Characterization and Remediation for Soft Error Reliability on GPU

A Dissertation Presented
by

Fritz Gerald Previlon

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University
Boston, Massachusetts

August 2019

ProQuest Number: 22621894

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 22621894

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

To my wife and my family.

Contents

List of Figures

List of Tables

List of Acronyms

GPU Graphics Processing Unit. a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second. [4]. GPU's are now widely used for general purpose applications.

Acknowledgments

Here I wish to thank those who have supported me during the completion of this thesis. First of all, I would like to thank my advisor, Prof. David Kaeli, for his help, support, and advice throughout this process. I would also like to acknowledge Prof. Devesh Tiwari for his cooperation on key parts of this thesis. I would also like to thank Dr. Charu Kalra, our collaboration contributed to the refinement and improvement of this work.

Finally, I would like to thank my wife Michelle for her love, support, and encouragement while going through graduate school.

Abstract of the Dissertation

Characterization and Remediation for Soft Error Reliability on GPU

by

Fritz Gerald Previlon

Doctor of Philosophy in Computer Engineering

Northeastern University, August 2019

Dr. Kaeli, Advisor

Graphic Processing Units (GPUs) have become the accelerator of choice for improving the performance of many of the most demanding applications. While performance of these devices continues to improve generation after generation, reliability of these devices has not been studied rigorously. Several sources of errors can undermine the reliability of these devices, including radiation-induced transient faults, environmental perturbations, and process, temperature or voltage variations. In particular, transient faults in GPU execution have become a significant threat to high performance computing (HPC) and safety-critical applications. HPC systems experience transient faults every few tens of hours and the trend is expected to become worse.

A key point in the study of transient faults and their effects on user programs is that some faults do not cause undesirable results in the affected programs. This is especially true for GPU applications. Past efforts to study and understand GPU vulnerability to transient faults have demonstrated reliability can vary greatly across different applications. A significant amount of resilience resides intrinsically in some GPU applications. Transient faults in these applications are not likely to affect the results they produce. Other applications are highly sensitive to transient faults and are likely to crash or produce incorrect results when they are affected by transient faults.

While it is generally a good idea to protect the GPU hardware from transient faults, the penalties incurred in terms of performance, power and area are not always justifiable, depending on the applications utilizing the hardware resources. Understanding the relationship between the underlying program characteristics and their implications on vulnerability is crucial. The inherent resilience in applications should be carefully considered when making decisions about designing protection mechanisms to guard against nefarious transient faults.

In this thesis, we focus on program characteristics that contribute to their vulnerability. We offer several methodologies that aim at alleviating the prohibitively expensive process of quantifying

and estimating the vulnerability of GPU applications, as this is the first step toward improving the reliability of GPUs. Our analyses also demonstrate that beyond the variability between the vulnerabilities of different applications, the vulnerability of GPU applications also varies during their runtime, and present a phase behavior. This phase behavior opens new opportunities for not only more efficient vulnerability estimation, but also more efficient fault mitigation approaches. We demonstrate a methodology for application designers to reduce the cost of protection against transient faults.

Chapter 1

Introduction

The recent adoption of accelerators in high performance computing has resulted in tremendous gains in performance of a wide range of applications. Graphics Processing Units (GPUs) have been at the forefront of the field. GPUs provide much higher computational throughput as compared to CPUs, providing high memory bandwidth, better energy efficiency, and overall better power/performance. We have witnessed a steep rise in GPU deployments across a wide range of computing domains. GPUs are present in many of the Top500 computing platforms in the world [5], as well as in a growing number of general-purpose and safety-critical applications [6].

However, as with many breakthroughs, there are many obstacles to the continued adoption of GPUs for non-graphics, general-purpose, applications. GPU designers and researchers alike have devoted a significant amount of effort in finding solutions to these challenges. These challenges include a range of issues, including programmability, performance, energy efficiency and reliability. The last issue in this list is of particular concern, given that the reliable of graphic workloads is less of a concern. As grow the number of applications that are enjoying acceleration from GPU computing, there has been growing concerns related to the reliability of these devices [7, 8, 9, 10].

1.1 The growth of GPU Computing

As power and thermal constraints have made it very challenging to continue increasing clock frequencies of microprocessors, the microprocessor industry turned to parallelism in order to obtain higher performance. A number of new paradigms were explored and developed to facilitate the development of parallel applications [11]. Application developers started to leverage parallelism by utilizing middlewares, and targeting parallel hardware. Microprocessor manufacturers started delivering multi-core processors, ready to take advantage of inherent parallelism in the applications.

As multi-core processors grew in popularity, Graphics Processing Units (GPUs) were a natural next step, equipped with thousands of cores and streaming memory [12]. However, it was challenging to program a GPU due to the lack of programmable shaders. Programmers were forced to use graphics-oriented programming languages. But just in past 10 years, new programming languages were introduced, which greatly reduced programmer burden [13, 14] these devices have become an accelerator of choice for high-performance computing and other data-intensive applications.

GPUs are more effective than traditional CPUs in many applications where the workload is computation-bound, and where processing can be performed concurrently. In addition to high computational throughput, GPUs are able to access many memory locations in parallel, providing a high degree of memory parallelism. GPUs are able to hide memory latencies with fast context switching between threads.

1.2 Emerging GPU Applications

Beyond their graphics role, GPUs are now used in a growing range of computing domains, including high-performance computing [15, 16], automotive [6], and scientific supercomputing [5]. Given their inherent parallelism, GPUs are well-suited for safety-critical applications such as navigation, guidance, and image understanding [17].

GPUs are no longer only used as graphics engines, they have expanded their role to become an essential element in today's desktop computers and smartphones, accelerating a number of data intensive tasks. They are also used in datacenters, accelerating data mining, searches and queries, as well as in many financial and scientific applications.

1.3 The Growing Need for Reliability in Emerging GPU Applications

While GPUs are being aggressively deployed in a growing range of computing domains, their reliability remains an issue. More specifically, the reliability of a GPU to transient faults has been an area of increased concern. These transient faults manifest as bit flips in the hardware. Given that GPUs were primarily designed for graphic rendering, bit flips were not an issue. As pointed out by Sheaffer et al. [18], it is unlikely that a user will notice when a single-bit error in a single frame pixel is modified by the particle strike, even while playing a game or viewing a video.

For non-graphics applications however, bit flips are potentially more serious. For example, in scientific computing on high performance computing system, bit flips can lead to incorrect results of important scientific problems, or loss of computation time, as scientific applications can run for days or weeks. In the automotive industry, bit flips may lead to fatal accidents, crashes and even loss of life [19].

1.4 Motivation for this Thesis

This thesis addresses a growing reliability challenge in GPU computing: the GPU's ability to provide reliable execution in the presence of transient faults. Transient faults are intermittent malfunctions in hardware that manifest as bit flips. Transient faults have become a key challenge in computing and their importance increases with each new technology generation. They are caused by single event upsets, which originate from energetic particles such as neutron particles from cosmic rays and alpha particles from packaging materials. As these energetic particles pass through semiconductor devices, they generate electron-hole pairs. These charges can be collected by a transistor's source and diffusion regions. If the amount of accumulated charge is sufficient, it may invert the state of a logic device such as a latch, SRAM cell, or gate. This results in the introduction of a logical fault in the circuits operation [20]. This type of fault is called transient or soft because it does not result in permanent error in state or permanent damage to the device.

Reliability has long been an important focus in CPU design, but has not been heavily studied for GPU designs. However, recent studies have reported that computer systems with GPUs - where the GPU is used for general purpose computing - experience errors every few tens of hours [21, 22], and that the level of fault tolerance in general-purpose GPU applications varies significantly between applications [23, 3, 24]. Given that graphic applications and gaming have been the main driving force in the graphics card market, resilience to bit flips has not always been a primary concern in the design of GPUs. As pointed out by Wadden *et al.* [25], the sales for non-graphic GPUs account for only 5-8% of the NVIDIA's revenue. This makes it challenging to justify architectural changes to GPU design in order to accommodate general purpose computing, if these changes impact graphic applications. After all, a bit flip in a 3-D graphics program would only produce a wrong pixel, highly unlikely to disturb the flow of a graphic application. Furthermore, solutions and mitigation strategies for transient faults always come with additional overhead in either performance, area or power. Manufacturers, designers and GPU programmers need to consider a range of trade-offs when addressing GPU reliability, balancing efficiency/overhead and cost of the resulting design.

This thesis addresses a number of important aspects of GPU reliability, including:

1. characterization of the vulnerability of a wide range of GPU applications, and
2. mitigation of the effects of transient faults.

The first step in addressing the reliability problem is a thorough understanding of the level of vulnerability of GPUs, and the applications that run on them. While prior work has considered the issue of GPU reliability, a major drawback of these studies has been their use of lengthy fault-injection experiments to collect results, providing limited insight regarding the resilience characteristics of GPU applications [8, 22, 24, 26]. This thesis exploits unique properties of GPU applications to propose a novel fault injection method that reduces the number of fault injection runs needed, without sacrificing accuracy of the vulnerability assessment.

Furthermore, the same studies suggested that vulnerability of GPUs are highly correlated to the applications running on the GPUs. To mitigate the effects of transient faults run on GPUs, it is crucial to understand the resilience characteristics of GPU programs. This thesis studies the resilience properties of GPU applications and how vulnerability changes across program execution. We characterize the repetitive, time-varying behavior of vulnerability in these applications. We argue that these observations provide opportunities for designing more efficient resilience mitigation strategies for programs running on GPU systems.

1.5 Reliability Analysis for CPU vs. GPU

While radiation-induced faults have been well studied and understood in microprocessors, their impact on Graphic Processing Units (GPU) computations has received less attention. Methodologies for evaluating vulnerability in CPU applications [27, 28] work with single-threaded processes where one processing element operates on the data being computed by the application. GPU programs are multi-threaded programs where threads can 1) work independently to compute parts of a large chunk of an output data, 2) cooperate and communicate in the computation of the output data. GPU programs have an in-order execution where many threads execute the same set of instructions, with some occasional irregularities. There are a few cases in their execution where threads in an application may diverge, with one group of threads (or a single thread) executes a set of operations while another group executes a different set of operations. Moreover, threads in the same group may share data with other threads. This means that corrupted values in one thread may propagate to other threads in the program.

Reliability evaluation for GPUs needs to take into account the fact that different threads may present different vulnerability behaviors. Many times, threads may execute the same instructions, but with different input values. Moreover, the communication between threads is accomplished at a block level. For many GPU programs, the block size is adjustable and dictates how data is distributed between the streaming processors of the GPU. This can impact the performance and also the resilience of GPU programs, as the fault propagation between threads may be limited to fewer/larger number of threads. These considerations must be made when estimating vulnerability for GPU applications.

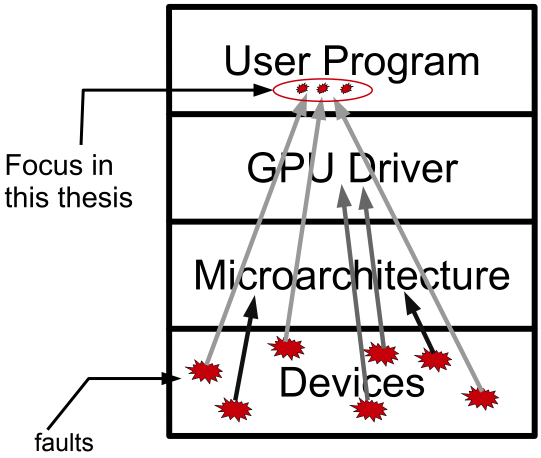


Figure 1.1: Layers of the system stack where transient faults can propagate in a GPU. Arrows indicate the extent of the propagation. In this thesis, we focus on faults that reach user programs.

1.6 Analysis of GPU Reliability

When a particle strike is incident on a computer system, causing a transient fault, it generally passes through many hardware/software before a user observes a failure in an application. In many cases, the transient fault resulting from the particle strike is masked and does not propagate to the application level. This masking can occur at different levels of a system stack, as shown in Figure 1.1.

Architecting resilience support requires a thorough understanding of the interaction between applications, system software, and the underlying hardware layer. This cross-layer reliability interaction is challenging to understand and fully characterize, since it is difficult to isolate the effects of one layer on other layers. For example, effects of transient faults are challenging to reproduce and vary significantly at each occurrence. The propagation of transient faults in hardware is heavily dependent on multiple layers including micro-architecture, system software, and application [27, 29].

Transient faults can sometimes lead to observable errors in program visible behavior, causing a program to crash, hang or produce an incorrect output (also referred to a Silent Data Corruption). In some cases, these faults can also remain unactivated (i.e., masked) depending upon the interaction among vulnerability layers in the system stack.

Researchers have begun to adopt multiple methods to assess the reliability of GPU devices, as well as the sensitivity of application workloads. For example, to assess the reliability of a hardware device, researchers and hardware manufacturers place the computing device into an electron beam of charged particles [30]. This methodology of reliability assessment allows to operate the software of interest on the real device and capture the interaction between underlying device, system software, and the application. While the neutron beam experiments potentially offer a more realistic picture of the effects of transient faults, they provide limited visibility into the application vulnerability behavior, one of the key factors toward understanding the interaction across the different hardware/software layers.

In this thesis, our focus is on vulnerability and remediation techniques that can be deployed at a software level, especially given that hardware-based remediation scheme typically come with signficant power and area costs, increasing the overall cost of the system.

Recent reliability studies on GPUs show that the overall impact of transient faults on GPU applications is highly correlated with the type of application running on the device [31, 3, 24]. In other words, some applications are inherently resilient and transient faults do not seem to impact them, while other applications are highly sensitive to any bit flips during their execution. Techniques to deal with transient faults exist, including special radiation-hardened circuit designs [32], localized error detection and correction [33], architectural redundancy [34, 35, 36, 37]. These techniques, however, introduce significant penalty in performance, power, die size, and design time. Resilience techniques introduced at a software level provide us with more flexibility and result in more efficient remediation solutions. A software-based scheme can be used adaptively (i.e., only when needed). Our studies uncover several interesting insights on how GPU program characteristics can affect the reliability of the underlying GPU device.

1.7 Scope and Contributions of this thesis

In this thesis, we focus on various aspects of the resilience characteristics in GPU applications. We specifically analyze the dependence of program resilience characteristics on their execution parameters, and their time varying behavior and how it correlates to program code execution. We propose a methodology for efficient assessment of vulnerability. Our end goal is to systematically and efficiently assess the resilience of GPU programs and provide robust remediation techniques.

1.7.1 Contributions

This thesis makes the following contributions:

- **We demonstrate that vulnerability is dependent on input and thread block sizes, as well as corner cases of input values.**

  We carry out an extensive fault injection campaign to characterize the vulnerability of a suite of GPU applications, when their input data changes. We found that a change in the input size of a program, as well as corner cases of biased input data values, can significantly affect program vulnerability. For example, the failure rate of some programs increased by as much as 30% when the input size was changed.

  As an example of a biased input data, the multiplication property of any value with a zero value (zero times any number is equal to zero) makes it a biased input for multiplication operations.

  When we examine the effects of changing the GPU thread-block size and its impact on vulnerability, we found that, similar to performance, the vulnerability of an application can depend on the block size of a kernel. In some applications, we found that the silent data corruption rate can vary by as much as 8% when changing the block size of a kernel.

- **PCFI: Automatic reduction of fault injection campaigns guided by program counter.**

  We propose a novel fault-injection method, *PCFI* [38], that automatically reduces the number of fault injections by exploiting the predictability in fault-injection outcome based on the program counter of the soft-error affected instruction. Evaluation on a variety of GPU programs covering a wide range of application domains shows that PCFI reduces the time to complete fault-injection campaigns by 22% on average, without sacrificing accuracy.

- **Characterization of time varying behavior of vulnerability in GPU applications.**

We show that the resilience characteristics of GPU programs change significantly during program execution and these characteristics show repetitive, time-varying behavior. Interestingly, these repetitive, time-varying, resilience characteristics of GPU programs do not align or correlate well with the performance phases of GPU programs. We support that phase changes in the vulnerability behavior during a program execution are due to changes in basic block execution paths. This allows us to capture these phases in a program within one single execution of the program.

- **Spoti-FI: Exploitation of the vulnerability time varying behavior for fault injection acceleration.**

  We demonstrate how characterization of the time-varying behavior of vulnerability can be exploited to accelerate a fault injection campaign for reliability assessment of GPU programs. We develop a methodology, *Spoti-FI*, which makes use of the characterization to reduce the number of fault injection experiments by an order of magnitude required in a campaign.

- **Exploitation of vulnerability phases for fault mitigation.**

  We present novel characterization data that also opens opportunities to design more effective resilience mitigation strategies. The characterization of the vulnerability phase behavior allows us to predict the vulnerability characteristics of time intervals of a program without performing exhaustive fault injections in all intervals of the program. We demonstrate how to use phase behavior characterization to dynamically enable or disable mitigation strategies in a program. This will significantly improve runtime overhead for mitigation strategies in GPU programs.

1.8 Organization of the Thesis

This thesis is organized as follows. In Chapter 2, we present a background on GPU computing as well as transient faults, and the traditional methodologies of reliability assessment for GPUs. We review the body of work and the state of the art research related to this thesis in Chapter 3. Chapter 4 describes our evaluation of the effects of execution parameters on resilience. In Chapter 5, we describe a methodology that reduces the number of faults for fault injections by using program counter execution in GPU programs. Chapter 6 details our characterization of the time-varying behavior of GPU program vulnerability, which constitutes the basis for our contributions to 1) systematically perform fault injections in selected intervals of a program (Chapter 7), and

2) alleviate the performance penalties of fault mitigation strategies by dynamically applying these strategies in executing GPU programs (Chapter 8).

Chapter 2

Background

In this chapter, we provide background information on *GPU computing* and the reliabilty challenges faced in GPU computing. We focus on a particular area plaguing GPU reliaiblity, specifically *transient faults* and the methods used to deal with these faults. We discuss techniques and paradigms used at the architectural level to assess the error rate of a processor, then we discuss the fault model as well as the evaluation framework we used throughout this thesis.

2.1 A brief history of GPU Computing

GPUs were originally designed to efficiently render 3-D graphics, providing highly optimized datapaths for generating frames of pixel data. The research community recognized that GPUs could also be used for massive data processing, and started migrating floating-point computations to the GPU using shader languages such as OpenGL and DirectX. The applications that were first ported to GPUs typically involved matrix-based operations. Matrix multiplication was one of the first microprocessor (Central Processing Units or CPU) programs that performed significantly better when run on a graphics card [39].

However, porting these general-purpose applications to GPUs was a very complex and daunting task, as it required that the programmers recast their algorithms in terms of the graphics Application Programming Interfaces (APIs). Industry leaders AMD and NVIDIA recognized this trend, and proposed general purpose programming languages that would allow GPUs to be used for a broader class of applications. OpenCL [14] and CUDA [13] have emerged as two standard programming frameworks that allow GPUs to be integrated in supercomputers and desktops as accelerators. Programmers were no longer tied to the underlying graphics programming model. They could focus more on high-performance computing, which attracted many more developers of general purpose applications to a GPU platform.

Experts from a wide variety of computing domains started converting their sequential CPU applications to GPU versions. One of the first ports of scientific computing to GPUs was the Monte Carlo simulation of photon migration by Alerstam *et. al* [40]. They achieved a speedup in the range of 1,000X over the CPU implementation by using a publicly available GPU (NVIDIA GeForce 8800GT). In bio-informatics, Schatz *et. al* [41] implemented the first high throughput parallel local sequence alignment (DNA sequencing), MUMerGPU, which runs on GPU and achieved more than a 10-fold speedup over a serial CPU version. Stone *et. al* [42] demonstrated the use of GPUs in molecular dynamics simulations, and showed that GPU-based calculations were typically 10-100 times faster than heavily optimized CPU-based implementations. In the financial domain, Grauer-Gray *et. al* [43] accelerated a popular library of computational finance applications, QuantLib, using hand-written GPU codes. They achieved orders of magnitudes of speedup over sequential implementations of the QuantLib applications.

Today, application developers use GPUs even in personal computers to accelerate ordinary applications. Popular desktop programs such as Adobe Photoshop, AutoCAD, SolidWorks and Matlab are exploiting this resource to accelerate their computations [11].

GPUs have also become prevalent in a number of computing systems. With the addition of error correcting codes to GPU structures in the NVIDIA's Fermi microarchitecture [12], GPUs were included in three of the top five supercomputers in 2010 [44]. Summit, an IBM-built supercomputer located at Oak Ridge National Laboratory, and the best performing supercomputer in the world as of the writing of this thesis, has 4,356 nodes, each one equipped with six NVIDIA Tesla V100 GPUs [5].

2.2 Challenges to GPU Computing

However, as mentioned in Chapter 1, there are many issues with GPU computing that need to be addressed to continue moving forward. These issues mostly pertain to the management of the large number of cores in one device, as both manufacturers and programmers seek to efficiently leverage these highly parallel devices for better performance, reliability and energy efficiency.

For example, it is increasingly difficult to design efficient programming models and system software that are able to completely leverage the opportunities offered by the numerous cores in a GPU. Moreover, GPU concurrency contains many nuances that make it difficult to correctly optimize

GPU code. These challenges include data races between threads, unfair scheduling across thread blocks and floating point accuracy [45].

Another important challenge in GPU computing is thread synchronization. As GPU programs become more complex, communication between the thousands of threads executing on the GPU becomes more essential. Newer GPU programs now require that individual threads collaborate more frequently to perform their computations, and accomplishing this in current GPUs is very challenging.

As GPUs are increasingly utilized in the large data-centers and supercomputers [44, 5], their energy consumption becomes a concern as system administrators look to reduce cooling costs. The US Department of Energy has a goal of 20 MW for an exa-scale supercomputer. It was estimated that Tianhe-2, the fastest supercomputer as of 2015, required at least a 26-fold improvement in power efficiency, if its performance were to scale with its power requirements [46]. While the current top supercomputers have better power efficiency, their current power requirement does not scale well to stay within the energy budget at exa-scale. Moreover, battery life is a crucial concern for handheld devices which utilize GPUs.

An often overlooked problem in computing devices, and especially GPUs is the problem of reliability to transient faults. While these faults rarely occur in normal consumer systems, their probability of occurrence increases significantly as more and more GPU nodes are included in a system. One of the worst outcomes of a transient fault occurring in hardware is a silent failure: a transient fault may cause a program to output an incorrect result without the knowledge of an end user. This is extremely undesirable for accuracy-sensitive and safety-critical applications. This thesis addresses the problem of transient faults in GPU applications.

2.3 Overview of Transient Faults

Transient faults are intermittent malfunctions of the hardware that cannot be reproduced. Transient faults are dynamic and are changes to a cell's contents, rather than changes in the circuitry. *Transient faults* can be generated by a plethora of causes, including environmental perturbations, software errors, and variations in process, temperature, or voltage.

In this thesis we focus on *radiation-induced transient faults*, which have been shown to be the most critical type of faults in modern electronic devices, given that they account for most of the faults in today's computing systems and produce a failure rate that is higher than all the other reliability mechanisms combined [47]. They are caused by single event upsets (SEUs) which are most often the result of particle strikes on silicon devices. These particles include high-energy neutrons, produced by the interaction of cosmic rays within the terrestrial atmosphere, and alpha particles that are emitted by the decay of radioactive impurities used in chip packaging.

Due to shrinking of transistor dimensions and the exacerbation of the amount of available resources, electronic devices are becoming more susceptible to transient faults induced by ionizing particles. As a result, today, high-energy neutrons generated by the interaction of cosmic rays with the terrestrial atmosphere are a major issue for the reliability of modern electronic devices [47] in general, and GPUs in particular [48, 49, 50, 51, 31].

When these strikes occur, the particles are able to inject a charge into the devices which can alter values in the devices. Each cell in a device has a minimum charge needed to change the stored value in the cell. This minimum charge is called the critical charge (Q_{crit}) for that cell. Following a particle strike, if the accumulated charge exceeds the critical charge of the cell, the value stored in the cell will be inverted; a transient fault occurs. Note that the cell containing this corrupted value is not permanently damaged and an operation can overwrite this cell to correct the faulty value. Since a radiation-induced fault does not cause a permanent failure, it is referred to as a soft fault, and is very challenging to detect.

2.4 The Transient Fault problem in the industry

While the occurrence of transient faults in one single device is very rare, clusters and datacenters with multiple CPU/GPU devices experience transient faults regularly. Moreover, the occurrence of one transient fault may be devastating for the affected system if proper fault tolerance is not implemented. The industry has seen firsthand the potential impact of transient faults.

Sun Microsystems lost a major customer to IBM after its flagship Enterprise server line experienced random crashes. In 2000, the company recognized that the crashes were caused by cosmic ray strikes on unprotected cache memories [52]. After the careful study of error logs of several large computer systems, Normand [53] reported numerous incidents of cosmic ray strikes.

To ensure highly reliable operation of their processor, Fujitsu protected 80% of its 200,000 latches in its 5th generation SPARC64 microprocessor with parity. All their caches were protected by some form of error detection [33].

In 2000, in the early days of Google, after one of their core systems had failed, the company was on the verge of losing a major contract with Yahoo. Google engineers figured out that the failures were caused by particle strikes. To keep costs down, the company did not use hardware with memory protection. However, as their computing systems scale, the occurrence of transient faults became more and more frequent [54].

2.5 Effects of Transient Faults

In this thesis, we want to clearly distinguish faults from errors. A fault is an undesired state change in hardware, and a fault in a particular layer in the computing stack may propagate to the next layer of the stack. Moreover, when a transient fault occurs in a bit, this bit can be overwritten to remove the fault. If the bit is not overwritten, the incorrect state that occurs as a consequence of this fault is termed an error. In this thesis, we focus on transient faults that manifest as errors at the user program level on the affected systems (see Section 1.1).

The occurrence of a transient fault can yield three outcomes in the program executing on the system, described as follows.

1. **Masked:** Some transient faults do not cause any perceivable error to the executing program. These faults are said to be *masked*. Masked outcomes occur when the occurrence of transient faults do not influence the output of a program. In other words, the output of a program in the presence of the transient fault is the same as the output in a fault-free environment. Transient faults produce *masked outcomes* when the location of the transient fault is: 1) not read or not used by the running program, 2) read, but corrected by hardware error correction mechanism (e.g., error correcting codes), or 3) read, but the value has no effect on the program output, i.e., the corrupted value was masked by subsequent operations after being read.

2. **Detected and Unrecoverable Error (DUE):** Some faults produce *Detected Unrecoverable Errors (DUE)*. A DUE occurs when a system is able to detect the presence of a fault, but can not correct or recover from this fault. The fault detection in a system can take place: 1) at the hardware level (through parity bits), 2) at the operating system level (e.g., when a transient fault causes a user program to request access to an unallocated memory location), or 3) at the user program level through verification checks in the program. In the case of a DUE, the hardware or software can decide to crash a system and restart the system from the last known clean, fault-free, state.

3. **Silent Data Corruption (SDC):** The third major class of outcome is a *Silent Data Corruption (SDC)*. An SDC occurs when the location of the transient fault was read and utilized by the

program, but the fault could not be detected. The faulty value eventually corrupts the dataflow of the program and produces an incorrect output. This outcome is called a silent corruption because both the user and the system are unaware that the program produced an incorrect output, until the output is verified against the golden output obtained via a fault-free execution.

2.6 Measuring Program Vulnerability to Transient Faults

Applications have different levels of resilience and exhibit different sensitivities to transient faults. It is important to understand the inherent levels of resilience in an application. In some applications, a fault may be more likely to generate an SDC. In other applications, a fault may be more likely to cause an DUE / crash. Other applications may be highly resilient, so that a fault in these applications is unlikely to generate any type of failure. Assessment of application resilience is an important step in developing highly resilient applications. To understand the inherent resilience based on program code, application developers need a way to quickly evaluate the vulnerability of their application. A developer can use either of the following techniques to measure the vulnerability of their software:

- Fault Injection [55], and
- Architecturally Correct Execution (ACE) analysis [56].

2.6.1 Fault Injection

Fault injection is the most widespread method for assessing reliability. A fault injection campaign compares the reference behavior of a system for a given workload (that is, the correct behavior validated by the designer) with the behavior obtained in the presence of each fault in a predetermined set of faults [57].

In a fault injection campaign, a fault is injected in a structure at a random time and at a random location, while a workload is being executed on the device being tested. The output of the workload is then examined against a golden output to determine whether the injected fault caused a visible failure. This process is then repeated a number of times and as the number of runs becomes statistically significant, the ratio between the number of failing runs to the total number of runs will be the probability that a fault reaching this structure will cause a failure in the program. Hardware fault injection and software fault injection are the most common approaches used to perform fault injection.

2.6.1.1 Statistical Significance

To determine the number of injections that is sufficient to achieve statistical significance, we utilize the formula presented by Leveugle *et al.* [57].

$$n = \frac{N}{1 + e^2 * \frac{N-1}{t^2 * p * (1-p)}} \tag{2.1}$$

According to the authors, given a confidence level, the sample size n, which is the number of faults to randomly select for injection, can be computed with the formula in 2.1. The variables in this formula are:

- N: initial population size. This is the number of all the potential injection sites.
- p: estimated probability of faults resulting in an error. The authors demonstrated that p=0.5 is a sufficient value to use in our experiments.
- e: margin of error. This is the most sensitive parameter in the formula. Reducing this parameter can increase the sample size very quickly. The margin of error is the amount of error that is allowed in case of miscalculations.
- t: cut-off point or confidence level. This number represents the level of confidence in our results, or the probability that our results are correct. For example, a 95% confidence level means that our results will be correct 95% of the time.

2.6.1.2 Hardware Fault Injection

The objective of performing hardware fault injection is to assess the resilience of hardware, with the end goal of designing more resilient hardware. For hardware fault injection, faults can be inserted either in the actual device silicon or in a simulated version of the device.

Injections in real device can be done by either using a dedicated custom hardware [58] or by injecting the faults into integrated circuits using heavy-ion radiation [30]. Radiation experiments help fully understand the impact of particle strikes on real hardware. Using beam testing, radiation-induced faults are generated across all layers of the computing stack.

Because the injection is done in actual hardware, it mimics the internals of the real system, meaning there is no need to know the internal details of the hardware. It is therefore very accurate; the effects of the operating system, the latency from I/O operations, and other non-deterministic effects are already taken into account. Furthermore, since injections are done in the actual hardware that is running the workloads, a fault injection campaign in actual hardware takes significantly less time than fault injection in simulated hardware.

However, there are also significant disadvantages to a fault injection campaign in real hardware. First, it needs to be done after the silicon phase in the design cycle, as we need at least a hardware prototype. This is usually too late considering that such reliability analysis is often needed during the architectural exploration phase of a design. However, the results can help make reliability decisions for future devices that use a similar technology or architecture. Second, it is very expensive and time-consuming to build a dedicated custom hardware and submit a hardware through an electron beam.

Injections in simulated hardware can be done in a performance simulator, which is usually available during the architectural exploration phase of a microprocessor design project. Therefore, the results of a software-implemented fault injection campaign can be used to influence the design of a new chip. Moreover, since we are using a software implementation of the hardware, we naturally have more visibility into the internals of the architecture under test.

However, simulated fault injection tends to be very slow compared to the execution of a workload on the native hardware. Moreover, it requires that an accurate performance simulator be available for the specific hardware under evaluation.

2.6.1.3 Software Fault Injection

Measuring program vulnerability implies quantifying the *masking* effects that are inherent in a program. In general, transient faults can be masked at many levels. Device-level masking reflects how likely it is for a fault not to propagate to the outputs of a device, and be exposed to the microarchitecture. Microarchitecture-level masking reflects how likely it is for a fault (not masked at device level) to propagate to the operating system. If the operating system does not mask this fault, it will be exposed to a user program. In this thesis, software fault injection aims to measure the vulnerability of programs in the presence of faults exposed to user programs.

A fault that is not masked by the microarchitecture of a system can manifest itself at an architecturally-visible resource. As defined by Sridharan *et al.* [28], an architecturally-visible resource is any ISA-visible structure or operation. Accordingly, architectural registers and memory that are addressable by the ISA, are considered architectural resources. Additionally, any structures that are part of the instruction datapath, such as the arithmetic logic unit or the load-store unit, can be regarded as an architectural resource.

A number of factors can influence the resilience properties of a program. One obvious factor is the type of operation that each instruction performs. Other factors that impact resilience directly correlate to the ordering of the instructions during the dynamic execution of the program. This ordering of instructions is often non-deterministic. A programmers choice of algorithm to implement a kernel and the compiler used to generate final binary are just two of the multiple components that influence the dynamic ordering and the types of the instructions executed.

Software fault injection can be done at different levels of the software stack (program source code, assembly instructions), depending on the goals of the analysis. The fault injection campaign used in this thesis is performed at native ISA level of the GPU (i.e., the SASS level of an NVIDIA GPU). This allows us to evaluate the the reliability of an application, as well as access any ISA-specific optimizations that have been performed to produce the application binary. Since the actual fault in hardware affect the final binary, working at this level should provide higher fidelity in terms of reliability metrics.

2.6.2 Architecturally Correct Execution (ACE) Analysis

ACE analysis was first described by Mukherjee *et al.* [56]. ACE analysis provides the ability to assess the *vulnerability* of individual pipeline structures, such as instruction queues and reorder buffers, to transient faults. Traditional ACE analysis is performed during execution-driven simulation, assessing the vulnerability of hardware structures by executing a single pass through a program.

In ACE analysis, the vulnerability of hardware structures is estimated by tracking the hardware state bits that are required for Architecturally Correct Execution (*ACE*). If any fault occurs in a storage cell containing these *ACE bits*, and if there is no error correction technique present on the system, there will be a visible error in the output of the program. The remaining state bits that are not ACE are called *un-ACE bits*; they are not required for architecturally correct execution of the program and a fault in a storage cell containing an un-ACE bit will not cause a visible error at the output of the program.

The vulnerability for a single-bit storage cell is the fraction of time it holds an ACE bit. Consequently, the vulnerability for a hardware structure is the average vulnerability of its storage cells. ACE analysis on a structure starts by conservatively assuming that all bits in the structure are ACE bits, then proceeds to identify bits that can be marked as un-ACE. Un-ACE bits can be categorized as either architectural or micro-architectural un-ACE bits. Examples of architectural un-ACE bits include bits from NOP instructions, performance-enhancing instructions (e.g., prefetches), predicated-false instructions, dynamically-dead code, and logical masking. Examples of microarchitectural un-ACE bits are idle or invalid bits, mis-speculated bits (wrong-path instructions or predictor structure bits), and microarchitecturally dead bits.

Because ACE analysis generates a conservative value for the vulnerability of a structure, the vulnerability estimation obtained through ACE analysis can very often be too conservative. It has been shown that even a refined ACE analysis can overestimate the error vulnerability of a structure by 2-3x [59]. This can result in an overly-conservative design of a structure, which can make the processor design less competitive. Furthermore, although ACE analysis gives more insight into the resilience of a structure, performing ACE analysis on certain structures can be a very involved process.

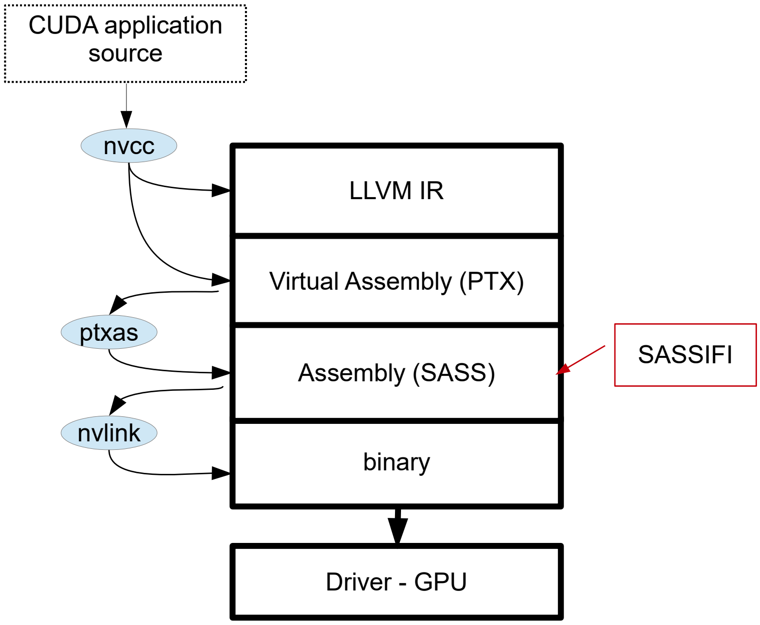
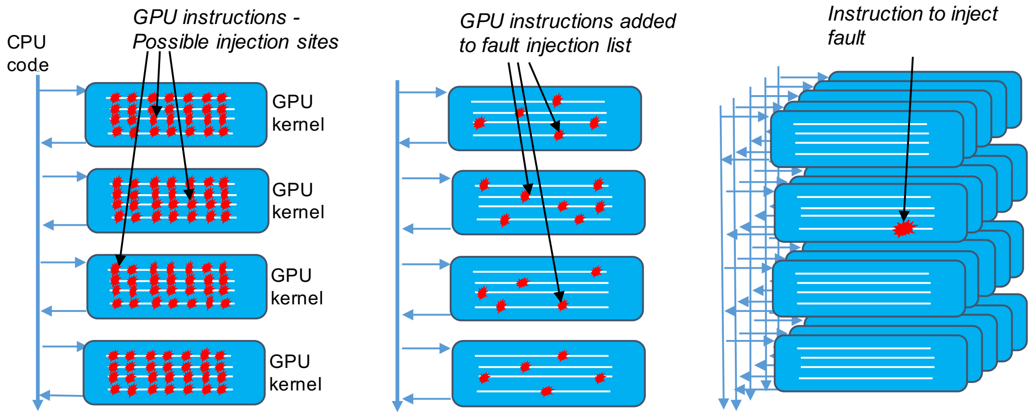


Figure 2.1: Stages during the compilation process. *ptxas* performs ISA-specific optimizations to generate SASS assembly code. The SASS assembly code directly runs on the GPU. SASSIFI performs fault injections at this level.

2.7 Evaluation Framework

In this section, we describe the framework used for vulnerability estimation throughout this thesis. We utilize SASSIFI [3], which is a widely used GPU software fault injection tool for evaluating the resilience of GPU programs. SASSIFI enables software-based fault injection on a real system by instrumenting the program at the assembly level. This methodology allows architects to assess the vulnerability of programs more accurately, as compared to approaches that perform fault injections at higher levels of the compilation process (see Figure 2.1), as reported by Tselonis *et al.* [60]. Injections at assembly level reflect the true effects of transient hardware faults.


CPU code
GPU instructions - Possible injection sites
GPU kernel
GPU kernel
GPU kernel
GPU kernel
GPU instructions added to fault injection list
Instruction to inject fault

(a) Profiling Stage (b) Fault list generation stage (c) Fault injection stage

Figure 2.2: Three stages of a fault injection campaign for an application in SASSIFI [3]

SASSIFI-based fault injection is built on top of SASSI [26], a compiler-based instrumentation framework which provides visibility into the application state, and retains information about the application state at the moment a fault is injected. Fault injection with SASSIFI consists of three stages: i) profiling, ii) fault generation, and iii) fault injection. Figure 2.2 provides an illustrated description of the three stages of SASSIFI.

1. **Profiling Stage** In the profiling stage, SASSIFI uses the compiler-based instrumentation framework (SASSI) to collect information about the application and identify all possible fault injection locations. The information collected includes the number of kernels in the application, the number of invocations for each kernel and the number of dynamic instructions

(per instruction opcode) for each invocation of a kernel. We note that the profiling stage is run only once for a chosen application-input pair.

Given that each dynamic instruction is a possible location for a fault injection, this stage gives us the population size of all fault injection locations. In order for a fault injection campaign to be statistically sound, it needs to inject a sufficiently high number of faults that is representative of all possible fault locations. The number of required fault injections to achieve statistical significance in terms of vulnerability assessment of an application depends on the target confidence interval, the error margin, and the total number of dynamic instructions [57]. Previous research has shown that a 99.8% target confidence interval and a 0.63% error margin yield accurate reliability estimates, although achieving this accuracy may require us to perform more than 60,000 injections, taking multiple days to complete fault injections on a single program [61, 62].

2. **Fault list generation stage**

   In the fault list generation stage, SASSIFI uses the profiling information from the first stage and generates a *statistically significant* number of faults. Statistical significance is determined using the population size obtained during the profiling stage. The generated faults are uniformly distributed across the entire execution time of the program for each kernel invocation, which helps to ensure that a program's vulnerability is sampled across the entire execution.

3. **Fault injection stage**

   In the fault injection stage, SASSIFI is fed a target fault from the fault list generated in the previous stage. The target location consists of a kernel, the invocation of the kernel and a dynamic instruction for the kernel invocation. SASSI is used to identify when the target dynamic instruction is reached. In this target dynamic instruction, a fault is injected as per the fault model described in Section 2.8. Fault injection is performed during program execution. SASSI is able to record the dynamic instruction, the program counter, and the basic block of the instruction where the fault was injected.

2.8 Fault Model Used in This Thesis

In this thesis, we evaluate the resilience of unprotected structures in a GPU, including pipeline stages, flip flops, arithmetic and logic units (ALUs), and load store units (LSU). We consider resilience in the presence of single-bit transient faults. To cover these structures, transient faults are introduced at the instruction level, specifically at the output destination of the affected instruction (a random bit is flipped in the destination register of the chosen instruction).

For our evaluations, we perform extensive fault injection campaigns, injecting faults into the destination registers of all instructions that write to a general purpose register. For each dynamic instruction, we estimate the probability that the program will not run correctly if the instruction datapath encounters a transient fault. A fault in the datapath of the instruction will likely produce an incorrect output. If this instruction writes to a register, a wrong value will be stored in the register, which may be used by future instructions, and potentially produce an incorrect output or program crash.

Our error model is similar to models used in other contemporary GPU fault injection studies and tools, such as GPU-Qin [24] and LLFI-GPU [63]. Both tools evaluate faults in the pipeline of a GPU by injecting single-bit faults at the output register of an executing instruction. Also, similar to both GPU fault injection tools, SASSIFI assumes that the GPU cache and memory are protected with Error Correction Codes (ECC). Note that our fault model does not specifically cover multi-bit faults or multiple faults in the same run, as previous studies have shown that: 1) the probability of multi-bit faults and multiple faults is relatively low, and 2) single-bit fault injection campaigns provide accurate vulnerability assessment of programs [64, 65].

As mentioned earlier in section 2.7, SASSIFI performs error injection at an binary level, specifically on SASS instructions. Note that the SASS instructions directly run on the GPU hardware. Fault injection experiments at the SASS level benefit from the fact that they can evaluate the code that directly runs on the GPU. This allows our thesis work to run on actual GPUs, versus on a GPU architecture simulator, enabling us to capture the vulnerability characteristics of live program execution.

2.9 Limitations

While our fault analysis methodology is rich, working on live hardware has some limitations, which we enumeration here.

- **Fault coverage:** Our fault injection framework covers only transient faults that corrupt the functional units in execution (e.g., the ALU and the LDS) datapaths, as described in Section 2.8. As GPUs are increasingly used in safety-critical applications, GPU manufacturers incorporate fault tolerance mechanisms in the newer GPU models. For example, in NVIDIA Voltas memory subsystem, the register file, shared memory, L1 cache and L2 cache are protected by a Single-Error Correct Double-Error Detect (SECDED) ECC code [66].

- **Hardware structure coverage:** Our fault injection framework is able to inject faults in hardware structures that are visible by a programmer. As a result, this thesis does not address faults that can influence the execution of a program, but are not visible by the programmer. While this remains an important limitation to be addressed, the aim in our analysis is to provide insights to programmers and hardware vendors alike. We believe these insights can also be instrumental in addressing reliability for physical structures, that are programmer-invisible.

- **Portability:** Our fault injection framework leverages SASSI [26]. SASSI is an instrumentation tool and is compatible only with NVIDIA binary programs (SASS). This implies that programs written for non-NVIDIA GPUs cannot be evaluated with our framework. However, NVIDIA GPUs are the most widely used today in the HPC market and the majority of GPU programs are written for CUDA.

- **Program modification:** Because our fault injection framework uses a compiler-based instrumentation framework, the program under evaluation is modified, and this can result in an interference with the program running in the hardware, compromising the accuracy in the assessment of the program resilience. However, SASSI applies instrumentation at the SASS assembly level, after all optimizations have been applied in the compilation process (see Figure 2.1). Since SASS code directly runs on the GPU, our framework has very minimal interference with the original application in terms of the final instruction schedule or register usage.

Chapter 3

Related Work

In this chapter we will provide a broad survey of prior work performed by other in topics related to this thesis work. While our main focus will be on GPUs and GPU reliability, we will also include related work in other areas.

3.1 Prior work in CPU Reliability

Transient fault reliability is a well-studied research area in the domain of CPUs. There is a large body of work on CPU-based reliability modeling and vulnerability assessment. Here, we summarize these studies.

3.1.1 Reliability assessment for CPU applications

Architecturally Correct Execution (ACE) analysis (Section 2.6.2) was first introduced for CPU reliability assessment by Mukherjee *et al.* [56], who used it to study various processor structures, including instruction queues and execution units. The methodology was further developed and extended by Biswas *et al.* to include assessment of address-based structures [67].

Li *et al.* investigated the validity of the ACE analysis, and demonstrated that at very high fault rates or in very large structures, the assumptions that failures of different components are independent of each other may not always hold true [68]. Wang *et al.* compared results from ACE analysis against results from fault injection campaigns and found that ACE analysis produced conservative reliability estimates [59].

Sridharan and Kaeli extended the ACE analysis methodology to introduce independent measurements of vulnerability at a hardware level [69] and at a program level [28]. Researchers have also applied fault injection to quantify vulnerability at a program level [70].

In this thesis, we evaluate vulnerability at a program level, using fault injections in GPU programs. Our work also offers a more efficient fault injection methodology, and characterizes the time varying behavior of vulnerability in GPU applications.

3.1.2 Reliability Studies on CPU Systems in the Field

Here, we highlight a few studies on failures in CPU systems and supercomputers in the field.

In 2006, Schroeder and Gibson published a study on failure data from high-performance computer systems at Los Alamos National Labs [71]. They found that the failure rates differ wildly depending on the system. In 2007, Li *et al.* presented a study of memory errors on real production systems in three different data sets [72]. They found that it is highly probable that the actual soft error rate on these production systems were at least two orders of magnitude lower than reported in other studies. In 2009, Schroeder *et al.* published a large-scale field study using Googles server fleet [73]. They found that memory errors in the field were dominated by hard errors rather than soft errors. In 2010, Li et al. published an expanded study of memory errors on an Internet server farm and other systems [74]. They found that non-transient errors form a majority of the memory errors that are exposed to system software. In 2012, Hwang *et al.* published an expanded study on Googles server fleet, as well as two IBM Blue Gene clusters [75]. Sridharan et al. conducted a precise study of DRAM failures on Los Alamos National Labs Jaguar supercomputer over the course of 11 months [76]. They found that approximately 30% of all DRAM failures are due to transient faults, and that the entire system experienced about 900 faults per month, or a little more than 1 fault per hour. Siddiqua *et. al* later found that transient faults are an even smaller proportion of overall faults, and may be as low as 1.5% when accounting for faults in the entire memory path (including memory controllers, buses, channels), and not just memory modules [77].

These studies above were mostly done on memory in high performance computing systems. While many errors in DRAM memory are hard, non-transient errors, the majority of SRAM faults in the field are transient faults, as reported by Sridharan *et al.* in 2015 [78].

3.2 Prior work in GPU Reliability

As GPUs become more pervasive in high performance computing and in safety-critical applications, researchers have investigated and analyzed their robustness to transient faults. The research community has:

1. developed techniques and tools that improve how to assess vulnerability in applications,
2. studied the propagation of faults in applications, and
3. developed methodologies for improving their resilience.

In this section, we review work that evaluates GPU vulnerability, and work that develops methodology to assess the reliability of GPU applications, then we present work that attempts to characterize the error propagation in GPU programs. Finally, we will review the latest development in transient fault remediation.

As mentioned before, research on assessing vulnerability in GPU programs can be done either by directly introducing faults into the GPU (fault injection) or by systematically tracking every bit of a GPU structure during the execution of a program (ACE analysis). We review GPU vulnerability assessment in both categories.

3.2.1 Studies on GPU systems in the field

Reliability is a crucial and challenging topic for Graphics Processing Units. Studies on real GPU systems have highlighted that transient faults are a real threat to GPU applications and computer systems that use GPUs. Here, we highlight a few studies.

Martino *et. al* studied failures in the Blue Waters supercomputer at the University of Illinois [8]. This supercomputer is equipped with 3,072 compute nodes with NVIDIA K20X GPU accelerators. They found that the DRAM memory in the GPUs had 10 times more transient faults than the DDR3 main memory. Their study suggested that GPU cards were the least reliable component in their system, and that their findings were concerning for the GPU-based supercomputers. Haque *et al.* [79] studied DRAM memory errors in a number of consumer-grade GPU cards on a network of distributed computers. Their study revealed patterned hard errors in two-thirds of the cards that were highly correlated with GPU architecture. Their analysis suggested that the memory systems of more modern GPU architectures, and server-grade compute GPUs such as the NVIDIAs Tesla line, were more reliable than gaming products. Tiwari *et. al* [22] conducted a more recent field study on GPU

DRAM and on-chip SRAM in nodes on the Titan supercomputer at Oak Ridge National Labs. They collected data over 18 months, and found that 899 of the 18,688 GPUs in the Titan supercomputer experienced at least 1 error, an average of 1.66 per day across the entire system.

While these studies were conducted on real systems in the field, they only report on transient faults that were either detected by hardware (through Error Correcting Codes or ECC), or that manifested into program-visible errors. Some transient faults cause non-visible program errors, resulting in incorrect value produced by programs executing on these systems. To capture all effects of transient faults, we resort to controlled fault injection experiments.

A variety of fault injection tools have been developed for studies in GPU applications. Examples are GPU-Qin [24], SASSIFI [3] and LLFI-GPU [63]. Fault injection campaigns with these tools reveal that some GPU applications inherently possess resilience properties to transient faults. In addition, studies with these tools have consistently found a wide variation in the level of vulnerability among the studied applications.

3.2.2 Beam Experiments

A realistic error rate of devices can be achieved by exposing them to controlled neutron beams while running workloads. The neutron flux at which the device is exposed during a radiation test is typically 6-8 orders of magnitude higher than the terrestrial flux. A statistically significant number of errors can then be observed in a relatively short time.

The first study that conducted radiation beam experiments in GPUs was by Tiwari *et al.* [22]. The authors use the neutron beam available at Los Alamos Neutron Science Center (LANSCE) and the ISIS (Rutherford Appleton Laboratories, UK) to measure the resilience of different generations of GPUs. The authors found that the per-bit failure rates for the older GPU cards were 2x to 3x higher than the newer ones. This is because newer cards have better bit-cell design (exact details regarding the improvements on older technology are confidential). The authors also found that the double-bit errors were more likely to occur in the newer cards. This finding can be explained by the fact that newer cards have a lower critical charge and thus neutrons are more likely to interact with multiple bit cells.

Oliveira *et al.* evaluated neutron sensitivity of two generations of GPUs in memory structures, L1, L2 caches, and register files [31]. They also found that the newer architecture shows better reliability due to better bit cell design. The authors also highlight that bits set to 0 are more prone to corruption than bits set to 1 in the L2 cache. This is because the L2 cache is designed to be dense and compact to minimize area. The authors also presented and evaluated hardening strategies (ECC vs. software hardening) for GPU applications, as well as their overhead. They found that software hardening can provide better resilience since they catch faults at program level and not just memory faults which ECC covers. However, the software hardening schemes come with a performance penalty.

Lunardi *et al.* experimentally investigated the effectiveness of using ECC in modern GPUs [80]. They considered GPUs fabricated in both CMOS and FinFET technologies. Their results showed that changes in transistor technology can be as beneficial as using ECC for reducing silent data corruption rates. They found that in some cases, improved transistor layout can even be more effective than ECC in reducing the failure rates in some applications.

Accelerated beam experiments do not provide an accurate fault model for GPU programs. Given that the resilience of GPUs is a factor of the executing application, we need better understanding of how faults will manifest at the program level. Fault injection studies and ACE analysis provide a more controllable environment to help develop this understanding.

3.2.3 Statistical Fault Injection Studies

Researchers use statistical fault injection [57] to perform fault injection campaigns and estimate the vulnerability of GPU programs. Statistical fault injection allows researchers to determine the number of injected faults necessary to achieve a certain error margin, given a confidence level. We highlight the studies on fault injections in GPU programs.

Farazmand *et al.* [81] use the Multi2Sim simulation framework [82] for a fault injection campaign in an AMD GPU model, the HD 5870. For this fault injection campaign, faults are injected in structures of the GPU microarchitecture. The results of this campaign show that a great number of resources are not utilized by the GPU, especially for the small applications that were used. This results in a very low rate of Silent Data Corruptions and crashes. For the injections in utilized resources, the GPU demonstrated high resilience, and in many cases, the applications were able to run to completion without any error in their output.

Tselonis *et al.* [23] implemented a comprehensive fault injection framework using a popular GPU simulation framework, GPGPU-Sim [83]. This framework, GUFI performs fault injections in any hardware components of the simulated GPU architecture. Their study also evaluates the differences in fault injection performed in virtual NVIDIA GPU instruction set (ptx) vs. actual instruction set (SASS) and found that these two methodologies yield remarkable differences. They found that the vulnerability is always underestimated in ptx mode. The differences in masked percentage range from 0.65 percentage unit to 21.50 percentage units.

Li *et al.* developed LLFI-GPU [63], a fault injection tool that uses LLVM to perform fault injections at the intermediate assembly level of GPUs. The authors also studied the propagation of faults across kernel calls between CPU and GPU. They found that error propagation in GPU applications is highly application specific.

GPU-Qin [24] is a fault injection tool for GPUs. The tool is built to perform fault injection studies on real GPUs running CUDA-based applications. It uses CUDA-GDB, the NVIDIA tool for debugging GPU applications. The applications are first profiled, and then instructions are selected as fault injection sites. At runtime, GPU-Qin injects a fault into the selected instructions.

We use SASSIFI in our studies, unless specified otherwise. SASSIFI [3] is a fault injection tool for NVIDIA GPU's. It is based on SASSI, a low-level, compiler-based assembly-language instrumentation framework that allows the injection of code at specific points in a program [26]. SASSIFI injects faults in the destination values of executing instructions of a running program at the architectural level. This allows for faster fault injection, increased visibility into the applications and the possibility for a detailed study and analysis of the magnitude of Silent Data Corruptions (SDC).

SASSIFI provides the user with the ability to trace an SDC all the way back to the specific fault which produced it, and also the ability to correlate program properties with program vulnerabilities, which is a key to develop low cost error mitigation schemes. Because SASSIFI injects faults at the architecture level (as opposed to the microarchitecture level), fault injection experiments with SASSIFI can only measure the derating that occurs at the application level.

Kalra *et al.* used SASSIFI to study the vulnerability behavior of vector and scalar instructions on a GPU [84]. They implemented a toolset that can identify dynamic scalar instructions in a program and inject faults in these operations. Their study provided an understanding of error propagation characteristics when faults are injected in scalar, versus vector, instructions. They also found that errors in some opcodes show positive correlations with certain types of outcomes while some others do not. These understandings can lay the foundation for predicting resiliency profile in an application.

3.2.4 ACE Analysis studies

Tan *et al.* developed GPGPU-SODA [85], a framework to evaluate the vulnerability of a GPU to transient faults. It is built on the cycle-accurate, open-source and publicly available, simulator, GPGPU-Sim. GPGPU-SODA is capable of estimating the vulnerability of the major microarchitecture structures in a Streaming Multiprocessor using ACE analysis. GPGPU-SODA attempts to characterize the vulnerability of different micro-architectural structures a GPU to transient faults through architecture vulnerability factor (AVF) analysis. The authors found that the GPU microarchitecture vulnerability is highly related to workload characteristics such as the percentage of un-ACE instructions, the per-block resource requirements and the degree of branch divergence. They also concluded that several structures are highly susceptible to transient faults, and that the entire GPU should be considered for protection.

Jeon *et al.* developed a framework to perform ACE analysis on an integrated CPU-GPU, or accelerated processing unit (APU). The framework is built in an in-house AMD simulator. Their study highlights that the vulnerability of both CPU and GPU must be accounted for, given that they share memory. They show how the reliability of the APU changes over time as computations transfer between CPU and GPU.

Wilkening *et al.* proposed a methodology to calculate vulnerability for spatial multi-bit transient faults, faults in multiple adjacent bits [86]. The authors found that vulnerability measurements for multi-bit faults are not derivable from single-bit measurements. They also found that the more adjacent bits that are affected in a multi-bit fault, the more likely it is for the multi-bit fault to cause program failure. Their methodology did not fully consider whether a single bit fault bit may lead to a single bit fault in a neighboring bit (ACE interference). Previlon *et al.* later showed that this interference is a rare event in typical GPU benchmarks [87], confirming that the previously proposed methodology for multi-bit vulnerability estimation is accurate.

Wilkening *et al.* also propose a practical method to measure resilience for multi-threaded applications using ACE analysis [88]. Their work specifically targeted architectural structures that are shared between multiple threads and can be accessed in various orders. They extended an ACE analysis framework to measure the vulnerability of these shared resources.

3.3 Efficient Fault Injection

Several studies have proposed fault injection acceleration using FPGAs [89, 90]. However, FPGA-based fault injection is not practical for modern microprocessors, and especially GPU's, as we cannot fit a design into an FPGA device. Moreover, these studies do not apply to faults that propagate to the instruction level; mitigation techniques at instruction level offer more flexibility.

Ebrahim *et al.*. [91], Cioroaica *et al.* [92] and Kaliorakis *et al.* [61] employ importance sampling for fault injection acceleration. They focus on performing fault injections only on portions of an application that are critical by first performing an ACE analysis. ACE analysis identifies non-vulnerable parts of the application, with the identified parts removed from consideration for the fault injection campaign. These studies perform fault injections in both live and dead architectural state. The pruning steps in these studies allows for removing any faults that would be injected in dead architectural state. We found that by injecting faults only at outputs of instructions that effectively execute on the GPU, we can already account for any non-vulnerable portions of a program.

Kaliorakis *et al.* perform a fault list reduction by running an ACE-like analysis and removing the faults that fall into the non-vulnerable intervals of a program. Subsequently, they group the remaining faults based on the target instruction pointers and byte positions. They then select representative faults from each group for the final fault injection campaign. Their work however focuses on applications that run on CPU's and their evaluation targets both hardware and software levels of the stack.

Nie *et al.* have proposed a multi-stage technique to prune the GPU fault injection sites [62]. In the first stage of pruning, they randomly choose one representative thread from a thread-block assuming that all threads in the block have similar reliability behavior. In the next stage, they prune instructions assuming that common instruction blocks are likely to have similar resilience characteristics. Through our evaluation on a diverse set of regular and irregular GPU applications, we found that this assumption may not always hold true as faults in the same instruction can often lead to different outcomes.

Similarly, Hari *et. al* propose a tool *Relyzer* [93] aimed at significantly reducing the number of faults necessary for application resiliency analysis. Relyzer prunes an exhaustive fault injection list using two methods. First, *Relyzer* finds instructions in a program in which faults would lead to the same outcome (fault equivalence). This is accomplished by pruning: a) faults in instructions that would follow the same control flow paths, b) faults in store instructions where the values stored are used similarly, and c) faults in instructions that use a value for the first time (this fault is equivalent to a fault in the instruction that defined the value). Secondly, *Relyzer* prunes faults that result in instructions accessing an invalid address. These faults are predicted to cause exceptions in the program. With Relyzer, a reduction of 3-6 orders of magnitude is achieved for the tested applications. However, this pruning is done on a full list of fault sites. The length of the final fault list after applying *Relyzer* is still on the order of millions of fault sites.

Kalra *et al.* introduced PRISM [94], a framework that uses machine learning to predict failures in GPU programs. Their framework utilizes instruction level features, such as the proportion of control flow, load, store and arithmetic instructions, to characterize program resiliency and predict failures in GPU applications without running fault injection campaigns.

Li *et al.* introduced Trident, a model that captures error propagation at instruction level using a static analysis [95]. Their model tries to predict application vulnerability by evaluating the probability for a fault to propagate at the granularity of each instruction.

3.4 GPU Program Phase Analysis

Since the introduction of phase-based performance characterization by Sherwood *et al.* [96], many studies have characterized phase-based behavior of the performance for diverse CPU workloads. A few studies have proposed microarchitecture-independent techniques, alternative to basic block vectors, to capture the phase behavior of CPU workloads. These studies propose the usage of working sets [97], program counters [98], subroutine invocations or loop iterations [99], and control branch counters [100]. Dhodapkar *et al.* compared these various phase detection techniques and observed that techniques based on basic block vectors tend to perform better at characterizing the performance phase behavior of a program [101]. Other studies have also identified a phase behavior in power dissipated during program execution and characterize the power phases using performance counters [102, 103].

Phase analysis has also been evaluated in the context of multi-threaded and GPU applications for acceleration of architectural simulation. Carlson *et al.* developed BarrierPoint [104], which partitions a multi-threaded program into regions separated by global synchronization points. They use a variety of techniques to characterize similar regions in a program, namely basic block vectors, LRU stack distance, and signature vectors. We have identified two studies that focus primarily on GPUs. Huang *et al.* [105] introduce TB-Point (TB stands for thread block) for accelerating simulation of GPU kernels. To characterize similarity between different program regions, they use a technique that takes into account the total amount of work to be done and the memory accesses associated with a GPU kernel. Kambadur *et al.* [106] explored combinations of different feature vectors, including a combination of Basic Block Vectors per kernel and memory accesses within a basic block.

3.5 Time-varying Reliability Characterization

Fu *et al.* evaluated the phase behavior of vulnerability in CPU applications [107], and found that a single performance metric cannot indicate program vulnerability. They showed that both program code-structure (basic block vectors) and runtime events (obtained through hardware performance counters) can classify program reliability phases. However, they found that runtime events were better at predicting reliability phases.

Oliveira *et al.* presented an experimental study in reliability for Intel Xeon Phi processors using both radiation and fault injection experiments [108]. They also presented a time varying vulnerability behavior for their applications. Their work includes both full application execution and fault injections, which can be very time consuming. In contrast, the analysis developed in this thesis presents a phase behavior characterization of resilience using only a single profile run of a GPU application, without performing fault injections.

As mentioned before, Jeon *et al.* [109] evaluated the time-varying behavior of program vulnerability in accelerated processing units in a performance simulator. Their studies evaluated cross-layer vulnerability behaviors and their analysis did not characterize this behavior to program specific behaviors.

3.6 Summary

This thesis provides an improved methodology to perform reliability assessment efficiently for GPU programs executing on a live GPU. This work uses properties of GPU programs to improve resilience analysis.

This thesis also addresses time varying behavior of a GPU in terms of reliability, developing a methodology to characterize this behavior. It demonstrates that understanding phase behavior can be beneficial in terms of improving the reliability for GPU programs.

Chapter 4

Impact of Execution Parameters on GPU Program Vulnerability [1]

If our goal is to provide resiliency to soft errors in hardware, we need to understand how software will use that hardware, propagating bit flips to impact program correctness. In this chapter we address two particular aspects of the evaluation of vulnerability of applications run on Graphics Processing Units (GPUs): i) their dependence on input data, and ii) their dependence on thread-block size.

We demonstrate that the characteristics of input data should be taken into consideration when evaluating the resilience of a program. We found that a change in the input size of a program, as well as corner cases of input data values, can significantly affect program vulnerability. For example, the failure rate of some programs increased by at least 30% when the input size was changed. If we ignore the impact of input size on reliability, it can result in an incomplete or misguided reliability evaluation.

In this chapter, we also study how changing the GPU thread-block size impacts vulnerability. We found that, similar to performance, the vulnerability of an application can depend on the block size of the kernels in the application. In some applications, we found that the silent data corruption rate can vary by as much as 8% when changing the block size of a kernel.

The degree of resilience variability in an application suggests that application-specific measures should be developed to better understand the reliability of GPUs. We argue that application developers that are working on reliability-sensitive applications should have guidance on how best to design resilient applications. A programmer can use either methodology described in Chapter 2 to measure the level of vulnerability in their code, then iteratively redesign their code until the level of vulnerability corresponds to a pre-established reliability target.

While it is possible to only evaluate the vulnerability of a program, independent of hardware parameters, program vulnerability is dependent on the input parameters as well as the specific binary generated by the compiler (which is also dependent on user-specified compiler flags). Input values that enable logical masking errors have the ability to effectively influence fault propagation. A change in the input values of a program can change the vulnerability of this program. In this chapter, we aim to develop a better understanding of this dependence.

The concern for the programmer is to effectively measure the vulnerability of their program. Accounting for all possible combinations of input data values and optimization flags is unfeasible. Our study aims at identifying reliability similarities based on patterns in both the input values and the optimizations selected. Following this methodology we can significantly reduce the number of cases to test. Our analysis can help inform future simulation studies, or help guide reliability ability engineers in terms of qualifying reliability of their project.

Sridharan *et al.* investigated the effects of program input values on program vulnerability [110] and concluded that the primary cause for changes in program vulnerability across varying input data values directly relates to the execution profile (i.e., which code regions were executed) of the program. The authors argued that an input that causes a program to touch all code regions should provide a fairly accurate measure of the vulnerability of the program. Additionally, Jones *et al.* examined the impact of compiler optimizations on system vulnerability and found that a compiler flag, *-freorder-blocks* can reduce the vulnerability of a system across a number of applications [111].

Prior experiments carried out by Sridharan and Kaeli [110] and Jones [111] were conducted on CPUs. In this work, we evaluate program sensitivity to changes in program inputs for applications that run on GPUs, where the variability in the execution profile tends to be minimized.

We evaluate the effects of input data changes in three different aspects: 1) input size, 2) general/generic input values, and 3) highly program-specific biased input values. Our analyses show that, for the GPU workloads studied, the vulnerability of a program can change when the input size is significantly increased, increasing the execution frequency of a portion (or portions) of the code. We also find that the resilience of the code under evaluation is insensitive to changes in input values, whenever these values are randomly generated. However, biased input data values can have a deep influence on program vulnerability.

Additionally, we found that changing the thread-block size of the kernels in a GPU application can change the rate of silent failures by as much as 7% on average for the impacted applications.

The contributions of this chapter can be summarized in the following points:

1. we characterize how input sizes change the execution profile of a program, as well as the vulnerability profile of the program,
2. we characterize how vulnerability changes with different input data values across a suite of popular GPU benchmarks, while exploring biased data value cases, using program-specific biased input data, and
3. we characterize how vulnerability changes as we modify the thread-block size of an application.

4.1 Input Data and Program Vulnerability

We found that input data can impact several aspects of program execution that can have a large influence on resilience: 1) the ordering of instructions in the program (i.e., control flow), 2) the amount of logical masking performed by the individual instructions in the program [110], and 3) the dynamic replication of code sections.

4.1.1 Program Instruction Order

The dynamic instruction order in a program determines how a program manipulates its data, which contributes to its resilience properties. While the instructions grouped into a basic block always execute in the same order, changing the input parameters of a program has the potential to influence the execution order across multiple basic blocks, modifying the control flow of the program. If this occurs, different input values can cause a program to have a different dynamic execution pattern, changing the traversal order of the basic blocks, changing the program's vulnerability profile.

Given the SIMT nature of GPU execution, input values can sometimes influence the control flow path taken by individual threads in a GPU program (e.g., a conditional statement that is based on a function input). In such cases, a change in the input value may cause a different path to execute in a program and therefore affect the resilience of the program.

Because GPU programs contain many threads, their resilience may not be significantly affected if a change in control flow occurs in only one thread. This helps to explain why commonly used input values do not influence resilience. However, in cases of extreme bias in the input values (e.g., an input of all zeros), it is likely for all threads in the program see a change in the control flow paths that are followed.

4.1.2 Logical Masking

Logical masking refers to the property of certain values and logical operations to impede the propagation of faults. As an example, in a 2-input AND operation where one of its inputs is equal to 0, a fault that occurs in its second input will not propagate, and the program will continue its normal execution.

Different input values will change the output of arithmetic and logical operations of a program. This has direct impact on the ability of a program to mask potential faults that may occur during execution. For example, if a MUL (multiply) instruction receives a value of ZERO as one of its input register operands, a fault in the second input register will always be masked and will not affect the correct execution of the program. However, if we change the input value of the program in such a way that the same MUL instruction does not receive a ZERO as an input operand, the amount of masking present in the program will be changed. If the application developers want to develop more resilient and reliable applications, they have to be able to estimate the reliability of their code with every possible combination of input data values and program parameters, which is not always possible. In this thesis we analyze if there are subsets of input data combinations that possess similar reliability profiles. Such a result would significantly reduce the time needed to evaluate the reliability of a program.

4.1.3 Dynamic Replication of Code Sections

The GPU programming model is tailored for highly data intensive applications. As a result, general-purpose programs benefit as the size of heir input data grows. Changes in input sizes of a program typically result in simply replicating the dynamic execution of the program. However, this is not always the case. We found that changing the input size of a program sometimes changes the execution behavior of selected code regions, causing some parts of a code to execute more (or less) frequently. This leads to a change in the resilience behavior of the program if a particularly highly vulnerable code is disproportionately replicated.

Table 4.1: Benchmarks and description of their input type and size

Benchmark	**Description**	**Input Type**
BFS	Find the minimum number of edges needed to reach every vertex in an undirected graph	Graph of *N* nodes with the list of edges and their weights
Gaussian	Solves a system of equations using the Gaussian elimination method	*NxN* Matrix and a *1xN* vector
Hotspot	Estimates processor temperature based on an architectural floorplan and simulated power measurements	*NxN* temperature values and *NxN* power values
KMeans	Performs fuzzy k-means clustering on a set of data points	*N* data points of *m* features per data point
LavaMD	Calculates particle potential and relocation due to mutual forces between particles within a large 3D space	*NxNxN* boxes with their respective charges and distances between them
NW (Needleman-Wunsch)	Nonlinear global optimization method for DNA sequence alignments	2 sequences of length *N*

Table 4.2: Outcome Categories for Injections

Outcome Category	**Explanation**
Masked	All output files match the correct output files
DUE	Detected and Unrecoverable Errors (Crash), or App does not terminate in the allocated time
Potential DUE	App exits with non-zero status, error messages were recorded
SDC	Error values at the program output

4.2 Results

Next, we present results of our experiments in two categories. In the first category, we present variations in resilience when we change the input data for our programs. In the second category, we examine the effects of adjusting the size of a thread block on the resilience of the program.

For each application, we inject 1,000 single-bit faults at the output of instructions that write to a General Purpose Register, which are sufficient to guarantee the worst case statistical error bars at a 95% confidence level to be at most 3.1% (Figure 2.6.1.1). The benchmarks selected are from the Rodinia suite and are representative across a range of application domains: linear algebra, physics simulation, and fluid dynamics. A more detailed description of the benchmarks, along with the type of input that is used, is provided in Table 4.1. The description for each outcome of an injection is provided in Table 4.2.

4.2.1 Impact of Input Data

We evaluate 3 different aspects of the input workload data: 1) input size, 2) input value, and 3) biased configuration. We present the outcome for each program for each aspect of our evaluation. We differentiate between these three aspects of the input data because they will produce different impacts on program execution. Varying input sizes has the potential to scale the dynamic execution of particular sections of a program. Input values have the ability to influence the logical masking in individual operations and potentially influence the control flow of a program. Finally, we identify specific *biased* input values and configurations that may induce irregular behavior in a program. For example, for `gaussian`, we found that for some input matrix values, the system of equations accepts an infinite number of solutions. In these cases, any fault that does not crash this application will likely not influence the correctness of the resulting computation. Such an input matrix is a biased input. The results of our experiments are presented next.

4.2.1.1 Input Sizes

Figure 4.1 shows how the resilience characteristics of our applications change when the size of their inputs change. It is important to emphasize that our analyses only examine how the resilience properties of an application change **after** a fault occurs in the application. By resilience properties, we mean the probability for a fault to be masked or to cause an error. We do not take into account the change in the likelihood for a fault to occur in the application in the first place, which can change if an application experiences a significant increase/decrease in the number of dynamic instructions.

The results in Figure 4.1 show that resilience characteristics can significantly change for an application when its input sizes change. The application `gaussian` experiences a change in its SDC rate from 3.8% with a 4x4 matrix, to 36.7% with a 32x32 matrix. For `k-means`, the SDC rate changes from 8.5% (with 100 objects) to 22.9% (with 3,000 objects). For `Needleman-Wunsch (nw)`, while the SDC rate stays constant, the DUE rate reduces from 50.4% (with an input sequence of 512) to 36.8% (with an input sequence of length 8,000).

Contrary to the hypothesis that input sizes only scales the execution time of a program and not its resilience [63], our results are in accordance with beam experiment results presented by

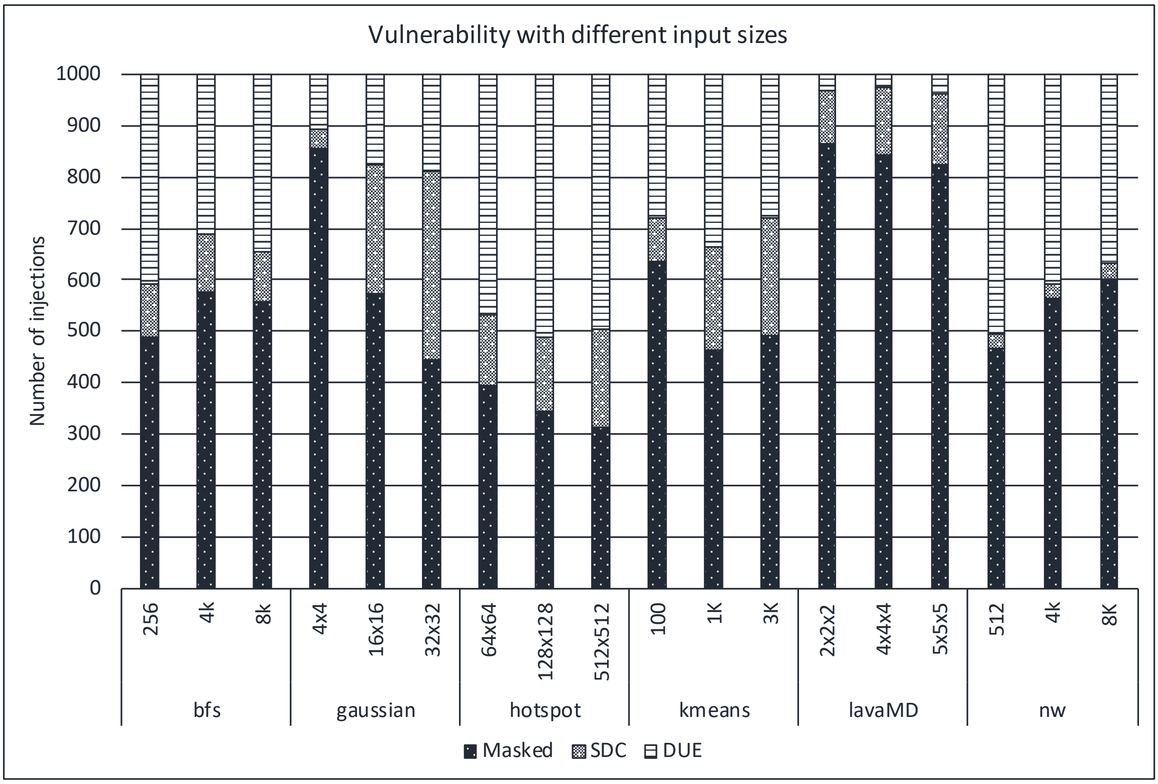
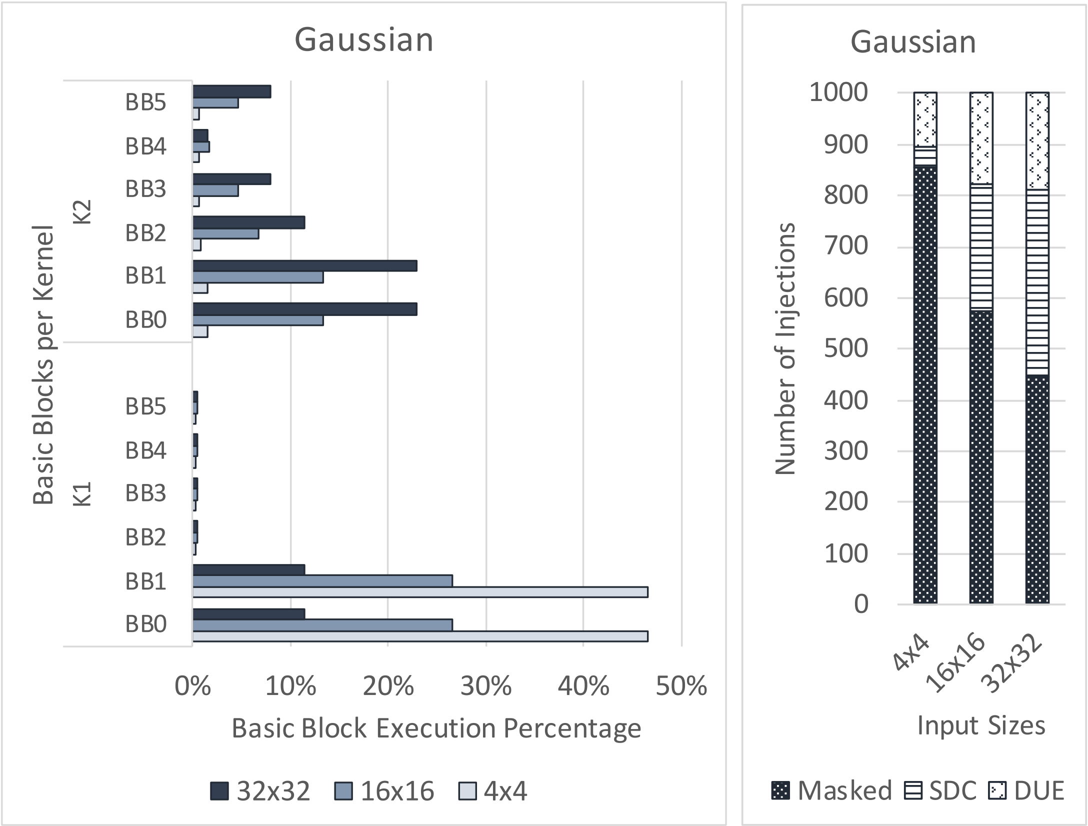


Figure 4.1: Fault injection outcomes for applications with different input sizes.

Rech et al. [17], showing that vulnerability of a program can significantly change when its input size changes. We found that the reason for the change in resilience resides in the non-uniform scaling that occurs in the dynamic execution of specific code sections. In other words, as the input size changes in an application, some sections of the application code account for more (or less) of the dynamic instruction count.

For example, the application `gaussian` experiences a significant increase in the SDC rate as the input size increases. Figure 4.2 shows the dynamic execution weight for each basic block in the two kernels of `gaussian` as the input size changes. What we find is that, for small input sizes, the majority of the dynamic instructions are due to the first kernel (K1). However, as the input size increases, a majority of the dynamic instructions are from the second kernel (K2). Our fault injection experiments in `gaussian` found that a fault in the second kernel (K2) is more likely to induce an SDC than a fault in K1. As the results show, the SDC rate grows as the input size grows, with K2 dominating execution. Similar trends are observed for other applications where there is a significant change in the dynamic execution weight of individual basic blocks.



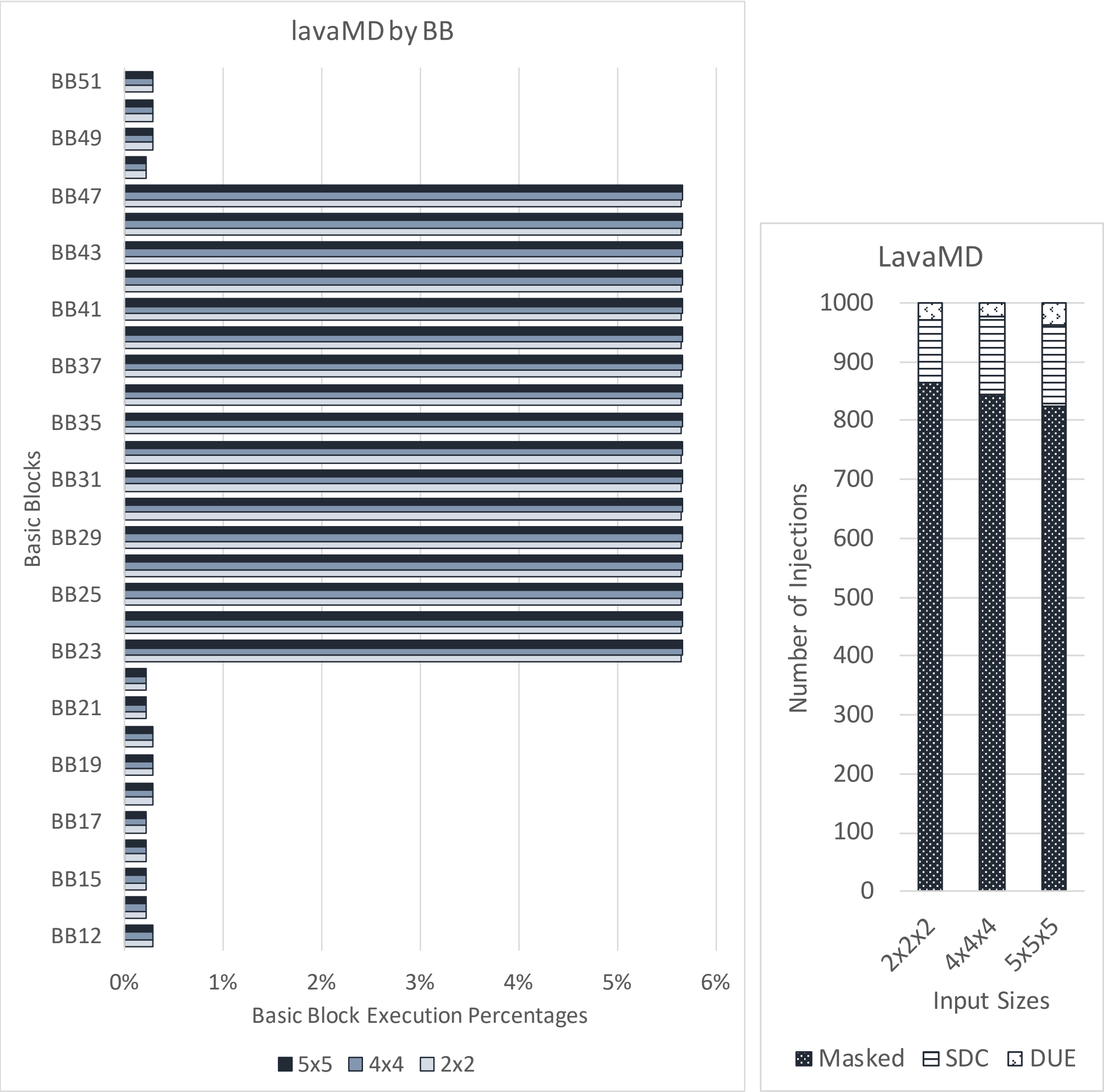
(a) Dynamic execution percentage for each basic block in both kernels.

(b) Fault injection results for each input size.

Figure 4.2: Dynamic execution percentage for each basic block in `gaussian`, when the input size changes and the fault injection results for the corresponding input sizes. Our experiments found that a fault in the second kernel (K2) is more likely to induce an SDC than a fault in K1. As the results show, the SDC rate grows as the input size grows, with K2 dominating the execution.

It is important to note however, that if one continues to increase the input size for this application, the vulnerability characteristics will reach a more stable point. We noticed this occurred for gaussian. The results for 64x64 input size are slightly different from the results of 32x32 (with a 3% increase in DUE outcomes), but beyond this point the dynamic behavior of this application reached a more stable behavior, and as a result its vulnerability remained unchanged past this input size.

In contrast, some applications experience no change in their resilience characteristics. This is the case for `lavaMD`. Our analysis of the dynamic behavior of this application for different input



(a) Dynamic execution percentage for each basic block in both kernels.

(b) Fault injection results for each input size.

Figure 4.3: Dynamic execution percentage for each basic block in `lavaMD`, when the input size changes and the fault injection results for the corresponding input sizes. This application scales well and increasing its input sizes simply results in repeatedly executing the same basic blocks several times.

sizes shows that the larger/smaller input sizes uniformly scale the dynamic execution of the different code sections of the application. This is illustrated in Figure 4.3. As we change the input size of the application, we do not see a significant change in the execution profile (as measured as a percentage of the overall execution) of any individual basic block. This is because this GPU application is highly

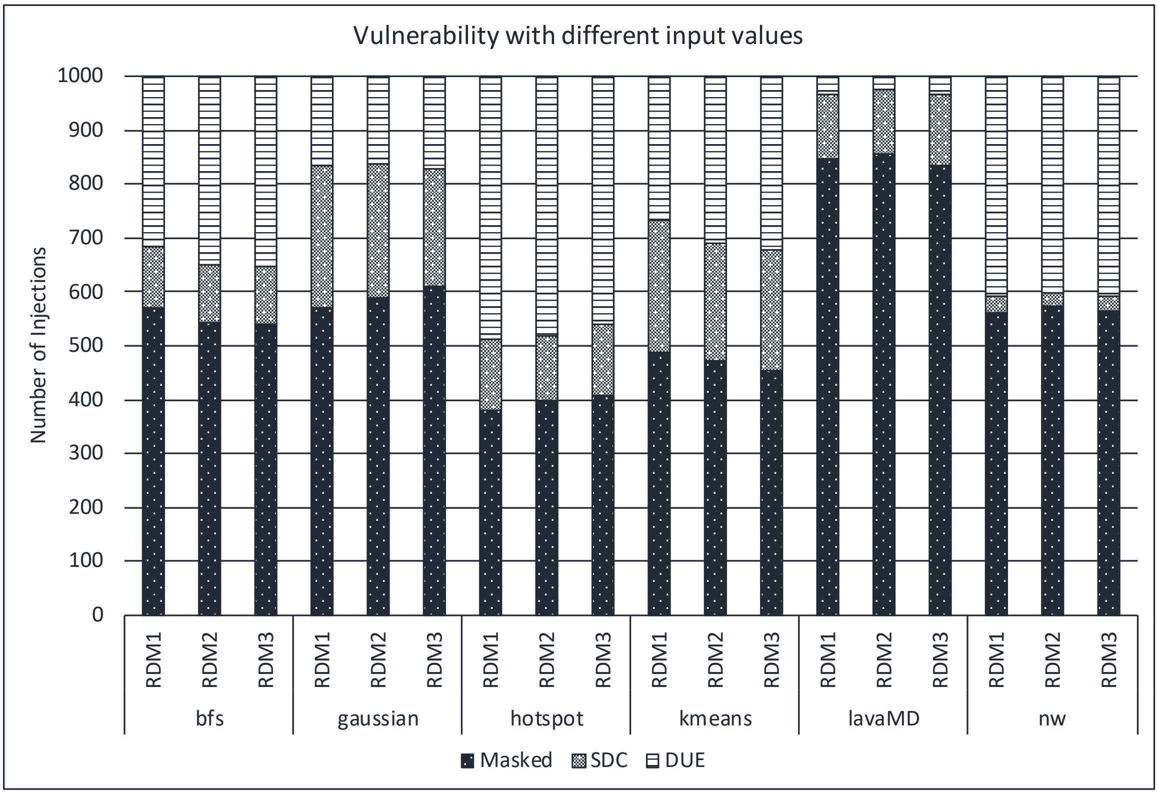


Figure 4.4: Fault injection outcomes for applications with different randomly generated input values. The input values were randomly generated through the program's own input generator.

regular, and increasing its input sizes simply results in repeatedly executing the same basic blocks several times.

4.2.1.2 Input Values

Figure 4.4 shows how the vulnerability of applications changes when the input values change. We randomly changed the input values (shown by *RDM 1* through *RDM 3*). These input values were obtained by using each workload's own input generator. Our programs allow users to change their input values by re-running the input generator scripts. We generated the new input values and verified that both sets of values in *RDM1* - *RDM3* were completely different from each other.

The results show that random changes in the input values do not significantly alter the vulnerability behavior of the applications. The difference between outcome rates using *RDM1* - *RDM3* is about 3%, which is within the error margin in our experiments.

Based on our results we can claim that, when evaluating the reliability of generic GPU applications, it is reasonably sufficient to use randomly generated inputs. However, it is worth noting that, whenever available, realistic inputs and configurations are preferred.

Our investigation shows that changing the input values for the programs under investigation does not significantly alter their control flow. In other words, when the input values are changed, though remain in a restricted range of values, we found that the control flow graph of our applications was not altered enough to significantly affect their vulnerability. A similar trend was observed for programs run on CPUs [110]. This work reported that different program inputs lead to different vulnerability behaviors for a program, because different program inputs were inducing different execution profiles (different instruction streams) for that program.

4.2.1.3 Biased Inputs

We evaluate two types of biased inputs in our experiments: 1) the extreme cases of biased input values, *ZERO* and *ONE* input arrays/matrices where all the elements are 0 or 1, respectively; and 2) the application-specific biased input data where the input is specifically crafted to influence the vulnerability of an application.

ZERO/ONE Biased Input Data

[112] For the two input sets of all ZEROs and all ONEs in `bfs`, we gave all the nodes in the input graph 0 or 1 edge, respectively. Consequently, all the nodes in the graph are visited in one iteration of the kernels. For typical inputs, with an average of 8 edges per node, the program converges after 8 iterations of the kernels. Faults in `bfs` are more likely to be propagated with more iterations of these kernels. To obtain a worst case result for SDC's, we should select for an upper-bound a realistic number of edges per node for `bfs`.

For ZERO and ONE input datasets for `gaussian`, we set the values in the input matrix and vector to be all ZEROs and all ONEs, respectively. For these input values, there exist an infinite number of solutions to the system of equations. A fault in this program will not likely produce an incorrect result when the input is ZERO or ONE. Biased inputs for gaussian include the set of inputs for which there exist an infinite number of solutions.

For `kmeans`, using the ZERO and ONE input datasets utilize data points such that the associated features are all ZERO and ONE, respectively. A large portion of this algorithm iteratively computes the distance of each data point to its assigned cluster center. For a ZERO input set, the

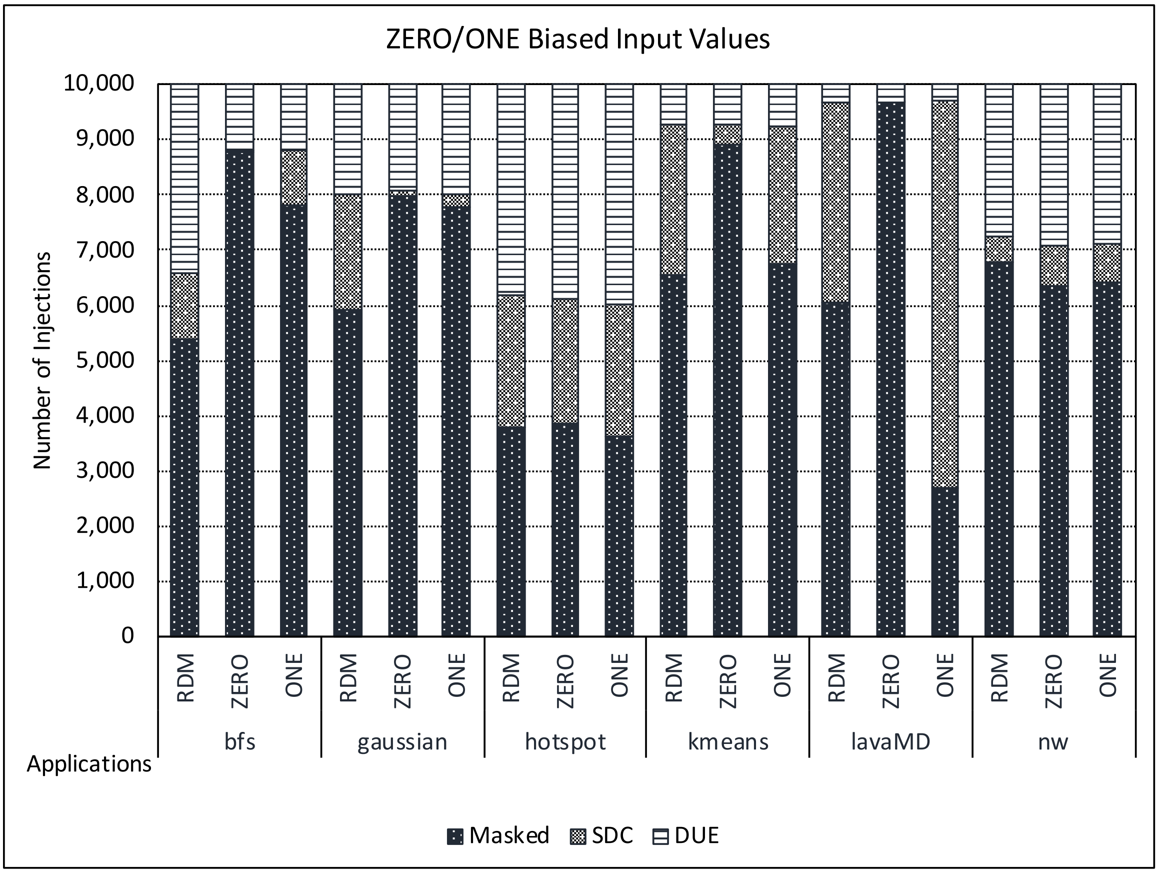


Figure 4.5: Fault injection outcomes for applications with randomly generated input values vs. extremely biased values. *ZERO* and *ONE* input represent input values of all 0 or 1, respectively.

distance is always 0, as all points are located at the cluster centers. A fault is not likely to change the membership of the data points.

A similar case arises for `lavaMD`, where an input data set of ZERO means that the distances and charges of the particles are all ZEROs. Consequently, there are no relocations of any particles and it is unlikely for a fault to change that result. The ONE input set, however, which assigns the distances and charges of all the particles to 1, corresponds to the maximum value for both distances and charges. This leads to a very large displacement of the particles, and more opportunities for fault propagation.

Overall, the changes in resilience caused by our *ZERO/ONE* biased inputs, with the exception of the *ONE* input set for `lavaMD`, correspond to an increase in masked outcomes. This is because the *ZERO* and *ONE* biased input sets are extreme corner cases for our applications, and they happen to maximize fault masking. Some should never actually occur, such as in `kmeans`.

Application-specific biased inputs and configurations

Table 4.3: A description of Workload specific biased input data.

App	Biased input description
BFS	Higher edge density. The average number of edges per node is 200, compared to 8.
GAUSSIAN	The input values were chosen such that the system accepts infinitely many solutions.
HOTSPOT	Input parameters were changed to increase the number of iterations of the kernel.
KMEANS	The maximum number of clusters possible was changed to 50, instead of the default 5.

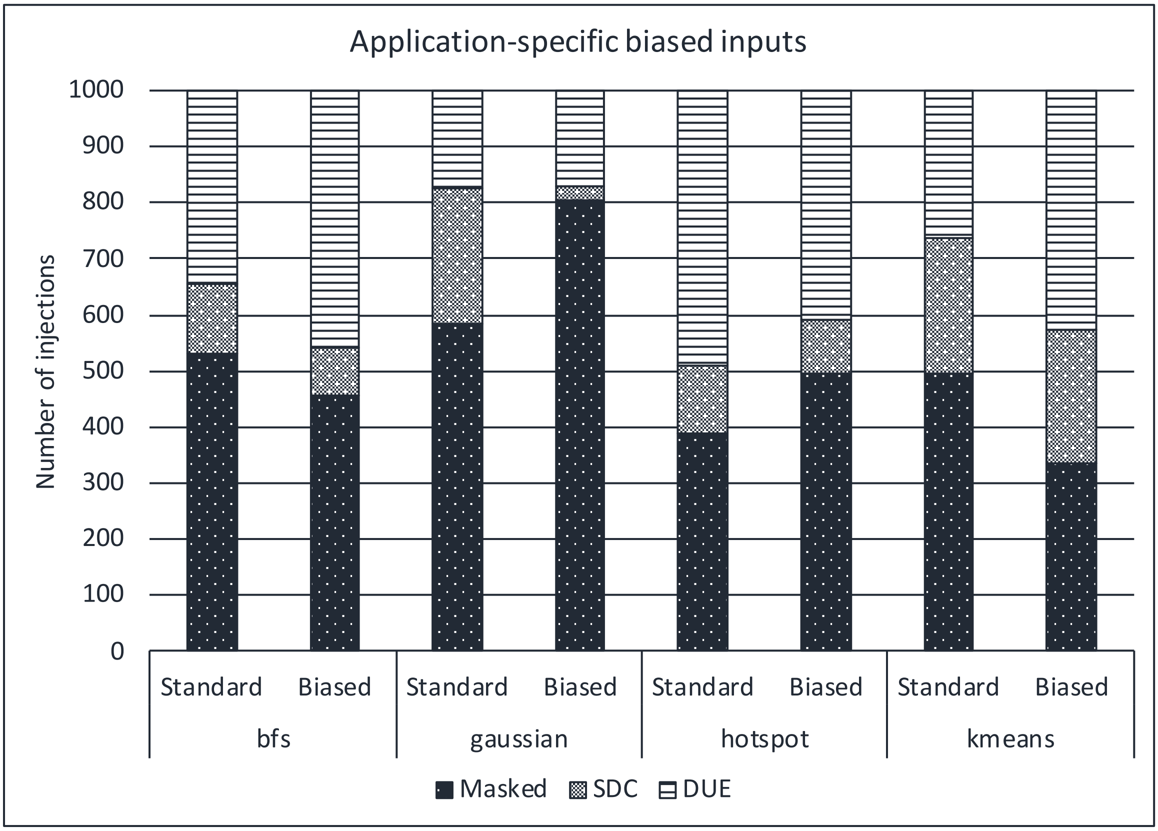


Figure 4.6: Fault injection outcomes for applications with randomly generated input values. The input values were randomly generated using the workload's own input generator.

We investigated the effects of biased inputs that are especially crafted to influence the resilience of our applications. This section shows that, some inputs, which do not exhibit extreme cases of all zero inputs, can still produce irregular execution behavior in applications.

We analyze cases where specific changes in the inputs of our applications could have a significant impact on their resilience behavior. Table 4.3 highlights the specific changes that we made to turn the standard inputs into biased ones. Figure 4.6 shows how the resilience of our applications change when we perform assessment with the biased inputs.

For `bfs`, we regenerated the graph input such that the edge density becomes much higher than the standard input. The first kernel of `bfs` contains a loop in which every node in the input graph visits the adjacent edges. Increasing the average number of edges in the graph results in a significant increase in the number of loop iterations inside the first kernel of the application. Consequently, the number of DUEs significantly increases, as faults in the first kernel are more likely to yield a DUE.

The application `gaussian` solves a system of linear equations. We exploited a corner case of a system of equations which has infinite solutions. We created an input for the program that would yield the case of infinite solutions. Because there exists an infinite number of correct solutions with this particular input, a fault that does not cause a program crash would likely produce a solution to the program that is acceptable as a correct solution. This particular case of input for `gaussian` makes the the program more tolerant to faults that do not result in termination of the program.

For `hotspot`, we explored execution parameters that can impact program behavior. Specifically, we doubled the number of iterations of the application kernel. This also further increased the number of iterations in a loop inside the kernel. Faults that occur during the execution of this loop are likely to be masked, and in turn, the increase in iterations of this loop yielded a more resilient application, as shown by the increase in **masked** outcomes in Figure 4.6.

For `kmeans`, we also changed an execution parameter and allowed the application to test for up to $k = 50$ to find the best number of clusters, as opposed to using only 5. This simply replicates the number of iterations of both kernels of the application. However, there is a loop inside one of the kernels where the number of iterations depends on the number of clusters being used. As the number of clusters increases, so does the number of iterations of this loop. This portion of the code therefore becomes a more significant percentage of the dynamic execution of the application. Faults in this portion of the loop also will likely yield a DUE outcome, as this portion of the code contains memory accesses and a fault can easily corrupt the address used to access memory. Consequently, there is an increase in the number of **DUE** outcomes with this change to k in `kmeans`, as shown in Figure 4.6.

The results in Figure 4.6 show that specific cases of input values can influence our resilience assessment. When evaluating the resilience of an application, we need to take into account whether there is a program input that could yield an unusual dynamic behavior (e.g., an increase in the

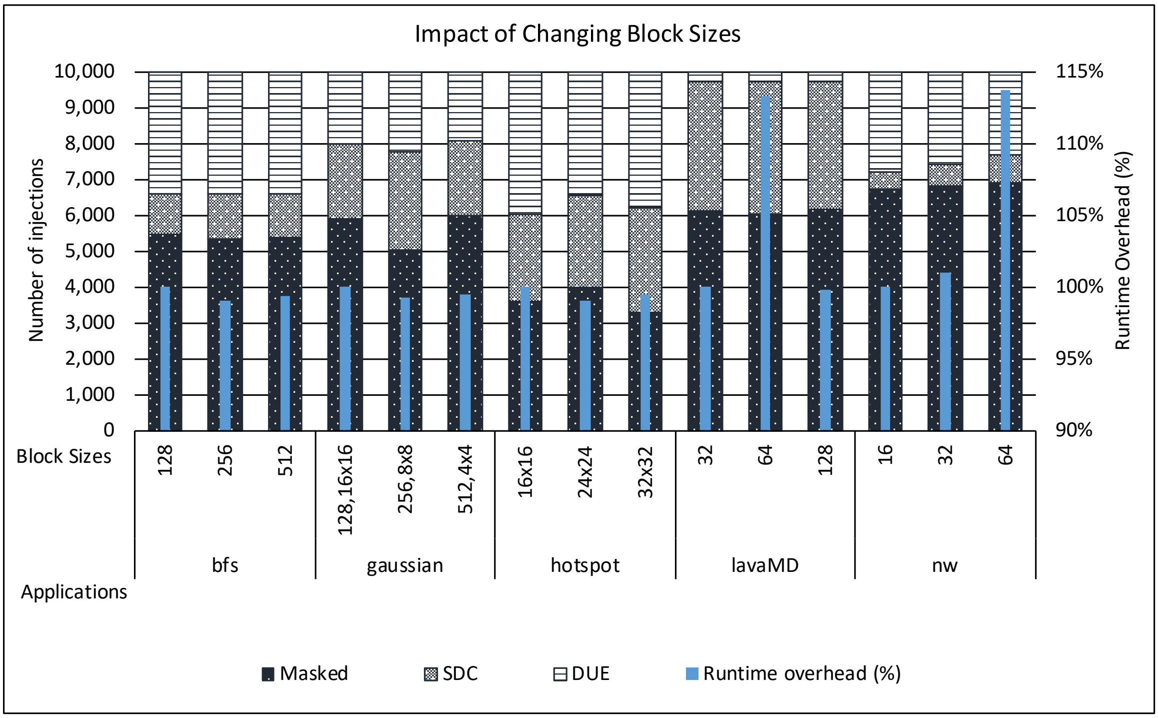


Figure 4.7: Fault injection outcomes for applications when block sizes are changed. The x-axis represents the thread-block sizes (1D and 2D) for the kernels. Runtime overhead for respective block sizes over the baseline configuration (configuration with the smallest block size) is also shown for each block size.

number of loops executed or an increase in the number of kernel launches), significantly impacting our conclusions from a reliability assessment. To come up with biased inputs, we chose inputs or configurations that would cause a change in behavior in the program dynamics. For most of our programs, the change in their input affected the number of iterations executed of an individual kernel or specific loops inside a kernel.

4.2.2 Effects of changes in kernel block sizes

A common practice for GPU programmers is to change the number of threads that each instance of their kernel can launch, also called the block size. The block size can significantly influence the performance of the program. A programmer often chooses the optimal block size depending on the complexity of their code, the pressure placed on system resources (e.g., registers and memory) and the compute capability of their device.

Some of the applications used in this experiment allow a user to adjust the block size of the kernel. A user can then experiment with the program by increasing (decreasing) the number of threads in each block while decreasing (increasing) the number of blocks in a grid.

Changing the block size may improve the performance of a program, depending on the capabilities of the hardware. Block size has already been demonstrated to significantly impact the reliability of GPUs [113]. Using our architectural-level fault injection framework, we aim at better understanding the reasons for the dependence of reliability on block size.

In Figure 4.7, we show the results of a fault injection campaign for our applications while changing the block size of their kernels. We also show the runtime overhead for the respective block sizes, as compared to the baseline configuration (i.e., the configuration with the smallest block size). Our results show that, just as performance of an application is affected by adjusting the block size for a kernel, the resilience of an application is also affected by this configuration parameter. For example, the SDC rates experience a change of as much as 8% for (`hotspot` and `gaussian`). These results show that changing a block size in an application can affect an application's resilience.

Our results suggest that the changes observed in reliability when we modify the block size are not only due to corruption at the microarchitecture level (see Figure 1.1), specifically in the block scheduler, as proposed by Rech et al. [113]. Our fault model does not account for faults occurring at the microarchitecture level.

Changing the block size of a kernel may change the number of blocks that can be assigned to a particular streaming multiprocessor of a GPU. The exact number of blocks assigned to a streaming multiprocessor depends on the amount of resources needed by each thread (e.g., registers) in the block, as well as the amount of shared resources needed by the threads (e.g., shared memory). Some blocks may be queued for later execution if the number of blocks in the kernel exceeds the number of blocks that can be scheduled in the GPU streaming multiprocessors. This potentially increases the vulnerability exposure time for data in the blocks that are scheduled for later execution.

Additionally, changing the thread block size in a program changes the distribution of data across the Streaming Multiprocessors. This results in a new arrangement of threads within the new blocks. Any resilience that came from the collaboration between threads in the same block is also impacted.

The degree of error propagation displayed is application dependent. In `gaussian`, for example, *512,4x4* means that the first kernel has 512 threads per block, while the second kernel has thread blocks of 2 dimensions (4x4). We found that the thread divergence present in the first kernel code made it more likely for a fault to be masked in this kernel with a larger block size.

Our results also suggest that in reliability estimation, we should not only use realistic input values, but we should also be mindful of the block size that is to be used with the application. In our programs, there is no direct correlation between the change in performance and the change in resilience. We plan to investigate this relationship further in the remainder of this thesis.

4.3 Summary on the Impact of Execution Parameters on Program Vulnerability

This chapter examined how changes in input data values of a GPU program impact vulnerability. We also evaluated the impact of changing the thread-block size of a program, a common practice among developers when tuning an application for performance.

When GPU programmers start to test the resilience of their applications, it is important that they consider whether the results of their testing will hold true with different types of input sizes and values. Our investigation here has found that resilience in GPU applications can change with changes in input sizes and using biased input values. These changes are mostly due to changes in the scaling the dynamic contributions of individual code sections in an application. Changes in input size that cause a portion of an application to scale disproportionately will likely influence the resilience of an application. Similarly, programmers should be wary of specific values that can affect the number of iterations of a single kernel, or input values that may induce a corner use case in an application.

Our experiments on the variation of the thread block size found that the SDC rates can change by as much as 8% for some applications. A programmer needs to carefully evaluating the reliability trade-offs of tuning the performance of an application when modifying the block size of the kernels. For the experiments carried out in this thesis, we use realistic input sizes and values for all vulnerability estimation experiments.

Chapter 5

PCFI: Program Counter Guided Fault Injection for Accelerating GPU Reliability Assessment

The software fault-injection process is prohibitively expensive, requiring multiple days to complete a statistically sound fault-injection campaign. To address this challenge, this chapter proposes a novel fault-injection methodology named *Program Counter Guided Fault Injection* (PCFI). PCFI is designed to reduce the number of fault injections by exploiting redundancy in fault injection experiments. PCFI utilizes the program counter value to characterize corrupted instruction execution. We consider the behavior of a range of GPU programs covering diverse application domains. Our goal is to design PCFI to reduce the time to complete fault-injection campaigns, without sacrificing accuracy.

5.1 PCFI Overview

5.1.1 Motivation

The software fault-injection process is prohibitively expensive since the number of faulty runs in a fault injection campaign needs to be sufficiently high to produce statistically significant results. Previous research papers have reported the number of fault injection runs to be more than 60,000 for achieving high accuracy [62, 61]. Prior work has reported that 10,000 injections is the right number of injections in order to obtain low error margins (close to 1%) and a 95% confidence interval [57] [63] (see Section 2.6.1.1). Assuming the average execution time for each of our applications is around 15 seconds on a GPU, it will take more than 45 days on a single GPU to perform 10,000 fault injection runs for the 26 benchmarks considered in this study.

As the execution time of a GPU program increases, and such is the case for long-running GPU-based HPC applications, this overhead makes reliability assessment impractical or even infeasible in some cases. We note that when researchers devise new resilience mitigation strategies, they need to run multiple 10,000 fault injection runs in order to compare trade-offs of competing solutions.

To address this challenge, we propose a novel fault-injection method, PCFI, that reduces the number of fault injection runs needed during a fault injection campaign to assess the vulnerability of GPU programs to soft errors. PCFI attempts to reduce the time and effort associated with soft error analysis, without comprising the accuracy of the results.

5.1.2 PCFI Key Idea

Traditional fault injection methods perform a large number of fault-injected runs during a fault injection campaign. In each run, one fault is injected and the effect on the program outcome is observed (i.e., crash, incorrect result, no effect). Across all the runs in a campaign, faults are uniformly and randomly distributed over all the dynamically executed instructions to achieve high statistical significance [3].

We show that GPU programs tend to execute a small subset of static instructions (i.e., a subset of all possible program counter values) multiple times during their dynamic execution. This small set of static instructions (PCs) often constitute a significant fraction of the total dynamic instructions. This tendency follows the 90/10 rule described in many Computer Architecture textbooks [114]. Hence, these PCs also account for a major fraction of injected faults, even when faults are uniformly and randomly distributed over all dynamically executed instructions.

Furthermore, we discovered that the outcome of injected faults in these PCs remain the same across different dynamic execution instances of the same static instruction. PCFI exploits these observations to reduce the number of fault injection runs. PCFI identifies frequently executed PCs and carefully limits the number of fault injections in those PCs if the outcome of a fault in those PCs does not change across different dynamic execution instances. PCFI is able to exploit the predictability in outcomes to maintain the same accuracy, but significantly reduces the time by eliminating many fault injection runs in the same PC that are likely to result in the same outcome behavior. We discuss the challenges and trade-offs involved in designing and implementing PCFI to achieve its goal. To the best of our knowledge, this is the first work to exploit PC-based behavior to reduce the fault injection campaign effort.

5.2 PCFI: Design and Implementation

Traditional fault-injection methods involve running a large number of fault injections, where a single fault is injected in each run and the effects on program correctness are observed (i.e., crash, incorrect result, no effect). Widely-used GPU fault injectors, such as SASSIFI (described in Chap. 2.7), take the following steps to perform fault injection. SASSIFI identifies all possible vulnerable locations (i.e., all the dynamic instructions of the program under evaluation). SASSIFI randomly and uniformly selects threads and instructions within a thread to cover different program phase behavior in space and time. Finally, SASSIFI injects faults in each chosen instruction. Traditional software fault injection methods use such an approach and inject high number of faults in a fault-injection campaign [24, 3, 63]. Therefore, we use the fault-list obtained from this method as the baseline fault list that PCFI prunes.

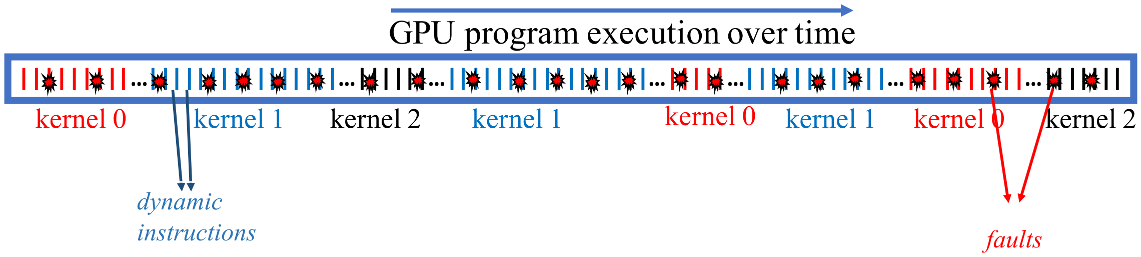


Figure 5.1: Traditional fault injection methods uniformly and randomly distribute faults across a program's dynamic instructions.

PCFI exploits the following two observations to reduce the number of fault-injected runs from an original fault injection list. First, GPU programs tend to have a small set of static instructions (PCs) that often constitute a significant fraction of total dynamic instructions. Consequently, this set of PCs also account for a major fraction of the injected faults in the baseline case. Second, the outcome of injected faults in many frequently executed PCs remain identical across different dynamic instances of the same static instruction.

Next, we provide quantitative evidence to support our observations. We show results for three representative applications: i) `reduction (red)`, ii) `dwtHaar1D`, and iii) `minimum spanning tree (mst)`. We obtained similar results and trends for other benchmarks during their execution.

Figures 5.2a, 5.2c, and 5.2e show the execution breakdown across each PC in these programs. These results show that some PCs are highly dominant. That is, their execution frequencies can be an order of magnitude higher than other PCs in the same application. This is due to the fact that these PCs belong in code blocks that are in loops with high trip counts. Using dynamic profiling of the `red`, `dwtHaar1D`, and `mst` applications, we found that 86, 83, and 615 unique PCs are executed, respectively. However, very few PCs dominate most of the execution in these programs. For example, in `red`, 99% of the dynamic instructions were execution from only 12 unique PCs, because the main code executed by each thread for this application resides in a loop with a high number of iterations.

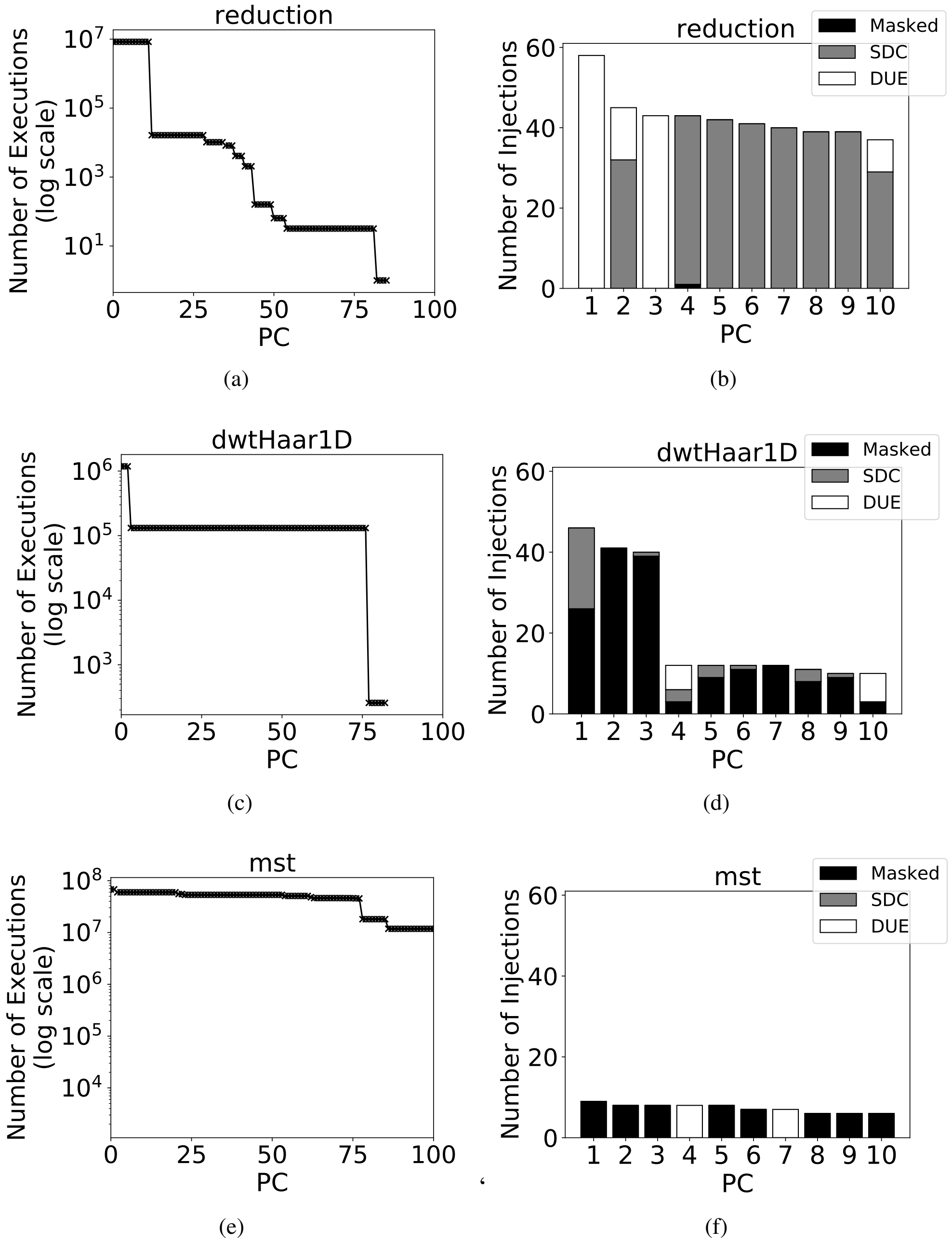
Figures. 5.2b, 5.2d, and 5.2f show the number of faults injected into unique PCs for a full fault injection campaign with SASSIFI. During this campaign, we chose a 95% confidence interval, and a 5% error margin, and the faults were randomly and uniformly distributed.

We only show the PCs with the highest execution frequencies (dominant PCs) to highlight their predictability. Many of the top 10 PCs are tied to a significantly higher number of faults. For example, in the application `red`, the top 10 PCs account for 80% of total faults out of all the injections. Moreover, we observe that the outcome of the injected faults is identical for many of the frequently executed PCs across all three benchmarks: `red`, `dwtHaar1D`, and `mst`. To exploit this opportunity, one could limit the number of injections in these "hot"-PCs and extrapolate the results for each PC according to its respective execution frequency.

We note, however, that some PCs do not follow this model, mainly due to changes in control flow behavior over the lifetime of the program. For example, in `dwtHaar1D` (see Figure 5.2d), injections in PC1, the most frequently executed PC, sometimes get masked and other times lead to SDCs. We further investigated this behavior and found that the different outcomes depend on the execution phase of the program. The injections over time in PC1 demonstrate a phase-based behavior, resulting in phases with a high probability for SDC outcomes and phases with a high probability of fault masking. We further examine the resilience phase behavior in Chapter 6.

When we encounter control-flow depend fault profiles, the number of injections should not be limited, since we will need capture all phases of their behavior. So before we reduce the number of injections in a PC, PCFI injects faults across the whole execution time of the application.

Leveraging these insights, PCFI identifies frequently executed PCs via a profile run and

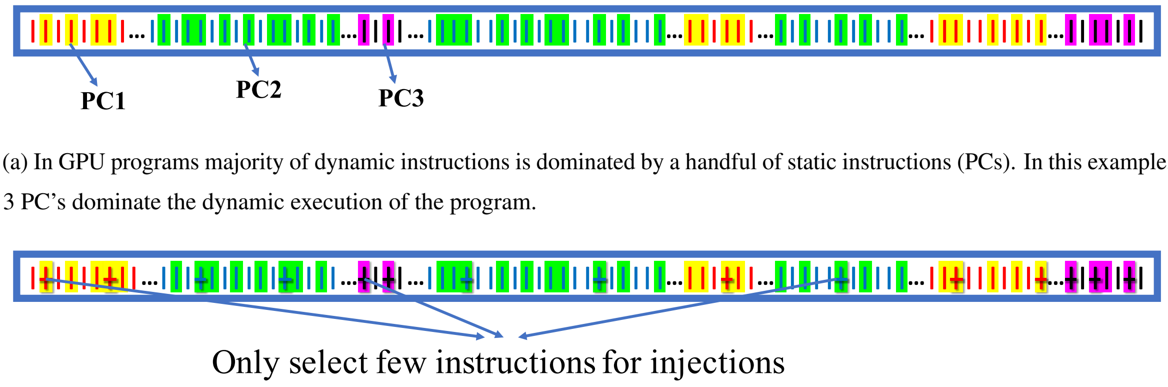


(a) (b)

(c) (d)

(e) (f)

Figure 5.2: Applications from different domains, `reduction (red)` (data processing), `dwtHaar1D` (signal processing), and `mst` (graph traversal), show that (1) few static instructions (PCs) dominate overall dynamic execution of instructions, (2) vulnerability outcomes in highly frequently PCs are likely to remain across injections for many PCs.



(a) In GPU programs majority of dynamic instructions is dominated by a handful of static instructions (PCs). In this example, 3 PC's dominate the dynamic execution of the program.

(b) PCFI limits the number of injections into instances of individual static instructions.

Figure 5.3: PCFI approach for PC-guided fault injections

carefully limits the number of fault injections in the those PCs if the outcome of a fault in those PCs does not change across different dynamic execution instances. Figure 5.3b shows that PCFI limits the number of injections in PCs for the baseline shown in Figure 5.3a, where most of the dynamic instructions in this example are different dynamic occurrences of 3 dominant PCs. The outcome of faults in PCFI-friendly PC's is extrapolated to match the execution frequency of the PC. PCFI accounts for cases where faults at different instances of a PC do not have the same outcome by keeping a history of outcomes for each PC. PCFI is able to exploit the predictability in outcomes to maintain the same accuracy, but significantly reduces the time by eliminating many fault injection runs in the same PC that are likely to result in the same outcome behavior. Our results show that by limiting the number of injections based on history outcomes, we do not compromise the accuracy of our results.

PCFI Implementation Details

Next, we describe how PCFI is implemented in our open-source GPU fault injector, SASSFI [3]. The steps below detail how PCFI can be adapted for any generic GPU fault injector, and is independent of the detailed implementation of the GPU fault injector.

- **Profiling Stage:** First, PCFI performs a profiling step which records the count of all dynamic instances of each static instruction in the given GPU program. Note that the overhead of this step is equivalent to performing a single fault injection run. At the end of this step, PCFI has identified all static instructions (PCs) that were executed most frequently, and their respective execution frequencies.

- **Fault List Generation Stage:** In this step, SASSFI generates the fault list - the dynamic instruction list where faults will be injected. SASSIFI generates these faults such that they are randomly and uniformly distributed across the dynamic executions of each PC. This also ensures that PC behavior across all different program phases is captured. Note that the number of injections that map to a PC is directly proportional to the dynamic execution frequency of the PC. The total number of faults is provided by the user. The user decides the number based on the error margin and confidence level she/he desires.

  We note, that up to this stage, PCFI would yield the same vulnerability profile (i.e., the same ratio for each type of outcome of fault injections) as the traditional methodology (as implemented in earlier studies [3, 63], and described in Section 2.7), since the fault list has not been pruned yet.

- **Fault Injection Stage:** In this step, PCFI instruments the fault injection handler in SASSIFI to inject a fault based on the PC value. Once the correct execution count for a specific PC is reached, the PCFI handler injects a fault in a destination register of the instruction, as per the fault model described in Section 2.8.

  To reduce the number of fault-injected runs, PCFI keeps track of the vulnerability outcomes of faults injected in each PC. If a particular PC is a frequently executed PC (i.e., it receives a greater number of faults than a threshold), then PCFI first randomly and uniformly picks a "threshold" for the total number of faults corresponding to this PC and executes these runs. If the outcomes of these runs are identical (e.g., all producing DUE), then PCFI eliminates further fault injections to this particular PC, and thus, reduces the number of fault-injected runs. If the outcomes differ for first "threshold" number of faults, then, PCFI continues to inject faults.

  PCFI chooses the threshold to be 1% of total number of faults in the base case. For a 10,000 fault injection campaign, PCFI will track and observe PCs which receive 100 or more faults. This choice is guided by our experiments where we find that 1% of uniformly distributed faults serve as a good indicator if a particular PC results in the same outcome, without affecting the accuracy, independent of the fault list size for the applications studied (Section 5.3).

Advanced optimization techniques, such as tracking the relative proportion of outcome types for PCs that produce different outcomes, can be used to limit the number of injections in PCFI-unfriendly PCs. This approach can further reduce the number of fault injections, but only produces very moderate time savings.

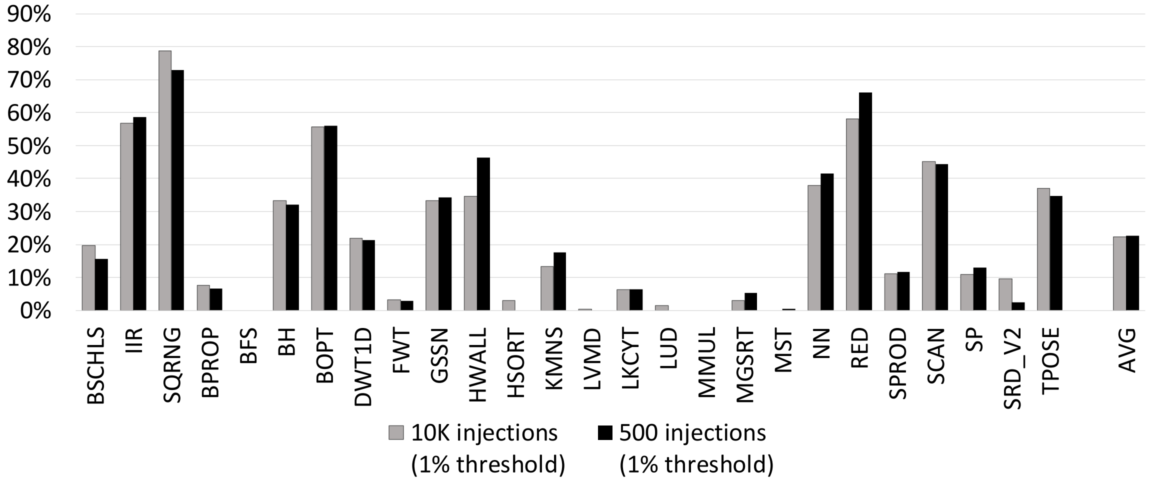


Figure 5.4: Percent reduction of the total number of injections, after filtering out excess faults in PCs.

5.3 PCFI Evaluation and results

We evaluate PCFI on a mix of 26 applications, selected from 2 well-known benchmark suites for GPU applications, the Rodinia suite of applications [115] and the CUDA-SDK [116]. These applications are representative across a range of application domains: linear algebra, graph traversal, image processing, physics simulation, fluid dynamics, signal processing, financial computation, data mining, and pattern recognition.

We present our results in two categories. First, we look at the fault list reduction and corresponding time savings achieved from the baseline fault list (i.e., 10K random fault injections). Then, we will evaluate the accuracy of our methodology by comparing our results to a traditional fault injection campaign performed using SASSIFI.

5.3.1 Fault List Reduction

Starting with the original number of injections, we apply our methodology to skip fault injections in dominant PCs, where multiple fault injections produce the same outcome. We compare our results to two different baselines: 10,000 random injections (a 95% confidence level, with a 0.98% error margin) and 500 random injections (a 95% confidence level, with a 4.4% error margin). We limit the maximum number of injections in the frequently executed (or dominant PCs) which produce identical outcomes to using 1% of the total number of injections performed in the baseline/traditional case. This threshold can be adjusted to a higher or lower value, however we observed that changing the threshold does not affect the accuracy of the results for the baseline since there are a sufficient number of injections (e.g., 500 faults). In our experiments, 1% produces effective results for both a high number and low number of baseline random injections, with minimal impact on accuracy, as is shown later.

Figure 5.4 shows the percent reduction in the number of faults injected. Our results show that PCFI is effective in both cases as compared to the baseline case of 10K injections, and even when we reduce the number to 500 injections. For 10K injections, PCFI achieves a fault list reduction of 22.38% on average, resulting in an average of 7,762 faults injected per application. For a 500 injections baseline, PCFI reduces the number of injections by 22.68% on average, resulting in 387 faults injected per application on average.

The magnitude of the fault list reduction changes based on the specific workload characteristics. For our 10,000 fault baseline, the fault list reduction varies between 0 and 78.8% (SQRNG).

Our analysis reveals that the wide range in fault list reductions can be attributed to the following factors:

1. **The Number of Dominant PCs in the Application**: If there are fewer dominant PCs in the application, it means that the majority of the execution is confined to a small set of the PCs. This increases the opportunity for PCFI to reduce the number of injections that target those PCs.

2. **Execution Frequency of the Dominant PCs**: As we mentioned earlier, the number of faults targeting a particular PC is proportional to its execution frequency. Therefore, execution frequency of all PCs dictates how faults are distributed across the application. This implies that having fewer dominant PCs that are executed significantly more often, compared to other PCs, offer a bigger margin for fault reduction through PCFI. In the case where execution is

spread out across many dominant PCs, the margin for fault reduction per PC reduces and PCFI will not be able to offer much savings. It is important to note that in GPU programming, best practices include developing code with high temporal locality, where most of the run time is spent in a limited amount of code [117].

Embarrassingly parallel applications tend to spend most of their execution time in a limited number of code blocks. These applications have *highly dominant PCs*: dominant PCs, where the number of executions that are not only significantly higher (by several orders of magnitude in terms of their frequency) than the non-dominant PCs, but also represent a large percentage of all dynamic instructions. Additionally, when the number of highly dominant PCs is small, the majority of the faults will be injected into the highly dominant PCs, increasing the opportunity for PCFI to reduce injections. These instances represent applications that have loops with high trip counts. These applications also have fewer conditional branches during dynamic execution.

The application `reduction (red)` experiences a very significant fault list reduction (58% for 10K faults, and 66% for 500 faults). We find that this application spends 99% of its dynamic execution in 12 unique PCs (out of a possible 86 unique PCs present in the application), with the number of executions 3 orders of magnitude greater than the remaining PCs in the application (see Figure 5.2a). This application is a classic parallel GPU application, that can greatly benefit from using PCFI.

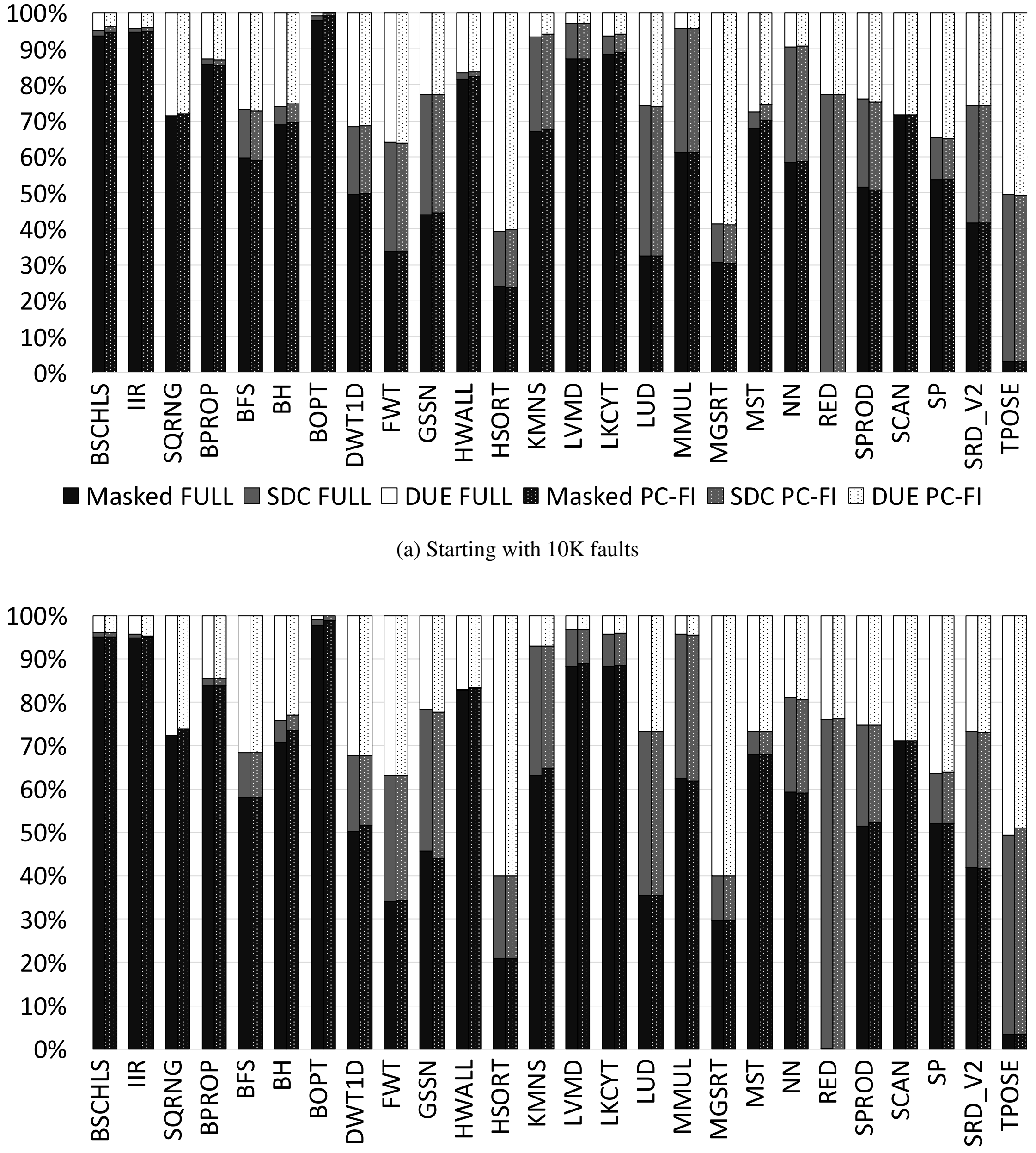
On the other hand, the execution time for the `minimum-spanning-tree (mst)` application is spread out fairly evenly across a large number of unique PCs (see Figure 5.2e). In this program, 95% of dynamic instructions are confined to 233 unique PCs (out of a total of 615 unique PCs). This application does not present many opportunities to use PCFI for fault list reduction.

Table 5.1: Time savings using PCFI

N	Traditional	PCFI
10,000	48.15 days	*36.16 days*
500	57.73 hours	*42.57 hours*

5.3.2 Time savings

Table 5.1 shows time for a fault injection campaign for the 26 applications in our testing framework. We compare the time expended using the traditional approach versus PCFI. Even with an already small number of injections per application (i.e., 500), PCFI offers significant time savings of 1̃5 hours. This time savings can rapidly increase when researchers perform multiple fault injection runs, as mitigation efforts need to be evaluated and new resilience mitigation strategies are being devised.



(a) Starting with 10K faults

(b) Starting with 500 faults

Figure 5.5: Results for PCFI compared to a traditional fault injection campaign. In 5.5b, we show outcomes for PCFI when starting with 500 faults (N=500). In 5.5a, we start with a list of 10K faults (N=10K).

5.3.3 Accuracy of PCFI compared against traditional fault injection, with 10K and 500 random injections.

Table 5.2: Min, Max and Average error observed between traditional fault injection and PCFI for Masked, SDC and DUE outcomes. The error is the difference in the observed fault breakdown percentage between PCFI and the traditional fault injection methods.

N	Outcome Type	Min Error	Max Error	Avg Error
10K	Masked	0%	2.49%	0.46%
	SDC	0%	0.90%	0.17%
	DUE	0%	2.04%	0.42%
500	Masked	0%	2.66%	0.53%
	SDC	0%	1.82%	0.45%
	DUE	0%	1.82%	0.32%

Next, we evaluate the accuracy of the fault injection campaign performed using PCFI. In Figure 5.5, we show the results of a fault injection campaign performed using PCFI, along with a traditional fault injection campaign. These results show that PCFI achieves the same results as the traditional fault injection campaign. They also show that for all 26 applications, a fault injection campaign performed with PCFI can produce results that are consistently well within the error margin for all fault outcome categories. We also show the average, maximum, and minimum error between PCFI results, and the results from a traditional fault injection in Table 5.2. Notably, PCFI achieves an average error in accuracy of less than 0.55% across all cases, and maximum accuracy error of 2.49%.

To demonstrate that PCFI produces better results than a traditional fault injection strategy using a reduced fault list (the same number of faults as PCFI), but injected using the traditional method (as described in Section 2.7), we selected the 3 applications which represent the smallest number of faults, starting with the original fault list of 500 injections: `SobolQRNG` (136 faults), `reduction` (170 faults) and `IIR` (207 faults). We evaluate whether the reduced number of injections obtained with PCFI could provide similar error margins as compared with a traditional methodology. *PCFI* uses 136, 170, and 207 faults, whereas the baseline uses a fault list of 500. Now, these faults (136, 170, and 207 faults) are being injected randomly in the respective benchmarks to demonstrate that PCFI is more effective than randomly pruning the fault list by the same amount.

We perform a fault injection campaign for each approach (PCFI and the traditional method with a reduced fault list) 5 times to check the consistency of the results. For each campaign, we take the difference between the results of the technique and the results of a baseline using 500 random

		PCFI			**Traditional**		
App	**Outcome**	**Avg. Error**	**Max Error**	**Std Dev**	**Avg Error**	**Max Error**	**Std Dev**
SQRNG	Masked	1.145%	1.731%	0.406%	3.124%	4.591%	1.135%
	SDC	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%
	DUE	1.145%	1.731%	0.406%	3.124%	4.591%	1.135%
RED	Masked	0.163%	0.285%	0.109%	0.527%	1.485%	0.536%
	SDC	1.196%	1.531%	0.237%	2.363%	5.245%	2.450%
	DUE	1.148%	1.449%	0.272%	2.889%	5.553%	2.287%
IIR	Masked	0.887%	1.452%	0.373%	2.118%	2.936%	1.022%
	SDC	0.476%	1.000%	0.385%	0.780%	1.899%	0.713%
	DUE	0.604%	1.246%	0.548%	1.560%	2.828%	0.883%

Table 5.3: Comparing two techniques: 1) PCFI and 2) traditional fault injection with a reduced fault list (as offered by PCFI with 500 faults) against a baseline of 500 fault injection campaign.

fault injections. We find the differences in the maximum, average, and standard deviation across the 5 fault injection campaigns. The baseline is obtained by taking the average results of 5 fault injection campaigns of 500 faults. The results are presented in Table 5.3, showing PCFI consistently produced similar results as the baseline (max error being 1.73%), while the equivalent number of faults in the traditional methodology shows a much higher variation over the five fault injection run (max error being 5.55%). These results indicate that, compared to the full traditional methodology, PCFI offers a lower margin for error (2x better in many cases), and a more consistent assessment (lower standard deviation).

5.4 Summary of PCFI

PCFI significantly reduces the number of fault injection runs needed during a fault injection campaign, without comprising the accuracy of vulnerability assessment of GPU programs. This is the first work that exploits PC-based behavior for predicting the soft-error vulnerability of instructions. We show we can reduce the time spent in fault injection campaigns by up to 78% for a wide variety of GPU benchmarks. The proposed method is implemented in SASSIFI, an open-source GPU fault injector. This methodology is able to help researchers accelerate live-system fault-injection studies and evaluate the resilience of different mitigation strategies.

Chapter 6

Characterizing Vulnerability Phase Behavior in GPU applications

In this chapter we explore resilience characteristics of GPU programs. We find that they can significantly change during program execution and that these characteristics repeat over time. Interestingly, these temporal-based resilience characteristics of GPU programs do not align or correlate well with the performance phases of GPU programs. Furthermore, we have identified that temporal changes in the vulnerability behavior during a kernel execution tend to coincide more with changes in basic block execution paths.

6.1 Motivation

Time varying behavior of applications is a well explored subject in the CPU domain. Previous work [96, 118, 119, 120] has shown that performance and power features of CPU applications exhibit a time-varying behavior characterized as *phases*. These phases are time periods during which applications exhibit the same performance or power behavior.

Researchers in both academia and industry have used the phase-based behavior of workloads to reduce the time needed for architectural simulation and performance evaluation. While many studies have explored application phase behavior in terms of performance, only a few of them have been applied to GPU execution. Huang *et al.* introduced TBPoints [105] for accelerating simulation of GPU kernels. TBPoints uses a feature vector that takes into account the total amount of work to be done and the memory accesses associated with a GPU kernel. Kambadur *et al.* introduced GTPin [106], a tool which accelerates the simulation of large GPU programs. GT-Pin profiles a program and collects a number of features that include memory operations and kernel parameters. These features are then utilized to select a few representative regions of the program, serving as a model of the full program for simulation.

Nonetheless, in terms of understanding reliability characteristics, time-varying behavior is an under-explored area. Few studies have explored the time-varying behavior of performance and reliability for various CPU programs, but not for GPU programs, which are fundamentally different from their CPU counterparts due to architecture and execution style differences [107]. Previous work has also explored time-varying behavior of GPU applications from the performance analysis perspective [106, 105]. However, time-varying vulnerability behavior of GPU applications has not been well explored. There is little guidance on which program features can best characterize the runtime vulnerability of GPU programs. This thesis is the first work to evaluate how basic block vectors, in the context of GPU applications, are related to time-varying reliability behavior.

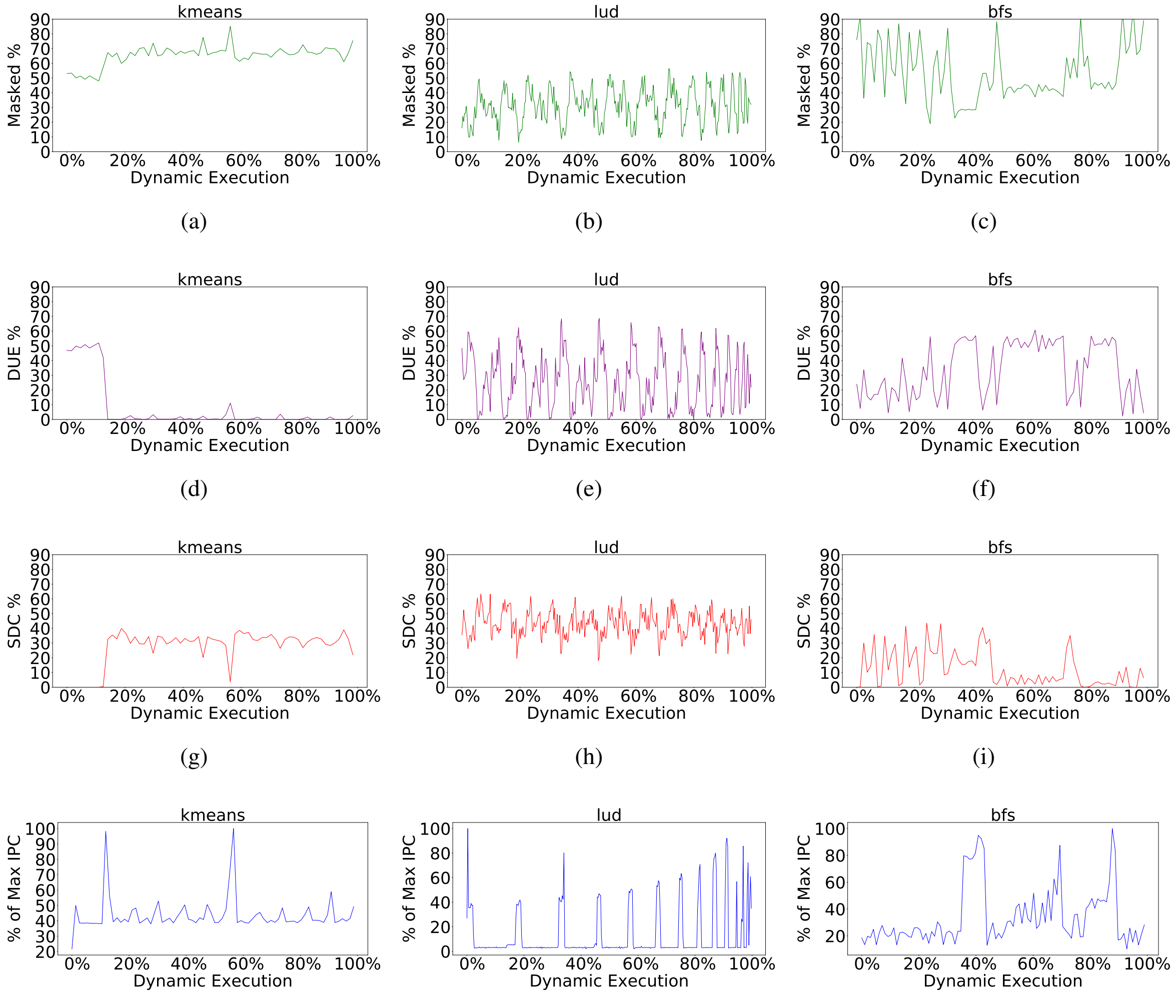
In Chapter 7, we present how the time-varying behavior can be exploited for accurately capturing the resilience characteristics of GPU programs Chapter 8. We will present a use-case of the application of this time-varying behavior analysis for efficient placement of mitigation strategies, contributing to a lower overhead of these strategies.
This chapter makes the following contributions:

- We show that the vulnerability of GPU programs exhibit phase-based behavior, (i.e., GPU program vulnerability behavior changes and repeats over time).
- We leverage execution profiles of GPU programs to capture this time varying behavior of their vulnerability characteristics. We group dynamic instruction sequences in GPU applications based on their resilience characteristics.

The results of this characterization study can be further leveraged in the design of remediation mechanisms to handle transient faults in GPU applications. System administrators can make use of our findings to apply selective protection techniques (e.g., checkpointing) during the phases of high vulnerability during the execution of GPU applications.

6.2 Time-Varying Vulnerability Behavior of GPU Programs

Prior research has shown that: 1) GPU and CPU programs have repetitive, time-varying performance characteristics (i.e., performance-based phases are present) [105, 106], and 2) GPU programs show different resilience characteristics (i.e., different GPU programs have different fault



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 6.1: Three applications taken from three different problems domains, k-means clustering (`kmeans`) (pattern recognition), LU Decomposition (`lud`) (linear algebra), and Breadth-First-Search (`bfs`) (graph traversal). Both vulnerability and performance (IPC) metrics are plotted, illustrating their time-varying and repetitive behavior over program execution. Notably, GPU program vulnerability and performance do not seem to be well correlated throughout their execution lifetime.

injection outcome probabilities in terms of masked, SDC, and DUE) [3, 24]. This chapter will show that: 1) the resilience characteristics of GPU programs change significantly during program execution and these characteristics show repetitive, time-varying behavior, and 2) these repetitive, time-varying, resilience characteristics of GPU programs do not align or correlate well with the performance phases of GPU phases.

To demonstrate the time-varying resilience characteristics of GPU programs, we conduct a simple experiment where we randomly inject faults throughout the program execution and observe outcomes (i.e., masked, SDC, DUE). We divide the dynamic execution of a program into **dynamic instruction sequences**. An instruction sequence contains of a number of dynamic instructions in program execution order. The size of an instruction sequence is chosen on to be from 1M to 100M instructions, depending upon the total number of dynamic instructions in the program, which can range from 100M to 100B for the workloads we evaluate. The choice of these instruction sequence sizes is in line with related work studying program runtime behavior [96, 106]. To characterize the vulnerability behavior present in each instruction sequence, we injected enough faults into each instruction sequence to reach a 5% error margin and a 95% confidence level [57]. This corresponds to approximately 350 faults injected in each instruction sequence and over 50,000 total injections per application. The faults in each instruction sequence are evenly distributed across the dynamic execution of the program.

Figure 6.1 shows: (a) the percentage of fault injection outcomes in each instruction sequence that are either masked, or resulted in DUEs, or SDCs, and (b) the performance metric (IPC) for each sequence for the applications `kmeans`, LU decomposition (`lud`), and `bfs`. We obtained similar trends for other applications in our test suite of 24 applications, but do not show them for brevity. The IPC measurements were obtained using SASSI [26] instrumentation, collected by recording a count of the number of GPU cycles used during each instruction sequence.

From our results, we make a few observations. First, as expected, the vulnerability outcome probabilities change across different applications (e.g., `bfs` has higher avg. DUE rate than `kmeans`). However, these characteristics also change significantly during program execution for a given application, although such phases are not straight-forward to extract from the vulnerability outcome profile in Figure 6.1.

Second, as expected, GPU programs exhibit phase-based behavior when considering performance characteristics such as IPC, as shown in Figure 6.1 (we saw similar trends for cache miss rates and other performance-related metrics). Interestingly, we observe that the time-varying behavior of vulnerability and performance do not necessarily always track each other or align with each other. In other words, these results indicate that a program's performance behavior can not be used as a proxy for the program's resilience behavior. This motivates the need to better understand the repetitive, time-varying, resilience characteristics of GPU programs and investigate ways to exploit such an understanding as demonstrated in this work.

To further explore the degree of variation in program vulnerability across instruction

	Masked		SDC		DUE	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
backprop	74.62%	17.06%	0.07%	0.49%	25.31%	17.12%
bfs	55.86%	22.22%	16.31%	15.37%	27.83%	18.45%
gaussian	21.20%	5.27%	49.21%	6.16%	29.58%	5.93%
kmeans	43.88%	13.67%	16.03%	9.72%	40.09%	13.41%
lud	19.56%	8.34%	42.15%	10.19%	38.28%	16.07%

Table 6.1: GPU programs show a large standard deviation in terms of the number of Masked, SDC and DUE outcomes across dynamic instruction sequences.

sequences, we measured the percent of masked, DUE and SDC outcomes for all the instruction sequences and report the standard deviation for all three fault-injection outcome types. Table 6.1 shows the probability of the different outcome types, as well as the standard deviation, for five applications from different application domains. We only include results for applications where the fault injection campaigns could be completed in a reasonable amount of time, while ensuring a significant number of injections in each dynamic instruction sequence (more than a thousand injections per instruction sequence were needed in many cases to achieve high accuracy).

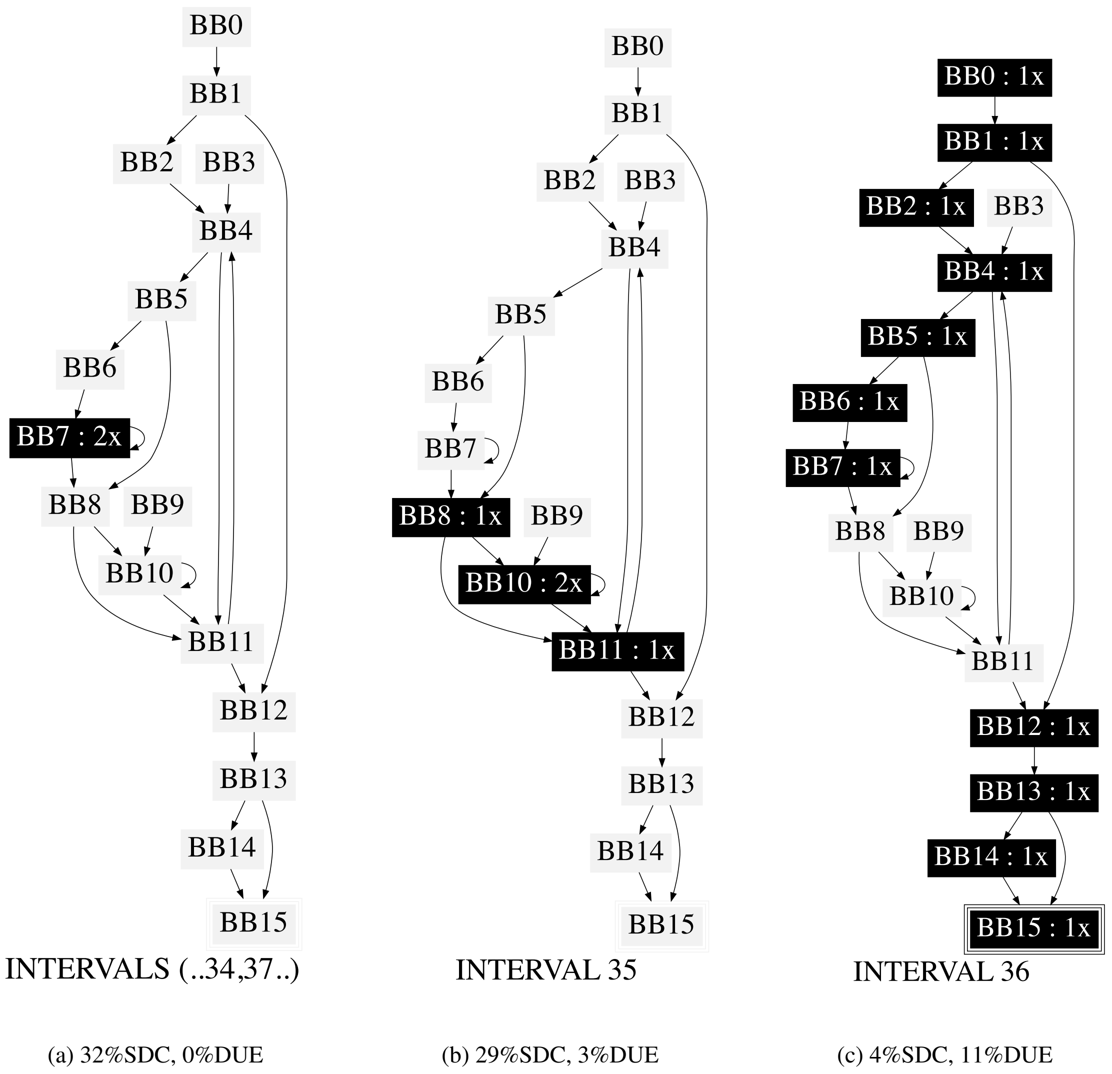
Our result shows that vulnerability in GPU programs varies significantly during program execution (i.e., across different dynamic instruction sequences). For example, in `backprop`, the average probability for a fault to be masked, an SDC or a DUE is 74.62%, 0.07% and 17.12%, respectively. However, the standard deviations for these outcomes are 17.06%, 0.49% and 13.00% over all instruction sequences. To understand the reasons for the variation in outcomes, we perform program-level analysis.

Factors Affecting the Variance of Vulnerability Behavior

Our analysis of the program runtime behavior reveals that the variability in resilience characteristics during program execution can be attributed to the differences in execution characteristics across "kernels" of a given program, as well as changes in the characteristics of the dynamic execution within a kernel execution.

1. **Different kernels show different vulnerability outcomes**

   GPU programs are composed of one or more kernels which can be executed multiple times and in different orders. As expected, the variability in resilience characteristics during program execution is correlated with the kernel being executed.



(a) 32%SDC, 0%DUE

(b) 29%SDC, 3%DUE

(c) 4%SDC, 11%DUE

Figure 6.2: Code block executions for instruction sequences surrounding execution sequence #36 in the `kmeans` application. From sequence #34 to sequence #36, the DUE probability goes up from 3% to 11%, and the SDC probability goes down from 32% to 3%. Black boxes represent basic blocks that are executed during the sequence.

For example, the application `kmeans` (Figures 6.1a, 6.1d, 6.1g) is composed of two kernels. The dynamic execution of the kernels of this application is in this order: `kernel1 - kernel2 - kernel2`. The first kernel is executed during the first 15% of the dynamic execution, where most of the faults causing DUEs occur. The second invocation of `kernel2` occurs around the 55% mark in the dynamic execution, where there is a significant drop in the SDC probability. These two kernels exhibit drastically different resilience characteristics. All of the SDC's in this application come from faults in the second kernel.

Other applications also exhibit this same behavior. For example, `lud` exhibits 15 repeated patterns (Figures 6.1b, 6.1e, 6.1h), which become shorter towards the end of the execution. This coincides with 15 invocations of 3 kernels in the application, with their execution becoming shorter (fewer dynamic instructions) towards the end of the application. For `bfs`, two kernels that make up this application are invoked 8 times. However, starting with their 5th invocation (around the 30% mark in the dynamic execution), one of the kernels executes significantly more dynamic instructions for its 5th, 6th and 7th invocation. This can be observed in the resilience patterns in Figures 6.1c, 6.1f, 6.1i, the 3 invocations with long execution are apparent in Figure 6.1f.

2. **The same kernel may also exhibit different resilience characteristics during its execution**

   While differences in the resilience characteristics across different kernels is expected, our analysis reveals that the resilience characteristics may change significantly within a kernel's execution, depending on the chain of basic blocks executed. For example, as shown in Figure 6.1), we found that around the 55% mark in the dynamic execution of the `kmeans` application, the probability for a fault to result in an SDC drops to 3%, from 32%. During the same dynamic instruction sequence (at sequence #36, sequence numbers are not shown in the figure), the probability for a fault to result in a DUE spikes to 11% (from 3%). As we noted, this change in vulnerability characteristics occurred during the re-invocation of `kernel2`. We looked closer into the reasons behind this change in resilience behavior.

   Our code profile showed that a different code region from the same kernel dominated execution during this instruction sequence (sequence #36). Figure 6.2 shows a snapshot of the control flow graph from the instruction sequences before and after sequence #36. We see the basic blocks (BBs) that executed, along with the normalized number of executions for each basic block. For `kmeans`, the majority of the dynamic executions is confined to basic block 7 (BB7), which is an inner loop that updates the distance of a point to a respective cluster. This can also be noted in the graph for the instruction sequences before #34, and after #37. However, the execution graph of this application changes during sequences #35 and #36, which leads to a change in the vulnerability behavior of the application during these instruction sequences. Upon further analysis, we discovered and validated that these trends also occur in other applications: sudden changes in the vulnerability behavior during a kernel execution tends to coincide with changes in basic block execution paths. These observations motivated us to investigate how basic block execution paths can be utilized to assess the vulnerability of programs toward transient faults.

6.3 Resilience Groups: Capturing Dynamic Vulnerability Behavior

In the previous section, we found that the execution path, analyzed at a basic block granularity, can identify and distinguish instruction sequences which show different vulnerability behaviors. In this section, we describe how we can leverage basic block based information to identify and group dynamic instruction sequences that have similar characteristics. More specifically, we propose the concept of **resilience groups** which can be used to accurately represent the resilience characteristics of the whole program.

While the vulnerability behavior in a program tends to exhibit repeated patterns, the length of these patterns varies throughout the execution of a program and across different programs, as discussed in Sec. 6.2. Therefore, capturing the vulnerability behavior accurately, and representing it concisely, is a challenging task. Our proposed concept of **resilience groups** addresses this challenge. A program's resilience characteristics can be represented by a number of resilience groups, where each resilience group is a set of dynamic instruction sequences that have similar vulnerability behavior (i.e., the masked, SDC, and DUE outcome probabilities from a fault-injection campaign).

We build resilience groups without having to perform a large number of fault injections at every instruction sequence. We had earlier shown that basic block level analysis can identify and distinguish instruction sequences which show different vulnerability behaviors. Therefore, we leverage **Basic Block Vectors (BBV)** to capture a program's behavior during an instruction sequence [96]. The BBV is a vector that keeps information about the basic block execution frequency for each sequence of a program. The basic block vectors are then weighted by the number of instructions present in the basic block.

Figure 6.3 describes the steps involved in our characterization of time-varying vulnerability behavior. After the collection of BBV's for each instruction sequence in a program, our next step is to group the instruction sequences with similar characteristics. We call a group of instruction sequences with similar characteristics a **resilience group**. We apply k-means clustering to produce this grouping, with the number of clusters (*m*) varied from 1 to 15. For each *m*, we note the *silhouette score* [121]. The silhouette score is a measure of how well each sequence fits within its cluster. The number of clusters *m* that yields the highest silhouette score is selected and the corresponding number of k-means clusters is the number of resilience groups contained in the program.

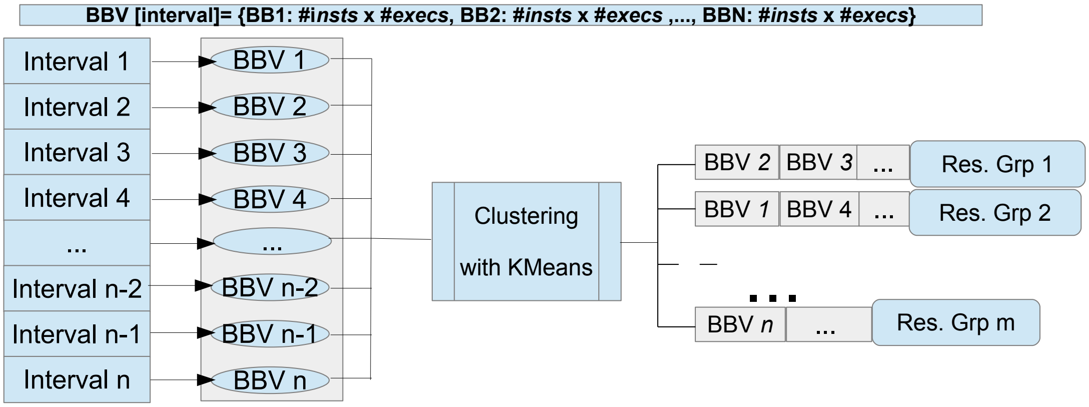


Figure 6.3: Forming resilience groups by combining different dynamic instruction sequences.

Traditionally, researchers have used BBVs for performance characterization and prediction purposes for CPU architectures. Previous studies have also shown that for GPU architectures, BBVs are not suitable for performance prediction, unless they are combined with other metrics such as memory access intensity [105, 106]. However, we show that BBVs are effective in capturing resilience characteristics of programs, even without augmenting them with other techniques. To support this claim, we present results to show that instruction sequences within the same resilience group, formed using our BBV methodology, have very similar resilience characteristics.

We perform fault injections into 10 randomly selected instruction sequences belonging to the same resilience group in our applications. We injected faults in 10 instruction sequences per resilience group as an alternative to performing an exhaustive campaign of 800K+ injections per application, in order to reach a 95% confidence level, with a 5% error margin, in all instruction sequences across all applications.

Next, we sum the standard deviations from each resilience group, weighted by the percentage of total application instruction sequences that each resilience group represents. A low standard deviation means that there was little variation between the instruction sequences in each resilience group of the program, and confirms that our methodology appropriately grouped the dynamic instruction sequences with similar resilience characteristics.

In Table 6.2, we list all the applications evaluated in this study. To achieve a representative range of parallel programming styles and application domains, we selected twenty-four applications with a range of different characteristics from three GPU benchmark suites: the *Rodinia* benchmark suite for heterogeneous computing [115], the *CUDA-SDK* suite of applications [116], and the

LonestarGPU suite of irregular GPU applications [122]. The benchmarks selected are representative across a range of application classes: algebraic applications, particle simulation, physics simulation and fluid dynamics.

Table 6.2 also presents the number of resilience groups obtained for each application using our method, and includes the standard deviations within the resilience groups for each type of outcome. We observe that the standard deviations are quite low (less than 5% in most cases), indicating that our methodology correctly groups sequences with similar resilience characteristics. Our results reveal that the average number of resilience groups per application is less than four for many benchmarks.

In the next chapter, we exploit this finding to reduce the number of fault injections when assessing how vulnerable a program is to transient faults.

Table 6.2: Benchmarks used in our experiments and the number of resilience groups found in each application. A resilience group is a set of instruction sequences with similar resilience characteristics.

Application	**Domain**	**Number of resilience groups**	**Standard Deviation in Resilience groups**		
			Masked (%)	**SDC (%)**	**DUE (%)**
BlackScholes	Financial computation	4	2.38%	1.87%	3.21%
IIR	Signal processing	2	1.21%	0.4%	1.06%
SobolQRNG	Financial computation	2	4.87%	2.98%	2.06%
bfs	Graph traversal	4	3.54%	3.24%	2.76%
bh	Astrophysics	2	1.76%	2.56%	0.8%
binomialOptions	Financial computation	2	0.73%	0.75%	0.03%
cfd	Fluid dynamics	2	3.23%	2.10%	1.32%
dwtHaar1D	Image processing	3	6.35%	2.71%	4.18%
gaussian	Linear algebra	5	7.45%	4.34%	5.3%
heartwall	Image processing	2	1.98%	1.23%	0.74%
hotspot	Physics simulation	2	4.67%	2.83%	3.03%
hybridsort	Sorting	2	3.45%	2.18%	1.87%
kmeans	Data mining	2	2.78%	2.99%	0.85%
lavaMD	Molecular dynamics	2	4.31%	3.88%	2.14%
leukocyte	Fluid dynamics	3	1.82%	0.65%	1.45%
lud	Linear algebra	2	3.86%	4.85%	4.02%
matrix_mul	Linear algebra	3	1.32%	1.03%	0.78 %
mst	Graph traversal	10	2.33%	1.91%	2.09%
nw	Bioinformatics	2	3.05%	1.08%	3.16%
reduction	Data reduction	6	1.45%	0.54%	0.93%
sp	Graph traversal	2	2.32%	1.12%	1.28%
srad_v1	Image processing	9	2.21%	0.23%	2.09%
srad_v2	Image Processing	2	1.09%	2.42%	1.27%
transpose	Linear algebra	4	1.87%	1.23%	3.98%

6.4 Summary on the Characterization of the Vulnerability Phase Behavior of GPU applications

In this chapter, we described our observations on phase behavior of vulnerability in GPU applications, as well as the factors contributing to these phases. We presented a characterization methodology and experiments for characterizing this runtime vulnerability of GPU applications, with just one single application execution.

With our characterization methodology, we are able to form resilience groups. Resilience groups are portions of a program that are similar in resilience characteristics. Resilience groups allow us to capture the resilience of a program with only a few representative parts. The knowledge of which parts of a program where resilience will behave similarly is powerful in addressing the reliability problem, as it can greatly contribute to powerful mitigation strategies. Chapter 8 demonstrates how we can use this concept for efficient application of a mitigation strategy.

Chapter 7

Spoti-FI: Reducing Fault Injection Time via Resilience Groups

The characterization of the runtime vulnerability behavior of GPU applications, as presented in Chapter 6, is useful in both vulnerability estimation and remediation. Next, we present *Spoti-FI* [123], a methodology that leverages our proposed runtime vulnerability characterization to guide systematic fault injection campaigns in a range of GPU applications. Using *Spoti-FI*, we can reduce the number of fault injections needed by an order of magnitude, as compared to the traditional fault injection campaign. *Spoti-FI* can maintain the same accuracy of vulnerability estimation as an exhaustive fault injection campaign. We use the time-varying behavior in an application to identify representative sections of that GPU application, thereby reducing the total number of faults necessary for fault injection campaigns.

7.1 Spoti-FI Methodology

To reduce the required number of injections for a fault injection campaign, we leverage the resilience group concept (Chapter 6, Section 6.3). Our approach exploits the observation that faults occurring in sequences in the same resilience group result in the same outcome, and hence, can be pruned intelligently by leveraging the repetitive, though time-varying, resilience characteristics of GPU programs.

For each resilience group, we carefully select one representative instruction sequence in which to perform fault injections, as described below. The steps required to exploit the resilience groups for fault injections are as follows:

1. First, we select one target instruction sequence for each resilience group in the program. We make use of k-means clustering to select the target instruction sequence within a group. Our k-means implementation uses the Euclidean distance between the instruction sequences to establish the similarity between them. We also use the Euclidean distance, scannning all the instruction sequences in each group (or in a k-means cluster) and select the instruction sequence that is the closest to the center of each cluster.

2. We then perform a fixed number of injections in the selected instruction sequences. We instrumented our fault injector to allow it to inject faults during specific instruction instruction sequences for the application of interest. The faults injected into each instruction sequence are uniformly distributed across the dynamic instructions of the sequence.

3. Finally, we compute the final fault injection outcome rates for the target applications. To do this, we use the occurrence frequency of each resilience group. We call this frequency $Freq(group_i)$. The frequency of group i is the ratio of the number of instruction sequences that make up group i to the total number of instruction sequences in the program:

   $$Freq(i) = \frac{\text{number of sequences of group } i}{\text{total number of sequences}}$$

   The **final rate for each outcome category** (SDC, DUE or Masked) is then computed as:

   $$\sum_{i=1}^{n} \text{Outcome Rate}_{\text{targetInstSequence}(i)} * Freq(i)$$

   where i is a resilience group, targetInstSequence(i) is the representative instruction sequence for group i, selected for injection (step 1), and $Freq(i)$ is the frequency of occurrence of group i (step 3).

7.2 Results and Analysis

Our experiments were performed with SASSIFI on an NVIDIA Kepler K20 GPU, a device based on the GK110 architecture [124]. The K20 has 5GB of global memory. The configuration of the GPU does not impact our results because faults are injected into the live application state. All the applications use the CUDA 7.0 toolkit.

We compare our fault injection results with fault injection campaigns performed with 10,000 random faults in each of the studied applications. As mentioned before, this number of injections is in accordance with other studies on resilience of GPU applications [63, 125, 29].

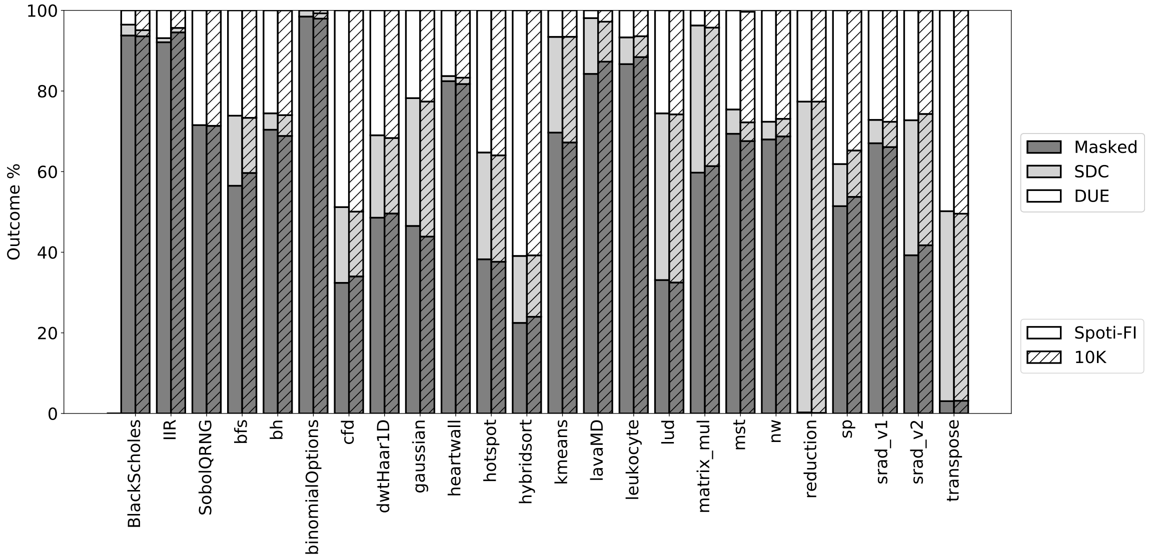


Figure 7.1: Results for fault injections performed using Spoti-FI, compared with results of traditional fault injections with 10K faults per program. Using Spoti-FI, we injected an average of 1,317 faults per application, for an average error of 1.42% for masked outcomes, 0.88% for DUE outcomes, and 3.92% for SDC outcomes, compared to a fault injection experiment of 10,000 injections per application.

Figure 7.1 shows the results of the faults injected using Spoti-FI, and compares against traditional fault injections results with 10K injections per application. For the Spoti-FI experiments, we injected a number of faults sufficient to achieve a 95% confidence level and a 5% error margin in one instruction sequence of each resilience group, for each application [57]. These results show that Spoti-FI is almost as accurate as an exhaustive fault injection campaign, with an average error rate below 1.5% across different outcome types and all applications.

The accuracy of Spoti-FI, as compared to 10K random injections, is also summarized in Table 7.1. Our results demonstrate that Spoti-FI provides accurate reliability measurements via leveraging resilience groups. Notably, Spoti-FI is able to provide accurate results even for cases where a particular outcome type has a low probability. For example, a fault in `BlackScholes` has the chance 2.70% of the time to result in an SDC, and 3.53% in a DUE. In `IIR, binomialOptions` and `heartwall`, the chances that an injected fault results in a DUE is 1.02%, 1.53% and 1.27%, respectively. With resilience groups, Spoti-FI is able to accurately predict these probabilities without running an extensive fault injection campaign.

Table 7.1: Min, Max and Average error observed between traditional fault injection with 10K faults and our methodology for Masked, SDC and DUE outcomes. The error is the difference in the observed fault breakdown percentage between our methodology and the traditional fault injection methods.

Outcome	Min Error	Max Error	Avg Error
Masked	0.01%	3.18% (*bfs*)	1.42%
SDC	0%	3.92% (*lavaMD*)	1.22%
DUE	0.001%	3.37% (*sp*)	0.88%

Table 7.2: Number of faults injected based on the number of resilience groups per application and fault list reduction from the standard 10K injections. For each resilience group, one representative instruction sequence is selected for injection. Because the resilience characteristics within one resilience group are stable, we injected only enough faults in each resilience group to ensure 5% margin of error with a 95% confidence level for the error injection.

Application	Number of Spoti-FI injections	Fault List Reduction	Application	Number of Spoti-FI injections	Fault List Reduction
BlackScholes	1,540	6.49x	kmeans	766	13.05x
IIR	768	13.02x	lavaMD	768	13.02x
SobolQRNG	770	12.99x	leukocyte	1,152	8.68x
bfs	1,536	6.51x	lud	768	13.02x
bh	768	13.02	matrix_mul	1,149	8.70x
binomialOptions	770	12.99x	mst	3,850	2.60x
cfd	770	12.99x	nw	768	13.02x
dwtHaar1D	1,155	8.66x	reduction	1704	5.07x
gaussian	1,920	5.21x	sp	768	13.02x
heartwall	772	12.95x	srad_v1	3,465	2.89x
hotspot	770	12.99x	srad_v2	766	13.05x
hybridsort	768	13.02x	transpose	1,538	6.5x

7.3 Discussion

Table 7.2 shows the number of injections that we performed in each application using Spoti-FI. With our approach, we discovered that vulnerability characteristics of most GPU applications can be captured by a small number of resilience groups. For example, for 14 out of 24 applications, two resilience groups were enough to characterize the resilience properties. This implies that we are able to group the BBV's of each of these 12 applications in two categories, and capture the resilience of these programs by limiting the injected faults to only two dynamic instruction sequences.

Spoti-FI can be leveraged by application developers to obtain early vulnerability estimates, which can help guide developing more resilient code during the development cycle of a program. Our methodology can also be applied to not only identify vulnerable regions of a program, but also accelerate a fault injection campaign.

Using our methodology, we are able to greatly reduce the duration of fault injection experiments. As shown in Table 7.2, on average, Spoti-FI injects approximately 1,317 faults in each application, achieving a difference (compared to results from 10K injections) of 1.42% on masked outcomes, 0.88% on DUE outcomes, and 3.92% on SDC outcomes. With significantly fewer injections, Spoti-FI can cut down the vulnerability estimation time by a significant factor. This makes it feasible to perform a fault injection experiment on a single GPU in 5.5 days.

7.4 Summary on Spoti-FI

Leveraging the vulnerability characterization methodology presented in Chapter 6, we developed Spoti-FI, a framework that allows us to spot the best places (dynamic instruction sequences) for fault injection. We described Spoti-FI, and demonstrated its utility with fault injection experiments on workloads selected from three benchmark suites.

Our novel methodology provides vulnerability measurements that are very close in fidelity to exhaustive fault injection campaigns, injecting 1-2 orders of magnitude fewer injections. Carrying out a proper fault injection campaign involves injecting 10,000 faults into a program that runs for 15 seconds, and would consume approximately 42 hours. Applying Spoti-FI, and injecting 1,317 faults on average per application, a fault injection campaign for a single application would be reduced to approximately 5.5 hours.

Assuming the worst case where we needed to inject 3,850 faults, a fault injection campaign would be reduced to 16.04 hours, a 2.5x reduction. In the best case, we only needed to inject 766 faults in 3.2 hours. In the future, we plan to explore new use cases for our characterization that will contribute to mitigate the effects of transient faults at application runtime.

Chapter 8

Exploiting Resilience Groups for Efficient Fault Mitigation

Table 8.1: Vulnerability characteristics per resilience group

Resilience Group #	Res. Grp. Frequency	Masked %	SDC %	DUE %
gaussian				
1	41.32%	23.25%	49.25%	27.5%
2	17.56%	53.75%	27.5%	18.75%
3	12.03%	40.25%	35.75%	24%
4	14.31%	80.75%	5.5%	13.75%
5	14.77%	75%	9.75%	15.25%
kmeans				
1	13.28%	58.75%	0%	41.25%
2	86.7%	71%	27.25%	1.75%
bfs				
1	10.53%	60.75%	24.75%	14.5%
2	25.00%	66.75%	15.75%	17.5%
3	23.68%	56.25%	32%	11.75%
4	40.79%	49.25%	8%	42.75%
hybridsort				
1	44.44%	32.75	5.5%	61.75%
2	55.56%	14.25	25.5%	60.25%
srad_v2				
1	66.67%	44.5%	34.75%	20.75%
2	33.33%	28.75%	31%	40.25%

Our phase analysis can also be leveraged to adaptively turn on resilience mitigation strategies such as crash or SDC detectors, depending on which instruction sequence is being executed. Spoti-FI identifies multiple resilience groups, which by definition, have different resilience characteristics, while dynamic instruction sequences belonging to the same resilience group have similar resilience characteristics. Hence, without performing an exhaustive fault injection (i.e., faults in all instruction sequences), Spoti-FI can predict the vulnerability outcomes for all instruction sequences based on which resilience group they belong to. Our results (Table 8.1) show that resilience characteristics of different resilience groups differ and present an opportunity to exploit dynamically adjustable resilience mitigation strategies.

Table 8.1 shows the results of fault injections in each resilience group for five applications. In `gaussian`, for two out of five resilience groups, the probability for a fault to be masked is more than 75% and the SDC probability is less than 10%. These two resilience groups account for 29% of the execution of this application. In the other three resilience groups, the probability for a fault to be masked is 53% or less, while the SDC probability is as high as 49%. This knowledge can be useful for applying adaptive resilience mitigation techniques to strike a balance between fault coverage and runtime overhead. In `kmeans`, one of the two identified resilience groups is not prone to DUEs, while the other is significantly more prone to DUEs. Similarly, other applications also exhibit a range of resilience characteristics across different resilience groups. It is important to note that DUEs (*Detected and Unrecoverable Errors*) are crash-causing bit flips that are detected by an application or the device driver, and from which it (application or driver) can not recover. It is worthwhile to detect these bit flips before they lead to crashes in an application.

In this chapter, we explore applying an adaptive resilience mitigation technique in order to strike a balance between fault coverage and runtime overhead. We will demonstrate how combining resilience group characterization with a mitigation strategy, we can dynamically reduce overhead of vulnerability mitigation strategies.

8.1 Mitigation Strategies for GPU

In this section, we describe the mitigation strategy that we use to show how our phase analysis can be beneficial in reducing mitigation overhead. We use ArmorAll [2] as our mitigation strategy.

Mitigation strategies for transient faults are based on redundancy. Examples of these strategies include using Error Correction Codes (ECC) and exploiting the benefits of Redundant

Multithreading (RMT) [126, 127]. A few studies have employed other techniques to enhance the reliability of GPUs [36, 128, 129]. However, as previously mentioned, these techniques introduce substantial overhead in terms of area, power, and performance.

These techniques are generally hardware-based or software-based. Hardware solutions for reliability can be more efficient and lightweight than their software-based counterparts, but they are inflexible and add unnecessary overhead for naturally resilient applications which do not require high levels of reliability (i.e., image processing applications). On the other hand, software-directed approaches such as compiler-based RMT are more flexible, but can introduce significant slow-down due to high synchronization overhead among threads. A more efficient software solution is to apply redundancy at an instruction granularity, as opposed to thread granularity. This approach (instruction-level replication) has been studied extensively on CPUs, but it is still an under-explored research area on GPUs. Prior research has leveraged hardware techniques to optimize instruction duplication on GPUs [130]. Kalra *et al.* introduced pure software solution to bridge this gap by proposing ArmorAll [2]. ArmorAll is a set of pure compiler-based redundancy schemes designed to optimize instruction duplication on GPUs, thereby enhancing their reliable execution.

8.1.1 Choice of Mitigation Strategy: ArmorAll [2]

ArmorAll is a light-weight and portable software solution to protect GPUs against soft errors. ArmorAll consists of a set of pure compiler-based redundancy schemes designed to optimize instruction duplication on GPUs. Based on the resilience scheme opted for by the user, ArmorAll will select a subset of the instructions for duplication in an application. For example, an application that is more susceptible to produce a DUE in the presence of transient faults will benefit more from a resilience scheme that is optimized for crash protection. Similarly, an application that is likely to produce an SDC will benefit from a resilience scheme optimized for SDC protection. This allows adaptable fault coverage for different applications. ArmorAll can provide good fault coverage for an application, with an accuracy of 91.7%. The high coverage provided by ArmorAll comes at an average improvement of 64.5% in runtime when using the selected redundancy scheme, as compared to the existing redundancy schemes [129, 127].

8.1.2 ArmorAll: Resiliency Schemes

ArmorAll works by running a static analysis that selects appropriate instructions for duplication, depending on the goals of the analysis. ArmorAll has three resiliency schemes:

- **AddressArmor** protects the addresses used by memory instructions. Wang *et al.* found that illegal memory accesses and addressing exceptions can be caused by transient faults [131]. Our analyses have also shown that most DUE's are the result of an addressing exception. They are often the result of an address being corrupted either due to bit flips in registers that hold addresses, or the propagation of a fault into one of those registers. Address Armor tracks all instructions that participate in memory address computation. Interestingly, Address Armor is also able to detect SDCs. Since addresses are protected by Address Armor, it can prevent threads from accessing incorrect (but not illegal) memory addresses and reading/writing wrong values, which can eventually lead to an SDC. Therefore, Address Armor is able to protect more than what it was originally designed for.

- **ValueArmor** protects the values written to memory by store instructions. An incorrect output is often a result of corruption caused by either a bit flip in a register that holds an output value, or propagation of a fault into one of those registers. Therefore, the goal of Value Armor (VA) is to track all instructions that contribute to the computation of the output value.

- **HybridArmor** protects both the values and addresses of load and store instructions. The goal of Hybrid Armor (HA) is to track all instructions that contribute to the computation of the output value and addresses.

8.1.3 ArmorAll: Evaluation and Overhead

Table 8.2: ArmorAll: Evaluation and Overhead

ArmorAll **Scheme**	**Increase in** **Dynamic Instructions (%)**	**Effectiveness:** **Failures detected (%)**
AddressArmor	60.7%	100% crash-causing faults
ValueArmor	42.8%	80% SDCs
HybridArmor	90.1%	100% crash-causing faults and 98% SDCs

Although ArmorAll allows programs to detect transient faults, it affects the performance of the applications, as it increases the number of dynamic instructions. This increase in the number of dynamic instructions comes from the duplicated instructions, verification instructions, as well the notification instructions. The percent increase in the dynamic instruction count varies across applications, with an average increase of 60.7%, 42.8% and 90.1% for Address Armor, Value Armor, and Hybrid Armor, respectively. As expected, Hybrid Armor protects both addresses and values, hence it adds more duplication and verification.

Address Armor protects the addresses used by load and store instructions. It is highly efficient and is able to detect 100% of the crash-causing bit flips. Value Armor is able to detect more than 95% the SDCs, for some applications. In other applications, Value Armor can detect over 70% the SDCs. Similar to Address Armor, Hybrid Armor is able to detect all crash-causing bit flips. In terms of SDC detection, Hybrid Armor provides the combined detection capability of Address Armor and Value Armor, and can detect over 98% of the SDCs across all applications. We summarize the overhead and effectiveness of ArmorAll in Table 8.2.

8.2 Application of Phase Analysis to ArmorAll

ArmorAll is an efficient software mitigation strategy. However, the overhead of running ArmorAll is significant (Table 8.2). In this section, we show how we apply our phase analysis, as described in Chapter 6, to ArmorAll to reduce the overhead. This new approach is termed *Selective ArmorAll*.

8.2.1 Selective ArmorAll: Methodology

Our approach is similar to Spoti-Fi (Chapter 7), in that we start out with one representative instruction sequence for each resilience group. Our methodology has two stages. In the first stage, we use our phase analysis to find the instruction sequences that need to be protected. In the second stage, we apply ArmorAll to specific instruction sequences, in order to reduce the number of dynamic instructions that are duplicated, and reduce the overhead of this mitigation strategy.
The detailed steps are outlined as follows:

1. We repeat steps 1 and 2, as outlined in Section 7.1, to select a representative instruction sequence from each resilience group. We use k-means clustering to select the instruction sequence for each resilience group. We then perform fault injections in each representative instruction sequence. The results of the fault injection campaign for each instruction sequence are applied to its entire resilience group. The results of the fault injections in each resilience group are recorded. As pointed out in Table 8.1, vulnerability between resilience groups in the same application can vary widely. Our methodology marks resilience groups with high DUE rates and high SDC rates.

2. In this step, we find the basic blocks that contribute to high DUE and high SDC rates.

   We profile the representative instruction sequence for each resilience group and find the most executed basic blocks in the instruction sequence. For resilience groups with high DUEs (crashes), we mark the most executed basic blocks in the instruction sequence to be protected by AddressArmor. For resilience groups with high SDCs, we mark the most executed basic blocks for protection with Value Armor.

In order to accomplish the second step, ArmorAll needs to be modified in order to allow protection for only a section of the code. Because ArmorAll is implemented at the LLVM level, this requires us to trace the basic blocks back to the application source code, then approximately link the source code to LLVM instructions.

Moreover, in the current implementation of ArmorAll, the automatic instruction duplication can only be applied globally to all the instructions in an application. In order to allow for local basic blocks, we first apply ArmorAll to the entire program and then prune the non-vulnerable basic blocks from the list of duplicated instructions. This is possible as ArmorAll works on static instructions and marks them for duplication.

Our current implementation is not yet automated and requires manual intervention to prune the basic blocks that do not need duplication. In the future, we plan on automating this process by adding a method to connect the original source code to the corresponding LLVM instruction(s). We provide the results of our evaluation in the following section.

Table 8.3: Vulnerability characteristics per resilience group in `bfs` after applying Selective ArmorAll to resilience groups 2 and 4 (see Table 8.1). A side-effect of applying ArmorAll is that it detects bit flips that would also result in Masked or SDC outcomes.

Res. Grp #	Res. Grp. Frequency	Masked %	Masked Detected	SDC %	SDC Detected	DUE %	Crash Detected
bfs							
1	10.53%	60.75%	0%	24.75%	0%	14.5%	0%
2	25.00%	40.25%	20.5%	10.45%	5.3%	0%	17.5%
3	23.68%	56.25%	0%	32%	0 %	11.75%	0%
4	40.79%	26.85%	22.4	5.4%	2.6%	0%	42.75%

8.3 Evaluation of Selective ArmorAll

In order to show the effectiveness of *Selective ArmorAll*, we use one of our applications: `bfs`. Our analysis in Table 8.1 showed that `bfs` has four resilience groups, one of which is highly susceptible to DUEs. Faults in this resilience group have a likelihood of 42% to lead to a DUE. This resilience group also accounts for 40.8% of all dynamic executions of this application. While faults in thie application are 26.156% likely to yield a crash (DUE), the crashes from Resilience Group 4 and from Resilience Group 2 account for 42.8% * 40.8% + 25%*17.5% = 21.84% of all crashes in the app. Applying the same resilience scheme to the whole application would result in unnecessary duplication of instructions.

Also, our analysis in Section 6.2 highlighted that `bfs` has two kernels that are invoked 8 times in sequence. Starting with their 5th invocation (around the 30% mark into the dynamic execution of the application), one of the kernels executes significantly more dynamic instructions for its 5th, 6th and 7th invocation (see the resilience patterns in Figures 6.1c, 6.1f, 6.1i).

To apply ArmorAll to `bfs`, we apply Address Armor (crash-causing fault detection) to the first two Resilience Groups in which faults are likely to result in crash. These resilience groups correspond to all but one basic block in `kernel1` of the application. We first apply Address Armor to `kernel1`, then remove the duplicated instructions that are not part of the resilience groups of interest. The results are shown in the following section.

8.3.1 Fault Coverage and Overhead

Next, we show the percentage of bit flips which would cause a crash that were detected with ArmorAll, as well as the increase in dynamic instructions that we achieved. We compare our results to the global use of Address Armor, without our selective usage.

Figure 8.1 shows the increase in dynamic instructions when the crash protection scheme of ArmorAll is applied both globally to the whole application, and to selected parts of the application. Our phase analysis guides this selection. The baseline is the application with no duplicated instructions. Using our methodology provides in a significant reduction in the overhead incurred with ArmorAll. With the selective application of AddressArmor, we were able to detect 80% of the faults that would cause crashes while incurring an overhead of only 30% in the number of dynamic instructions.

The use of Global Address Armor can be preferred in some circumstances as it has the ability to detect all possible crashes in an application. Users need to find the correct balance between

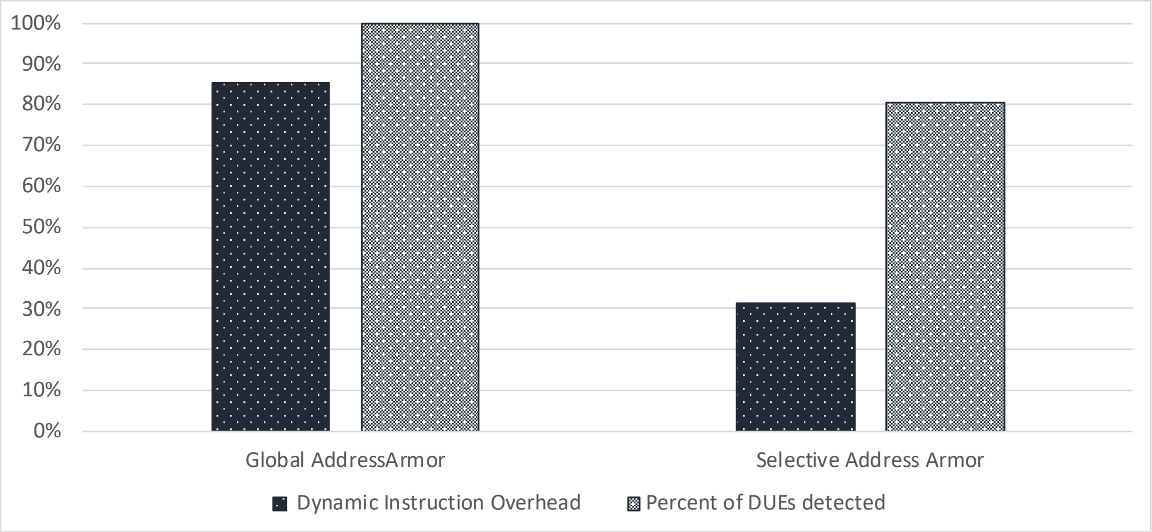


Figure 8.1: Overhead and fault coverage efficiency of global application vs our selective application of Address Armor to `bfs`. For the selective application, we chose two resilience groups with a high number of DUE cases. By applying AddressArmor to resilience groups 2 and 4, we were able to detect about 21% of the bit flips that would cause a crash in the appliction. This is equivalent to 80% of all the crash-causing bit flips.

performance and reliability. Furthermore, these current results correspond to selective Address Armor applied to two resilience groups. This implementation can be further refined to add protection to more resilience groups. While this will add more overhead, it can be beneficial for users looking for higher levels of reliability while keeping the performance overhead to a low level.

8.4 Summary on the Exploitation of Resilience Groups for Efficient Mitigation

Different resilience groups, by definition, have different resilience characteristics, while dynamic instruction sequences belonging to the same resilience group have similar resilience characteristics. Our results show that resilience characteristics of different resilience groups are significantly different and present an opportunity to exploit dynamically-adjustable resilience mitigation strategies.

This section introduces the first work on efficient local software mitigation strategies for GPU programs. Our technique utilizes a robust mitigation tool, ArmorAll [2] to apply our mitigation technique. Our results show that this technique provides a significant reduction in the overhead introduced by redundancy schemes. In the future, we plan on automating the process so that ArmorAll can automatically apply the appropriate resilience scheme to the correct code section.

Chapter 9

Summary and Conclusion

As GPUs continue to be used for General Purpose computing and deployed in High Performance Computers and supercomputers, their resilience to transient faults will continue to be a major concern for manufacturers and programmers alike. This thesis focuses on various aspects of the resilience of GPU programs, ranging from the efficient assessment of the vulnerability of GPU programs in presence of transient faults to the time varying behavior of the vulnerability of these faults as they occur during program execution. Understanding this behavior is critical in mitigating these faults for GPU programs. In this Chapter we will summarize the lessons learned in this thesis, and also provide directions for future work.

9.1 Major Contributions of this Thesis

9.1.1 Dependence of vulnerability on input size and configuration parameters

As discussed in this thesis in Section 5.1.1, the assessment of the effects of transient faults in GPU programs (fault injection campaigns) is a crucial step in addressing reliability. However, researchers generally report on a single set of data inputs and program parameters. Chapter 4 addressed this aspect.

This thesis shows that program vulnerability can be dependent on the size of the input used during the reliability assessment, as a larger data inputs can greatly increase the contributions of individual dynamic code sections in a non-uniform manner. This may result in increasing the execution weight of a specific code section that is more (or less) vulnerable.

Moreover, some corner cases of input data values can greatly influence the vulnerability of programs. We show specific cases of biased input values where all values were either equal to 0, or 1. We also show some cases of program-specific biased input values, which can be particularly chosen to influence the vulnerability of a program.

Our guidance for reliability researchers is to be aware of these pitfalls associated input values and input sizes that can dramatically change execution patterns in a program. Researchers should profile their target applications, taking note of the dynamic execution weights of individual basic blocks for different input sizes and values. A drastic change in the execution weight of individual basic blocks will generally indicate that the vulnerability of this program is susceptible to input changes.

9.1.2 Automatic reduction of fault injection campaigns using Program Counters

Assessment of vulnerability is also a prohibitively expensive step, as the number of faulty runs in a fault injection campaign needs to be sufficiently high to produce statistically significant results.

Chapter 5 presented a methodology, PCFI, that automatically reduces the number of faulty runs necessary to achieve assessment fidelity during a fault injection campaign. Our analysis was built on the predictability of faults in different instances of some static instructions.

The results obtained with PCFI do not introduce any compromise on the accuracy of the vulnerability assessment. PCFI helps researchers to accelerate live-system fault injection studies and quickly establish the resilience of different mitigation strategies.

9.1.3 Characterization of time varying behavior of vulnerability

Runtime vulnerability behavior has not been well explored for GPU applications. Understanding this behavior can help guide the use of selective application of mitigation strategies and selective fault injection campaigns.

In Chapter 6, we evaluated how vulnerability changes over time. We divide applications into instruction sequences and evaluated the vulnerability of individual instruction sequences. Our analyses show that vulnerability across different instruction sequences can widely vary. We leverage Basic Block Vectors to capture this vulnerability behavior and we introduced the term *resilience groups*, a group of instruction sequences with similar vulnerability behavior. Basic Block Vectors can be obtained with a profile of an application, without a fault injection campaign. The characterization of the time-varying behavior of vulnerability for GPU applications can be captured with one single execution of the application. This characterization can help reduce the performance overhead of mitigation by only applying the mitigation to resilience groups that are vulnerable.

9.1.4 Exploitation of time varying behavior for efficient vulnerability assessment

In Chapter 7, we showed how resilience groups can be leveraged to accelerate fault injections by only injecting faults in representative intervals from each resilience group. We build on the observation that faults at different intervals of the same resilience group result in the same outcome. We developed a methodology to allow us to select the best intervals for each resilience group for fault injection.

Our studies reveal that, for the set of applications studied in this thesis, that resilience groups account for a high percentage of the execution intervals of the applications. For example, some resilience groups represent up to 75% of all intervals in their applications. This means that we can study the resilience of one interval of execution to predict the resilience of up to 75% of all intervals of these applications. We use this methodology, and show we can greatly reduce the number of faults necessary versus a traditional fault injection approach. Our results show a fault reduction of an order of magnitude. Hence, without performing an exhaustive fault injection (i.e., faults in all execution intervals), we can predict the vulnerability outcomes for all execution intervals based on which resilience group it belongs to.

9.1.5 Exploitation of the time varying behavior for efficient fault mitigation

After performing fault injections in each representative instruction sequences per resilience group, and extending the vulnerability behaviors to all instruction sequences in the resilience groups, we identified the most vulnerable resilience groups. This knowledge is then transferred to an effective mitigation strategy, ArmorAll. We instrument ArmorAll to locally apply its resilience scheme. Our phase-based behavior applied to ArmorAll allows us to reduce the overhead of mitigation even further.

9.2 Future Work

- **Optimization of PCFI:** PCFI works by identifying static instructions in which faults lead to the same outcome and capping injections at these static instructions. Our current methodology does not make any attempt to predict static instructions which will have the same vulnerability profile. It is possible to optimize this methodology and even further reduce the number of injections in a campaign. We identify at least two ways to achieve this improvement.

1. Selected instructions that reside on the same execution control flow path are likely to lead to the same reliaiblity outcome. Identifying the set of these instructions can help when trying to reduce the number of injections needed, avoiding duplicate injections.
2. It is possible to predict the outcome of some faults in some instructions. For example, for instructions that access memory, it is possible to predict when a fault will cause the instructions to access illegal memory. Such faults can be pruned from an injection campaign.

- **Automatic placement of mitigation strategy:** In our current implementation for the efficient placement of ArmorAll resilient schemes, we manually pruned duplicated instructions that were not part of vulnerable instruction sequences. As such, it is challenging to apply these resilience schemes to all the applications in our test suite. It is possible to automate this process and directly receive feedback from the phase characterization analysis obtained from SASSIFI. It is possible to add a method to connect the original source code to its corresponding LLVM instruction(s), and this will help automate pruning of the unnecessary duplicated instructions.
- **Software Recovery Mechanisms:** The majority of DUEs that occur in GPUs are caused by an illegal memory accesses. In terms of mitigation strategies, it is worth exploring the development of a software-based recovery mechanism, developing a scheme to implement an error handler. This would intercept messages from the GPU driver before crashing the executing GPU program. An more robust solution would be to try to correct the faulty bit flip which caused an address to be incorrect in the first place. In general, error recovery is a challenging topic, and could be a ripe area to student as fault rates continue to grow in future technology nodes.

Bibliography

[1] F. G. Previlon, C. Kalra, D. R. Kaeli, and P. Rech, "A comprehensive evaluation of the effects of input data on the resilience of gpu applications," in *The 32nd IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, October 2019.

[2] C. Kalra, F. Previlon, N. Rubin, and D. Kaeli, "Armorall: Compiler-based resilience targeting gpu applications," in *Under Review*, 2019.

[3] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017.

[4] Nvidia, "Graphics processing unit (gpu)." [Online]. Available: http://www.nvidia.com/object/gpu.html

[5] S. E., D. J., S. H., and M. M., "Top500 lists," June 2018. [Online]. Available: https://www.top500.org/lists/2018/06/

[6] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 751–766. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173191

[7] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen,

"Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 129–173, May 2014. [Online]. Available: http://dx.doi.org/10.1177/1094342014522573

[8] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 610–621.

[9] R. Lucas, J. Ang, K. Bergman, and S. e. a. Borkar, "Top ten exascale research challenges," Feb 2014. [Online]. Available: http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf

[10] P. Rech, C. Aguiar, R. Ferreira, C. Frost, and L. Carro, "Neutron radiation test of graphic processing units," in *2012 IEEE 18th International On-Line Testing Symposium (IOLTS)*, June 2012, pp. 55–60.

[11] NVIDIA, "Gpu applications catalog." [Online]. Available: https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/

[12] P. N. Glaskowsky, "Nvidia's fermi : The first complete gpu computing architecture," 2009.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[14] T. K. G. T. O. Standard, *www.khronos.org/opencl*.

[15] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 908–916, Jul. 2003. [Online]. Available: http://doi.acm.org/10.1145/882262.882363

[16] C. Barnes, E. Cule, J. Liepe, K. Erguler, M. P. Stumpf, P. Kirk, and T. Toni, "ABC-SysBioapproximate Bayesian computation in Python with GPU support," *Bioinformatics*, vol. 26, no. 14, pp. 1797–1799, 06 2010. [Online]. Available: https://dx.doi.org/10.1093/bioinformatics/btq278

[17] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, "Impact of gpus parallelism management on safety-critical and hpc applications reliability," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 455–466.

[18] J. W. Sheaffer, D. P. Luebke, and K. Skadron, “The visual vulnerability spectrum: Characterizing architectural vulnerability for graphics hardware,” in *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, ser. GH ’06. New York, NY, USA: ACM, 2006, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/1283900.1283902

[19] F. F. d. Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, and P. Rech, “Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures,” in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, June 2017, pp. 169–176.

[20] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O’Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus, “Ibm experiments in soft fails in computer electronics (19781994),” *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 3–18, Jan 1996.

[21] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. S. Reorda, “Gpgpus: How to combine high computational power with high reliability,” in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–9.

[22] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, L. Carro, and A. Bland, “Understanding gpu errors on large-scale hpc systems and the implications for system design and operation,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 331–342.

[23] S. Tselonis and D. Gizopoulos, “Gufi: A framework for gpus reliability assessment,” in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 90–100.

[24] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, “Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications,” in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014.

[25] J. Wadden and K. Skadron, *Advances in GPU reliability research*, 12 2017, pp. 617–647.

[26] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 185–197.

[27] V. K. Sridharan, "Introducing abstraction to vulnerability analysis," p. 175, 2010, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2016-03-10.

[28] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *Int'l Symposium on High Performance Computer Architecture (HPCA-15)*, 2009, pp. 117–128.

[29] F. G. Previlon, B. Egbantan, D. Tiwari, P. Rech, and D. R. Kaeli, "Combining architectural fault-injection and neutron beam testing approaches toward better understanding of gpu soft-error resilience," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2017, pp. 898–901.

[30] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using heavy-ion radiation to validate fault-handling mechanisms," *IEEE Micro*, vol. 14, no. 1, pp. 8–23, Feb 1994.

[31] D. A. G. de Oliveira, L. L. Pilla, T. Santini, and P. Rech, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 791–804, March 2016.

[32] T. Calin, M. Nicolaidis, and R. Velazco, "Upset hardened memory design for submicron cmos technology," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2874–2878, Dec 1996.

[33] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, "A 1.3ghz fifth generation sparc64 microprocessor," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, June 2003, pp. 702–705.

[34] T. J. Slegel, R. M. Averill, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb, "Ibm's s/390 g5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, March 1999.

[35] E. Rotenberg, "Ar-smt: a microarchitectural approach to fault tolerance in microprocessors," in *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, June 1999, pp. 84–91.

[36] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proceedings 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 99–110.

[37] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, June 2000, pp. 25–36.

[38] F. G. Previlon, C. Kalra, D. Tiwari, and D. R. Kaeli, "Pcfi: Program counter guided fault injection for accelerating gpu reliability assessment," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 308–311.

[39] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Supercomputing, ACM/IEEE 2001 Conference*, Nov 2001, pp. 43–43.

[40] E. Alerstam, T. Svensson, and S. Andersson-Engels, "Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration," *Journal of biomedical optics*, vol. 13, p. 060504, 2008.

[41] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, no. 1, pp. 1–10, 2007. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-8-474

[42] J. E. Stone, J. C. Philips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, pp. 2618–2640, 2007.

[43] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos, "Accelerating financial applications on the gpu," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6. New York, NY, USA: ACM, 2013, pp. 127–136. [Online]. Available: http://doi.acm.org/10.1145/2458523.2458536

[44] S. E., D. J., S. H., and M. M., "Top500 lists," November 2010. [Online]. Available: https://www.top500.org/lists/2010/11/

[45] A. Donaldson, G. Gopalakrishnan, N. Chong, J. Ketema, G. Li, P. Li, A. Lokhmotov, and S. Qadeer, *Formal analysis techniques for reliable GPU programming: Current solutions and call to action*. United States: Elsevier Inc., 9 2016, pp. 3–21.

[46] P. E. Bailey, D. K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. R. d. Supinski, "Adaptive configuration selection for power-constrained heterogeneous systems," in *2014 43rd International Conference on Parallel Processing*, Sep. 2014, pp. 371–380.

[47] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 305–316, Sept 2005.

[48] N. DeBardeleben, S. Blanchard, L. Monroe, P. Romero, D. Grunau, C. Idler, and C. Wright, "GPU Behavior on a Large HPC Cluster," *6th Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids in conjunction with the 19th International European Conference on Parallel and Distributed Computing (Euro-Par 2013), Aachen, Germany,*, August 26-30 2013.

[49] H. J. Wunderlich, C. Braun, and S. Halder, "Efficacy and efficiency of algorithm-based fault-tolerance on gpus," in *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, July 2013, pp. 240–243.

[50] L. A. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, R. Rech, and M. S. Reorda, "GPGPUs: How to Combine High Computational Power with High Reliability," in *DATE*, Dresden, Germany, 2014.

[51] D. Oliveira, P. Rech, H. Quinn, T. Fairbanks, L. Monroe, S. Michalak, C. Anderson-Cook, P. Navaux, and L. Carro, "Modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison," *Nuclear Science, IEEE Transactions on*, vol. 61, no. 6, pp. 3115–3122, Dec 2014.

[52] R. C. Baumann, "Soft errors in commercial semiconductor technology: Overview and scaling trends," 04 2002.

[53] E. Normand, "Single event upset at ground level," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, Dec 1996.

[54] J. Somers. (2018, Dec. 7) The friendship that made google huge. [Online]. Available: https://www.newyorker.com/magazine/2018/12/10/the-friendship-that-made-google-huge

[55] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, “Ferrari: a tool for the validation of system dependability properties,” in *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, July 1992, pp. 336–344.

[56] 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 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 29–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956570

[57] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, “Statistical fault injection: Quantified error and confidence,” in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE ’09.*, April 2009, pp. 502–506.

[58] J. Arlat, Y. Crouzet, and J.-C. Laprie, “Fault injection for dependability validation of fault-tolerant computing systems,” in *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, June 1989, pp. 348–355.

[59] N. J. Wang, A. Mahesri, and S. J. Patel, “Examining ace analysis reliability estimates using fault-injection,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA ’07. New York, NY, USA: ACM, 2007, pp. 460–469. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250719

[60] S. Tselonis and D. Gizopoulos, “Gufi: A framework for gpus reliability assessment,” in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 90–100.

[61] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, “Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment,” *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 241–254, Jun. 2017.

[62] B. Nie, L. Yang, A. Jog, and E. Smirni, “Fault site pruning for practical reliability analysis of gpgpu applications,” in *International Symposium on Microarchitecture (MICRO) 2018*, 2018.

[63] G. Li, K. Pattabiraman, C. Cher, and P. Bose, “Understanding error propagation in gpgpu applications,” in *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 240–251.

[64] 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 *2017 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[65] 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 *Proceedings of the 32Nd International Conference on Computer Safety, Reliability, and Security - Volume 8153*, ser. SAFECOMP 2013. New York, NY, USA: Springer-Verlag New York, Inc.

[66] NVIDIA, "NVIDIA TESLA V100 GPU ARCHITECTURE," 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[67] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing architectural vulnerability factors for address-based structures," in *32nd International Symposium on Computer Architecture (ISCA'05)*, June 2005, pp. 532–543.

[68] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "Architecture-level soft error analysis: Examining the limits of common assumptions," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, June 2007, pp. 266–275.

[69] V. Sridharan and D. R. Kaeli, "Using hardware vulnerability factors to enhance avf analysis," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 461–472. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816023

[70] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 1245–1254.

[71] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, Oct 2010.

[72] X. Li, K. Shen, M. C. Huang, and L. Chu, "A memory soft error measurement on production systems," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ser. ATC'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 21:1–21:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1364385.1364406

[73] B. Schroeder, E. Pinheiro, and W.-D. Weber, “Dram errors in the wild: A large-scale field study,” in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’09. New York, NY, USA: ACM, 2009, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/1555349.1555372

[74] X. Li, M. C. Huang, K. Shen, and L. Chu, “A realistic evaluation of memory hardware errors and software system susceptibility,” in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 6–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855846

[75] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, “Cosmic rays don’t strike twice: Understanding the nature of dram errors and the implications for system design,” *SIGPLAN Not.*, vol. 47, no. 4, pp. 111–122, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2248487.2150989

[76] V. Sridharan and D. Liberty, “A study of dram failures in the field,” in *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2012, pp. 1–11.

[77] T. Siddiqua, A. E. Papathanasiou, A. Biswas, and S. Gurumurthi, “Analysis and modeling of memory errors from large-scale field data collection,” in *SELSE 2013*, 2013.

[78] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, “Memory errors in modern systems: The good, the bad, and the ugly,” *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 297–310, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694348

[79] I. S. Haque and V. S. Pande, “Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu,” in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, May 2010, pp. 691–696.

[80] C. Lunardi, F. Previlon, D. Kaeli, and P. Rech, “On the efficacy of ecc and the benefits of finfet transistor layout for gpu reliability,” *IEEE Transactions on Nuclear Science*, vol. 65, no. 8, pp. 1843–1850, Aug 2018.

[81] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based analysis of a GPU architecture," in *Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2012.

[82] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, " Multi2Sim: A Simulation Framework for CPU-GPU Computing ," in *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[83] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Dec 2007, pp. 407–420.

[84] C. Kalra, F. Previlon, X. Li, N. Rubin, and D. Kaeli, "Analyzing the vulnerability of vector-scalar execution on data-parallel architectures," in *The 14th Workshop on Silicon Errors in Logic-System Effects, SELSE*, 2018.

[85] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on gpgpu microarchitecture," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2011, pp. 226–235.

[86] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli, "Calculating architectural vulnerability factors for spatial multi-bit transient faults," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 293–305.

[87] F. Previlon, M. Wilkening, V. Sridharan, S. Gurumurthi, and D. Kaeli, "Examining the impact of ace interference on multi-bit avf estimates," *Proceedings of SELSE: Silicon Errors in Logic-System Effects*, 2015.

[88] M. Wilkening, F. Previlon, and D. R. Kaeli, "Evaluating the resilience of highly parallel applications," in *The 12th Workshop on Silicon Errors in Logic-System Effects, SELSE*, 2016.

[89] W. Mansour and R. Velazco, "An automated seu fault-injection method and tool for hdl-based designs," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2728–2733, Aug 2013.

[90] A. Mohammadi, M. Ebrahimi, A. Ejlali, and S. G. Miremadi, "Scfit: A fpga-based fault injection technique for seu fault model," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2012.

[91] M. Ebrahimi, N. Sayed, M. Rashvand, and M. B. Tahoori, "Fault injection acceleration by architectural importance sampling," in *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2015, pp. 212–219.

[92] E. Cioroaica, J. Jahi, T. Kuhn, C. Peper, D. Uecker, C. Dropmann, P. Munk, A. Rakshith, and E. Thaden, "Accelerated simulated fault injection testing," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct 2017, pp. 228–233.

[93] 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," *SIGPLAN Not.*, vol. 47, no. 4, pp. 123–134, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2248487.2150990

[94] C. Kalra, F. Previlon, X. Li, N. Rubin, and D. Kaeli, "Prism: Predicting resilience of gpu applications using statistical methods," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 69:1–69:14. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291748

[95] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 27–38.

[96] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, pp. 45–57, Oct. 2002.

[97] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2002.

[98] M. Annavaram, R. Rakvic, M. Polito, J. . Bouguet, R. Hankins, and B. Davies, "The fuzzy correlation between code and performance predictability," in *International Symposium on Microarchitecture'04*.

[99] W. Liu and M. C. Huang, "Expert: Expedited simulation exploiting program behavior repetition," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ser. ICS '04, NY, USA.

[100] R. Balasubramonian, D. Albones, A. Buyuktosunoglu, and S. Dwarkadas, “Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures,” in *IEEE/ACM International Symposium on Microarchitecture. MICRO 2000.*

[101] A. S. Dhodapkar and J. E. Smith, “Comparing program phase detection techniques,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003.

[102] C. Isci and M. Martonosi, “Identifying program power phase behavior using power vectors,” in *2003 IEEE International Conference on Communications (Cat. No.03CH37441)*, Oct 2003, pp. 108–118.

[103] ——, “Phase characterization for power: evaluating control-flow-based and event-counter-based techniques,” in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, Feb 2006.

[104] T. E. Carlson, W. Heirman, K. V. Craeynest, and L. Eeckhout, “Barrierpoint: Sampled simulation of multi-threaded applications,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014.

[105] J. C. Huang, L. Nai, H. Kim, and H. H. S. Lee, “Tbpoint: Reducing simulation time for large-scale gpgpu kernels,” in *IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014.

[106] M. Kambadur, S. Hong, J. Cabral, H. Patil, C. K. Luk, S. Sajid, and M. A. Kim, “Fast computational gpu design with gt-pin,” in *2015 IEEE International Symposium on Workload Characterization*, Oct 2015, pp. 76–86.

[107] X. Fu, J. Poe, T. Li, and J. A. B. Fortes, “Characterizing microarchitecture soft error vulnerability phase behavior,” in *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, Sept 2006, pp. 147–155.

[108] D. Oliveira, L. Pilla, N. DeBardeleben, S. Blanchard, H. Quinn, I. Koren, P. Navaux, and P. Rech, “Experimental and analytical study of xeon phi reliability,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’17. New York, NY, USA: ACM, 2017, pp. 28:1–28:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126960

[109] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, and G. H. Loh, "Architectural vulnerability modeling and analysis of integrated graphics processors," in *Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2013.

[110] V. Sridharan and D. Kaeli, "The effect of input data on program vulnerability," 2009.

[111] T. M. Jones, M. F. P. O. 'boyle, and O. G. Ergin, "Evaluating the effects of compiler optimisations on avf," in *Workshop on Interaction Between Compilers and Computer Architecture (INTERACT-12)*, 2008.

[112] F. G. Previlon, C. Kalra, D. R. Kaeli, and P. Rech, "Evaluating the impact of execution parameters on program vulnerability in gpu applications," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 809–814.

[113] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, "Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability," in *IEEE International Conference on Dependable Systems and Networks (DSN 2014)*, Atlanta, USA, 2014.

[114] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[115] 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 *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[116] NVIDIA, "NVIDIA, CUDA SDK, V7.0."

[117] ——, "CUDA C BEST PRACTICES GUIDE," 2018. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[118] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[119] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: methodology and empirical data," in *IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*

[120] X. Shen, Y. Zhong, and C. Ding, “Locality phase prediction,” *SIGPLAN Not.*, vol. 39, no. 11, pp. 165–176, Oct. 2004.

[121] P. J. Rousseeuw”, “”silhouettes: A graphical aid to the interpretation and validation of cluster analysis”,” *”Journal of Computational and Applied Mathematics”*, vol. ”20”, no. ”Supplement C”, ”1987”.

[122] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on gpus,” in *IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2012.

[123] F. G. Previlon, C. Kalra, D. Tiwari, and D. R. Kaeli, “Characterizing and exploiting soft error vulnerability phase behavior in gpu applications,” in *Under Review*, 2019.

[124] NVIDIA, “NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110,” 2015. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[125] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, “Hauberk: Lightweight silent data corruption error detector for gpgpu,” in *IEEE International Parallel Distributed Processing Symposium*, May 2011.

[126] M. Dimitrov, M. Mantor, and H. Zhou, “Understanding software approaches for gpgpu reliability,” in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 94–104. [Online]. Available: http://doi.acm.org/10.1145/1513895.1513907

[127] C. Wang, H. Kim, Y. Wu, and V. Ying, “Compiler-managed software-based redundant multi-threading for transient fault detection,” in *International Symposium on Code Generation and Optimization (CGO’07)*, March 2007, pp. 244–258.

[128] S. Mukherjee, J. Emer, and S. Reinhardt, “The soft error problem: an architectural perspective,” in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, Feb 2005, pp. 243–247.

[129] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, “Real-world design and evaluation of compiler-managed gpu redundant multithreading,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 73–84.

[130] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 67:1–67:12. [Online]. Available: https://doi.org/10.1109/SC.2018.00070

[131] N. J. Wang and S. J. Patel, "Restore: symptom based soft error detection in microprocessors," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, June 2005, pp. 30–39.