spreading over time.

**RQ4: Effect of error masking** We found that there is substantial error masking within GPU kernels, and that many errors do not even affect the TM after they occur. This means that there may not be a need to deploy expensive error detection mechanisms such as Dual Modular Redundancy (DMR) or Error Detection by Duplicated Instructions (EDDI) [106] within the kernel, unless it is a safety-critical application. Instead one can check the results after the kernel's invocation. For example, ABFT-based detection algorithms [45, 73] can be used at the kernel boundary to detect errors, to determine whether GPU kernel re-execution should be initiated.

**RQ5 & RQ6: Crash-causing Faults and Checkpoint Scheme** Studies on error propagation on CPUs find that crash-causing faults can propagate for a long time before they cause crashes. Hence, checkpoints may be corrupted by these faults if the crash-latency in the program is not bounded [88, 89]. However, on GPGPU programs, we find that crash-causing faults do not propagate outside the kernel where faults occur. In other words, crash-latency of GPGPU programs is naturally bounded within one kernel invocation. Therefore, one can place checkpoints at kernel boundaries for crash recovery. As we find that many kernels do not propagate errors to other kernels, individual kernels could also recover from failures through re-execution at the kernel boundaries of copying. The application can be restarted locally on the same GPU or on a spare GPU. Further, as most kernels have short execution times in the range of milliseconds, the cost of re-executing a kernel would be insignificant.

**RQ7: Differences across platforms** From our findings, it appears that error resilience of GPGPU applications does not depend on the specific hardware platform (we have only validated it on platforms from the same manufacturer, which was Nvidia in our case). This suggests that one can perform resilience characterization on one platform and generalize the results to a different platform.

## 7.7 Summary

In this chapter, we study error propagation in GPGPU application with the goal of building targeted error detection and recovery mechanisms for them. We built a fault injection tool LLFI-GPU, and defined metrics for quantifying propagation in GPGPU applications. We empirically studied error propagation across ten GPGPU applications using LLFI-GPU. The main findings are: (1) Crash-causing faults in GPGPU are naturally kernel-bounded, (2) Error spreading in memory is highly application dependent, (3) Most memory data corruptions lead to output corruption unlike what is observed in CPU programs, and (4) The majority of faults are masked within a single kernel execution, and do not propagate across kernels.

# Chapter 8

# Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications

This chapter investigates error propagation in DNN accelerators and applications, which are recently deployed in safety-critical environment such as self-driving cars. Many studies have focused on the performance aspects of the accelerators, but their reliability is not well understood. In this chapter, we first build the fault injection infrastructure for DNN accelerators and applications, then characterize error propagation through an empirical study. We find that DNN accelerators and applications have very unique error propagation characteristics compared with general purpose applications. Based on our investigation, we propose cost-effective error mitigation techniques for DNN accelerators and applications.

## 8.1 Introduction

Deep learning neural network (DNN) applications are widely used in high-performance computing systems and datacenters [19, 44, 118]. Researchers have proposed the use of specialized hardware accelerators to accelerate the inferencing process of

DNNs, consisting of thousands of parallel processing engines [34, 36, 62]. For example, Google recently announced its DNN accelerator, the Tensor Processing Unit (TPU), which they deploy in their datacenters for DNN applications [145].

While the performance of DNN accelerators and applications have been extensively studied, the reliability implications of using them is not well understood. One of the major sources of unreliability in modern systems are soft errors, typically caused by high-energy particles striking electronic devices and causing them to malfunction (e.g., flip a single bit) [20, 40]. Such soft errors can cause application failures, and they can result in violations of safety and reliability specifications. For example, the IEC 61508 standard [74] provides reliability specifications for a wide range of industrial applications ranging from the oil and gas industry to nuclear power plants. Deep neural networks have promising uses for data analytics in industrial applications [149], but they must respect the safety and reliability standards of the industries where they are employed.

A specific and emerging example of HPC DNN systems is for self-driving cars (i.e., autonomous vehicles), which deploy high performance DNNs for real-time image processing and object identification. The high performance, low power, high reliability, and real-time requirements of these applications require hardware that rivals that of the fastest and most reliable supercomputer (albeit with lower precision and memory requirements). For instance, NVIDIA Xavier—a next-generation SoC with a focus on self-driving applications—is expected to deliver 20 Tops/s at 20W in 16nm technology [126]. We focus our investigation into DNN systems[1] in this emerging HPC market due to its importance, stringent reliability requirements, and its heavy use of DNNs for image analysis. The ISO 26262 standard for functional safety of road vehicles mandates the overall FIT rate[2] of the System on Chip (SoC) carrying the DNN inferencing hardware under soft errors to be less than 10 FIT [119]. This requires us to measure and understand the error resilience characteristics of these high-performance DNN systems.

This chapter takes a first step towards this goal by (1) characterizing the propagation of soft errors from the hardware to the application software of DNN sys-

[1] We use the term DNN systems to refer to both the software and the hardware accelerator that implements the DNN.

[2] Failure-in-Time rate: 1 FIT = 1 failure per 1 billion hours

tems, and (2) devising cost-effective mitigation mechanisms in both software and hardware, based on the characterization results .

Traditional methods to protect computer systems from soft errors typically replicate the hardware components (e.g., Triple Modular Redundancy or TMR). While these methods are useful, they often incur large overheads in energy, performance and hardware cost. This makes them very challenging to deploy in self-driving cars, which need to detect objects such as pedestrians in real time [83] and have strict cost constraints. To reduce overheads, researchers have investigated software techniques to protect programs from soft errors, e.g., identifying vulnerable static instructions and selectively duplicating the instructions [52, 64, 88]. The main advantage of these techniques is that they can be tuned based on the application being protected. However, DNN software typically has a very different structure compared to general-purpose software. For example, the total number of static instruction types running on the DNN hardware is usually very limited (less than five) as they are repeatedly executing the multiply-accumulate (MAC) operations. This makes these proposed techniques very difficult to deploy as duplicating even a single static instruction will result in huge overheads. Thus, current protection techniques are DNN-agnostic in that they consider neither the characteristics of DNN algorithms, nor the architecture of hardware accelerators. *To the best of our knowledge, we are the first to study the propagation of soft errors in DNN systems and devise cost-effective solutions to mitigate their impact.*

We make the following major contributions in this paper:

- We modify a DNN simulator to inject faults in four widely used neural networks (AlexNet, CaffeNet, NiN and ConvNet) for image recognition, using a canonical model of the DNN accelerator hardware.

- We perform a large-scale fault injection study using the simulator for faults that occur in the data-path of accelerators. We classify the error propagation behaviors based on the structure of the neural networks, data types, positions of layers, and the types of layers.

- We use a recently proposed DNN accelerator, Eyeriss [35], to study the effect of soft errors in different buffers and calculate its projected FIT rates.

- Based on our observations, we discuss the reliability implications of designing DNN accelerators and applications and also propose two cost-effective error protection techniques to mitigate Silent Data Corruptions (SDCs) i.e., incorrect outcomes. The first technique, symptom-based detectors, is implemented in software and the second technique, selective latch hardening, in hardware. We evaluate these techniques with respect to their fault coverage and overhead.

Our main results and implications are as follows:

- We find that different DNNs have different sensitivities to SDCs depending on the topology of the network, the data type used, and the bit position of the fault. In particular, we find that only high-order bits are vulnerable to SDCs, and the vulnerability is proportional to the dynamic value range the data type can represent. The implications of this result are twofold: (1) When designing DNN systems, one should choose a data type providing just-enough dynamic value range and precision. The overall FIT rate of the DNN system can be reduced by more than an order of magnitude if we do so - this is in line with recent work that has proposed the use of such data types for energy efficiency. (2) Leveraging the asymmetry of the SDC sensitivity of bits, we can selectively protect the vulnerable bits using our proposed selective latch hardening technique. Our evaluation shows that the corresponding FIT rate of the datapath can be reduced by 100x with about 20% area overhead.

- We observe that faults causing a large deviation in magnitude of values likely lead to SDCs. Normalization layers can reduce the impact of such faults by averaging the faulty values with adjacent correct values, thereby mitigating SDCs. While the normalization layers are typically used to improve performance of a DNN, they also boost its resilience. Based on the characteristics, we propose a symptom-based detector that provides 97.84% precision and 92.08% recall in error detection, for selected DNNs and data types.

- In our case study of Eyeriss, the sensitivity study of each hardware component shows that some buffers implemented to leverage data locality for

performance may dramatically increase the overall FIT rate of a DNN accelerator by more than 100x. This indicates that novel dataflows proposed in various studies should also add protection to these buffers as they may significantly degrade the reliability otherwise.

- Finally, we find that for the Eyeriss accelerator platform, the FIT rates can exceed the safety standards (i.e., ISO 26262) by orders of magnitude without any protection. However, applying the proposed protection techniques can reduce the FIT rate considerably and restore it within the safety standards.

## 8.2 Exploration of Design Space

We seek to understand how soft errors that occur in DNN accelerators propagate in DNN applications and cause SDCs (we define SDCs later in Section 8.3.6). We focus on SDCs as these are the most insidious of failures and cannot be detected easily. There are four parameters that impact of soft errors on DNNs:

(1) **Topology and Data Type:** Each DNN has its own distinct topology which affects error propagation. Further, DNNs can also use different data types in their implementation. We want to explore the effect of the topology and data type on the overall SDC probability.

(2) **Bit Position and Value:** To further investigate the impact of data type on error propagation, we examine the sensitivity of each bit position in the networks using different data types. This is because the values represented by a data type depend on the bit positions affected, as different data types interpret each bit differently (explained in Section 8.3.5). Hence, we want to understand how SDC probabilities vary based on the bit corrupted in each data type and how the errors result in SDCs affect program values.

(3) **Layers:** Different DNNs have different layers - this includes the differences in type, position, and the total number of layers. We investigate how errors propagate in different layers and whether the propagation is influenced by the characteristics of each layer.

(4) **Data Reuse:** We want to understand how different data reuses implemented in the dataflows of DNN accelerators affects the SDC probability. Note that unlike

other parameters, data reuse is not a property of the DNN itself but of its hardware implementation.

## 8.3 Experimental Setup

### 8.3.1 Networks

**Table 8.1:** Networks Used

| Network | Dataset | No. of Output Candidates | Topology |
|---|---|---|---|
| ConvNet [41] | CIFAR-10 | 10 | 3 CONV + 2 FC |
| AlexNet [81] | ImageNet | 1,000 | 5 CONV(with LRN) + 3 FC |
| CaffeNet [24] | ImageNet | 1,000 | 5 CONV(with LRN) + 3 FC |
| NiN [94] | ImageNet | 1,000 | 12 CONV |

We focus on convolutional neural networks in DNNs, as they have shown great potential in solving many complex and emerging problems and are often executed in self-driving cars. There are four neural networks that we consider in Table 8.1. They range from the relatively simple 5-layer ConvNet to the 12-layer NiN. The reasons we chose these networks are: (1) They have different topologies and methods implemented to cover a variety of common features used in today's DNNs, and the details are publicly accessible, (2) they are often used as benchmarks in developing and testing DNN accelerators [35, 77, 128], and (3) they are well known to solve challenging problems, (4) and the official pre-trained models are freely available. This allows us to fully reproduce the networks for benchmarking puposes. All of the networks perform the same task, namely image classification. We use the ImageNet dataset [75] for AlexNet, CaffeNet and NiN, and the CIFAR-10 dataset [39] for ConvNet, as they were trained and tested with these datasets. We use these reference datasets and the pre-trained weights together with the corresponding networks from the Berkeley Vision and Learning Center (BVLC) [22].

We list the details of each network in Table 8.1. As shown, all networks except NiN have fully-connected layers behind the convolutional layers. All four networks implement ReLU as the activation function and use the max-pooling method in their sub-sampling layers. Both AlexNet and CaffeNet use a Local Response Nor-

malization (LRN) layer following each of the first two convolutional layers - the only difference is the order of the ReLU and the sub-sampling layer in each convolution layer. In AlexNet, CaffeNet and ConvNet, there is a soft-max layer at the very end of each network to derive the confidence score of each ranking, which is also part of the network's output. However, in NiN, there is no such soft-max layer. Hence, the output of the NiN network has only the ranking of each candidate without their confidence scores.

**Table 8.2:** Data Reuses in DNN Accelerators

| | **Weight Reuse** | **Image Reuse** | **Output Reuse** |
|---|---|---|---|
| Zhang et al. [153], Diannao [34], Dadiannao [36] | N | N | N |
| Chakradhar et al. [28], Sriram et al. [131], Sankaradas et al. [122], nn-X [57], K-Brain [107], Origami [27] | Y | N | N |
| Gupta et al. [60], Shidiannao [49], Peemen et al. [109] | N | N | Y |
| Eyeriss [35] | Y | Y | Y |

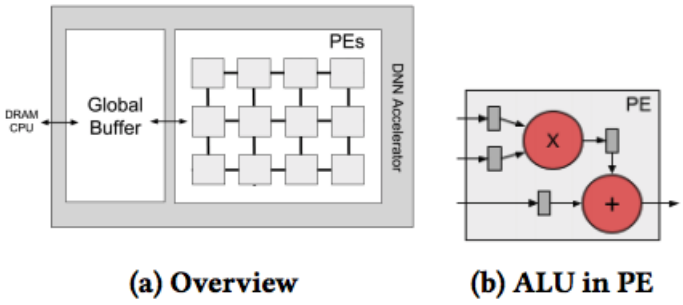



**Figure 8.1:** Architecture of general DNN accelerator

### 8.3.2 DNN Accelerators

We consider nine of DNN accelerators mentioned in Table 8.2. We separate the faults in the datapaths of the networks from those in the buffers. We study datapath faults based on the common abstraction of their execution units in Figure 8.1B. Thus, the results for datapath faults apply to all nine accelerators.

For buffer faults, since the dataflow (and buffer structure) is different in each accelerator, we have to choose a specific design. We chose the Eyeriss accelerator for studying buffer faults because: (1) The dataflow of Eyeriss includes all three data localities in DNNs listed in Table 8.2, which allows us to study the data reuse seen in other DNN accelerators, and (2) the design parameters of Eyeriss are publicly available, which allows us to conduct a comprehensive analysis on its dataflow and overall resilience.

### 8.3.3 Fault Model

We consider transient, single-event upsets that occur in the data path and buffers, both inside and outside the processing engines of DNN accelerators. We do not consider faults that occur in combinational logic elements as they are much less sensitive to soft errors than storage elements shown in recent studies [56, 125]. We also do not consider errors in control logic units. This is due to the nature of DNN accelerators which are designed for offloaded data acceleration - the scheduling is mainly done by the host (i.e., CPU). Finally, because our focus is on DNN accelerators, we do not consider faults in the CPU, main memory, or the memory/data buses.

### 8.3.4 Fault Injection Simulation

Since we do not have access to the RTL implementations of the accelerators, we use a DNN simulator for fault injection. We modified an open-source DNN simulator framework, Tiny-CNN [142], which accepts Caffe pre-trained weights [23] of a network for inferencing and is written in C++. We map each line of code in the simulator to the corresponding hardware component, so that we can pinpoint the impact of the fault injection location in terms of the underlying microarchitectural components. We randomly inject faults in the hardware components we consider

by corrupting the values in the corresponding executions in the simulator. This fault injection method is in line with other related work [64, 82, 88, 93, 147].

### 8.3.5 Data Types

Different data types offer different tradeoffs between energy consumption and performance in DNNs. Our goal is to investigate the sensitivity of different design parameters in data types to error propagation. Therefore, we selected a wide range of data types that have different design parameters as listed in Table 8.3. We classify them into two types: floating-point data type (FP) and fixed-point data type (FxP). For FP, we choose 64-bit double, 32-bit float, and 16-bit half-float, all of which follow the IEEE 745 floating-point arithmetic standard. We use the terms DOUBLE, FLOAT, and FLOAT16 respectively for these FP data types in this study. For FxPs, unfortunately there is no public information about how binary points (radix points) are chosen for specific implementations. Therefore, we choose different binary points for each FxP. We use the following notations to represent FxPs in this work: 16b_rb10 means a 16-bit integer with 1 bit for the sign, 5 bits for the integer part, and 10 bits for the mantissa, from the leftmost bit to the rightmost bit. We consider three FxP types, namely 16b_rb10, 32b_rb10 and 32b_rb26. They all implement 2's complement for their negative arithmetic. Any value that exceeds the maximum or minimum dynamic value range will be saturated to the maximum or minimum value respectively.

**Table 8.3:** Data types used

| Data Type | FP or FxP | Data Width | Bits (From left to right) |
|---|---|---|---|
| DOUBLE | FP | 64-bit | 1 sign bit, 11 bits for exponent, 52 bits for mantissa |
| FLOAT | FP | 32-bit | 1 sign bit, 8 bits for exponent, 23 bits for mantissa |
| FLOAT16 | FP | 16-bit | 1 sign bit, 5 bits for exponent, 10 bits for mantissa |
| 32b_rb26 | FxP | 32-bit | 1 sign bit, 5 bits for integer, 26 bits for mantissa |
| 32b_rb10 | FxP | 32-bit | 1 sign bit, 21 bits for integer, 10 bits for mantissa |
| 16b_rb10 | FxP | 16-bit | 1 sign bit, 5 bits for integer, 10 bits for mantissa |

### 8.3.6 Silent Data Corruption (SDC)

We define the SDC probability as the probability of an SDC given that the fault affects an architecturally visible state of the program (i.e., the fault was activated). This is in line with the definition used in other work [52, 64, 93, 147].

In a typical program, an SDC would be a failure outcome in which the application's output deviates from the correct (golden) output. This comparison is typically made on a bit-by-bit basis. However, for DNNs, there is often not a single correct output, but a list of ranked outputs each with a confidence score as described in Section 3.5.1, and hence a bit-by-bit comparison would be misleading. Consequently, we need to define new criteria to determine what constitutes an SDC for a DNN application. We define four kinds of SDCs as follows:

- SDC-1: The top ranked element predicted by the DNN is different from that predicted by its fault-free execution. This is the most critical SDC because the top-ranked element is what is typically used for downstream processing.
- SDC-5: The top ranked element is not one of the top five predicted elements of the fault-free execution of the DNN.
- SDC-10%: The confidence score of the top ranked element varies by more than +/-10% of its fault-free execution.
- SDC-20%: The confidence score of the top ranked element varies by more than +/-20% of its fault-free execution.

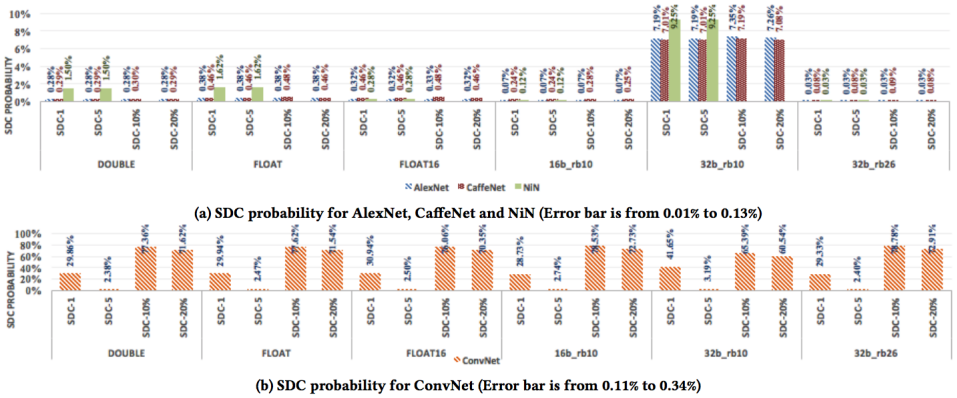
### 8.3.7 FIT Rate Calculation

The formula of calculating the FIT rate of a hardware structure is shown in Equation 8.1, where $R_{raw}$ is the raw FIT rate (estimated as 20.49 FIT/Mb by extrapolating the results of Neale et al. [103]. The original measurement for a 28nm process is 157.62 FIT/MB in the paper. We project this for a 16nm process by applying the trend shown in Figure 1 of the Neale paper[3]). $S_{component}$ is the size of the component, and $SDC_{component}$ is the SDC probability of each component. We use this

[3]We also adjusted the original measurement by a factor of 0.65 as there is a mistake we found in the paper. The authors of the paper have acknowledged the mistake in private email communications with us.

formula to calculate the FIT rate of datapath components and buffer structures of DNN accelerators, as well as the overall FIT rate of Eyeriss in Section 8.4.1 and Section 8.4.2.

$$FIT = \sum_{component} R_{\text{raw}} * S_{\text{component}} * SDC_{\text{component}} \tag{8.1}$$




(a) SDC probability for AlexNet, CaffeNet and NiN (Error bar is from 0.01% to 0.13%)

(b) SDC probability for ConvNet (Error bar is from 0.11% to 0.34%)

**Figure 8.2:** SDC probability for different data types in different networks (for faults in PE latches).

## 8.4 Characterization Results

We organize the results based on the origins of faults (i.e., datapath faults and buffer faults) for each parameter. We randomly injected 3,000 faults per latch, one fault for each execution of the DNN application. The error bars for all the experimental results are calculated based on 95% confidence intervals.

### 8.4.1 Datapath Faults

#### Data Types and Networks

Figure 8.2 shows the results of the fault injection experiments on different networks and different data types. We make three observations based on the results.

First, SDC probabilities vary across the networks for the same data type. For

example, using the FLOAT data type, the SDC probabilities for NiN are higher than for other networks using the same data type (except ConvNet - see reason below). This is because of the different structures of networks in terms of sub-sampling and normalization layers which provide different levels of error masking - we further investigate this in Section 8.4.1. Further, ConvNet has the highest SDC propagation probabilities among all the networks considered (we show the graph for ConvNet separately as its SDC probabilities are significantly higher than the other networks). This is because the structure of ConvNet is much less deep than for other networks, and consequently there is higher error propagation in ConvNet. For example, there are only 3 convolutional layers in ConvNet, whereas there are 12 in NiN. Further, ConvNet does not have normalization layers to provide additional error masking, unlike AlexNet and CaffeNet.

Second, SDC probabilities vary considerably across data types. For instance, in AlexNet, SDC-1 can be as high as 7.19% using 32b_rb10, and as low as 0.38% using FLOAT - the maximum difference is 240x (between SDC-1 in 32b_rb10 and 32b_rb26). The reasons are further explored in Section 8.4.1.

Finally, for networks using the ImageNet dataset (all except ConvNet), there is little difference in the SDC probability for the four different kinds of SDCs for a particular network and data type. Recall that there are 1,000 output dimensions in the ImageNet DNNs. If the top ranked output is changed by the error, the new ranking is likely to be outside of the top five elements, and its confidence score will likely change by more than 20%. However, in ConvNet, which uses the CIFAR-10 dataset, the four SDC probabilities are quite different. This is because there are only 10 output dimensions in ConvNet, and if the top ranked output is affected by errors, it is still highly likely to stay within the top five elements. As a result, the SDC-5 probability is quite low for this network. On the other hand, the SDC-10% and SDC-20% probabilities are quite high in ConvNet compared to the other networks, as the confidence scores are more sensitive due to the small output dimensions compared to the other networks. Note that since NiN does not provide confidence scores in its output, we do not show SDC-10% and SDC-20% for NiN.

Because there is little difference between the SDC types for three of the four networks, we focus on SDC-1 in the rest of the chapter and refer to them as SDCs

unless otherwise specified.

### Bit Position

We show the results by plotting the SDC probabilities for each bit in each data type. Due to space constraints, we only show the results for NiN using FLOAT and FLOAT16 data types for FP, and for CaffeNet using 32b_rb26 and 32b_rb10 for FxP. We however confirmed that similar observations apply to the rest of networks and data types.

The results of NiN using FLOAT and FLOAT16 are shown in Figure 8.3A and Figure 8.3B respectively. For the FP data types, only the high-order exponent bits are likely to cause SDCs (if corrupted), and not the mantissa and sign bits. We also observed that bit-flips that go from 0 to 1 in the high-order exponent bits are more likely to cause SDCs than those that go from 1 to 0. This is because the correct values in each network are typically clustered around 0 (see Section 8.4.1) and hence, small deviations in the magnitude or sign bits do not matter as much. This is also why the per-bit SDC probability for FLOAT16 is lower than that for FLOAT. A corrupted bit in the exponent of the latter is likely to cause a larger deviation from 0, which in turn is likely to result in an SDC.

For FxP data types, we plot the results for CaffeNet using 32b_rb26 and 32b_rb10 in Figure 8.3C and Figure 8.3D respectively. As can be seen, only bits in the integer parts of the fixed point data types are vulnerable. Both FxP data types have 32-bit data widths but different binary point positions. We observed that the per-bit SDC probability for the data type 32b_rb10 is much higher than that of 32b_rb26. For example, the 30th bit in 32b_rb10 has an SDC probability of 26.65% whereas the same bit position only has an SDC probability of 0.22% in 32b_rb26. This is because 32b_rb26 has a smaller dynamic range of values compared to 32b_rb10, and hence the corrupted value in the former is likely to be closer to 0 than the latter. This is similar to the FP case above.

### Value

We chose AlexNet using FLOAT16 to explain how errors that result in SDCs affect program values. We randomly sampled the set of ACTs in the network that were

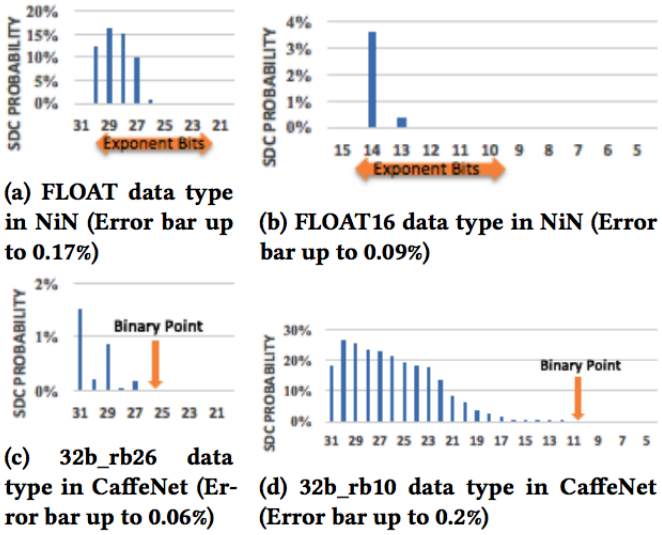
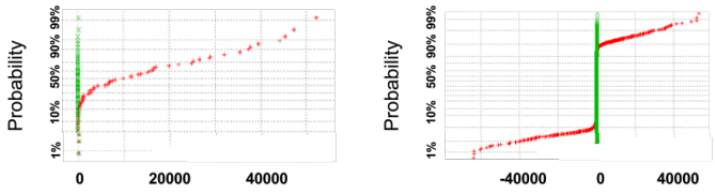



**Figure 8.3:** SDC probability variation based on bit position corrupted, bit positions not shown have zero SDC probability (Y-axis is SDC probability and X-axis is bit position)

affected by errors, and compared their values before (in green) and after (in red) error occurrence. We classified the results based on whether the errors led to SDCs (Figure 8.4A) or were benign (Figure 8.4B). There are two key observations: (1) If an error causes a large deviation in numeric values, it likely causes an SDC. For example, in Figure 8.4A, more than 80% of errors that lead to a large deviation lead to an SDC. (2) In Figure 8.4B, on the other hand, only 2% of errors that cause large deviations result in benign faults. This is likely because large deviations make it harder for values in the network to converge back to their correct values which are typically clustered around 0.

We now ask the following questions. How close together are the correct (error-free) values in each network, and how much do the erroneous values deviate from the correct ones? Answering these questions will enable us to formulate efficient error detectors. We list boundary values of ACTs profiled in each layer and network in Table 8.4. As can be seen, in each network and layer, the values are bounded within a relatively small range in each layer. Further, in the example of




**(a) Values before (green dots) and after (red dots) errors resulting in SDCs. X-axis represents values.**

**(b) Values before (green dots) and after (red dots) errors resulting in benign. X-axis represents values.**

**Figure 8.4:** Values before and after error occurrence in AlexNet using FLOAT16

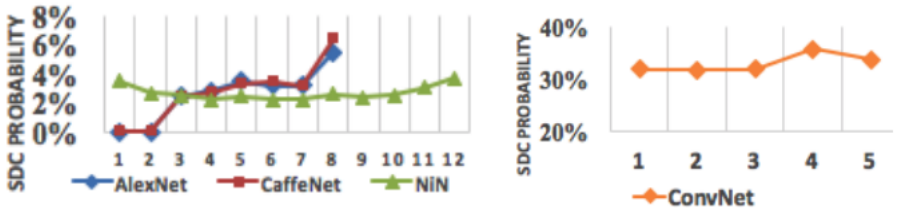
AlexNet using FLOAT16, 80% of the erroneous values that lead to SDCs lie outside this range, while only 9.67% of the erroneous values that lead to benign outcomes do so. Similar trends are found in AlexNet, CaffeNet, and NiN using DOUBLE, FLOAT, FLOAT16, and 32b_rb10. This is because the data types provide more dynamic value range than the networks need. The redundant value ranges lead to larger value deviation under faults and are more vulnerable to SDCs. This indicates that we can leverage symptom-based detectors to detect SDCs when these data types are used (Section 8.5.2). On the other hand, 16b_rb10 and 32b_rb26 suppress the maximum dynamic value ranges. ConvNet is an exception: ConvNet has a limited number of outputs and a small stack of layers, as even a small perturbation in values may significantly affect the output rankings.

**Table 8.4:** Value range for each layer in different networks in the error-free execution

| Network | Layer 1 | Layer 2 | Layer 3 | Layer 4 | Layer 5 | Layer 6 | Layer 7 | Layer 8 | Layer 9 | Layer 10 | Layer 11 | Layer 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AlexNet | -691.813 ~662.505 | -228.296 ~224.248 | -89.051 ~98.62 | -69.245 ~145.674 | -36.4747 ~133.413 | -78.978 ~43.471 | -15.043 ~11.881 | -5.542 ~15.775 | N/A | N/A | N/A | N/A |
| CaffeNet | -869.349 ~608.659 | -406.859 ~156.569 | -73.4652 ~88.5085 | -46.3215 ~85.3181 | -43.9878 ~155.383 | -81.1167 ~38.9238 | -14.6536 ~10.4386 | -5.81158 ~15.0622 | N/A | N/A | N/A | N/A |
| NiN | -738.199 ~714.962 | -401.86 ~267.8 | -397.651 ~388.88 | -1041.76 ~875.372 | -684.957 ~1082.81 | -249.48 ~244.37 | -737.845 ~940.277 | -459.292 ~584.412 | -162.314 ~437.883 | -258.273 ~283.789 | -124.001 ~140.006 | -26.4835 ~88.1108 |
| ConvNet | -1.45216 ~1.38183 | -2.16061 ~1.71745 | -1.61843 ~1.37389 | -3.08903 ~4.94451 | -9.24791 ~11.8078 | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

### Layer Position and Type

To investigate how errors in different layers propagate through the network, we study the error sensitivity of both the positions and types of layers. We show the SDC probability per layer, ordered by the position for AlexNet, CaffeNet, and NiN in Figure 8.5A and for ConvNet in Figure 8.5B.




**(a) AlexNet, CaffeNet and NiN (Error bar is up to 0.39%)** **(b) ConvNet (Error bar is up to 0.77%)**

**Figure 8.5:** SDC probability per layer using FLOAT16, Y-axis is SDC probability and X-axis is layer position

In Figure 8.5A, in AlexNet and CaffeNet, we observed very low SDC probabilities in the first and second layers, compared to the other layers. The reason is the Local Response Normalization (LRN) layers implemented at the end of the first

and second layers in these networks normalize the faulty values, thus mitigating the effect of large deviations in the values. However, there is no LRN or similar normalization layer in the other layers. NiN and ConvNet do not have such a normalization layer, and hence, NiN (in Figure 8.5A) and ConvNet (in Figure 8.5B) have a relatively flat SDC probability across all convolutional layers (layers 1 to 12 in NiN and layer 1 to layer 3 in ConvNet). We also observe there is an increase in the SDC probabilities in layers in AlexNet and CaffeNet after the LRNs. This is because the later layers require narrower value ranges (Table 8.4), and hence wider value ranges and bits are likely to result in SDCs. Note that the fully-connected layers in AlexNet (layer 6 to layer 8), CaffeNet (layer 6 to layer 8), and ConvNet (layers 4 and 5) have higher SDC probabilities. This is because (1) they are able to directly manipulate the ranking of the output candidates, and (2) ACTs are fully-connected, and hence faults spread to all ACTs right away and have much higher impact on the final outputs. Recall that there is no fully-connected layer in NiN, however, and hence this effect does not occur.

To further illustrate the effect of LRN on mitigating error propagation, we measured the average Euclidean distance between the ACT values in the fault injection runs and the golden runs at the end of each layer after faults are injected at the first layer in different networks using the DOUBLE data type. The results are shown in Figure 8.6. We choose the DOUBLE data type for this experiment as it accentuates any differences due to its wide value range. As we can see, the Euclidean distance decreases sharply from the first layer to the second layer after LRN in AlexNet and CaffeNet. However, neither NiN nor ConvNet implement LRN or similar layers, and hence the Euclidean distances at each layer are relatively flat.

Recall that the POOL and ReLU layers are implemented in all four networks we consider, and are placed at the end of each convolutional and fully-connected layer after MACs. Recall also that POOL picks the local maximum ACT value and discards the rest of the ACTs before forwarding them to the next layer, while ReLU resets values to zero if the value is negative. Since POOL and ReLU can either discard or completely overwrite the values, they can mask errors in different bits. Therefore, we study the bit-wise SDC probability (or error propagation rate) per layer after each POOL or ReLU structure in convolutional layers in Table 8.5. We measured this rate by comparing the ACT values bit by bit at the end of the

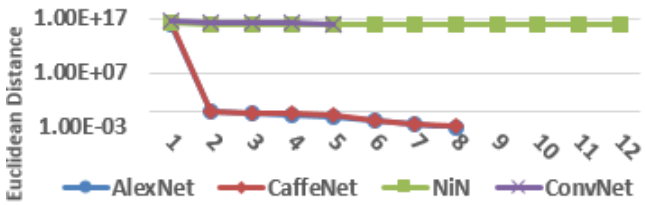



**Figure 8.6:** Euclidean distance between the erroneous values and correct values of all ACTs at each layer of networks using DOUBLE, Y-axis is Euclidean distance and X-axis is layer position (Faults are injected at layer 1)

last layer. Due to space constraints, we only show the result of AlexNet using FLOAT16, though similar results were observed in the other cases.

**Table 8.5:** Percentage of bit-wise SDC across layers in AlexNet using FLOAT16 (Error bar is from 0.2% to 0.63%)

| Layer 1 | Layer 2 | Layer 3 | Layer 4 | Layer 5 |
|---|---|---|---|---|
| 19.38% | 6.20% | 8.28% | 6.08% | 1.63% |

There are three main observations in Table 8.5: (1) In general, there is a decreasing propagation probability across layers, as faults that occur in earlier layers have a higher probability of propagating to other layers and spreading. (2) Even through many faults spread into multiple locations and reach the last layer, only a small fraction of them (5.5% on average, compared with AlexNet in Figure 8.5A) will affect the final ranking of the output candidates. The reason, as explained in Section 8.4.1, is that the numerical value of ACTs affected by faults is a more influential factor affecting the SDC probability than the number of erroneous ACTs. (3) A majority of the faults (84.36% on average) are masked by either POOL or ReLU during the propagation and cannot even reach the last layer. Therefore, error detection techniques that are designed to detect bit-wise mismatches (i.e., DMR) may detect many errors that ultimately get masked.

### Datapath FIT rate

We calculate the datapath FIT rates for different networks using each datatype based on the canonical model of datapath in Figure 3.1B, and the formula in

Eq. 8.1. Note that the latches assumed in between execution units are the minimum sets of latches to implement the units, so our calculations of datapath FIT rate are conservative. The results are listed in Table 8.6. As seen, the FIT rate varies a lot depending on the network and data type used. For example, it ranges from 0.004 to 0.84 in NiN and ConvNet using 16b_rb10, and from 0.002 to 0.42 in AlexNet using 16b_rb10 and 32b_rb10. Depending on the design of the DNN accelerator and the application, the datapath's FIT rate may exceed the FIT budget allowed for the DNN accelerator and will hence need to be protected. We will further discuss this in Section 8.5.1.

**Table 8.6:** Datapath FIT rate in each data type and network

| | **ConvNet** | **AlexNet** | **CaffeNet** | **NiN** |
|---|---|---|---|---|
| FLOAT | 1.76 | 0.02 | 0.03 | 0.10 |
| FLOAT16 | 0.91 | 0.009 | 0.009 | 0.008 |
| 32b_rb26 | 1.73 | 0.002 | 0.005 | 0.002 |
| 32b_rb10 | 2.45 | 0.42 | 0.41 | 0.54 |
| 16b_rb10 | 0.84 | 0.002 | 0.007 | 0.004 |

### 8.4.2 Buffer Faults: A Case Study on Eyeriss

Eyeriss [35] is a recently proposed DNN accelerator whose component parameters are publicly available. We use Eyeriss in this case study - the reason is articulated in Section 8.3.2. Other than the components shown in Figure 3.1, Eyeriss implements a *Filter SRAM, Img REG* and *PSum REG* on each PE for data reuses listed in Table 3.1. The details of the dataflow are described in Chen et al. [35]. We adopted the original parameters of the Eyeriss microarchitecture at 65 nm and projected it to 16nm technology. For the purpose of this study, we simply scale the components of Eyeriss in proportion to process technology generation improvements, ignoring other architectural issues. In Table 8.7, we list the microarchitectural parameters of Eyeriss at the original 65nm and the corresponding projections at 16nm. We assume a scaling factor of 2 for each technology generation, and as there are 4 generations between 65nm and 16nm technology (based on published values by the TSMC foundry), we scaled up the number of PEs and the sizes of buffers by a factor of 8. In the rest of this work, we use the parameters of Eyeriss at 16nm.

**Table 8.7:** Parameters of microarchitectures in Eyeriss (Assuming 16-bit data width, and a scaling factor of 2 for each technology generation)

| Feature Size | No. of PE | Size of Global Buffer | Size of One Filter SRAM | Size of One Img REG | Size of One PSum REG |
|---|---|---|---|---|---|
| 65nm | 168 | 98KB | 0.344KB | 0.02KB | 0.05KB |
| 16nm | 1,344 | 784KB | 3.52KB | 0.19KB | 0.38KB |

### Data Reuse and Buffer

Here we measure and compare SDC probabilities of different buffers in Eyeriss by randomly injecting 3,000 faults in each buffer component for different network parameters. By analyzing the results, we found that faults in buffers exhibit similar trends as datapath faults for each parameter of data type, network, value, layer, though the absolute SDC probabilities and FIT rates are different (usually higher than for datapath faults due to the amount of reuse and size of the components). Hence, we do not repeat these sensitivity results in this section. Rather, we present SDC probabilities and FIT rates for each buffer of the different networks in Table 8.8. The calculation of the FIT rate is based on Eq. 8.1. We show the results using the 16b_rb10 data type as a 16-bit FxP data type is implemented in Eyeriss.

**Table 8.8:** SDC probability and FIT rate for each buffer component in Eyeriss (SDC probability / FIT Rate)

| Network | Global Buffer | Filter SRAM | Img REG | PSum REG |
|---|---|---|---|---|
| ConvNet | 69.70%/87.47 | 66.37%/62.74 | 70.90%/3.57 | 27.98%/2.82 |
| AlexNet | 0.16%/0.20 | 3.17%/3.00 | 0.00%/0.00 | 0.06%/0.006 |
| CaffeNet | 0.07%/0.09 | 2.87%/2.71 | 0.00%/0.00 | 0.17%/0.02 |
| NiN | 0.03%/0.04 | 4.13%/3.90 | 0.00%/0.00 | 0.00%/0.00 |

As seen, as the network becomes deeper, its buffers are much more immune to faults. For example, ConvNet is less deep than the other three networks, and the FIT rate of *Global Buffer* and *Filter SRAM* are respectively 87.47 and 62.74, compared to 0.2 and 3.0 for AlexNet. This sensitivity is consistent with datapath faults. Another observation is that *Img REG* and *PSum REG* have relatively low

FIT rates as they both have smaller component sizes and a short time window for reuse: a faulty value in *Img REG* will only affect a single row of fmap and only the next accumulation operation if in *PSum REG*. Finally, we find that buffer FIT rates are usually a few orders of magnitude higher than datapath FIT rates, and adding these buffers for reuse dramatically increases the overall FIT rate of DNN accelerators. The reasons are twofold: (1) Buffers by nature have larger sizes than the total number of latches in the datapath, and (2) due to reuse, the same fault can be read multiple times and lead to the spreading of errors to multiple locations in a short time, resulting in more SDCs. Both lead to a higher FIT rate (See in Eq. 8.1). We will further discuss its implication in Section 8.5.1.

## 8.5 Mitigation of Error Propagation

We explore three directions to mitigate error propagation in DNN systems based on the results in the previous section. First, we discuss the reliability implications in designing DNN systems. Second, we adapt a previously proposed software based technique, *Symptom-based Error Detectors (SED)*, for detecting errors in DNN-based systems. Finally, we use a recently proposed hardware technique, *Selective Latch Hardening (SLH)* to detect and correct datapath faults. Both techniques leverage the observations made in the previous section and are optimized for DNNs.

### 8.5.1 Implications to Resilient DNN Systems

(1) **Data Type:** Based on our analysis in Section 8.4.1, we can conclude that DNNs should use data types that provide just-enough numeric value range and precision required to operate the target DNNs. For example, in Table 8.6, we found that the FIT rate of datapath can be reduced by more than two orders of magnitude if we replace type 32b_rb10 with type 32b_rb26 (from 0.42 to 0.002).

However, existing DNN accelerators tend to follow a *one-size-fits-all* approach by deploying a data type representation which is long enough to work for all computations in different layers and DNNs [34, 35, 49]. Therefore, it is not always possible to eliminate redundant value ranges that the data type provides across layers and DNNs. The redundant value ranges are particularly vulnerable to SDCs

as they tend to cause much larger value deviations (Section 8.4.1). We propose a low-cost symptom-based error detector that detects errors caused by the redundant value ranges regardless of whether conservative data types are used. A recent study has proposed a reduced precision protocol which stores data in shorter representations in memory and unfolds them when in the datapath to save energy [77]. The approach requires hardware modifications and may not be always supported in accelerators. We defer the reliability evaluation of the proposed protocol to our future work.

(2) **Sensitive Bits:** Once a restricted data type is used for a network, the dynamic value range is suppressed, mitigating SDCs caused by out-of-range values. However, the remaining SDCs can be harder to detect as erroneous values hide in normal value ranges of the network. Fortunately, we observe that the remaining SDCs are also caused by bit-flips at certain bit positions (i.e., high bit positions in FxP) (Section 8.4.1). Hence, we can selectively harden these bits to mitigate the SDCs (Section 8.5.3).

(3) **Normalization Layers:** The purpose of the normalization layers such as the LRN layer is to increase the accuracy of the network [81]. In Section 8.4.1, we found that LRN also increases the resilience of the network as it normalizes a faulty value with its adjacent fault-free values across different fmaps to mitigate SDCs. Therefore, one should use such layers if possible. Further, one should place error detectors after such layers to leverage the masking opportunities, thus avoiding detecting benign faults.

(4) **Data Reuse:** Recently, multiple dataflows and architectures have been proposed and demonstrated to provide both energy and performance improvements [34, 35]. However, adding these local buffers implementing more sophisticated dataflow dramatically increases the FIT rate of a DNN accelerator. For example, the FIT rate of *Filter SRAM*s (3.9 in NiN, Table 8.8) can be nearly 1000x higher than the FIT rate of the entire datapath (0.004 in NiN, Table 8.8). Unfortunately, however, protecting small buffers through ECC may incur very high overheads due to smaller read granularities. We propose an alternative approach, SED, to protect these buffers at low cost (Section 8.5.2).

(5) **Datapath Protection:** From Table 8.6, we found the datapath FIT rate alone can go up to 2.45 without careful design, or 0.84 even with a resilient data

type (16b_rb10, in ConvNet). The safety standard, for example in ISO 26262, mandates the overall FIT rate of the SoC carrying DNN accelerator to be less than 10 FIT. Since a DNN accelerator is often a very small area fraction of the total on the SoC. The FIT budget allocated to the DNN accelerator should be only a tiny fraction of 10. Hence, datapath faults cannot be ignored as they stretch the FIT budget allocated to the DNN accelerator.

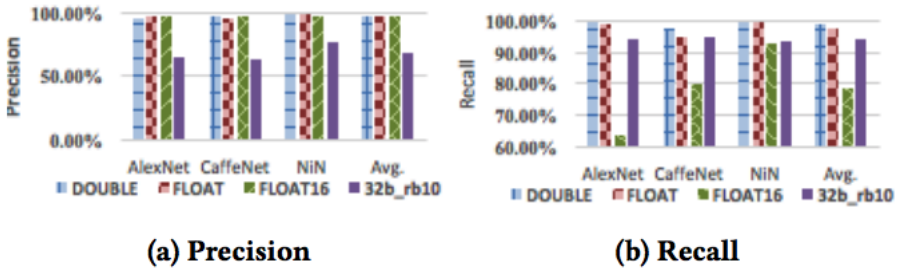
### 8.5.2 Symptom-based Error Detectors (SED)

A symptom-based detector is an error detector that leverages application-specific symptoms under faults to detect anomalies. Examples of symptoms are unusual values of variables [64, 108], numbers of loop iterations [64, 88], or address spaces accessed by the program [88, 108]. For the proposed detector, we use the value ranges of ACTs as the symptom to detect SDC-causing faults. This is based on the observation from Section 8.4.1: *If an error makes the magnitude of* ACTs *very large, it likely leads to an SDC, and if it does not, it is likely to be benign*. In the design of the error detector, there are two questions that need to be answered: *Where* (which program locations) and *What* (which reference value ranges) to check? The proposed detector consists of two phases - we describe each phase along with the answers to the two questions below:

**Learning:** Before deploying the detector, a DNN application needs to be instrumented and executed with its representative test inputs to derive the value ranges in each layer during the fault-free execution. We can use these value ranges, say -X to Y, as the bounds for the detector. However, to be safe, we apply an additional 10% cushion on top of the value ranges of each layer, that is (-1.1*X) to (1.1*Y) as the reference values for each detector to reduce false alarms. Note that the learning phase is only required to be performed once before the deployment.

**Deployment:** Once the detectors are derived, they are checked by the host which off-loads tasks to the DNN accelerator. At the end of each layer, the data of fmaps of the current layer are calculated and transferred to the global buffer from the PE array as the input data of ifmaps for the next layer. These data will stay in the global buffer during the entire execution of the next layer for reuse. This gives us an opportunity to execute the detector asynchronously from the host, and check the values in global buffer to detect errors. We perform the detection

asynchronously to keep the runtime overheads as low as possible.




**(a) Precision** **(b) Recall**

**Figure 8.7:** Precision and recall of the symptom-based detectors across networks (Error bar is from 0.03% to 0.04%)

**Evaluation:** After deploying the detector, we measure its coverage by randomly injecting 3,000 single bit-flip faults in each hardware component, using all 3 FP data types and 32b_rb10 in AlexNet, CaffeNet and NiN, one fault per fault injection run. As we explained earlier, we do not include the 16b_rb10 and 32b_rb26 data types or the ConvNet network as they do not exhibit strong symptoms, and hence symptom-based detectors are unlikely to provide high coverage in these cases. So there are a total of 3,000*5*4*3=180,000 faults injected for this experiment.

We define two metrics in our evaluation: (1) Precision: 1 - (The number of benign faults that are detected by the detector as SDC) / (The number of faults injected), and (2) Recall: (The number of SDC-causing faults that are detected by the detector) / (The number of total SDC-causing faults). The results are shown in Figure 8.7A and Figure 8.7B for the precision and the recall respectively averaged across the data types and components (due to space constraints, we only show the average values). As can be seen, the average precision is 90.21% and the average recall is 92.5%. Thus, we can reduce the FIT rates of Eyeriss using FLOAT and FLOAT16 by 96% (from 8.55 to 0.35) and 70% (from 2.63 to 0.79) respectively using the symptom-based detector technique (based on Equation 8.1).

### 8.5.3 Selective Latch Hardening (SLH)

Latch hardening is a hardware error mitigation technique that adds redundant circuitry to sequential storage elements (i.e., latches) to make them less sensitive to errors. Protecting the latches in the datapath can be vital for highly dependable systems as they become the reliability bottleneck once all buffers are protected (e.g., by ECCs). Given that the technology keeps scaling to smaller feature sizes [20, 40], it will be more important to mitigate datapath faults in the future. There have been a number of different hardened latch designs that differ in their overheads and levels of protection, and latch hardening need not be applied in an all-or-nothing manner. For example, Sullivan et al. [135] developed an analytical model for hardened latch design space exploration and demonstrated cost-effective protection by hardening only the most sensitive latches and by combining hardening techniques offering differing strengths in an error-sensitivity proportional manner. Since we observed and characterized asymmetric SDC sensitivity in different bits in Section 8.4.1, we can leverage this model to selectively harden each latch using the most efficient hardening technique to achieve sufficient error coverage at a low cost.

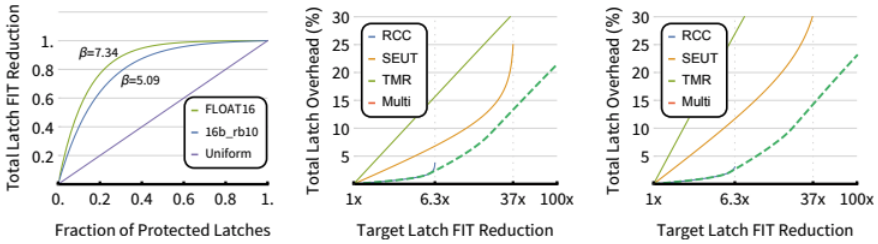
**Design Space Exploration:** There are a wide variety of latch hardening techniques that vary in their level of protection and overheads. Table 8.9 shows the three hardening techniques used by [135]; these same techniques are considered in this chapter, though the methodology should apply with any available hardened latches. The baseline design in the table refers to an unprotected latch.

**Table 8.9:** Hardened latches used in design space exploration

| **Latch Type** | **Area Overhead** | **FIT Rate Reduction** |
|---|---|---|
| Baseline | 1x | 1x |
| Strike Suppression (RCC) | 1.15x | 6.3x |
| Redundant Node (SEUT) | 2x | 37x |
| Triplicated (TMR) | 3.5x | 1,000,000x |

**Evaluation:** Figure 8.8A shows the AlexNet FIT rate reduction versus the fraction of protected latches assuming a perfect hardening technique that completely eliminates errors. We plot this curve to illustrate the maximum benefit one can achieve by protecting bits that are more sensitive to SDC with priority. $\beta$

characterizes the asymmetry of SDC FIT rate in different bits (latches)—a high $\beta$ indicates that a small number of latches dictate the overall SDC probability. As can be seen, there are negative exponential curves in the figure, such that we can selectively protect only the most sensitive latches for area-efficient SDC reduction.




**(a) FIT reduction versus the fraction of protected PE latches** **(b) Overhead versus FIT reduction using FLOAT16** **(c) Overhead versus FIT reduction using 16b_rb10**

**Figure 8.8:** Selective Latch Hardening for the Eyeriss Accelerator running AlexNet

Figure 8.8B and Figure 8.8C show the AlexNet FIT rate reduction versus the latch area overhead when using each hardened latch, and the optimal combination of the hardened designs (*Multi*) generated by the model. Due to space constraints, we only show the AlexNet result for the FLOAT16 and 16b_rb10 data types (the other networks and data types exhibit similar trends). By exploiting the asymmetry of error sensitivity in data type bits and combining complementary hardening techniques, one can achieve significant protection while paying only modest area costs. For example, applying the three hardening techniques together can reduce the latch FIT rate by 100x, while incurring about 20% and 25% latch area overheads in FLOAT16 and 16b_rb10, respectively. This translates to a chip-level area overhead roughly akin to that required for ECC protection of the larger (and more easily protected) SRAM structures.

## 8.6 Summary

DNNs (Deep Neural Networks) have gained prominence in recent years and they are being deployed on hardware accelerators in self-driving cars for real-time im-

age classification. In this chapter, we characterize the impact of soft errors on DNN systems through a large-scale fault injection experiment with 4 popular DNNs running on a recently proposed DNN hardware accelerator. We find that the resilience of a DNN system depends on the data types, values, data reuse, and the types of layers in the design. Based on these insights, we formulate guidelines for designing resilient DNN systems and propose two efficient DNN protection techniques to mitigate soft errors. We find that the techniques significantly reduce the rate of Silent Data Corruption (SDC) in DNN systems with acceptable performance and area overheads.

# Chapter 9

# Conclusion

In this chapter, we first summarize the dissertation and describe its expected impact. We then briefly delineate possible directions for future work.

## 9.1 Summary

We had two main goals in this dissertation. First, we wanted to understand how errors propagate in programs and lead to different types of failures. The understanding helped us come up with techniques that can evaluate programs' resilience and guide the protection of programs, which was our second goal. We considered both general programs (i.e, CPU programs) and the ones executing on hardware accelerators (i.e., GPUs and DNN accelerators). We applied both empirical and analytical approaches to achieve our goals. The dissertation had three parts as follow.

We first targeted an important but often neglected type of failure in CPU programs — LLCs. We conducted an empirical study in Chapter 4 to investigate error propagation that lead to LLCs, and then characterized the code patterns propagating the faults. We found that it was possible to identify these code patterns through program analyses, and to protect the code and eliminate LLCs in programs. Based on the above observation, we proposed a heuristic-based technique that was able to identify program locations that were responsible for more than 90% of LLCs in programs. The proposed technique pruned the fault injection space by more than 9

orders of magnitude compared with an exhaustive fault injection approach.

Secondly, we targeted SDCs, which are the most insidious type of failure, and are challenging to identify. We started our investigation in CPU programs as they are the most common applications. In Chapter 5, we explored an analytical approach to model error propagation that lead to SDCs in programs. Our proposed model is able to accurately estimate the SDC probabilities of both programs and individual instructions without any fault injections. The model can be also used to guide selective protection in a given program in a fast and accurate manner. In Chapter 6, we discussed how the error propagation can be affected by multiple program inputs, and extended the analytical model to support multiple inputs. We showed that the extended analytical model can be used to bound the SDC probability of a give program with multiple inputs without performing extensive fault injections.

Finally, we investigated error propagation in the applications that run on hardware accelerators such as GPUs and DNN accelerators (Chapter 7 and 8). Because these accelerators and applications have different architectures and programming models, we observed different error propagation patterns compared with CPU programs. We first built the tools that can be used to inject faults on the applications, then characterized their unique error propagation patterns. Based on the observations, we proposed error mitigation techniques that were specifically targeted to the accelerators and applications running on them, and hence more cost-effective than generic techniques.

## 9.2 Expected Impact

The first impact of this dissertation is to provide insights regarding identifying vulnerable program locations that lead to different types of failures, for application developers to evaluate programs' resilience and mitigate errors. Traditionally, vulnerable program locations were identified through extensive fault injection simulations which are extremely time-consuming. Because of this, developers were loathe to integrate resilience techniques into the software development process. We demonstrated (Chapter 4) that the code that lead to certain type of faults such as LLCs mostly fall into certain code patterns, which can be identified by program

analysis techniques. This insight became the driving force for the rest of the thesis. It inspired us to investigate program-level characterization of error propagation that lead to different types of faults on different platforms, which in turn allowed us to build automated techniques to identify the vulnerable parts of the program for the protection based on different reliability targets.

Furthermore, our research in Chapter 5, and 6 demonstrated that a systematic characterization of error propagation also enables us to build an analytical model to track error propagation in programs. The analytical model not only identifies the vulnerable parts that propagate errors, but also quantitively analyzes their propagation probabilities. Using TRIDENT, we showed that it is even possible to completely get rid of fault injection and hence significantly shorten the time taken in the whole evaluation process from a few days to a few minutes. The high-speed performance of the model and its capability of quantification imply that the technique may be integrated into compiler toolchains for developers to fine tune programs' resilience online.

Other than the practicability, the analytical model, TRIDENT and vTRIDENT, also revealed the detailed steps of error propagation. Traditionally, researchers reply on FI approaches to study error propagation. In FI, faults are repeatedly injected during the executions of programs, and the users wait util the end of the executions in order to observe failures, if any. This is a black-box technique, because it does not provide much insight into what happens during the propagation of the faults that are injected. Hence, for decades, researchers do not really have a solid understanding of error propagation, not to mention how to design highly cost-effective error detectors. In contrast, the analytical aspects of the models allow researchers to understand the details of error propagation and obtain detailed insights. Therefore, we believe that in the future, more efficient and cost-effective error detection and mitigation techniques can be developed based on the knowledge of error propagation characteristics.

Last but not least, this dissertation places the reliability problem of accelerator applications in the limelight of system research. In the past, researchers focused on the performance aspects of the accelerators because they were initially designed for performance. However, with the rising deployment of accelerators in safety-critical applications, the reliability of accelerators has started playing a much more

important role. As discussed in Chapter 7 and Chapter 8, hardware accelerators and applications can be vulnerable to hardware errors without protections. Our work has focused on building handy tools for studying error propagation in the accelerators and applications, and demonstrated that their error propagation has very different propagation patterns. We show it is possible to mitigate the error propagation in accelerators based on their unique characteristics in a cost-effective manner. We expect other researchers will leverage our tools to conduct reliability studies, and design more reliable accelerators and applications in the future.

## 9.3 Future Work

We propose four potential future work directions as follows.

**Direction 1: Modeling Error Propagation in GPU Programs**

In this thesis, CRASHFINDER, TRIDENT and VTRIDENT in Chapter 4, 5 and Chapter 6 focused on error propagation that lead to LLCs and SDCs in CPU programs. As mentioned in Chapter 7, the increasing error rate and rising popularity of GPUs accelerate the demand of developing fault-tolerant GPU programs. Since GPU programs typically contain hundreds of thousands of threads, they have a much larger space of fault injection sites compared with the CPU programs with similar lines of code. Hence, the resilience evaluation of GPU programs can be much more time-consuming in the development of fault-tolerant GPU applications [50, 104]. One way to speed up the evaluation process is to extend our models in this thesis to support GPU programs. LLFI-GPU introduced in Chapter 7 can be used to obtain further program-level insights of error propagation in GPU programs in order to achieve the goal. Those future techniques can be used to design cost-effective error detectors for GPU programs and integrated into GPU compiler toolchains to enable online tuning of GPU programs' resilience in the software development process.

**Direction 2: Pruning Profiling Space**

Another direction to extend our analytical models, TRIDENT and VTRIDENT, are through pruning profiling space in programs. As discussed in Chapter 5, the performance bottleneck in TRIDENT is the profiling phase which records a large amount of information of different program states etc. Studies [65, 104,

124] have shown that it is possible to select only a small subset of representative states to project the overall error resilience of a program. Hence, one direction to pursue is to find the subset of the representative states to profile when constructing the model, in order to speed up the model.

**Direction 3: Other Fault Models**

This dissertation mainly focused on the faults that occur in the data path of processors, which is one of the most challenging types of faults to detect and mitigate. Faults originating in memory are another major source. Current protection techniques leverage Error Correction Code (ECC) to mitigate memory faults. Since ECC memory ncurs non-negligible overheads in area, performance and energy consumption, they are mainly deployed in high-end systems such as supercomputers and aerospace applications. However, with increasing error rates, it becomes necessart to protect systems from memory faults in a tuneable and cost-effective way in future commodity systems. Therefore, selective protections at the levels of instruction, process, and applications need to be developed, which require one to understand error propagation caused by memory faults - this is an interesting future research direction.

**Direction 4: Secure-Enough Software Systems**

One of the main advantages of software error mitigation techniques is to provide flexible and selective protection in programs. Through selective protection, one can provide "reliable-enough" computation for commodity systems with high error coverage and low overheads, just like what demonstrated in this thesis. In contrast, today's security techniques are mostly designed to be either on and off, which may incur huge runtime overhead. We believe the idea of the reliable-enough computation can also be expanded to the security domain. For example, in one of the prevalent security attacks, row-hammer attack, malicious users can leverage hardware deficiencies of memory modules to flip bits in the logic values that are stored in memory, and manipulate the normal computation of programs [101]. These bit-flip errors, while similar to the soft error that we discussed in this dissertation, however, may have different impact on program executions and their consequences. Characterizing and understanding error propagation of the bit-flip errors introduced by row-hammer attacks in programs can be an interesting direction.

**Long-Term Direction: Formal Methods**

In this thesis, our proposed techniques guiding selective protection are based on empirical observations. As a result, they do not provide any guarantees on the error coverage and performance overhead of the protections. One way to provide such guarantees is to build a mathematically rigorous model of fault-tolerant applications, so that developers can not only verify the property of error resilience in a more formal way, but also use mathematical proof as a complement to test the efficiency of the protections and ensure their correct behaviors.

# Bibliography


[1] Parallel circuit solver. [Online; accessed Apr. 2016]. → page 115

[2] NVIDIA CUDA-GDB Documentation. [Online; accessed Apr. 2016]. → pages 18, 108

[3] NVIDIA Multi-GPU Programming. [Online; accessed Apr. 2016]. → page 115

[4] NVIDIA SDK Samples. [Online; accessed Apr. 2016]. → page 108

[5] Enabling on-the-fly manipulations with LLVM IR code of CUDA sources. https://github.com/apc-llc/nvcc-llvm-ir. [Online; accessed Apr. 2016]. → page 107

[6] IEEE standard for floating-point arithmetic. https://standards.ieee.org/findstds/standard/754-2008.html, 2008. IEEE Std 754-2008. → page 63

[7] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Goofi: Generic object-oriented fault injection tool. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 83–88. IEEE, 2001. → page 17

[8] H. M. Aktulga, J. C. Fogarty, S. A. Pandit, and A. Y. Grama. Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques. *Parallel Computing*, 38(4):245–259, 2012. → page 64

[9] Alippi, Cesare, Vincenzo Piuri, and Mariagiovanna Sami. Sensitivity to errors in artificial neural networks: A behavioral approach. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 1995. → page 13

[10] Amazon. Amazon s3 availability event: July 20, 2008. 2008. URL https://status.aws.amazon.com/s3-20080720.html. → page 1

[11] R. Ashraf, R. Gioiosa, G. Kestor, R. DeMara, C.-Y. Cher, and P. Bose. Understanding the propagation of transient errors in HPC applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis(SC)*. IEEE, 2015. → pages 104, 105, 107, 115, 119, 120

[12] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose. Understanding the propagation of transient errors in hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 72. ACM, 2015. → pages 17, 65

[13] Autonomous car facts. Keynote: Autonomous Car A New Driver for Resilient Computing and Design-for-Test, 2016. URL https://nepp.nasa.gov/workshops/etw2016/talks/15WED/20160615-0930-Autonomous_Saxena-Nirmal-Saxena-Rec2016Jun16-nasaNEPP.pdf. → page 22

[14] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson. A study of the impact of single bit-flip and double bit-flip errors on program execution. In *Computer Safety, Reliability, and Security*, pages 265–276. Springer, 2013. → page 117

[15] C. Basile, L. Wang, Z. Kalbarczyk, and R. Iyer. Group communication protocols under errors. In *22nd International Symposium on Reliable Distributed Systems.*, pages 35–44, Oct 2003. → page 24

[16] E. Battenberg and D. Wessel. Accelerating nonnegative matrix factorization for audio source separation on multi-core and many-core architectures. In *10th International Society for Music Information Retrieval Conference (ISMIR 2009)*, 2009. → page 115

[17] Bettola, Simone, and Vincenzo Piuri. High performance fault-tolerant digital neural networks. *IEEE transactions on computers*, 1998. → page 13

[18] C. Bienia, S. Kumar, J. P. Singh, and K. 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. → pages 25, 27, 38, 64, 83

[19] Bojarski, Mariusz, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel and others. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. → page 129

[20] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6): 10–16, 2005. → pages 1, 104, 130, 153

[21] S. Borkar. Electronics beyond nano-scale cmos. In *Proceedings of the 43rd annual Design Automation Conference*, pages 807–808. ACM, 2006. → page 1

[22] BVCL. BERKELEY VISION AND LEARNING CENTER, 2014. URL http://bvlc.eecs.berkeley.edu. → page 134

[23] Caffe Model. Caffe Model Zoo, 2014. URL http://caffe.berkeleyvision.org/model_zoo.html. → page 136

[24] CaffeNet. CaffeNet, 2014. URL http://caffe.berkeleyvision.org/model_zoo.html. → page 134

[25] J. Calhoun, L. Olson, and M. Snir. Flipit: An LLVM based fault injector for HPC. In *Euro-Par: Parallel Processing Workshops*, pages 547–558. Springer, 2014. → page 105

[26] J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998. → page 17

[27] Cavigelli, Lukas, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. Origami: A convolutional network accelerator. In *In Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, 2015. → pages 22, 135

[28] Chakradhar, Srimat, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, 2010. → pages 22, 135

[29] S. Chandra and P. M. Chen. How fail-stop are faulty programs? In *Fault-Tolerant Computing. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 240–249. IEEE, 1998. → page 9

[30] S. Chandra and P. M. Chen. The impact of recovery mechanisms on the likelihood of saving corrupted state. In *Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE.*, pages 91–101. IEEE, 2002. → page 46

[31] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez. Evaluating and accelerating high-fidelity error injection for hpc. In *Evaluating and Accelerating High-Fidelity Error Injection for HPC*, page 0. IEEE, 2018. → page 27

[32] Chatterjee, Avhishek, and Lav R. Varshney. Energy-reliability limits in nanoscale neural networks. In *The 51st Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6, 2017. → page 14

[33] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC 2009)*, pages 44–54. IEEE, 2009. → pages 52, 57, 64, 83, 114

[34] Chen, Tianshi, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, 2014. → pages 14, 20, 22, 130, 135, 149, 150

[35] Chen, Yu-Hsin, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016. → pages 14, 20, 21, 22, 131, 134, 135, 147, 149, 150

[36] Chen, Yunji, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li and others. Dadiannao: A machine-learning supercomputer. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014. → pages 22, 130, 135

[37] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller. Understanding soft error resiliency of Blue Gene/Q compute chip through hardware proton irradiation and software fault injection. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, November 2014. doi:10.1109/SC.2014.53. → page 117

[38] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10. IEEE, 2013. → pages 107, 117

[39] CIFAR dataset. CIFAR-10, 2014. URL https://www.cs.toronto.edu/~kriz/cifar.html. → page 134

[40] C. Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *Reliability and Maintainability Symposium*, pages 370–374. IEEE, 2008. → pages 1, 104, 130, 153

[41] ConvNet. High-performance C++/CUDA implementation of convolutional neural networks, 2014. URL https://code.google.com/p/cuda-convnet. → page 134

[42] J. J. Cook and C. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *International Conference on Dependable Systems and Networks(DSN)*, pages 482–491. IEEE, 2008. → pages 9, 15, 75, 101

[43] E. W. Czeck and D. P. Siewiorek. Observations on the effects of fault manifestation as a function of workload. *IEEE Transactions on Computers*, 41(5):559–566, 1992. → pages 10, 79, 81, 82

[44] Dahl, George E., Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1): 30–42, 2012. → page 129

[45] S. Di and F. Cappello. Adaptive impact-driven detection of silent data corruption for hpc applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2809–2823, 2016. → page 127

[46] D. Di Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson, and R. Johansson. On the impact of hardware faults–an investigation of the relationship between workload inputs and failure mode distributions. *Computer Safety, Reliability, and Security*, pages 198–209, 2012. → pages 11, 76, 79

[47] C. Di Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer. Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36. IEEE, 2015. → page 104

[48] M. Dimitrov, M. Mantor, and H. Zhou. Understanding software approaches for GPGPU reliability. In *Workshop on General Purpose Processing on Graphics Processing Units*, pages 94–104. ACM, 2009. → page 13

[49] Du, Zidong, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, 2015. → pages 22, 135, 149

[50] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. Gpu-qin: A methodology for evaluating the error resilience of GPGPU applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 221–230. IEEE, 2014. → pages 12, 104, 105, 108, 113, 115, 159

[51] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. ePVF: An enhanced program vulnerability factor methodology for cross-layer resilience analysis. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 168–179. IEEE, 2016. → pages 4, 9, 10, 17, 55, 58, 65, 69, 72, 73, 74, 75, 76, 77

[52] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 385–396. ACM, 2010. → pages 2, 9, 10, 12, 14, 15, 17, 24, 49, 50, 69, 72, 74, 76, 84, 101, 131, 138

[53] Fernandes, Fernando and Weigel, Lucas and Jung, Claudio and Navaux, Philippe and Carro, Luigi and Rech, Paolo. Evaluation of histogram of oriented gradients soft errors criticality for automotive applications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):38, 2016. → page 14

[54] P. Folkesson and J. Karlsson. The effects of workload input domain on fault injection results. In *European Dependable Computing Conference*, pages 171–190, 1999. → pages 11, 79, 81, 82

[55] A. Geist. How to kill a supercomputer: Dirty power, cosmic rays, and bad solder. ACM, 2016. → page 104

[56] Gill, B., N. Seifert, and V. Zia. Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node. In *Proceedings of the International Reliability Physics Symposium (IRPS)*, 2009. → page 136

[57] Gokhale, Vinayak, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 g-ops/s mobile coprocessor for deep neural

networks. In *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014. → pages 22, 135

[58] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2003. → pages 9, 24, 26, 104

[59] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan and Timothy Tsai. Modeling soft-error propagation in programs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018. → pages vi, 12, 81, 85, 91, 96, 98

[60] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015. → pages 22, 135

[61] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 44. ACM, 2017. → pages 10, 11

[62] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016. → pages 14, 20, 130

[63] S. Hari, T. Tsai, M. Stephenson, S. Keckler, and J. Emer. Sassifi: Evaluating resilience of GPU applications. In *SELSE: IEEE Workshop of Silicon Errors in Logic*. IEEE, 2015. → pages 12, 104, 105, 108, 113, 115

[64] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *Dependable Systems and Networks (DSN), 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012. → pages 2, 24, 46, 65, 131, 137, 138, 151

[65] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 123–134. ACM, 2012. → pages 2, 9, 12, 15, 50, 81, 115, 159

[66] S. K. S. Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi. Ganges: Gang error simulation for hardware resiliency evaluation. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 61–72. IEEE, 2014. → pages 9, 12, 17, 50, 81

[67] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015. → page 19

[68] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000. → pages 25, 27, 38, 64, 83

[69] M. Hiller, A. Jhumka, and N. Suri. Propane: an environment for examining the propagation of errors in software. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 81–85. ACM, 2002. → page 105

[70] https://asc.llnl.gov/CORAL-benchmarks/. Coral benchmarks. → page 84

[71] https://github.com/coExp/Graph. Github. → page 84

[72] https://github.com/karimnaaji/fft. Github. → page 84

[73] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. volume 100, pages 518–528. IEEE, 1984. → page 127

[74] IEC 61508. Functional Safety and IEC 61508, 2016. URL http://www.iec.ch/functionalsafety/. → page 130

[75] ImageNet. ImageNet, 2014. URL http://image-net.org. → page 134

[76] H. Jeon and M. Annavaram. Warped-DMR: Light-weight error detection for GPUGPU. In *IEEE/ACM International Symposium on Microarchitecture*, pages 37–47. IEEE Computer Society, 2012. → page 13

[77] Judd, Patrick, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. 2016. → pages 134, 150

[78] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on computers*, (2):248–260, 1995. → page 17

[79] I. Karlin. Lulesh programming model and performance ports overview. https://codesign.llnl.gov/pdfs/lulesh_Ports.pdf. [Accessed Apr. 2016]. → pages 64, 120

[80] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, et al. Lulesh programming model and performance ports overview. 2012. → page 115

[81] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012. → pages 19, 134, 150

[82] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson. Ipas: Intelligent protection against silent output corruption in scientific applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 227–238. ACM, 2016. → pages 65, 137

[83] Lane, Nicholas D., and Petko Georgiev. Can deep learning revolutionize mobile sensing? In *In Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, 2015. → page 131

[84] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. An empirical study of injected versus actual interface errors. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 397–408. ACM, 2014. → page 8

[85] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization. International Symposium on*, pages 75–86. IEEE, 2004. → pages 17, 25, 37, 50, 105

[86] LeCun, Yann, Koray Kavukcuoglu, and Clment Farabet. Convolutional networks and applications in vision. In *Proceedings of IEEE International Symposium on Circuits and Systems*, 2010. → page 18

[87] G. Li and K. Pattabiraman. Modeling input-dependent error propagation in programs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2018. → page vi

[88] G. Li, Q. Lu, and K. Pattabiraman. Fine-grained characterization of faults causing long latency crashes in programs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 450–461.

IEEE, 2015. → pages
v, 10, 12, 17, 81, 84, 101, 104, 107, 127, 131, 137, 151

[89] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose. Experience report: An application-specific checkpointing technique for minimizing checkpoint corruption. In *26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 141–152. IEEE, 2015. → pages
101, 104, 105, 125, 126, 127

[90] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose. Understanding error propagation in GPGPU applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 240–251. IEEE, 2016. → pages v, 9, 17, 69

[91] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 8. ACM, 2017. → page vi

[92] G. Li, K. Pattabiraman, and N. DeBardeleben. Tensorfi: A configurable fault injector for tensorflow applications. In *IEEE International Workshop on Software Certification (WoSoCer)*, 2018.

[93] Li, Guanpeng and Pattabiraman, Karthik and Cher, Chen-Yong and Bose, Pradip. Understanding error propagation in gpgpu applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2016. → pages
137, 138

[94] Lin, Min and Chen, Qiang and Yan, Shuicheng. Network in network. *arXiv preprint arXiv:1312.4400*, 2013. → page 134

[95] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 213–224. ACM, 2011. → pages 1, 2

[96] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman. LLFI: An intermediate code level fault injector for hardware faults. In *International Conference on Quality, Reliability and Security (QRS)*. IEEE, 2015. → page 117

[97] Q. Lu, G. Li, K. Pattabiraman, M. S. Gupta, and J. A. Rivers. Configurable detection of sdc-causing errors in programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(3):88, 2017. → pages 2, 4, 10, 12, 14, 15, 24, 50, 72, 73, 75, 84, 101

[98] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. V. Adve, D. Marinov, and S. Misailovic. Leveraging software testing to explore input dependence for approximate computing. *Workshop on Approximate Computing Across the Stack (WAX)*, 2017. → page 11

[99] G. B. Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 1(1):486–490, 1896. → pages 72, 101

[100] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 29–40. IEEE, 2003. → pages 4, 10

[101] O. Mutlu. The rowhammer problem and other issues we may face as memory becomes denser. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 1116–1121. European Design and Automation Association, 2017. → page 160

[102] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones. Scalable stochastic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 335–338. European Design and Automation Association, 2010. → page 2

[103] Neale, Adam, and Manoj Sachdev. Neutron radiation induced soft error rates for an adjacent-ECC protected SRAM in 28 nm CMOS. 2016. → page 138

[104] B. Nie, L. Yang, A. Jog, and E. Smirni. Fault site pruning for practical reliability analysis of gpgpu applications. 2018. → page 159

[105] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *Transactions on Reliability*, 51(1):111–122, 2002. → page 15

[106] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63–75, 2002. → pages 13, 106, 127

[107] Park, Seongwook, Kyeongryeol Bong, Dongjoo Shin, Jinmook Lee, Sungpill Choi, and Hoi-Jun Yoo. 4.6 a1. 93tops/w scalable deep learning/inference processor with tetra-parallel mimd architecture for big-data applications. In *International Solid-State Circuits Conference*. → pages 22, 135

[108] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-specific error detectors using dynamic analysis. volume 8, pages 640–655. IEEE, 2011. → pages 126, 151

[109] Peemen, Maurice, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for convolutional neural networks. In *IEEE 31st International Conference on Computer Design (ICCD)*, 2013. → pages 22, 135

[110] A. J. Peña, W. Bland, and P. Balaji. Vocl-ft introducing techniques for efficient soft error coprocessor recovery. In *International Conference for High Performance Computing, Networking, Storage and Analysis(SC)*, page 71. ACM, 2015. → page 13

[111] Piuri, Vincenzo. Analysis of fault tolerance in artificial neural networks. *Journal of Parallel and Distributed Computing*, 2001. → page 13

[112] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Modeling the propagation of intermittent hardware faults in programs. In *Dependable Computing (PRDC), IEEE 16th Pacific Rim International Symposium on*, pages 19–26. IEEE, 2010. → page 16

[113] Reagen, Brandon and Whatmough, Paul and Adolf, Robert and Rama, Saketh and Lee, Hyunkwang and Lee, Sae Kyu and Hernández-Lobato, José Miguel and Wei, Gu-Yeon and Brooks, David. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2016. → page 14

[114] D. A. Reed and J. Dongarra. Exascale computing and big data. volume 58, pages 56–68. ACM, 2015. → page 104

[115] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. 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. → page 13

[116] V. Reva. An efficient cuda implementation of the tree-based barnes hut n-body algorithm. Elsevier, 2014. → page 115

[117] V. Robert and X. Leroy. A formally-verified alias analysis. In *Certified Programs and Proofs*, pages 11–26. Springer, 2012. → page 47

[118] Russakovsky, Olga, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang and others. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3): 211–252, 2015. → page 129

[119] Safety Standard. ISO-26262: Road Vehicles Functional safety, 2016. URL https://en.wikipedia.org/wiki/ISO_26262. → pages 22, 130

[120] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011. → pages 1, 2

[121] B. Sangchoolie, K. Pattabiraman, and J. Karlsson. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In *International Conference on Dependable Systems and Networks (DSN)*, pages 97–108. IEEE, 2017. → pages 17, 64, 76

[122] Sankaradas, Murugan, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. A massively parallel coprocessor for convolutional neural networks. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2009. → pages 22, 135

[123] S. K. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mswat: low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 122–132. ACM, 2009. → page 27

[124] S. K. Sastry Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi. Ganges: Gang error simulation for hardware resiliency evaluation. *ACM SIGARCH Computer Architecture News*, 42(3):61–72, 2014. → pages 2, 160

[125] Seifert, Norbert, Balkaran Gill, Shah Jahinuzzaman, Joseph Basile, Vinod Ambrose, Quan Shi, Randy Allmon, and Arkady Bramnik. Soft error susceptibilities of 22 nm tri-gate devices. 2012. → page 136

[126] D. Shapiro. Introducing Xavier, the NVIDIA AI supercomputer for the future of autonomous transportation, 2016. URL https://blogs.nvidia.com/blog/2016/09/28/xavier/. → page 130

[127] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan. Towards formal approaches to system resilience. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 41–50. IEEE, 2013. → page 105

[128] Simonyan, Karen, and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. → page 134

[129] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, et al. Addressing failures in exascale computing. *Institute for Computing in Science (ICiS). More infor*, 4:11, 2012. → page 49

[130] V. Sridharan and D. R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *15th International Symposium on High Performance Computer Architecture*. → pages 9, 50, 55, 74, 76

[131] Sriram, Vinay, David Cox, Kuen Hung Tsoi, and Wayne Luk. Towards an embedded biologically-inspired machine vision processor. In *In Field-Programmable Technology*, 2010. → pages 22, 135

[132] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International*, pages 91–100. IEEE, 2000. → page 17

[133] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012. → pages 25, 27, 38, 114

[134] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908. → pages 67, 68

[135] Sullivan, Michael and Zimmer, Brian and Hari, Siva and Tsai, Timothy and Keckler, Stephen W. An analytical model for hardened latch selection and exploration. 2016. → page 153

[136] R. Taborda and J. Bielak. Large-scale earthquake simulation: computational seismology and complex engineering systems. *Computing in Science & Engineering*, 13(4):14–27, 2011. → page 64

[137] J. Tan, N. Goswami, T. Li, and X. Fu. Analyzing soft-error vulnerability on GPGPU microarchitecture. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 226–235. IEEE, 2011. → page 13

[138] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 193–204. IEEE Press, 2014. → page 111

[139] D. Tao, S. L. Song, S. Krishnamoorthy, P. Wu, X. Liang, E. Z. Zhang, D. Kerbyson, and Z. Chen. New-sum: A novel online abft scheme for general iterative methods. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 43–55. ACM, 2016. → page 11

[140] O. Temam. A defect-tolerant accelerator for emerging high-performance applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 356–367, 2012. → page 14

[141] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. In *Dependable Systems and Networks (DSN), 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013. → page 24

[142] Tiny-CNN. Tiny-CNN Framework. URL https://github.com/nyanp/tiny-cnn. → page 136

[143] Tithi, Jesmin Jahan, Neal C. Crago, and Joel S. Emer. Exploiting spatial architectures for edit distance algorithms. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014. → page 20

[144] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, et al. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 331–342. IEEE, 2015. → page 104

[145] TPU. Google supercharges machine learning tasks with TPU custom chip. URL https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html. → page 130

[146] N. J. Wang and S. J. Patel. Restore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):188–201, 2006. → page 24

[147] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Dependable Systems and Networks (DSN), 44rd Annual IEEE/IFIP International Conference on*, 2014. → pages 2, 4, 9, 17, 27, 37, 39, 64, 69, 76, 84, 105, 106, 110, 137, 138

[148] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995. → pages 25, 27, 38, 83

[149] Yann LeCun. Deep learning and the future of AI, 2000. URL https://indico.cern.ch/event/510372/. → page 130

[150] K. S. Yim, Z. T. Kalbarczyk, and R. K. Iyer. Quantitative analysis of long-latency failures in system software. In *Dependable Computing, PRDC'09. 15th IEEE Pacific Rim International Symposium on*, pages 23–30. IEEE, 2009. → pages 3, 4, 8, 16, 24, 26, 84, 104, 115, 125

[151] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *International Parallel & Distributed Processing Symposium (IPDPS)*, page 287. IEEE, 2011. → pages 12, 13, 84, 113

[152] L. Yu, Y. Zhang, X. Gong, N. Roy, L. Makowski, and D. Kaeli. High performance computing of fiber scattering simulation. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pages 90–98. ACM, 2015. → page 115

[153] Zhang, Chen, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *In Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015. → pages 22, 135