TOWARDS SCALABLE AND SPECIALIZED APPLICATION ERROR ANALYSIS

BY

ABDULRAHMAN HASSAN N MAHMOUD

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Abstract

Modern systems at scale are increasingly susceptible to transient hardware errors at current technology sizes from natural phenomena such as high-energy particle strikes (also called soft errors). Traditional solutions aimed at dealing with soft errors, however, typically rely on indiscriminate redundancy in space and/or time for resilience. Such techniques can incur high system overheads, whether in manufacturing cost, runtime performance, energy consumption, and/or area requirements. Moreover, the all-or-nothing protection offered by full redundancy may result in over-protection and inefficient use of resources. To that end, while it is critical to be able to protect against the effect of hardware errors, it is important to do so in an an efficient and low-cost manner.

One way to reduce the cost of protecting applications from hardware errors is to understand *how* errors propagate at finer granularities, and only protect vulnerable components via selective duplication. This raises three important questions:

1. What granularity of analysis is reasonable to target?
2. Which components at this granularity should be selected for protection?
3. How should the selective protection be implemented in a low-cost manner?

This thesis addresses these three questions with the design of multiple tools and techniques geared towards identifying and understanding how single-bit flip errors propagate and affect an application's output. First, we present a general-purpose tool called Approxilyzer. Approxilyzer uses the novel error pruning and equalization techniques pioneered by a prior tool, Relyzer, to quantify the impact of virtually every error site in an application. Targeting the instruction-level granularity for analysis and protection, Approxilyzer shows that not all errors are equally important, and that trading off a small output quality degradation (for example, 1%) can yield large resiliency overhead reduction (up to 55%) for 99% resiliency coverage.

While Approxilyzer is a promising tool for resiliency analysis, it initially took a long time to run due to the large number of error sites requiring exploration in an application. To accelerate error analysis tools (such as Approxilyzer), the second part of this thesis introduces a software-testing inspired toolkit called Minotaur. Minotaur bridges the gap between software testing and hardware resiliency by adapting multiple techniques from the software engineering domain to make hardware error analysis faster and thus more scalable.

We show that Minotaur can significantly improve the runtime of Approxilyzer (10.3× on average), while *simultaneously* improving its accuracy in identifying vulnerable instructions which need protection.

The third part of this thesis focuses on reducing the implementation overhead of instruction-level duplication, by taking into consideration the hardware platform and unique opportunities provided by the backend architecture. Specifically, we develop a tool called SInRG, or **S**oftware-managed **In**struction **R**eplication for **G**PUs. SInRG provides a family of instruction duplication techniques that exploit underutilized hardware resources for error detection. Inspired by CPU instruction-level duplication, SInRG establishes the first practical approach to software-directed instruction duplication for GPU-based systems, identifies GPU-specific opportunities for overhead reduction, and explores software and hardware performance optimizations to lower the overheads of replication significantly. The GPU-specific software optimizations trade off error containment for performance and reduce the average runtime overhead to 36%. We also propose new ISA extensions with limited hardware changes and area costs to further lower the average runtime overhead to just 30%.

General purpose error analysis and hardening techniques provide the benefit of being universally applicable to general purpose code. Given additional information about the application, however, can further enable low-cost resiliency solutions by leveraging domain knowledge. The fourth part of this thesis uses this premise to perform a *specialized* resiliency analysis for convolutional neural networks (CNNs), due to their prevalence in many safety-critical application such as self-driving cars. We develop and evaluate two selective protection techniques at different target granularities in CNNs (feature map level and inference level), and show that the combination of both techniques is better than the sum of its parts. Our results show that the specialized, domain-specific error analysis and hardening techniques can achieve very high error coverage of 99.78% on average for the CNNs explored, while incurring as low as 20% overhead, or 5× less overhead compared to full duplication.

Overall, this thesis focuses on understanding *how* hardware errors propagate to corrupt an application's output. We develop multiple tools and techniques for error analysis, and advocate for specialized, selective protection solutions as a means to achieve low overheads while maintaining high error coverage in applications.

To Mama, Baba, Hanan, Musa, and my entire family for their love and support.

Acknowledgments

This Ph.D. journey would not have been possible without the help of so many people along the way. Most of all, I need to thank my advisor, Dr. Sarita Adve, for all her kindness, patience, dedication, and support throughout this process. I have learned so much during this time having worked with her, and I have become a much better researcher and person as a result. Thank you, Sarita, for everything you have taught me and coached me over the years.

Many parts of this thesis were done in collaboration with Dr. Siva Hari. Besides being an excellent researcher, collaborator and mentor, Siva is one of the kindest and most genuine people I have ever met. I am lucky to have worked with him on multiple projects, and even more fortunate to have gotten to know him and benefit from his great mentoring.

I also sincerely appreciate all the feedback and interactions from the rest of my Ph.D. committee. Dr. Darko Marinov, thank you for providing me a quote for the ages: "Be with the force. Read the source." Dr. Chris Fletcher, your enthusiasm and passion are contagious, and I always look forward to our many chats. Dr. Sasa Misailovic, my results are finally here, put together in thesis form. Dr. Luis Ceze, thank you for always having time for me, and constantly providing positive feedback and encouragement.

The entire Computer Science department staff at Illinois is incredible. Thank you for all the assistance throughout, and for making the time I spent here during my Ph.D. very enjoyable and conducive to success. Special thanks to Kim Baker, Jennifer Dickens, Dana Garard, Samantha Hendon, Joe Jeffries, Mary Beth Kelly, Viveka Kudaligama, Kara MacGregor, Rhonda McElroy, Maggie Metzger Chappell, Michelle Osbourne, Colin Robertson, Kathy Ann Runck, and Glen Rundblom. Any success I have had is a tribute to their incredible efforts along the way.

My work has been supported by the National Science Foundation under grants CCF 13-20941 and CCF 19-56374, by the Center for Future Architectures Research (C-FAR), one of the six centers of STARnet, a Semiconductor Research Corporation (SRC) program sponsored by Microelectronics Advanced Research Corporation (MARCO) and Defense Advanced Research Projects Agency (DARPA), and by the Applications Driving Architectures (ADA) Research Center, a Joint University Microelectronics Program (JUMP) Center co-sponsored by SRC and DARPA. A portion of this work was performed while I interned at NVIDIA.

I am very thankful to all my collaborators and/or labmates: Neeraj Aggarwal, Khalique Ahmed, Dr. John Alsop, Wesley Darvin, Dr. Iuri Frosio, Aditi Ghosalkar, Sam Grayson, Dr.

Steve Keckler, Alex Nobbe, Dr. Pavlo Molchanov, Paul Reckamp, Charbel Sakr, Giordano Salvador, Dr. Naresh Shanbhag, Dr. Matthew Sinclair, Rahul Singh, Dr. Michael Sullivan, Phoebe Tang, Dr. Timothy Tsai, and Dr. Radha Venkatagiri.

I would like to particularly thank Radha for all her work and collaboration together throughout the years. Her encouraging spirit and approachable nature have made bouncing ideas off of one another and working together an enjoyable and enlightening experience!

Special thanks to Muhammad Huzaifa. An amazing friend, buddy, teammate, collaborator, soccer player, motorcycle enthusiast, and generally a great person to be around. You have made this ride very enjoyable my friend.

Finally, a very special thank you to my wife, Hanan, for all her love and support throughout. You have always been my number one fan, my pillar of support, and my better half. To my son, Musa, you helped make this journey more exciting than it ever could have been, and your smile and laugh are my favorite things in the whole wide world. To my loving family and community: thank you for always being there for me, and carrying me to the finish line.

Table of Contents

Chapter 1: Introduction

1.1 MOTIVATION

As we approach the end of conventional transistor scaling, hardware is becoming increasingly susceptible to errors in the field [1, 2, 3, 4, 5, 6]. Commodity hardware is used in systems with a range of reliability requirements, from entertainment devices such as personal phones to stringently safety-critical systems such as self-driving cars. Modern systems at scale are especially susceptible to transient hardware errors (also called *soft errors*) at current technology sizes from natural phenomena such as high-energy particle strikes, wear out, and/or voltage droops [3, 7, 8]. With exascale systems scaling to hundreds of thousands of nodes, automated driving systems on the roads, and wearable medical devices increasing in use, it is critical to protect against the increased likelihood of hardware events causing system errors and/or crashes.

Studies have shown that hardware errors can have severe unintended consequences unless the system is designed to detect these errors [9, 10]. Many high-performance computing (HPC) field studies [11, 12, 13] and exascale reports and challenges [3, 7, 8] assert the importance of designing error-tolerant systems. Traditional reliability solutions, however, typically rely on full and indiscriminate redundancy in hardware or software to ensure high resilience. This can be both costly and inefficient at the system level. Although it is crucial to detect hardware errors in such systems to ensure high reliability guarantees, hardware errors themselves occur infrequently such that full, indiscriminate redundancy may be overkill for many systems.

Early work recognized that a large majority of hardware errors are either masked at the software level (i.e., they did not change the output of the executing program) or result in easily detectable anomalous software behavior (e.g., exceptions due to unaligned or out-of-bounds addresses) [14, 15, 16, 17, 18, 19, 20]. The former errors require no action and the latter can be detected using zero to very low-cost detection mechanisms. While such software-centric resiliency techniques show immense promise, unfortunately, some hardware errors escape detection and result in undetected and potentially unacceptable silent data corruptions (*SDCs*) of the program output.

SDCs have been an obstacle in the widespread adoption of software-centric resiliency techniques; therefore, significant recent research has focused on characterizing and reducing these SDCs either through hardware solutions (e.g., use of error-correcting code, or ECC, in hardware memory structures) or software solutions (e.g., insertion of software checks and assertions in application code regions determined to be too vulnerable to SDCs) [19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32].

Underlying all of these solutions is the need for techniques that find SDCs in the applications of interest. Software resiliency (or just *resiliency*) is the ability of a given piece of software to avoid an SDC for a given hardware error, and can be divided into two primary components. The first component is *resiliency analysis*, which involves characterizing the resiliency of a given piece of software for a given set of hardware errors. The second component is *resiliency hardening*, whereby either software or hardware modifications are made to make the software more resilient. Each component offers unique research challenges in the identification and mitigation of SDCs.

In order to avoid the heavy hammer of indiscriminate redundancy for software resilience, a developer can perform an offline analysis of the workload to help pinpoint vulnerabilities before deployment in the field. The primary drawback of such a resiliency analysis, however, is that the number of potential error sites requiring exploration can be extremely large. For example, assuming a single-bit flip error model in an application would require studying the impact of every bit flip of every register of every dynamic instruction in the application – clearly an intractable proposition.

Prior work in resiliency analysis imposes a significant trade-off between speed and accuracy. Statistical analyses based on dynamic error-free execution traces or static code [27, 33, 34, 35, 36] are unable to precisely model error propagation paths. Randomized error injection campaigns [37, 38, 39, 40, 41] can only provide statistical information and are unable to predict resilience for code portions where errors were not injected. More systematic and comprehensive error-injection techniques [42, 43] can precisely identify SDC-causing instructions but are much slower than the previous techniques.

Resiliency analysis can have a direct (positive) influence on hardening overheads, by reducing the amount of work required by hardening techniques to only the vulnerable error sites from the analysis. Resiliency hardening however, does not need to solely depend on an offline analysis: various software-hardware co-designed protection techniques can be employed which operate in an "always-on" mode. Such a hardening solution instead depends on understanding the hardware platform running the code, and exploiting spare resources to "hide" protection overheads effectively. While such a technique can be beneficial for some workloads, combined with an offline analysis can make it even more powerful as a low-cost deployment resilience solution.

While the aforementioned analysis and hardening techniques can be universally applicable to general-purpose code, a general purpose approach which does not take into consideration the executing workload may perform sub-optimally in terms of minimizing the cost of resilience while ensuring high-error coverage. Instead, by tuning resiliency to the application, we can further reduce resiliency overheads. Major challenges involved in such an endeavor include understanding how errors propagate in the application, and crafting solutions that guard against the most egregious output corruptions.

This thesis aims to address many of the challenges described above by introducing tools and techniques that help understand how hardware errors propagate, and how to mitigate such errors using low-cost protection mechanisms. This thesis argues for scalable and specialized application error analysis tools to enable low-cost hardening techniques. Further, given the recent meteoric rise of machine learning and deep learning in many safety-critical applications, this thesis performs a deep-dive into convolutional neural network (CNN) resilience. Overall, we show that software-directed resiliency techniques not only have high SDC detection capabilities, but can also do so with lower overheads than traditional mechanisms.

1.2 SUMMARY OF CONTRIBUTIONS

To address the challenges involved in identifying and protecting against hardware errors, it is important to categorize and understand how errors propagate at various granularities in a system. Moreover, it is critical to design tools and techniques which capitalize on the gathered insights and understandings, while simultaneously ensuring the usability of such tools in terms of their performance and accuracy. The following sections summarize the contributions made in this thesis.

All the following work has been a collaborative effort, with shared contribution among fellow students, faculty, and industry researchers.[1] Radha Venkatagiri is the lead author on the Approxilyzer tool described in Chapter 2. Khalique Ahmed led the effort in the gem5-Approxilyzer work described in Section 2.6. I co-led the work described in Chapter 3 with Radha. I am the lead author of the work described in Chapters 4, 5, and 6.

1.2.1 Principled and General Purpose Error Analysis Techniques

Despite the fact that errors carry a negative connotation, not all errors are created equal. While some errors are egregious and must be identified and mitigated, other errors are tolerable and may have little or no impact on the system or application. In order to identify which errors are "bad" and must guarded against while dismissing errors which are "good"

[1]Chapters 2 – 6 are heavily based on the publications I led and co-led.

and can be tolerated, it is necessary to understand how errors propagate once a fault manifests in the hardware. This requires *principled* and comprehensive tools to perform such an analysis.

One challenge these tools must overcome is the extremely large space of possible errors, given an error model. For example, if we assume an alpha particle strike can flip a random bit of a register during program execution, the plurality of errors would include studying the impact of every bit flip of every register of every dynamic instruction for a running application. This is prohibitively expensive.

In addition to reducing the error space, we also need to understand the impact an error has on the *quality* of an application's output, an important contribution of this thesis. This type of analysis can significantly help in reducing the typically high overhead associated with reliability. Further, it can help in identifying which errors are especially important to protect and which can be dismissed due to their low impact on the application's output quality.

To address these issues, the first contribution of this thesis is the development of Approxilyzer [44] and gem5-Approxilyzer [45]. Using the novel error-site pruning techniques pioneered by Relyzer [42], Approxilyzer quantifies the quality impact of a single-bit error in all dynamic instructions of an execution with high accuracy. We demonstrate two uses of Approxilyzer. First, we show how Approxilyzer can be used to quantitatively tune output quality versus resiliency versus overhead to enable ultra-low cost resiliency solutions (for a single bit flip error model). For example, we show that Approxilyzer determines that a very small loss in output quality (1%) can yield large resiliency overhead reduction (up to 55%) for 99% resiliency coverage. Second, we show how Approxilyzer can be used to provide a first-order estimate of the approximation potential of general-purpose programs. It does so in an automated fashion while requiring minimal user input and no program modifications. We extend Approxilyzer to gem5-Approxilyzer, an open-source implementation that enables support for various instruction set architectures (ISAs); we study the error profile of applications in the x86 and Scalable Processor Architecture (SPARC) ISAs for comparison. Effectively, Approxilyzer is a comprehensive, general-purpose tool which provides instruction-level error analysis with quality assessment.

1.2.2 Scaling Up Error Analyses Through Software Testing Techniques

Even with principled analysis, the space of possible errors may still be very large, and a comprehensive analysis can still be slow. To promote the adoption of highly effective error analysis tools such as Approxilyzer, an underlying requirement is that the tool must be fast, without sacrificing accuracy of error analysis. To address this challenge, the second contribution of this thesis is to leverage the extensive line of work from the domain of software testing, and adapt it to hardware resiliency. The novel insight here is that analyzing software for resiliency to hardware errors is similar to testing software for software bugs; therefore, adapting techniques from the rich software testing literature can substantially improve the state-of-the-art in resiliency analysis.

The result is called Minotaur [46], a software testing inspired toolkit which can accelerate error analysis tools such as Approxilyzer. Minotaur bridges between software testing and hardware reliability by adapting four software testing techniques to make hardware error analysis faster and thus more scalable. As a result, we show that Minotaur improves the runtime of Approxilyzer by an order of magnitude (10.3×) on average while *simultaneously* improving the accuracy of Approxilyzer in identifying hardware errors.

1.2.3 Optimizing General Purpose Protection Through Software-Managed Techniques

Error analysis forms only part of the equation when it comes to making application resilient in the face of hardware errors. Another important aspect involves hardening the application to avoid errors at runtime. For example, one mechanism for hardening an application is duplicating all vulnerable instructions in order to detect errors and notify the system at runtime. Although advantageous from a resiliency perspective, this can incur significant overheads which may not be acceptable, especially for mission critical applications with real-time constraints.

To study the implementation overhead of instruction-level duplication, the third contribution of this thesis is the development of a tool called SInRG [47], or **S**oftware-managed **In**struction **R**eplication for **G**PUs. SInRG provides a family of instruction duplication techniques that exploit underutilized hardware resources for duplication. This work describes a practical methodology to employ instruction duplication for graphics processing units (GPUs) and identifies implementation challenges that can incur high overheads (69% on average). It explores GPU-specific software optimizations that trade fine-grained recoverability for performance. It also proposes simple ISA extensions with limited hardware changes and area costs to further improve performance, cutting the runtime overheads by more than half to an average of 30%.

We find that the optimal SInRG duplication strategy is workload dependent: based on the hardware resource utilization of the workload, a particular SInRG techniques performed better or worse. For example, a workload with heavy register utilization would benefit from a duplication strategy that avoids increasing register pressure. Thus, by leveraging application-specific information, developers can potentially design lower-cost resiliency solutions.

1.2.4 Specialized, Application-Driven Reliability for CNNs

The fourth contribution of this thesis is an end-to-end, specialized reliability toolset for CNN workloads. As CNNs become more prevalent in safety-critical applications such as self-driving vehicles, it is imperative that they behave reliably in the face of hardware errors. Thus, rather than relying on general purpose tools for analysis and protection, we ask the question: can we leverage domain-specific knowledge for resiliency analysis and hardening?

We develop and open-source a tool called PyTorchFI [48], which allows us to study the impact of transient errors on the outcome of a CNN inference. Using PyTorchFI, we introduce two software-driven, selective protection techniques that target different protection granularities of a CNN. First, we develop a feature map level resilience technique (FLR) [49], which identifies and statically protects the most vulnerable feature maps in a CNN. Second, we develop an inference level resilience technique (ILR), which selectively reruns vulnerable inferences by analyzing their output. Finally, we show that the combination of both techniques (called FILR) is highly efficient. Our results show that the combination can achieve very high error coverage of 99.78%, while incurring only 48% overhead on average (and as low as 20% for ResNet50, or 5× less overhead compared to full duplication).

1.3 THESIS ORGANIZATION

This thesis is organized as follows. Chapter 2 introduces Approxilyzer and gem5-Approxilyzer, two general-purpose hardware error analysis tools that quantitatively measure the impact of errors at the instruction-level. While Approxilyzer and gem5-Approxilyzer can attain orders-of-magnitude gains via error pruning and equalization of errors (leveraging Relyzer techniques), these tools can still take a long time to complete the analysis. To address this issue, Chapter 3 proposes and evaluates the toolset Minotaur, showing how employing and adapting software testing techniques to the field of hardware error analysis can dramatically speed up application error analysis without loss of accuracy. Next, Chapter 4 presents SInRG, a family of low-cost, instruction-level duplication techniques which leverage spare hardware resources to effectively hide error detection overheads in general-purpose code.

Our results in Chapters 2–4 show that incorporating information about an application can help optimize resiliency analysis and hardening techniques. Chapter 5 introduces PyTorchFI, a specialized runtime perturbation tool for deep learning. Using PyTorchFI, Chapter 6 introduces and promotes a fully application-based error analysis and detection approach for CNNs. Chapter 6 explores different granularities of analysis and protection for software-directed error analysis of CNNs, and shows that a combination of techniques at different granularities is better than the sum of its parts. Chapter 7 highlights related work, and Chapter 8 summarizes the contributions of this thesis and discusses potential future research directions.

Chapter 2: Principled and General Purpose Hardware Error Analysis

2.1 MOTIVATION

Traditional solutions for dealing with hardware errors in the field have relied on indiscriminate redundancy in space and/or time [50, 51, 52, 53]. For example, Tesla released a Full Self Driving (FSD) system in March 2019 which deploys two fully redundant FSD chips along with accompanying redundant control logic, power, and peripheral packaging on the board for reliability [53]. This solution of indiscriminate redundancy, however, is in direct tension with a key implication of the end of the Moore's Law and Dennards scaling era: hardware designers are now severely constrained by power and area, and must obtain improved efficiency to stay on the performance and functionality curves to enable future applications [54]. Indiscriminate redundancy incurs too much overhead in cost, area, power, and/or performance for most systems [1, 27, 30, 36, 55, 56, 57, 58, 59, 60]. This overhead is particularly onerous since in many cases, it unnecessarily overprotects against hardware errors.

A promising alternative to full application redundancy is the use of software for the detection for hardware errors. Software-centric solutions aim to identify anomalous software behavior due to hardware errors with the use of low-cost techniques [14, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]. However, many hardware errors still manage to escape detection, resulting in unacceptable silent data corruptions, or SDCs, of the program output [30, 61, 62, 63]. Therefore, significant research has focused on analyzing these SDCs, and reducing them through either hardware or software solutions.

Resiliency analysis techniques used in practice to help uncover SDCs can generally be categorized into two major categories: *experimental error injection campaigns* or *analytical error propagation models*. An *experimental error injection campaign* involves performing many error injection experiments, each of which emulates a hardware error by perturbing internal program state, and then executing the program to completion to evaluate the effect of the error [64, 65, 66]. Since a program can consist of trillions of operations and there are a plurality of errors possible for each operation, a primary challenge of this approach is the large amount of time and resources needed to completely characterize the resilience of an application. On the other hand, *analytical error propagation* models attempt to reduce the resource intensity of error injection by estimating the vulnerability of different operations through higher-level models that take into account architecture or domain knowledge [67, 68, 69]. Section 7 goes into more detail on specific resiliency analysis techniques from the literature.

In order to obtain the accuracy obtained from experimental error injection while avoiding the associated overhead involved, we need to systematically address the large error space and reduce it in a *principled* manner. Furthermore, we need to understand the *impact* an error has on an application's output quality, to determine whether or not it requires protection. To that end, we developed an open-source tool called Approxilyzer [44, 70].

Approxilyzer builds on a resiliency-driven tool called Relyzer [42] which uses a combination of error injection and program analysis to predict the outcomes of errors, assuming all deviations from the error-free output are unacceptable. Relyzer uses an instruction-level transient single bit error model, and determines outcomes for all such errors in the operand registers of all dynamic instructions. We use *error site* to refer to a specific bit in a specific operand register in a specific dynamic instruction. Using program analysis and some heuristics, Relyzer identifies error sites that behave similarly in the presence of a single-bit error and groups these together into an equivalence class. It then performs an error injection experiment on just one representative error site (called a *pilot*) and uses its result to predict the outcome for all the error sites in the equivalence class. Hence, Relyzer is able to predict the resiliency characteristics of virtually all the error sites in the application with relatively few error injection experiments and high accuracy.

Approxilyzer builds upon Relyzer by introducing the notion of quality, determining the quality degradation for each predicted SDC, and applying this information in the areas of low-cost resiliency and approximate computing. It takes as input an unmodified program (with input), a quality metric, and an optional acceptable quality threshold, and produces a comprehensive *output quality profile.* This profile provides the outcome of a transient single bit error in each error site in an execution. The outcome can be masked (an output is produced and is the same as the golden, error-free output), detected (e.g., a fatal trap was invoked), or an output corruption (an output is produced, but is different from the golden output). For the last case, Approxilyzer further categorizes the output as a *detectable* data corruption (outputs that are visibly incorrect and could be detected; e.g., a not-a-number or NaN) or a *silent* data corruption. SDCs are further binned into buckets based on the output quality.

We show two applications of Approxilyzer's output quality profile in this chapter. First, for low-cost resiliency (assuming our error model), we use the observation that error sites that result in SDCs that are binned above an acceptable quality threshold do not need any protection. We show how this observation can be used by the programmer to quantitatively tune output quality versus resiliency versus overhead to enable ultra-low cost resiliency solutions.

Second, for a broader application of approximate computing, we observe that an error site is not amenable to approximation if it produces an outcome that is detected, detectable, or an SDC binned below an acceptable quality threshold. Although our error model is limited, it is reasonable to assume that if even a single bit perturbation produces such an unacceptable outcome, then a stronger perturbation will likely also produce the same, making the error site an unlikely candidate for approximation. Thus, Approxilyzer allows the system or programmer to focus on the remaining error sites (and constituent instructions) as candidates for approximation. These sites may or may not result in acceptable outcomes with stronger perturbations than single bit flips, but they provide a smaller subset for further (and potentially easier) analysis with other tools. This pruning of the space of approximable instructions is particularly valuable since it is completely automated – the only requirement from the programmer is the end-to-end quality metric. Knowledge of a threshold for acceptable quality is beneficial, but it is not necessary and can also be conservative. The more conservative the threshold, the more SDCs are deemed as not approximable (in the limit of no threshold, all SDCs are deemed not approximable). In this way, Approxilyzer enables a first-order, automated estimation of the potential for approximation for any general-purpose program.

While impactful as a general purpose resiliency tool, Approxilyzer's adoption was limited due to its implementation using the proprietary Simics infrastructure [71] and the SPARC [72] instruction set architecture (ISA). To that end, we also developed and open-sourced *gem5-Approxilyzer* [45], a re-implementation of Approxilyzer using the open-source gem5 simulator [73]. gem5-Approxilyzer can be extended to different ISAs, starting with x86 in this work. We show that gem5-Approxilyzer is both efficient (up to two orders of magnitude reduction in error injections over a naïve campaign) and accurate (average 92% for our experiments) in predicting the program's output quality in the presence of errors. We also compare the error profiles of five workloads under x86 and SPARC to further motivate the need for a tool like gem5-Approxilyzer.

2.2 BACKGROUND: PRINCIPLED ERROR PRUNING TECHNIQUES

Compared to a naïve campaign that performs an error injection for every error site, principled error pruning can dramatically reduce the number of error injections necessary to predict the error outcome for all error sites. Using the novel heuristics pioneered by Relyzer [42], we leverage two sets of pruning techniques: *equivalence-based* and *known-outcome* pruning techniques, which we incorporate for use in Approxilyzer. This section briefly describes these techniques; detailed explanations and examples can be found in prior work [42].

Equivalence-based pruning techniques use program analysis (static and dynamic) and heuristics to prune errors that are likely equivalent to others. These techniques partition error sites into *equivalence classes*, where each class requires an error injection into just a single representative error site (called a *pilot*) to predict the error outcome for all other error sites in the class. Approxilyzer implements two equivalence-based pruning techniques: control equivalence and store equivalence.

- **Control equivalence** groups error sites based on the observation that errors that propagate through similar code sequences are likely to have similar error outcomes. This technique records the next N branches for dynamic instances of a given static instruction in the original execution (with no error injection). Corresponding error sites of dynamic instances that share the same control path (up to depth N) are grouped in an equivalence class.

- **Store equivalence** is used to equalize dynamic instances of store instructions and instructions that a store depends on within a basic block based on the observation that errors in a store instruction propagate through the loads that read the erroneous value. This technique records the subsequent loads that read from a store address and groups corresponding error sites of dynamic instances of store instructions that have the same list of subsequent loads together.

Known-outcome pruning techniques largely use static (and some dynamic) program analyses to determine the outcome of an error. Approxilyzer uses two known-outcome pruning techniques:

- **Address-bound pruning** uses the observation that single-bit errors that appear outside the address range of an application result in *detected* outcomes. Thus, their outcomes are known a priori and these errors can be pruned.

- **Def-use pruning** uses the observation that an injection in a def is equivalent to an injection in the first use at the same register and bit position, so only one needs to be explored.

By combining all these pruning techniques, error analysis tools can dramatically reduce the total number of error sites that need detailed analysis. Specifically, Relyzer uses these techniques to prune 99.78% of error sites in an application, reducing the number of error sites requiring detailed study by 3 to 5 orders of magnitude. Relyzer uses error injection simulations on the remaining error sites to identify the outcome of errors, and shows 96% validation accuracy on average across all the applications studied.

2.3 APPROXILYZER: A FRAMEWORK FOR INSTRUCTION-LEVEL ERROR ANALYSIS

Approxilyzer builds upon Relyzer by introducing the notion of quality, determining the quality degradation for each predicted SDC, and applying this information in the areas of low cost resiliency and approximate computing. It takes as input an unmodified program (along with the application's input), a quality metric, and an optional acceptable quality threshold, and produces a comprehensive *output quality profile.* This profile provides the outcome of a single transient bit error in each error site in an execution. An outcome can be masked, detected, or an output corruption (OC), where output corruptions are further classified as either detectable data corruptions (DDCs) or silent data corruptions (SDCs). Additionally, SDCs are binned into buckets based on their output quality.

We demonstrate two uses of Approxilyzer. First, we show how Approxilyzer can be used to quantitatively tune output quality versus resiliency versus overhead to enable ultra-low cost resiliency solutions (with a single bit error model). For example, we show that Approxilyzer determines that a very small loss in output quality (1%) can yield large resiliency overhead reduction (up to 55%) for 99% resiliency coverage. Second, we show how Approxilyzer can be used to provide a first-order estimate of the approximation potential of general-purpose programs. It does so in an automated way while requiring minimal user input and no program modifications. This enables programmers or other tools to focus on the promising subset of approximable instructions for further analysis.

This section provides details of the Approxilyzer framework and its usage model, as illustrated in Figure 2.1.

2.3.1 Inputs to Approxilyzer

Underlying any approximate computing solution is the need to quantify output quality through an end-to-end quality metric. This metric is domain-specific [74, 75, 76] and Approxilyzer assumes that the programmer or user will supply it. Approxilyzer uses this metric to calculate the quality degradation of the erroneous output with respect to an error-free output.

Another parameter pertinent to many use cases for approximation is the quality threshold that sets a bound on the maximum quality degradation that is acceptable to the user. This is an optional parameter that Approxilyzer can take as input from the user. Since programmers may want to use Approxilyzer for analysis or tuning, Approxilyzer enables them to specify quality threshold ranges if they so desire. In the limit, no threshold range may be specified,

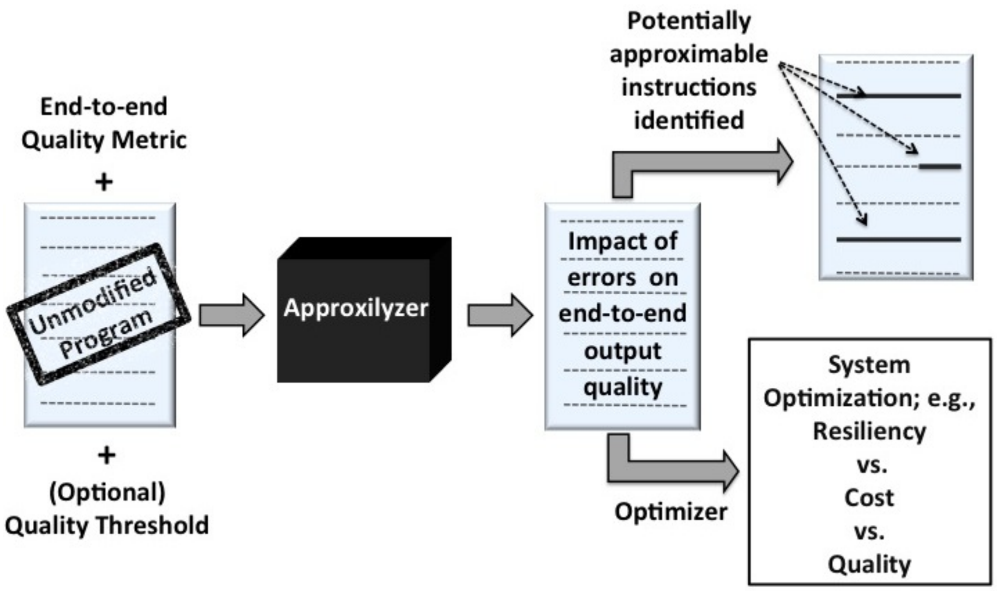

End-to-end Quality Metric
+
Unmodified Program
+
(Optional) Quality Threshold
Approxilyzer
Impact of errors on end-to-end output quality
Potentially approximable instructions identified
Optimizer
System Optimization; e.g., Resiliency vs. Cost vs. Quality

Figure 2.1: Overview of the Approxilyzer framework and its usage.

in which case Approxilyzer will perform its analysis for the full range of quality degradation.

To assist the user, we incorporate simple domain-specific libraries in our framework that include common sense quality metrics and thresholds that a user can choose. For example, the maximum of the relative (percentage) difference between the golden and error-free output components, L2-norms of matrices, and absolute differences are examples of quality metrics that quantify the deviation of the erroneous output from the error-free one. Negative values and infinities are examples of obviously unacceptable outputs for many financial applications – the user can choose to apply acceptable thresholds based on such criteria. Section 2.4.1 describes specific metrics and thresholds used in our evaluation.

Thus, Approxilyzer places the absolute minimum burden on the user, only requiring an end-to-end quality metric and, optionally, acceptable quality thresholds.

2.3.2 Classification of Errors

Approxilyzer aims to quantify the impact of a single bit transient error on the program's end-to-end output quality, for error sites comprising each register bit of each dynamic instruction in an execution. To accomplish this, Approxilyzer builds upon Relyzer [42], a tool to predict the outcomes of errors in all of the above error sites. Relyzer's predictions, however, only consider whether an error results in being *masked* (an output is produced and is the same as the golden, error-free output), *detected* (e.g., a fatal trap was invoked), or a *silent data corruption* (SDC). Relyzer does not consider output quality, marking all

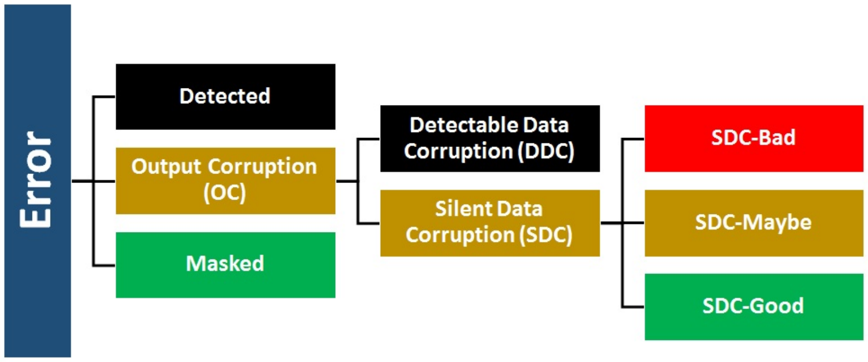


Figure 2.2: A classification of errors.

corruptions in an output (no matter how trivial) as an SDC outcome.

Approxilyzer introduces a new categorization of error outcomes to incorporate the notion of output quality into the category of errors traditionally known as SDCs. Figure 2.2 depicts the new taxonomy of errors. We use the term **output corruption (OC)** to indicate the outcome of an error where the execution runs to completion without crashing the program, but where the output does not match up identically to that of the golden output. In the literature, such outcomes have previously been uniformly referred to as SDCs. However, we observe that there is a subclass of these previously classified SDCs that is, in fact, detectable and not strictly silent. Such outcomes can be detected using a variety of low-cost mechanisms such as range detectors [19, 21]. We introduce additional detectors in Approxilyzer to catch NaNs, infinity values, negative outputs (if not expected by an application), and a check to see if the final output of the erroneous execution generates the same number of values as the golden output, irrespective of deviation. Our categorization refers to the output corruptions detected through the above means as **detectable data corruptions (DDC)**. It refers to the remaining output corruptions, which are not detectable and truly silent, as silent data corruptions.

The SDCs are further categorized as follows:

- ***SDC-Good:*** These SDCs are *highly tolerable* output corruptions which produce negligibly small quality degradations. This category also includes outcomes where the deviations from the golden output occur only in non-significant portions of the output (e.g., program related statistics and timing information). These SDCs do not require any resiliency protection since errors produced by them are inherently tolerable. The threshold for "small errors" is application and metric dependant.

- ***SDC-Maybe:*** These are potentially tolerable SDCs. The entire class is not outright tolerable, but a subset of SDCs in this class may be tolerable based on user-provided application quality constraints – usually in the form of an acceptable quality degradation threshold. Using Approxilyzer we show how we can extract SDCs that are tolerable from this class, tuning the output quality in accordance with cost- and reliability-benefits.

- ***SDC-Bad:*** These produce such large quality degradations that it can be reasonably assumed that they are not tolerable for most applications and users.

The above categorization of the SDCs is dependent on the domain-specific quality metric (required) and acceptable quality thresholds (optional) provided by the user. If a quality threshold is provided by the user, then whether an SDC is tolerable or not is a binary decision based on whether the resulting output quality degradation falls below or above the quality threshold. In the absence of user-provided quality thresholds (e.g., in cases where the user wants to undertake program analysis or tuning), Approxilyzer classifies the SDC error sites into SDC-Good, SDC-Bad, and SDC-Maybe. Note that the classification into SDC-Good and SDC-Bad occurs only if the user chooses to apply common sense domain-specific thresholds provided by the tool (Section 2.4.1). Otherwise, all the SDC error sites are classified as SDC-Maybe. For each error site belonging to the SDC-Maybe error class, Approxilyzer also records its associated output quality degradation. It assesses the quality of the corrupted/erroneous output by measuring its deviation from the error-free/precise output using the application-specific quality metrics provided by the user. For example, an error that produces a quality degradation of 20% is said to have a different outcome from one with a quality degradation of 25%. Hence, the output quality for a given error site is characterized by its error outcome class and additionally, in the case of SDC-Maybe, by the amount of quality degradation introduced in the output.

Approxilyzer hypothesizes that Relyzer's main insight (that errors propagating "similarly" through the program are likely to result in similar outcomes) also holds true when considering quality as part of the error outcome. That is, errors propagating "similarly" through the program are likely to generate program outputs of similar quality. Approxilyzer uses Relyzer's heuristics related to control and data flow to predict similarity and to divide error sites into equivalence classes (Section 2.2). We define validation experiments to test this hypothesis and show that this is indeed the case (Section 2.4). Thus, Approxilyzer is able to enumerate, with high confidence, the output quality generated when each single error site in the program is perturbed by a single bit corruption. We use the phrase "output quality of error site" to refer to the quality degradation in the output generated when an error is injected in the error site.

Table 2.1: Error outcomes and their potential for approximation and resiliency overhead savings.

Error outcome category	**Is this class of error sites approximable?**	**Does this class of error sites need resiliency protection?**
Masked	✓	✗
SDC-Good	✓	✗
SDC-Maybe	Maybe	Maybe
SDC-Bad	✗	✓
DDC	✗	✗
Detected	✗	✗

2.3.3 Instruction-Level Approximation Opportunities

Given an unmodified program and end-to-end quality metrics, Approxilyzer analyzes the program and automatically provides the programmer with a set of instructions that are potential first order candidates for approximations. Approxilyzer does this by eliminating instructions that have unacceptable output quality. The underlying argument that Approxilyzer makes is that if an instruction produces an unacceptable quality output in the presence of single bit corruptions, then it is highly unlikely to generate an output of acceptable quality with more vigorous perturbations introduced by approximation.

Table 2.1 provides a classification of which error outcome categories are approximable and which are not. Error sites that produce Detected, DDC and SDC-Bad outcomes are clearly not acceptable and Approxilyzer marks them as not approximable. SDC-Good and Masked error sites are marked as approximable. SDC-Maybe error sites are potential candidates for approximation depending on whether their quality meets the acceptable quality threshold set by the user. The approximation potential of an instruction is decided based on the nature of its constituent error sites. If any error site in an instruction is deemed not approximable then the instruction is marked by Approxilyzer as not approximable. Otherwise, the instruction is marked as a potential candidate for approximation.

Since each error site in the application contains the information regarding which dynamic instance of an instruction it belongs to, Approxilyzer can identify dynamic instructions that can be approximated. This can be useful to determine if the application will benefit from approximation techniques at the dynamic instruction granularity (e.g., task skipping [77] and loop perforation [78]). Section 2.5.4 shows a case study for how Approxilyzer can be used to analyze applications for approximation potential.

Since our framework uses transient single bit errors as the error model, instructions marked as approximable by Approxilyzer may be false positives, since they may produce unacceptable quality output with approximation techniques that use different error models. False negatives, however, are expected to be rare since in most cases if a single bit upset in an instruction causes an unacceptable outcome, then it is highly likely that multi-bit upsets will also result in unacceptable outcomes. While our approach is aggressive, we believe it is still useful since it narrows the huge exploration space for approximation to a manageable smaller set of instructions on which it is feasible to do further rigorous and targeted analysis. Another benefit of our approach is that the identification of approximable instructions is automatic and needs only minimal programmer input – end-to-end quality metrics and quality thresholds – and no program modifications. This makes it feasible even for novice programmers to analyze any program for hidden approximation opportunities.

2.3.4 Tuning Quality vs. Resiliency vs. Overhead

Approximate computing environments often trade accuracy in the program output for gains in other system parameters such as energy or performance. A framework like Approxilyzer, which quantifies the output quality of each error site in the program, can be used to tune the loss in output accuracy with respect to other system benefits. We study one such system benefit; namely, the reduction in the overhead costs related to resiliency. We describe how Approxilyzer can be used to tune overhead costs with respect to desired resiliency protection for different output quality requirements and show that this can enable ultra-low cost resiliency solutions (Section 2.5.3).

Approxilyzer uses its knowledge of each error site's output quality to decide whether that error site needs protection from transient errors (Table 2.1). Error sites that result in masked outcomes do not need to be protected since they produce the golden output even in the presence of transient errors. Low cost detectors (as discussed earlier) can be used to catch the detected category of errors and hence the associated error sites do not need to be protected.

In the absence of Approxilyzer, we would have to protect all OC error sites. With Approxilyzer's quality information, the system can selectively protect only those OCs that are neither tolerable by the user/application, nor can be protected by low cost detectors. Since SDC-Good is inherently tolerable and DDC (like detected) can be captured using other low cost detectors, these error sites need not be protected. SDC-Bad error sites produce intolerable outputs and hence they always have to be protected. SDC-Maybes may or may not need protection based on whether they meet the user's quality threshold. This reasoning about which error sites need protection from transient errors can be extended to instructions based on the quality of their constituent error sites. If an instruction contains an error site that needs to be protected, then we say that the instruction needs to be protected.

Thus, based on the type and quality of the OCs produced, Approxilyzer can selectively tune the set of OC causing instructions chosen for protection from transient errors.

2.4 EVALUATION METHODOLOGY

2.4.1 Quality Assessment and Metrics

Table 2.2 details the applications, inputs, quality metrics, and quality threshold ranges used in our evaluations. To quantify the quality of the corrupted output, we must find a measure of its difference from the golden (error-free) output. We refer to this "difference measure" as the quality metric – technically, this is a quality degradation metric since the higher the value of the difference, the lower the quality.

Table 2.2: Applications, quality metrics, thresholds, and quality bins.

Application	**Input**	**Metric**	**DDC**	**SDC-Good**	**SDC-Bad**	**SDC-Maybe QB : {Error range}**
Blackscholes [79]	sim-large	max-rel-err max-abs-diff	$F_i > \$500$ $F_i < \$0$	max-abs-diff $< \$10^{-4}$	max-rel-err $> 100\%$	max-rel-err: 1: $\{10^{-4}\% \leftrightarrow 1\%\}$ 2: $\{1\% \leftrightarrow 2\%\}$ ... 99: $\{98\% \leftrightarrow 99\%\}$ 100: $\{99\% \leftrightarrow 100\%\}$
Swaptions [79]	sim-small	max-abs-diff	$F_i > \$500$ $F_i < \$0$	max-abs-diff $< \$10^{-4}$	max-abs-diff $> \$1$	max-abs-diff: 1: $\{10^{-4} \leftrightarrow 10^{-3}\}$ 2: $\{10^{-3} \leftrightarrow 10^{-2}\}$ 3: $\{10^{-2} \leftrightarrow 10^{-1}\}$ 4: $\{10^{-1} \leftrightarrow 1\}$
LU [80]	512x512 matrix 16x16 blocks	max-rel-err	No App-Specific Detectors	max-rel-err $< 10^{-4}\%$	max-rel-err $> 100\%$	max-rel-err: same binning as Blackscholes
Water [80]	512 molecules	max-rel-err	No App-Specific Detectors	max-rel-err $< 10^{-4}\%$	max-rel-err $> 100\%$	max-rel-err: same binning as Blackscholes
FFT [80]	64K points	rel-l2-norm	No App-Specific Detectors	rel-l2-norm $< 10^{-4}\%$	rel-l2-norm $> 100\%$	rel-l2-norm: same binning as Blackscholes
Common to all apps			F_i = NaN F_i = Inf #F != #G	Errors in non-significant portions of the output		

In the absence of specific domain studies and standardization [74, 75, 76], we have done our best to choose quality metrics that strike a balance between over- and under-estimating an application's tolerance to errors. For example, consider outputs with multiple components. Without further guidance, we must first determine a difference function for each component and then a method to aggregate across the components. Depending on the magnitude of the individual components, we use the absolute difference (small magnitude) or the relative difference (large magnitude) for the per-component difference function. To aggregate across components, we use the maximum instead of the average. In cases where there is an established common practice to analyze the output, we use the corresponding quality metric. For example, FFT produces a matrix and we use the relative difference in the bounded L2 norm to determine the quality.[1] More precisely, given a golden output G and a faulty output F, both having n components, where $n \geq 1$, Table 2.2 uses the following three quality metrics.

(1) ***max-abs-diff*:** This metric calculates the maximum absolute difference between the components of the golden and faulty outputs.

$$\textit{max-abs-diff} = \textit{max}(|G_1 - F_1|, |G_2 - F_2|, \ldots, |G_n - F_n|) \tag{2.1}$$

(2) ***max-rel-err*:** This metric calculates the maximum of the relative error between the individual components of the golden and faulty outputs.

$$\textit{rel\_err}_i = \frac{|G_i - F_i|}{G_i} \times 100 \tag{2.2}$$

$$\textit{max-rel-err} = \textit{max}(\textit{rel\_err}_1, \textit{rel\_err}_2, \ldots, \textit{rel\_err}_n) \tag{2.3}$$

(3) ***rel-l2-norm*:** This metric is typically used in mathematics to directly compare two matrices. The metric estimates the relative difference in the bounded L2 norms (BL2N) of the golden and erroneous matrices. For any matrix A, having n elements, $a_1, a_2, \ldots, a_n$, we define the following,

$$\|A\|_{BL2N} = \frac{\|A\|_{L2}}{n} \tag{2.4}$$

where,

$$\|A\|_{L2} = \sqrt{\sum_{i=1}^{n} a_i^2} \tag{2.5}$$

The rel-l2-norm is thus defined as:

$$\textit{rel-l2-norm} = \frac{\|G - F\|_{BL2N}}{\|G\|_{BL2N}} \times 100 \tag{2.6}$$

Table 2.2 also lists the quality threshold ranges for identifying SDC-Good and SDC-Bad. We use fairly conservative values that we believe will be reasonable for most users and applications.

Finally, although output quality is a continuous function, for ease of analysis and comparison, we discretize it into multiple *Quality Bins (QB)*, fine-grained enough to capture small

[1]We do not use this for LU because it effectively produces two triangular matrices and how the errors in the two are composed depends on how the output is used.

quality variations (last column of Table 2.2). For example, with the *max-rel-err* metric, the bins are 1% wide (except at the boundary), and quality values of 12.1%, 12.6% and 13.3% are assigned a QB of 13, 13, and 14 respectively. Thus, SDC-Maybe with QB10 refers to an SDC-Maybe outcome with an output quality degradation in the range specified by QB10.

2.4.2 Error Injection Framework

Our error injection simulation infrastructure for Approxilyzer is similar to that used for Relyzer [42]: it is based on Wind River Simics [71] and GEMS [81] for running our applications on OpenSolaris and compiled to the SPARC V9 ISA.

We inject single bit flips in integer and floating point architectural registers. Hence, we only consider instructions that employ either an integer or floating point register as an operand. For example, we do not inject errors in instructions such as *call* (no operands), *ret* (no operands) or branches that use special condition code registers. Such instructions will not be considered for approximation or resiliency protection.

We perform error injections only in the pilots of the generated equivalence classes. This can still lead to a large number of error injections, especially for longer applications. In order to reduce the simulation time, we only study 99% of the error sites in the application for Approxilyzer, thereby trading off simulation time for a modest loss in coverage [42]. The 1% of error sites not included in the study do not detract from the observations and gains reported.[2] For resiliency overhead tuning (Section 2.5.3), these unexplored error sites might represent missed opportunity (in the event that they produce SDCs) for further overhead reduction using Approxilyzer. When identifying approximable instruction (Section 2.5.4), these remaining error sites might introduce some false positives (in the event that they produce unacceptable errors). This is, however, consistent with our goal to tolerate some false positives, while minimizing false negatives, in the quest to uncover approximation opportunities in the application.

Approxilyzer retains Relyzer's speed benefits, with negligible additional overheads. Compared to a (hypothetical) framework that would perform an error injection for each error site, Relyzer is able to prune error injections by 3 to 5 orders of magnitude [42]. The remaining error injections complete on our cluster of 200 machines in a few days. Approxilyzer adds a few hours to this process to perform quality calculations and error outcome categorizations. The analysis to generate quality versus resiliency versus overhead curves for an application (Section 2.5.3) takes several minutes. Analyzing error outcomes and quality to

[2]We address the remaining 1% of error sites in Chapter 3 by performing *input minimization* to reduce the total number of error sites, making 100% error site evaluation feasible.

identify approximable instructions (Section 2.5.4) takes a few seconds per application.

2.4.3 Validation Methodology

Approxilyzer relies on Relyzer's heuristics to group error sites that produce similar quality outcomes into an equivalence class (Section 2.3.2). It predicts the quality of each element of an equivalence class based on the outcome of a fault injection experiment on its pilot. To validate these predictions, we perform experiments similar to those in Relyzer [42].

The validation experiment asks the question: how accurately does the output quality of the pilot predict the output quality of the other error sites in its equivalence class? For validating a single pilot, we perform error injections in a sample of error sites from the pilot's equivalence class. We then compare the output quality of the population with that of the pilot to gain confidence that the pilot accurately represents the population, and hence the equivalence class. For example, a pilot that produces a DDC has a 100% validation/prediction accuracy if the injection experiments for all of its associated population also produced DDCs. If only 90% of the population produced a DDC, the pilot's prediction accuracy is 90%.

To validate a pilot of an SDC-Maybe class, we further require that the QB of the pilot match that of the population to be considered a correct prediction. For example, consider a pilot X that generates an SDC-Maybe with QB12. Suppose 86% of its population is SDC-Maybe with QB12, 6% is SDC-Maybe with QB13, 5% is SDC-Maybe with QB10, and 3% is SDC-Bad. Then the prediction accuracy of pilot X is 86%.

The overall prediction accuracy for an application is obtained by calculating the average of the prediction accuracy across all the pilots studied, weighted by the size of their equivalence class.

Requiring the pilot's QB to exactly match the QB of the associated population is unnecessarily conservative and a tall order for any tool, especially for outcomes with quality at the QB boundaries. We therefore introduce a flexibility parameter, δ, that allows a fine-grained margin of error at QB boundaries. For the validation of pilot X described above, setting $\delta = x$ means that an error site in its population with QB of $12 \pm x$ would be considered as a correct prediction. Thus, pilot X's prediction accuracy with $\delta = 1$ is 92% and with $\delta = 2$ is 97%. The baseline validation uses the setting $\delta = 0$.

We can further loosen our constraints on validation by considering the context in which Approxilyzer is used as follows:

- **Validation for Resiliency:** In the first case, Approxilyzer is used to determine which instructions need to be protected for resiliency. We therefore do not need to distinguish between Masked, SDC-Good, DDC, and Detected outcomes since all of them do not require protection. We therefore group these outcomes together.

- **Validation for Approximation:** In the second case, Approxilyzer is used to determine which instructions are approximable. We therefore do not need to distinguish between Masked and SDC-Good outcomes since they are approximable, and can group them together. Similarly, we can group SDC-Bad, DDC, and Detected outcomes together since they are not approximable.

Thus, for a given use case, a pilot is said to have a correct prediction for a member of its equivalence class if both the pilot and the member produce an outcome within the same group as defined above for the use case. We use δ_{res} and δ_{approx} when considering use-specific validations for resiliency and approximation respectively. We continue to use δ for use-oblivious validations.

As an example, consider a Pilot Y that generates a DDC and has the following population outcome distribution: DDC: 85%, Detected: 7%, SDC-Good: 5% and SDC-Bad: 3%. The prediction accuracy of Y is 85% for $\delta = 0$, 97% for $\delta_{res} = 0$, and 95% for $\delta_{approx} = 0$.

We perform the validation experiments for ∼700 pilots from each application. This gives us a 99% confidence interval with a 5% error margin. We validate each pilot against a sample population size of 750 (drawn randomly from the equivalence class), which also gives us a statistical confidence of 99% with a 5% error margin. In all, we perform approximately 2.6 million error injection experiments for validating Approxilyzer.

2.5 RESULTS

We begin the evaluation in Section 2.5.1 by showing the output corruption distribution as analyzed by Approxilyzer, followed by validation results in Section 2.5.2. Section 2.5.3 describes Approxilyzer results for tuning an application's quality versus resiliency versus overhead tradeoffs. Section 2.5.4 shows how Approxilyzer can be used to identify approximable instructions in an application, using the duality of the tool for resilience and approximation purposes.

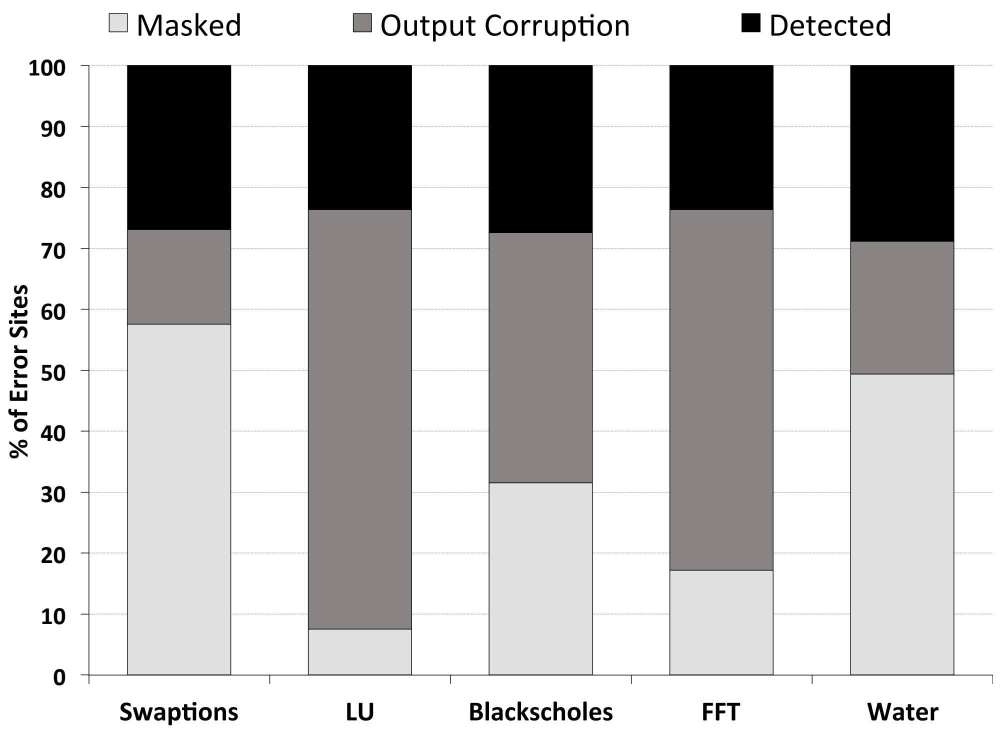


Figure 2.3: Distribution of error outcomes for the applications studied.

2.5.1 Output Corruption Distribution

Figure 2.3 shows the distribution of outcomes for error sites in the studied applications.[3] Each application exhibits a unique distribution of error outcomes. At 68.8%, LU contains the highest percentage of Output Corruption (OC) causing error sites and Swaptions, at 15.6%, the lowest.

Figure 2.4 shows the different categories of output corruptions, separately for integer and floating point register error sites. Swaptions and Water show very high percentage of SDC-Good at 76% and 58% respectively while Blackscholes produces 68% DDC (for both Integer and Float combined). Swaptions and FFT show an interesting dichotomy in the behavior of errors in the integer versus floating point registers, implying perhaps, a need for separate techniques for resiliency and approximation across the two different register classes. LU's OC error sites are almost exclusively (>98%) composed of SDC-Bad outcomes. This may either imply that LU is inherently not tolerant to errors or that the quality metric used to classify errors in the output of LU may not be the correct choice.

These results illustrate how Approxilyzer can be employed to automatically analyze an

[3]The OC (originally SDC) rates reported in this work are different from the rates reported in previous work [21, 42] as our error model is different. We study errors in both integer and floating point architectural registers, while our prior work only considered integer registers.

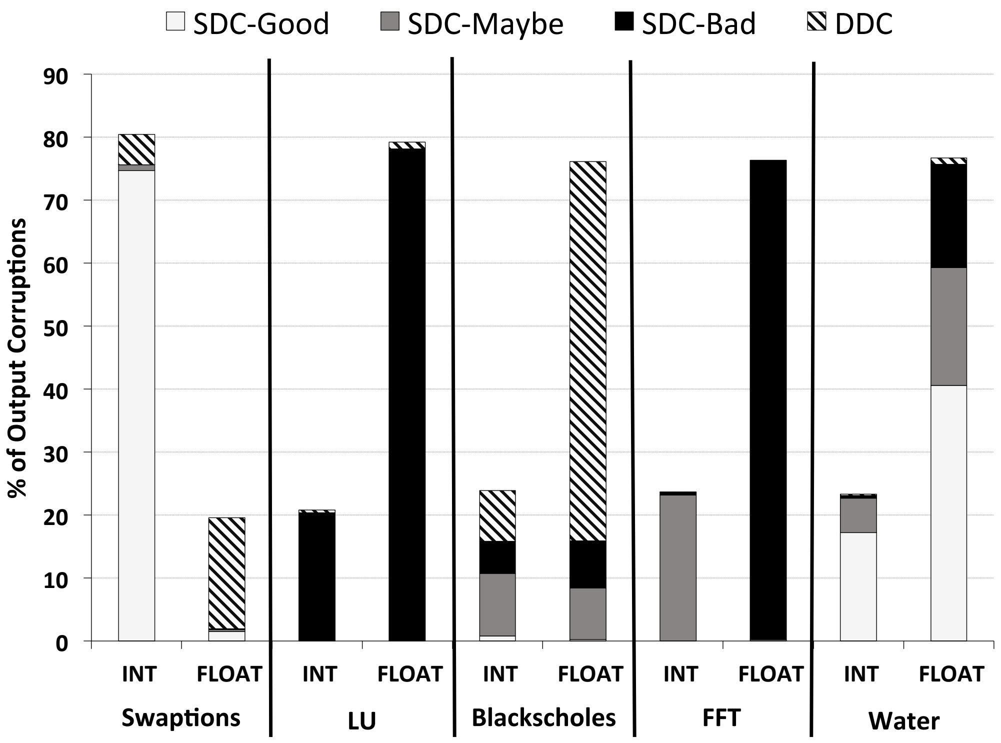


Figure 2.4: Distribution of output corruptions (OC) in integer (INT) and floating point (FLOAT) registers.

application to gain insights into its behavior in the presence of perturbations.

2.5.2 Validation

Validation results for resiliency: The results for validation geared towards resiliency are shown in Figure 2.5. All of the applications show very high pilot prediction rates with an average prediction rate of 96% across applications using a very fine quality window of 2 (i.e., $\delta_{res} = 2$).

Swaptions and Water see big gains in validation just by applying the δ_{res} optimization. Both of these applications have high SDC-Good rates (Figure 2.4) and therefore many pilots that are picked for validation belong to the SDC-Good outcome category. Some of these equivalence classes (with SDC-Good pilots) contain a mix of Masked and SDC-Good error sites, leading to lower overall validation for $\delta = 0$.

Water especially has a high rate of SDC-Good outcomes which are due to very small errors ($< 10^{-6}\%$) in the program statistics part of the output file. Approxilyzer heuristics (not surprisingly) combine these error sites with Masked outcomes into equivalence classes. As a result, applying the δ_{res} optimization causes the validation rate of Water to jump from

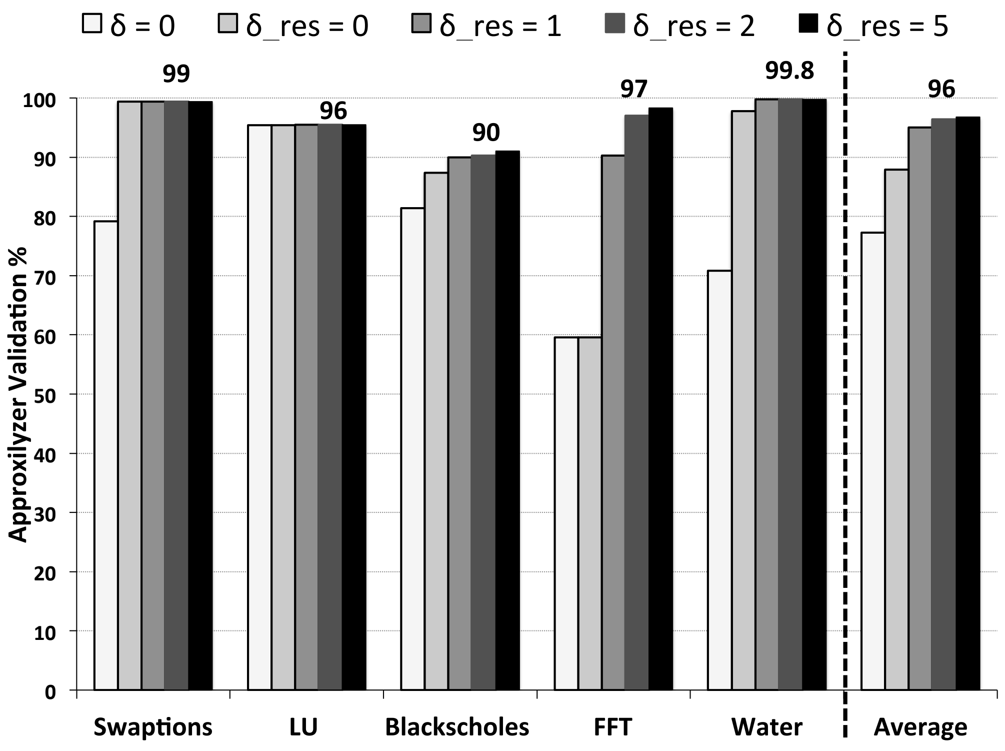


Figure 2.5: Approxilyzer validation geared towards resiliency.

71% to 98%.

In addition to having equivalence classes with mixed SDC-Good and Masked outcomes (as described above for Water), Swaptions also has some pilots with DDC outcomes (Figure 2.4) belonging to equivalence classes that feature a mix of DDC and Masked outcomes. These pilots represent error sites from a few floating point instructions that process randomly generated numbers. If the error causes the random number to exceed the (expected) range of 0 to 1, it causes floating point overflows which result in NaN values. Because Approxilyzer heuristics cannot accurately distinguish this special case, Swaptions contains some equivalence classes with a mix of Masked and DDC outcomes which results in poor validation for $\delta = 0$. Applying the δ_{res} optimization, causes the validation rate of Swaptions to go up from 79% to 99%.

While still high at 90% (for $\delta_{res} = 2$), Blackscholes shows the lowest validation accuracy of the applications studied. Further analysis shows that this is due to a few pilots whose equivalence classes have a mix of SDC-Maybe and SDC-Bad outcomes. This is why increasing the quality window size (δ) does not cause the prediction rate to increase. The reason behind the mixed equivalence class can be attributed to the fact that Blackscholes calculates the option price for a portfolio containing more than 64,000 options and hence, the same

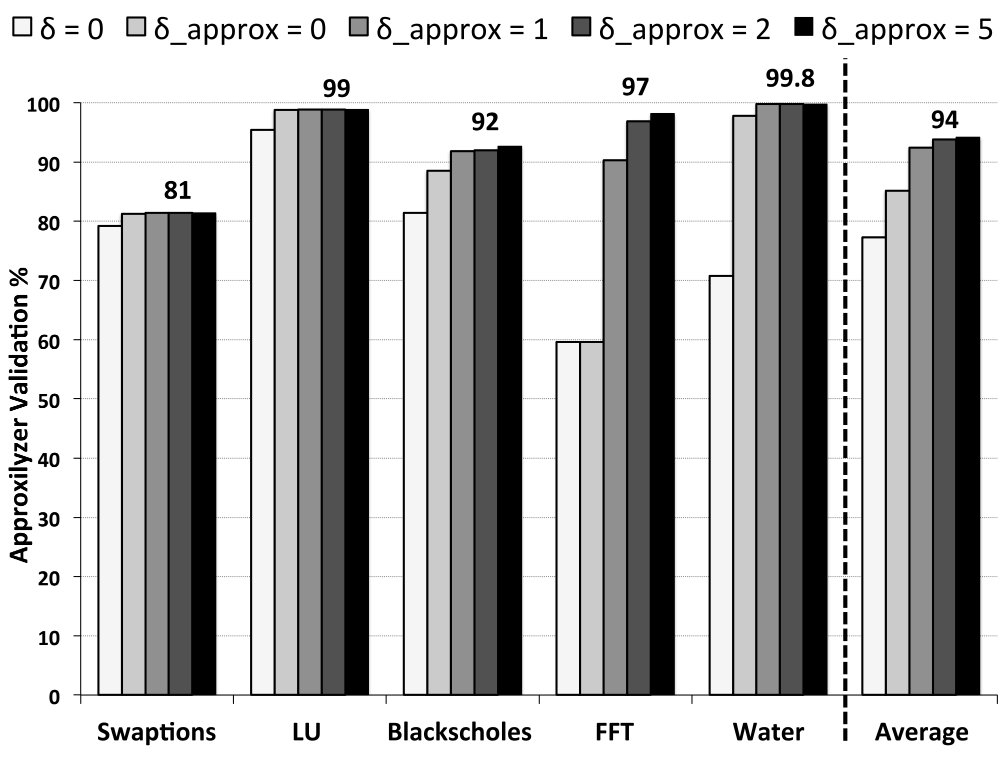


Figure 2.6: Approxilyzer validation geared towards approximation.

instructions produce OCs of different quality based on the input being processed at any given execution cycle. While range detectors can be applied to better capture variations in data patterns in certain SDC-Bad outcomes and specialized heuristics to handle some of these cases, we leave their implementation to future work. In spite of these special cases, Blackscholes shows high prediction rate.

Validation results for approximation: Figure 2.6 shows the graph for validation considering approximation. On average, the validation percentage for $\delta_{approx} = 2$ is 94% across all applications. Swaptions shows lower validation predominantly due to poorly validated DDC pilots belonging to a few floating point instructions (validation accuracy for integer pilots with $\delta_{approx} = 2$ is 97%) operating on random numbers, as described above. While the δ_{res} optimization equalizes these outcomes, the δ_{approx} considers DDC and Masked outcomes separately and hence the validation accuracy is not improved. Simple range detectors to check the range of the random numbers can resolve this issue and we leave its implementation to future work.

Overall, the average validation percentage, across $\delta_{approx} = 2$ and $\delta_{res} = 2$, for the applications studied is 95%. Thus, we conclude that Approxilyzer can capture the output corruption quality – at very fine granularities – with high precision for the purposes of both resiliency and approximate computing.

2.5.3 Tuning Quality vs Resilience vs Overhead

We demonstrate the ability of the user to harness Approxilyzer to tune application output quality versus other system attributes with a study targeted towards system resiliency. As explained in Section 2.3.4, we can target specific static instructions for resiliency protection based on the quality threshold specified. Given additional criteria regarding resiliency (the fraction of output corruption producing error sites in the application that must be protected, referred to as *"resiliency coverage"* or simply *"coverage"*) and the maximum overhead to be incurred for protection, an optimizer can pick the optimum balance of output quality, resiliency coverage, and overhead to target user requirements. We produce tuning curves that show the tradeoffs for different combinations.

To produce the different tuning curves, we first identify the instructions that need resiliency protection for different output quality thresholds (QT). Then we use a 0/1 knapsack algorithm to pick the instructions for resiliency protection that offer the specified coverage for the least overhead. A similar methodology is used by Relyzer [42] to tune resiliency versus overhead, but without relaxation of output quality.

We assume instruction redundancy as our error protection scheme and charge one instruction worth of overhead to protect a given instruction. Hence, the execution overhead cost for protecting static instruction X is equivalent to the dynamic instruction count of X.[4]

To illustrate the above with a simple example, consider two candidate static instructions A and B, each responsible for 30% and 20% of the output corruption error sites in the application, and producing 5% and 10% of the dynamic instructions, respectively. Assume Approxilyzer determines that the maximum quality degradation produced by an error in A and B is 1% and 4% respectively. Then for no quality loss, to cover 50% of the output corruption error sites (resiliency coverage), both A and B have to be protected and the (execution) overhead cost of doing so is their cumulative dynamic instruction count; i.e., 15%. If the user is willing to accept a quality loss of 2%, we do not have to protect A, and essentially get the resiliency coverage afforded by A (30%) at no additional overhead cost. For resiliency coverage of 50% (with acceptable quality loss of 2%), we will need to protect B and incur an overhead of 10%.

Figure 2.7 shows the resiliency overhead cost versus coverage for different levels of accept-

[4]Although this is a reasonable assumption, in Chapter 4, we show how actual instruction duplication overhead is not always precisely a unit overhead per instruction, and introduce different techniques to reduce the duplication overhead.

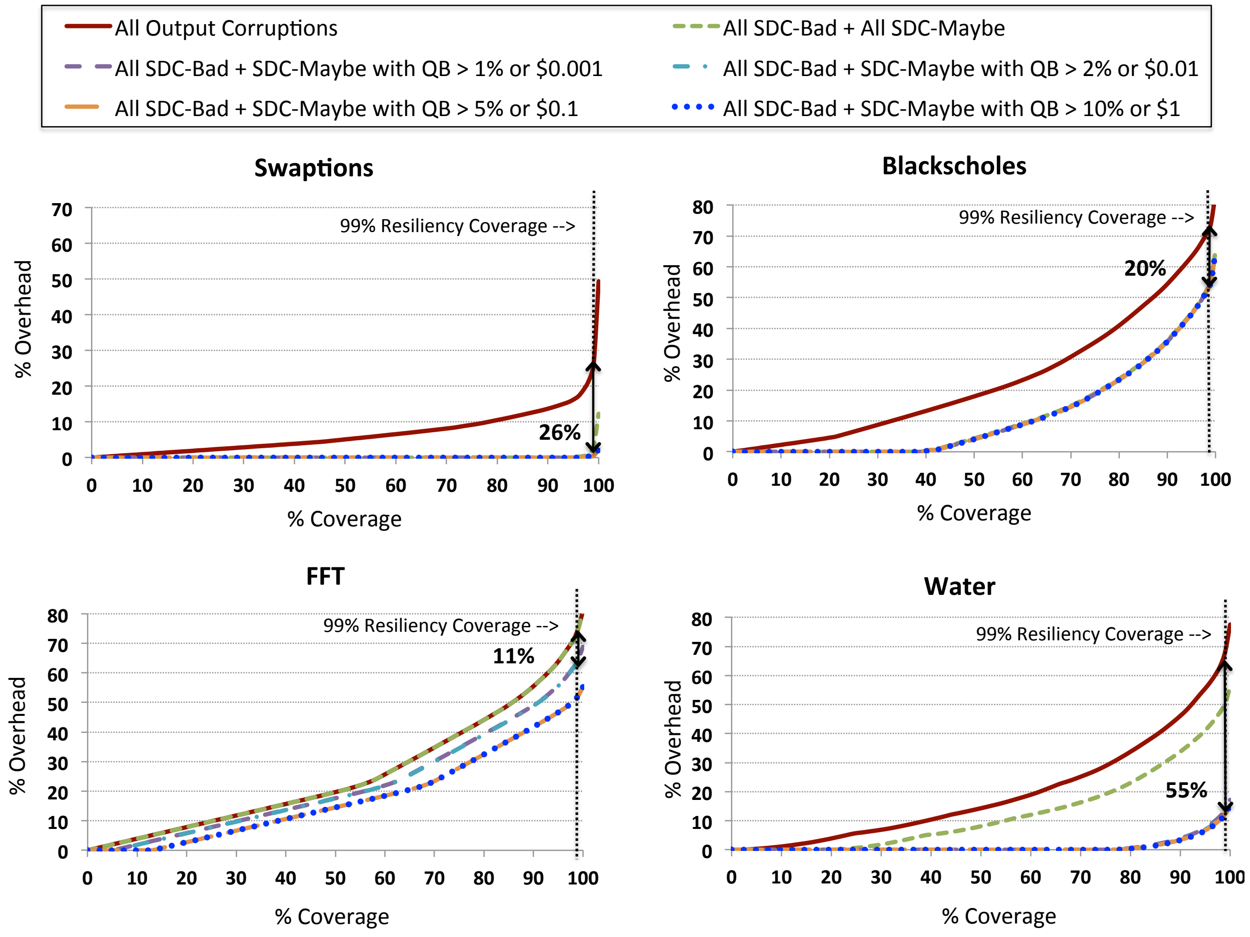


Figure 2.7: Tuning resilience versus execution overhead versus output quality for different applications.

able output quality degradation. We show graphs for four of our five benchmarks (the fifth, LU, is discussed later). Each graph shows the following curves corresponding to different levels of acceptable quality degradation.

- ***All Output Corruptions***: This curve represents the optimal overhead versus coverage when all OC causing instructions are protected. It represents the state-of-the-art in the absence of Approxilyzer's output quality impact information to distinguish instructions that produce acceptable quality output.

- ***All SDC-Bad + SDC-Maybe***: This curve shows the optimal overhead versus coverage when all the instructions causing SDC-Bad and SDC-Maybe outcomes are protected. This is the graph that will be generated by Approxilyzer in the absence of specific user-defined quality thresholds. Approxilyzer automatically removes the instructions that only produce SDC-Good and DDC from the list of instructions to protect.

- ***All SDC-Bad + SDC-Maybe with QB>x***: These are the optimal overhead versus coverage curves with user-specified quality threshold x. For these curves, Approxilyzer does not protect instructions that produce SDC-Maybe with QB$\leq x$ from the list of instructions protected. This essentially means that if a user says that she is willing to tolerate x% quality loss in the output, then we need not protect the instructions that we know will not suffer a quality degradation greater than x% in the presence of transient errors. For convenience, the graphs show x as an actual application-specific quality threshold instead of a QB number.

The gaps between the various curves for each point along the x axis represent the overhead/cost savings by not applying resiliency protection to those instructions that produce acceptable quality loss when perturbed. The benchmarks shown in Figure 2.7 show significant overhead savings if the user can tolerate very small quality loss. For example, if the user can tolerate a 1% quality degradation in the output, then the resiliency overhead costs can be reduced by 20%, 55%, and 11% for Blackscholes, Water and FFT respectively, while still achieving 99% coverage (the difference between the *All Output Corruptions* and *All SDC-Bad + SDC-Maybe with QB>1%* curves at 99% on the x axis). Similarly, for a quality loss of less than one hundredths of a penny in final stock price (*All SDC-Bad + SDC-Maybe with QB>$0.001*), Swaptions achieves an overhead reduction of 26% while providing 99% coverage.

Swaptions has many instructions that exclusively contain SDC-Good error-sites. Hence the overhead is significantly reduced by not protecting those instructions (99% coverage for *All SDC-Bad + SDC-Maybe* has an overhead of 3%). Further increasing the application's quality degradation threshold provides marginal benefits (2% overhead reduction).

Blackscholes also does not show any benefit from increasing the quality degradation threshold (QB), but for a different reason. As mentioned in Section 2.4.3, many (static) instructions in Blackscholes produce a mix of SDC-Bad and SDC-Maybe outcomes (with wide QB ranges) and hence they are always protected. Blackscholes does, however, achieve a 20% overhead reduction by not protecting instructions that only generate DDC and/or SDC-Good outcomes.

FFT, on the other hand, displays a behavior contrary to Swaptions and Blackscholes – all its overhead reductions come from increasing the acceptable quality threshold of SDC-Maybes (the curves for *All Output Corruptions* and *All SDC-Bad + SDC-Maybe* sit on top of each other). This can be attributed to the fact that none of the instructions in FFT exclusively produce only DDC or SDC-Good outcomes. Changing the quality threshold from *QB>1%* to *QB>5%* results in an additional overhead reduction of 12% for the 99% coverage point.

Water shows the most overhead reduction while tolerating a small quality loss. This is because Water has many instructions that contain a mixture of SDC-Good and SDC-Maybe error sites that result in very small quality degradation. Hence even a small quality degradation threshold results in large gains.

LU shows no gains from either quality tuning or from not protecting SDC-Good and DDCs (not shown in the figure for brevity). This is because, as seen from Figure 2.4, LU produces only SDC-Bad corruptions and hence all of the instructions need protection.

In summary, most of the applications show significant resiliency overhead reductions while suffering very small accuracy losses. Thus, Approxilyzer can be used to target ultra-low cost resiliency solutions in an approximate environment.

2.5.4 Approximation Opportunities

As mentioned in Section 2.3.3, Approxilyzer can be used as a tool to analyze the first order approximation potential of an unknown application along different dimensions. We show one such use case where we use Approxilyzer to analyze the approximation potential along the dimension of static instructions for our five workloads. We use the same technique used to identify which static instructions are potentially approximate, to also identify approximation along different static instruction granularities. For example, if all the error sites related to a particular register in a static instruction were deemed approximable, then we say that the register is approximable. In this case study we do this analysis for the following static instruction granularities:

- ***Full Instruction (FI)*:** The entire static instruction (i.e., all register bits) is approximable.
- ***Partial Instruction, Full Register (PI_FR)*:** At least one full register in the static instruction is approximable.
- ***Partial Instruction, Partial Register, x bits (PI_xb)*:** At least one x bit long register chunk in the static instruction is approximable.

For the purposes of this study we do not assume a quality threshold. Instead, we estimate the best and worst case approximation bounds. For the best case, we assume that all the SDC-Maybes have acceptable quality and hence are approximable. For the worst case we assume that none of them have acceptable output quality and therefore are not approximable.

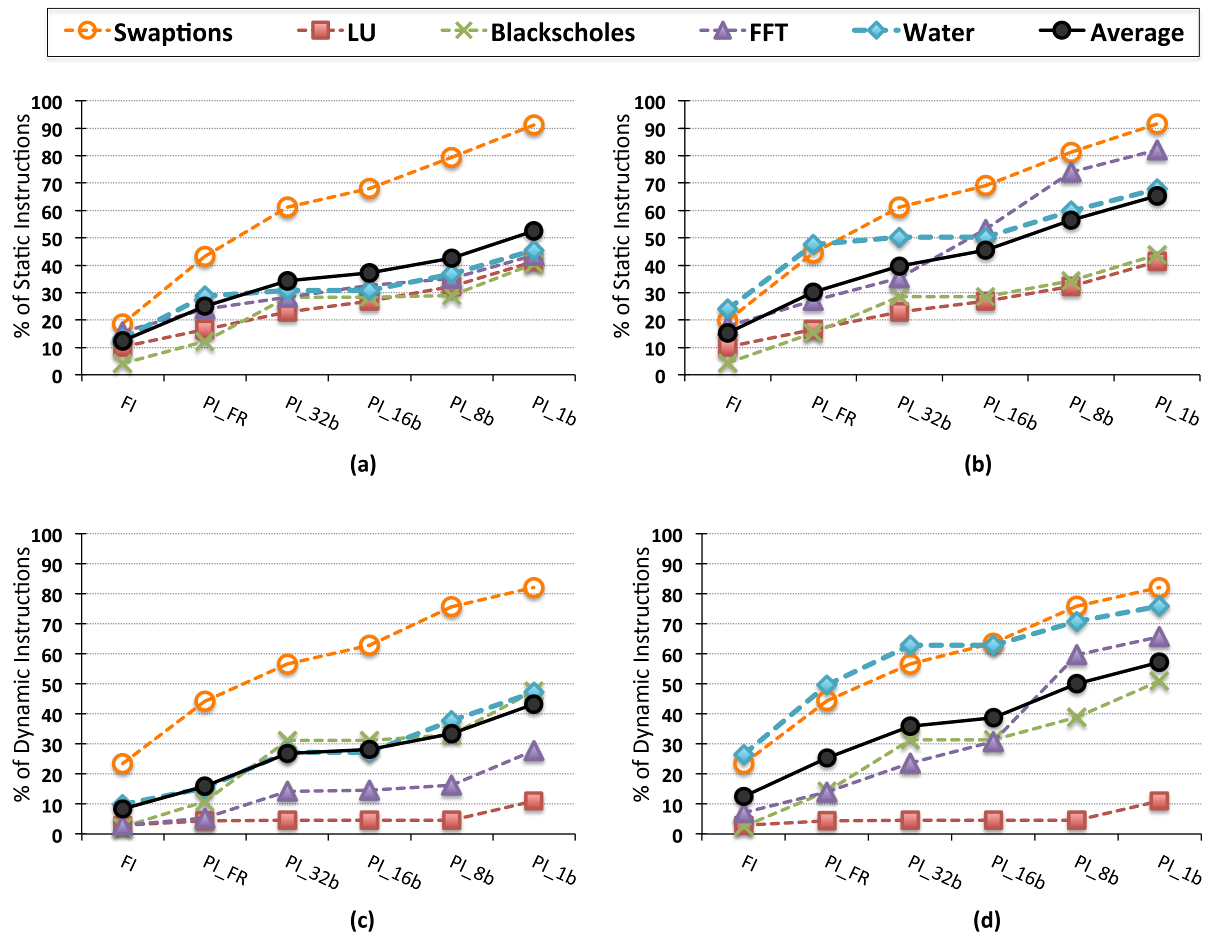


Figure 2.8: Graphs (a) and (b) show the first order worst and best case bounds respectively for the percentage of static instructions (studied) that are approximable for each application at different granularities of approximation. Graphs (c) and (d) show the percentage of dynamic instructions (in the full application) generated by these static approximable instructions in the worst and best case respectively.

Figures 2.8(a) and 2.8(b) show, for each application the worst and best case bound, respectively, on the number of static instructions, marked by Approxilyzer as candidates for approximation. In order to understand the potential impact of approximating these static instructions, Figures 2.8(c) and 2.8(d) show the proportion of dynamic instances produced by these static instructions in the full application. Note that while the static instruction percentage shown is for the fraction of static instructions studied, for a better insight, the dynamic instruction percentage reported is the fraction over the entire application, which includes dynamic instances of instructions we do not study (Section 2.4).

The graphs show that, on average, between 34% (worst case) to 40% (best case) of the static instructions studied have 32 bits of continuous register chunks that can be candidates for approximation (assuming a technique can exploit approximations at that granularity). These static instructions account for 27% (worst case) and 36% (best case) of dynamic instructions respectively. Of the applications studied, Swaptions shows the most potential for approximation. This is commensurate with its high SDC-Good and overall low OC error sites.

Another insight gained from this experiment is that even applications that do not have full instructions that are approximable, may contain pockets of smaller register chunks (partial instructions) that are tolerable to errors. Hence, techniques that can exploit approximation at these finer granularities can conceivably achieve big gains and unlock the hidden potential in many new applications traditionally not considered as candidates for approximate computing. For example, in the best case, while only 4% of the static instructions (producing 3% dynamic instructions) in Blackscholes are marked as candidates for (full instruction) approximation by Approxilyzer, this number goes up to 29% (31% dynamic instructions) when considering individual 32b register chunks. While in this work we only consider static instructions for approximation, such analysis can also be carried out along the dimension of individual dynamic instructions to further understand the application's approximation potential.

In summary, Approxilyzer can be used to understand the best and worst case bounds on the approximation potential of an application even without a clear quality threshold. Such analysis can unlock hidden approximation potential that can then be targeted by specialized techniques.

2.6 EXTENDING ERROR ANALYSIS TO OTHER INSTRUCTION SET ARCHITECTURES

Approxilyzer's unique features makes it a useful tool that can enable new avenues of research, but limitations in its current implementation hinder its usability. The tool relies on Wind River Simics [71], a proprietary full-system simulator, is also designed to handle only applications compiled for the SPARC instruction set architecture (ISA). The restrictions imposed from both the simulator and ISA make a wide adoption of the tool challenging.

To that end, we developed gem5-Approxilyzer [45], an open-source[5] implementation of Approxilyzer that enables support for more ISAs, beginning with x86 in this work. We build the foundations of our new tool using the open-source gem5 simulator [73] which facilitates (with relative ease) the future inclusion of more ISAs into the tool. Building gem5-Approxilyzer required significant engineering effort to support x86 error analysis on gem5. Approxilyzer's

[5]https://github.com/rsimgroup/gem5-Approxilyzer

original implementation built for SPARC (a reduced instruction set computing architecture, or RISC architecture) assumes constant register size and instruction encoding length, which is not the case for x86 (a complex instruction set computing architecture, or CISC architecture). This additional tool shows Approxilyzer's effectiveness/accuracy with a different ISA, namely x86. Further, we show that the error profiles of the same application can be rather different under different architectures, which in turn can require customized resiliency and approximation solutions. This result further motivates the need for open-source tools such as gem5-Approxilyzer that can enable such comparisons and aid in building better solutions and exploring new avenues of research.

2.6.1 gem5-Approxilyzer: An Overview

gem5-Approxilyzer is an implementation of Approxilyzer [44] using the open-source gem5 simulator. Hence, its interface, high-level design and techniques are the same as those described in the previous sections and prior work [42, 44]. We describe here the implementation details and associated challenges of gem5-Approxilyzer.

Similar to Approxilyzer, we use a single-bit transient error model in architectural registers for gem5-Approxilyzer. We study errors in bits of both source and destination registers of instructions. For gem5-Approxilyzer, we undertake error injection in registers of x86 macro-instructions. Modern CISC implementations like x86 often implement the complex machine instructions (macro-instructions) using low-level instructions called micro-instructions or micro-operations. Micro-instructions are generally specific and proprietary to the micro-architecture and not faithfully recreated in publicly available simulators. Hence, we restrict our analysis to macro-instructions.

For this study, we only consider general-purpose registers and SSE[6] registers in x86. We do not inject errors in special-purpose, status, and control registers (e.g., %rsp, %rbp, rflags) to simplify our error model and reduce the number of error injections required for a first-order analysis. We assume that these always need protection and can be hardened in hardware (e.g., with ECC). We also do not inject in implicit[7] registers in this work, although gem5-Approxilyzer is extensible and can support these registers in the future.

[6]The binaries we study do not explicitly use floating point stack registers (st0-st7) in the region of interest and hence we do not study them.

[7]For example, the instruction *imul rbx* performs the following signed multiplication: $rdx : rax \leftarrow rax{*}rbx$. We only inject errors in rbx and not in rax and rdx.

2.6.2 Implementation Details

To execute gem5-Approxilyzer end-to-end, the user provides an application, its inputs, and associated quality metrics that evaluate the application output. The user can optionally mark the beginning and end of a code region of interest (ROI) – either by annotating the source or providing static PCs marking the beginning and end of the ROI – for analysis. In the absence of an ROI, the full application is analyzed.

gem5-Approxilyzer executes four phases to produce an application's error resiliency profile:

Phase 1 extracts static and dynamic properties of instructions executed within the ROI. An instruction parser module analyzes static instructions in the application's disassembly to identify registers used, determine if the instruction affects control flow (jumps, conditional branches, function calls, etc.), and identify any registers that contain memory addresses (these are marked for address-bound pruning). Information from this static pass is used to build the def-use chains that are used by pruning techniques in Phase 2. Next, gem5 is used to produce a full dynamic execution trace of user-mode instructions and memory accesses. From this trace, only the (dynamic) instructions that are found within the static disassembly, along with their corresponding memory accesses, are extracted for analysis; gem5-Approxilyzer does not analyze external library code, system code, or calls to them. Further, it simplifies the trace to only contain the execution within the ROI (if an ROI is provided).

Phase 2 prunes error sites as mentioned in Section 2.2 by applying control- and store-equivalence as well as address-bound and def-use techniques. gem5-Approxilyzer processes the execution trace from Phase 1 to build control-equivalence classes and def-use chains. The memory accesses recorded in the trace are used to build store equivalence classes and perform address-bound pruning. At the end of this phase, gem5-Approxilyzer picks a pilot for each equivalence class and creates the set of error sites for error injections.

Phase 3 performs the error-injection experiments using our *error injector* module built for gem5. The error injector takes as input a string with the error-site description: dynamic instruction described using the cycle number of the simulation, register information (register name and whether it is used as a source/destination operand) and register bit number.

The error injector pauses the simulation at the specified dynamic instruction and flips the bit in the register. For source registers, the bit flip is performed before the instruction execution. For destination registers, the error is simulated by performing the bit flip after the instruction execution (otherwise the error would be overwritten by the instruction execution). The simulation then proceeds, checking for any hangs and crashes, or other symptoms to identify detected outcomes. If no detected symptoms are encountered before the simulation ends, gem5-Approxilyzer compares the generated output with the error-free execution's output to identify any OC. If there is an OC, gem5-Approxilyzer uses the user-provided quality metric to evaluate the output quality.gem5-Approxilyzer records each pilot injection and its outcome for post-processing in the next phase.

Phase 4 analyzes the outcome of each error injection and assigns it the appropriate error outcome, i.e., error outcome category and quality degradation (QD) score for OCs. gem5-Approxilyzer then assigns the same error outcomes to pruned error sites associated with the pilot, and finally outputs the application's comprehensive error profile containing all the error sites and their corresponding error outcomes.

For an end-to-end error analysis with gem5-Approxilyzer, the error injections in Phase 3 consume the most time – several days worth of CPU time versus only few minutes/hours consumed by all the other phases combined for the experiments reported here. Thus, using effective pruning techniques that can reduce the total number of error injections in Phase 3 is the most direct means of reducing the tool's analysis time.

x86 Implementation Challenges: While Phases 3 and 4 are largely ISA independent, Phases 1 and 2 in gem5-Approxilyzer require customization to support different ISAs. Since x86 is a CISC ISA, opcode lengths vary, and hence the instruction parser in Phase 1 must capture instruction semantics correctly to identify source and destination operands of different instructions. Depending on the complexity of the macro-instructions, a varying number of micro-instructions can be generated. Any memory accesses performed by these micro-instructions in the gem5 memory trace must be mapped to the correct macro-instruction. Since x86 allows for variable register sizes, another challenge in Phase 2 is to correctly associate registers of varying sizes with their aliased 64-bit registers. This must be done carefully to identify aliased def-use pairs which enables pruning the right set of error sites within an aliased register. For example, %ax and %eax both alias to %rax. While performing def-use pruning, only the lower 16 bits of %eax definition must be pruned if the first use is %ax.

Extensions to Other ISAs: We designed gem5-Approxilyzer to be reasonably modular (e.g., each phase is a separate module) to enable future extensions to support different ISAs, error models, and pruning techniques. Here, we briefly elaborates on some details for future extensions.

The gem5 simulator currently supports many ISAs, and gem5-Approxilyzer could support them with the following modifications:

1. The instruction parser in Phase 1 must be modified to capture the semantics of the new ISA. This modification ensures that instructions near branch boundaries belong to the correct control equivalence class.

2. ISA-specific behaviors that affect control flow (e.g., branch delay slots for SPARC) should be incorporated into the control-equivalence algorithm accordingly. If registers alias to one another, then def-use pairs must carefully track the currently used register bits within the larger register.

3. Register aliasing must be captured correctly to track def-use pairs.

The error-injector module in Phase 3 can be modified to support other error models such as multi-bit injections or injections to other system structures like dynamic random-access memory (DRAM). The error-pruning module in Phase 2 would need to be extended to support pruning algorithms appropriate for the chosen error model.

gem5-Approxilyzer performs Phase 2 analysis on the dynamic trace generated by gem5 in Phase 1. For very long executions, this may result in excessively long traces, requiring a tighter coupling of Phases 1 and 2 to trace and analyze parts of the execution at a time.

2.6.3 Evaluation

We implement gem5-Approxilyzer using gem5 [73] to simulate an Ubuntu-16.04 system, and we use GCC 7.3 with -O3 optimization to compile the applications. We validate gem5-Approxilyzer in a similar manner to Approxilyzer. Specifically, for SDC-Maybe, we equalize error cites based on if their output quality degradations (QD) are above or below user specified output quality thresholds (QT). In the absence of a quality threshold, prediction accuracy measurements for any SDC-Maybe pilot with a quality degradation of, say, Q measures the number of its population members that also result in SDC-Maybe with the same quality degradation Q. Table 2.3 lists the applications explored as well as their inputs for gem5-Approxilyzer evaluation.

For each application, the overall validation accuracy for a given equivalence based pruning technique (control, store, or combined = control+store) is obtained by calculating the average of the pilot prediction accuracy across a random sample of equivalence classes built using that technique (control, store, or combined), weighted by the size of the equivalence class.

We randomly pick 750 equivalence classes to validate each of the control- and store-equivalence techniques (1500 equivalence classes for combined control+store). This gives us a 99% confidence interval with a 5% error margin [84]. For each equivalence class we set the population size to 750. If the equivalence class size is less than 750 (error sites), then the pilot is validated using all the remaining error sites in the equivalence class. This again gives

Table 2.3: Benchmarks, inputs, and error-site pruning by technique (C: Control-Equivalence, S: Store-Equivalence, C+S+K: total pruning using control, store, and known-outcome techniques)

Application	**Input**	**Total Error Sites**	**Remaining Error Sites**	**Pruned Error Sites %**
Blackscholes [82]	21 options	232K	100K	C: 12.24 S: 9.45 C+S+K: 56.77
Swaptions [82]	1 simulation 1 option	10.3M	720K	C: 52.47 S: 7.85 C+S+K: 93.01
LU [80]	8x8 blocks 16x16 matrix	1.2M	268K	C: 23.49 S: 22.72 C+S+K: 77.91
FFT [80]	2^8 data points	4.4M	215K	C: 43.99 S: 21.50 C+S+K: 95.05
Sobel [83]	81x121 pixels	85.3M	300K	C: 62.74 S: 20.94 C+S+K: 99.65

us a 99% confidence interval with a 5% error margin [84]. In all, we perform approximately 1.6 million error-injection experiments to validate gem5-Approxilyzer.

Pruning Effectiveness: The number of error sites pruned is directly related to the reduction in the number of error injection experiments needed to analyze the application. For each application, we measure first the number of error sites in the application's region of interest and then the number of error sites remaining after the pruning phase to calculate the number of error sites that have been pruned by various heuristics. This metric evaluates the tool's effectiveness since the number of error sites pruned directly reduces the number of error-injection experiments needed to analyze the application. For the control heuristic, we set depth to N=50, as in prior work [43].

The last column of Table 2.3 shows the percentage of error sites pruned by gem5-Approxilyzer using the control-equivalence (C), store-equivalence (S), and known-outcome (K) pruning techniques. At 56.77%, Blackscholes has the smallest total (C+S+K) pruning. Blackscholes is a small application, which coupled with our choice of a small input leads to a very small execution footprint (as can be seen by the small number of total error sites). This translates to few dynamic instructions per static PC which leads to very small equivalence classes. The average size of the equivalence class in Blackscholes is just 1.96. Since the amount of pruning is directly proportional to the size of the equivalence class, it is not surprising that the pruning effectiveness for Blackscholes is limited. The maximum pruning is achieved in Sobel, at 99.65%. Apart from Blackscholes, all the other applications see a one to two orders of magnitude reduction in the number of error injections needed to comprehensively analyze them. Thus we show that these pruning techniques are also effective for x86.

Validation Results: Figures 2.9(a), 2.9(b), and 2.9(c) show the validation accuracy for the

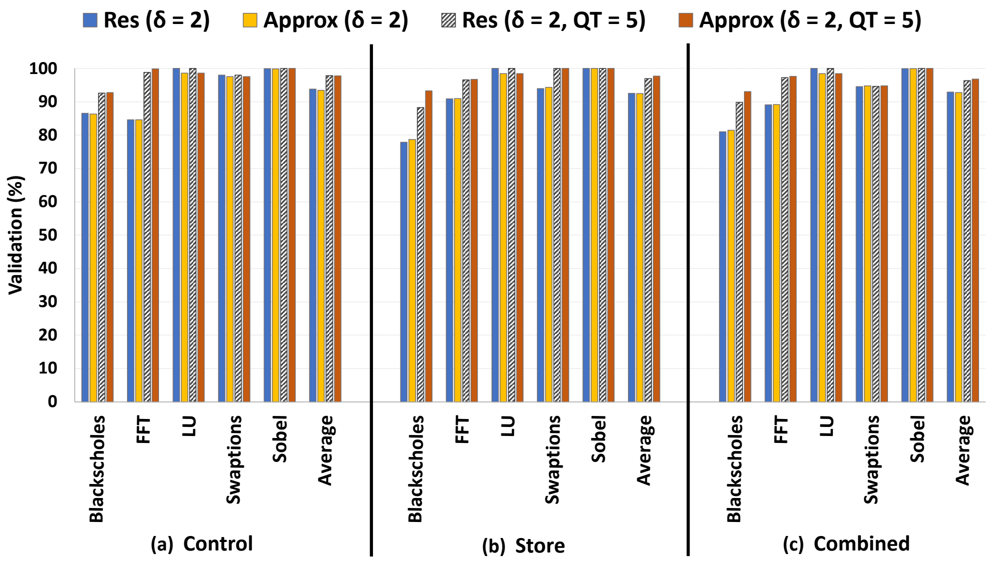


Figure 2.9: gem5-Approxilyzer validation for (a) control equivalence, (b) store equivalence, and (c) combined (control + store) equivalence.

control equivalence, store equivalence and their combination respectively. On average, both control and store equivalence techniques show high accuracy ($> 92\%$) for Res and Approx with a flexible quality parameter $\delta = 2$ that uses 2% for Blackscholes, FFT, LU, and Sobel, while for the financial applications, Blackscholes and Swaptions, uses the absolute difference in the dollar value set to \$0.01 (difference of 1 cent or less). In brief, gem5-Approxilyzer is able to correctly predict the output quality of the x86 application error sites with very fine granularity (2% or within a single cent). Swaptions ($> 94\%$), LU ($> 98.5\%$), and Sobel ($> 99.9\%$) show very high validation accuracy across the board. While still relatively high, Blackscholes shows the poorest validation accuracy for both control ($Res_{\delta=2} = 87\%$, $Approx_{\delta=2} = 86\%$) and store ($Res_{\delta=2} = 78\%$, $Approx_{\delta=2} = 79\%$). As mentioned before, Blackscholes has small equivalence classes which can lead to poor prediction accuracy even if a single error site is predicted incorrectly.

We observe that the pilots that have low prediction accuracy in Blackscholes and FFT (and a few in Swaptions) predominantly belong to two categories: (a) pilots are SDC-Maybe and the populations also produce SDC-Maybe but with quality degradations that have a wider range than allowed by the δ and (b) pilots of equivalence classes that have a mix of outcomes at the border of either SDC-Bad and DDC or SDC-Maybe and SDC-Bad. More sophisticated heuristics that combine control and data flow might capture specific patterns in these applications more accurately and we leave their exploration to future work. Across all applications, we observe that pilots with Masked, SDC-Good, and Detected outcomes show almost perfect ($> 99.9\%$) validation accuracy.

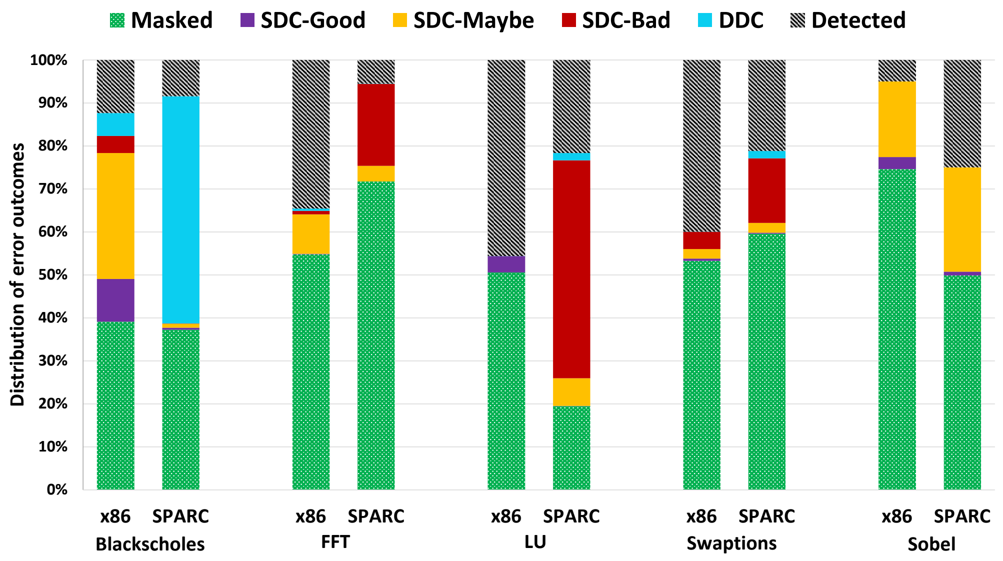


Figure 2.10: Distribution of error outcome categories for the applications studied using the x86 and SPARC ISAs.

Both Blackscholes and FFT show an improvement ($> 90\%$) when a user quality threshold is applied. For brevity we show results for QT=5, but we performed this experiment with a range of different QT values and observed a similarly high validation accuracy. This implies that even for pilots that fail to predict the quality at a fine granularity, the grouping of the equivalence classes is sufficiently accurate to be used in many realistic use cases. On average, we see that when a quality threshold is supplied, the validation accuracy is $> 97\%$ for both store and control heuristics (and hence their combination).

Hence, we show that the techniques used by gem5-Approxilyzer are very accurate in characterizing the error profiles for x86 applications.

2.6.4 Error Profiles for Different ISAs

We use gem5-Approxilyzer to perform an error analysis on our workloads compiled to x86 binary. We also analyze the same workloads compiled to a SPARC binary with of Approxilyzer using SIMICS [71], which allows us to perform an initial comparison of the resiliency and approximation characteristics of the same workloads for two different ISAs.

Figure 2.10 compares the distribution of error outcomes (for all the error sites) in each application for the x86 and SPARC ISAs. The error outcome profiles of the same application look rather different for the different ISAs. We note, however, that some differences are expected due to the CISC vs. RISC nature of the instructions as well as the fact that x86 uses many more implicit registers (that we do not inject into) compared to SPARC. The

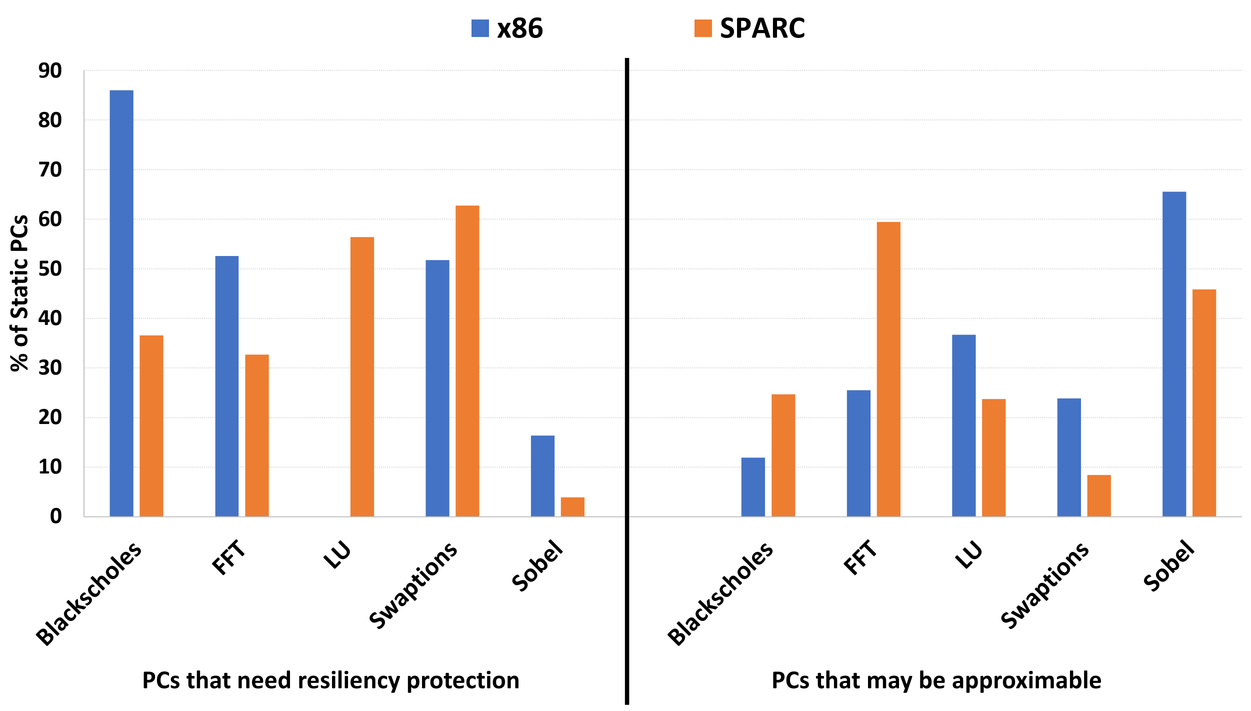


Figure 2.11: Percentage of static PCs in the application that need resiliency protection and percentage of PCs that are approximable across x86 and SPARC. We use the same QT across both ISAs: 5% for Blackscholes, Sobel, FFT, and LU; $0.001 for Blackscholes and Swaptions.

graph shows that SPARC has a higher percentage of more egregious outcomes. For example, while Blackscholes-x86 has many error sites that lead to SDC-Good, SDC-Maybe, and SDC-Bad outcomes, the error outcomes in Blackscholes-SPARC produce such bad quality output that they become DDCs. We leave a deeper analysis of the causes for these differences to future work.

Figure 2.11 further shows the percentage of static instructions that need resiliency protection and those that are approximable for the same QT across the two ISAs. The wide differences across the two ISAs and the lack of a clear trend further underscore the importance of resiliency analysis tools like gem5-Approxilyzer that can analyze applications at the binary level to devise customized resiliency and approximation solutions for different architectures. Source-code or IR-level error analysis may not lead to the most optimized solutions for different architectures.

2.7 SUMMARY

We present a systematic framework, Approxilyzer, for principled and general purpose instruction-level approximate computing and show its application to hardware resiliency. Approxilyzer uses a new scheme to classify error outcomes into categories based on approximation potential. This categorization is based on an end-to-end quality metric that is application-specific. We perform an extensive validation to show that Approxilyzer is able to predict the impact of an instruction-level error on output quality with high accuracy (average of 95% accuracy for fine-grained quality classification observed over 2.6 million error injections), for all dynamic instructions in a program execution.

Approxilyzer also presents a mechanism to quantitatively tune output quality, resiliency, and overhead to the user's target goals for the error model assumed. Furthermore, for general error models, Approxilyzer automatically identifies candidate instructions for approximation with no programmer burden (except for information on the quality metric), enabling a more focused analysis for the general error model by other tools or the user.

The extension to gem5-Approxilyzer, an open-source re-implementation of Approxilyzer for the gem5 simulation environment, enables support for multiple ISAs in the future within an open-source infrastructure. We start by supporting x86 in this work. We show that gem5-Approxilyzer is both effective and highly accurate in predicting the program's final output quality in the presence of soft errors in the execution. To additionally motivate the need for such tools, we perform a comparison of our workloads across two ISAs, by generating the error profiles for both x86 and SPARC. The differences in the error profiles for the same applications across ISAs further underscore the need for a tool like gem5-Approxilyzer.

Chapter 3: Scalable General Purpose Hardware Error Analysis

3.1 MOTIVATION

With principled error analysis techniques, the space of possible errors can shrink by orders of magnitude, as exemplified by Relyzer [42] and Approxilyzer [44]. Despite the reduction in the total number of error sites requiring exploration to make it a tractable problem, actual runtimes can still be high. For example, with Approxilyzer, we found that it took multiple days to complete the error analysis of an application – for a *single* input. Thus, it is important to accelerate this general-purpose analysis in order for it to be used across potentially even larger applications, as well as extending it to multiple inputs for a more comprehensive resiliency analysis.

To effectively scale up such principled error analysis techniques, we leverage insight from the domain of software testing. The software development workflow has, for many decades, dealt with the problem of vetting code to be bug-free before release, in spite of the challenges associated with uncovering elusive bugs. In the scope of hardware resiliency, this can be considered analogous to the detection of silent data corruptions (SDC), where the (challenging) process of analysis for soft errors also typically occurs before program deployment. Identifying this key bridge between software testing for software bugs and hardware reliability analysis for hardware errors is one of the key contributions of this thesis. By identifying and adapting key concepts from the software testing domain, hardware reliability analysis can scale up while maintaining the principled and comprehensive analysis enabled by tools such as Approxilyzer.

This chapter presents Minotaur [46], a toolkit that improves the speed of resiliency analyses while also precisely identifying more SDC-causing instructions (or, equivalently, the *program counters* for general purpose applications) referred to henceforth as *SDC-PCs*. Minotaur bridges between software testing and hardware reliability by adapting four software testing techniques to make hardware error analysis faster and thus more scalable. As a result, we show that Minotaur improves the runtime of Approxilyzer by an order of magnitude on average (10.3×) while *simultaneously* improving the accuracy of Approxilyzer in identifying hardware errors.

We identify, adapt, and evaluate the following four concepts which bridge between software testing and resiliency analysis:

Concept 1: Test-Case Quality $\rightarrow$ Input Quality

Concept 2: Test-Case Minimization $\rightarrow$ Input Minimization

Concept 3: Test-Case Prioritization $\rightarrow$ Error-Injection Prioritization

Concept 4: Test-Case Prioritization $\rightarrow$ Input Prioritization

Minotaur shows, for the first time, that leveraging software testing concepts for resiliency analysis enables principled and significant benefits in speed and accuracy. While our evaluation uses Approxliyzer as the underlying resiliency analysis, Minotaur and its concepts apply more generally. For example, Concepts 1 and 2 can be applied to speed up any dynamic resiliency analyses that typically study large inputs, by producing a smaller representative input for analysis. Error-injection analyses can greatly benefit from Concept 3, by prioritizing error-injections and employing early termination for SDC-PCs. Concept 4 can propel resiliency analyses to explore multiple inputs, a new direction which previously was daunting due to speed and accuracy concerns of existing techniques. Minotaur provides a foundation for a systematic methodology for efficient resiliency analysis based on software testing, and opens up many avenues for further research.

3.2 BACKGROUND: SOFTWARE TESTING TECHNIQUES

Software testing is the process of executing a program or system with the intent of finding failures [85]. The objective of testing can be quality assurance, verification, validation, or reliability estimation. We discuss some techniques and best practices adopted by the software testing community and associated definitions.

3.2.1 Test-Case Quality

In software testing, a *test case* is an input and an expected output used to determine whether the system under test satisfies some software testing objective. A *test set* is a collection of one or more test cases. The number of all test cases can be intractably large. Thus, selecting appropriate test cases has a significant impact on testing cost and effectiveness. Test cases are selected by evaluating them using quality criteria relevant to the testing objectives.

Selecting a quality criterion involves a tradeoff. A "stronger" criterion enables closer scrutiny of program behavior to find bugs, while a "weaker" criterion can be fulfilled faster using fewer test cases [86]. The choice of criterion depends on several factors, including the size of the program, cost requirements, and criticality and consequence of failure. Some popular quality criteria from the software testing literature [86], ordered from weaker to stronger, are: (1) statement coverage [87], which measures the fraction of program statements executed by tests; (2) branch coverage [85], which measures the fraction of branch edges executed; and (3) def-use coverage [86, 88], which measures the fraction of pairs of variable definitions and their corresponding uses executed. Despite being a weak criterion, statement coverage is typically used for testing commercial software due to its low resource overheads. Branch coverage is often used for safety-critical systems [89]. The software testing literature provides an extensive analysis of various testing criteria [87].

3.2.2 Test-Case Minimization

While running larger (or more) test cases is desirable for thorough testing, time and resources limit the size (or number) of test cases that can be executed. *Test-case minimization* is used to minimize the testing cost in terms of execution time [90, 91, 92, 93, 94, 95, 96]. The goal of test-case minimization is to generate a smaller test case that has similar or (ideally) the same quality as the original test case; e.g., covers the same statements.

3.2.3 Test-Case Prioritization

Resource constraints can sometimes make it infeasible to execute all planned test cases. It thus becomes necessary to prioritize and select test cases so that critical failures can surface sooner rather than later [90]. Test-case prioritization techniques schedule test cases in an order that allows the most important tests, by some measure, to execute first. For example, test-cases can be prioritized by their coverage. Many test-case prioritization techniques have been proposed in the literature [90].

3.3 MINOTAUR: ADAPTING SOFTWARE TESTING TECHNIQUES FOR HARDWARE ERRORS

Minotaur shows, for the first time, that leveraging software testing concepts for resiliency analysis enables principled and significant benefits in both speed and accuracy of such analysis. We identify, adapt, and evaluate four bridges between software testing and resiliency analysis described below. These concepts can benefit many resiliency analysis techniques; here we evaluate them by applying to the state-of-the-art Approxilyzer tool [44]. Figure 3.1 illustrates the complete system.

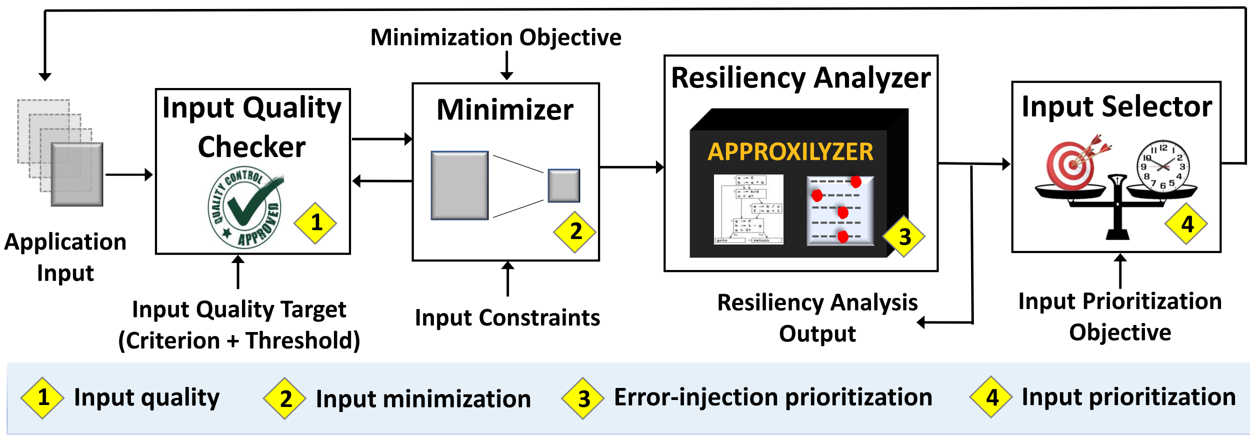


Figure 3.1: Overview of Minotaur. Approxilyzer may be replaced with another resiliency analyzer.

3.3.1 Input Quality

Evaluating and ensuring that "good" quality inputs are used for resiliency analysis increases the effectiveness of the analysis. We adapt the concept of test-case quality (Section 3.2.1) to build an *Input Quality Checker* (Box 1 in Figure 3.1) that measures the quality of the inputs used for resiliency analysis.

The software test quality criteria are typically expressed at the source-code level, to make it easier for developers to understand what is covered and what is not. There has also been some work on test coverage at the object-code level [97, 98], but it is not widely studied. Our resiliency analysis examines error models at the object code level and aims to find assembly instructions that are vulnerable to SDCs (SDC-PCs). Hence, it is desirable to measure the quality of the input used for resiliency analysis with quality criteria expressed at the object code.

Figure 3.2 demonstrates the difference between using input quality criteria at the source versus object code level. Suppose a ternary operator is used by the developer, such as in Line 1. Assuming a value of $True$ for the variable c, the statement coverage (Section 3.2.1) of the source code measures that this single input will cover (execute) 100% of the code. However, for the same code compiled to assembly, only 75% of the instructions are covered (executed). Analyzing resiliency with just this input does not provide full (100%) assembly instruction coverage, and an error in assembly instruction PC-4 would not be captured.

For resiliency analysis, we adapt three test (input) quality criteria to the object code level—statement, branch, and def-use coverage. The analog of statement coverage at the object code level measures the fraction of static assembly instructions (or *PCs*) executed by the input; we call it simply *PC coverage*. Branch and def-use coverage are analogously

```
// INPUT: c = True
// Source. 100% Statement Coverage
    1. v = c ? E1 : E2            // covered
// Assembly. 75% PC Coverage
       PC-1. beq c, $0, L2       # covered
   L1: PC-2. move v, E1          # covered
       PC-3. jump L3             # covered
   L2: PC-4. move v, E2          # not covered
   L3: …
```

Figure 3.2: Statement coverage vs. PC coverage.

adapted from the source to the object code level to consider assembly-level branches and def-uses pairs, respectively.

The Input Quality Checker evaluates whether a given input meets the desired *quality threshold* (e.g., 90%) for a specified *quality criterion* (e.g., PC coverage). We refer to the combination of the input quality criterion and the threshold as the *input quality target* (IQT).

3.3.2 Input Minimization

Minimizing the input size for an application can greatly speed up the resiliency analysis by reducing the time for each error-injection experiment and/or reducing the total number of error injections needed. Using insights from test-case minimization, we designed a systematic technique, a *Minimizer* (Box 2 in Figure 3.1), that Minotaur uses to generate a minimal input, *Min*, provided a reference input, *Ref*.

There is no general algorithm to minimize inputs across all application domains in software testing [87]. Our Minimizer algorithm is specialized for our workloads, but the underlying concepts are general and can be extended to other domains with appropriate modifications. Given a Ref, the goal of the Minimizer is to find a reduced input (Min) that minimizes a stated *minimization objective (MinObj)* (e.g., execution time) while satisfying an *input quality target* (e.g., 90% PC coverage relative to Ref). We chose a simple, greedy algorithm based on binary search for the Minimizer and found it effective. More sophisticated optimizers may find better Min inputs; we leave such an exploration to future work.

In addition to the minimization objective and input quality target, the Minimizer is provided with the list of input parameters (e.g., command line and other program-specific parameters) and a set of parameter constraints (e.g, range or boundary conditions) to ensure that the Min generated is both legal and realistic. A realistic Min enables the resiliency analysis to uncover SDC-PCs that are vulnerable for realistic conditions, avoiding over- or under-protection. Domain knowledge enables understanding the realistic range of input values and how to change them (e.g., choosing image shrinking instead of sub-sampling pixels or subsetting image inputs [99]) to achieve realistic inputs.

Algorithm 3.1: Input Minimization Pseudocode

```
$PList$: Parameter List, $C$: Constraints,
$IQT$: Input Quality Target, $MinObj$: Minimization Objective,
$PList_{Ref}$: Reference input's $PList$
Function Minimizer($PList_{Ref}$, $C$, $IQT$, $MinObj$):
    $PList \leftarrow OrderParams(PList_{Ref}, MinObj)$
    for $param \in PList$ do
        $lower \leftarrow$ Minimum value of $param$ provided $C$
        $upper \leftarrow$ Reference value of $param$
        $PList[param] \leftarrow BinarySearch(lower, upper, C, IQT)$
    end
    return $PList$
Function OrderParams($PList$, $MinObj$):
    return Ordered parameters of $PList$ with respect to $MinObj$
Function BinarySearch($lower$, $upper$, $C$, $IQT$):
    Search values between $lower$ and $upper$ provided $C$,
    checking if the candidate value satisfies $IQT$
    return minimum value that satisfies $IQT$
```

Algorithm 3.1 shows the pseudo-code of Minotaur's Minimizer. It first performs a pre-processing pass over the reference input's parameter list and orders the parameters according to their estimated impact on the minimization objective. Our current implementation determines this order by running the program with a few different values for each input parameter and measuring the impact on the minimization objective. This step can be accelerated with additional domain knowledge from the user or automated using more sophisticated optimizers.

Given the ordered parameter list, the Minimizer uses binary search to progressively change each input parameter (one with highest impact on the minimization objective first) while ensuring that the new input value meets the input quality target. Lines 6–10 of Algorithm 3.1 show this search for applications with (1) numeric inputs and (2) where reducing the value of input parameters reduces (or does not affect) the minimization objective. All applications we study (except Sobel, which takes as input an image) satisfy both characteristics, with binary search sufficing for the value exploration. We reduce images for Sobel using the *resize* utility in the ImageMagick suite [100], which accepts a numerical argument, adapting the binary search to adjust this argument. Similarly, other application domains could also require appropriate adaptation of the algorithm. At the end of this process, the Minimizer outputs the final parameter list for the minimized input.

3.3.3 Error Injection Prioritization with Early Termination

We next use insights from test-case prioritization to improve resiliency analysis for any input (minimized or not). We evaluate *error-injection prioritizations* that order error injections for a PC such that error sites which are more likely to be SDC-causing are examined earlier. Once an injection reveals an SDC, Minotaur does not perform injections for any other error sites for that PC. Hence, error-injection prioritization can lead to *early termination* of error-injection campaigns, leading to significant savings in error injections. Box 3 of Figure 3.1 shows the application of error-injection prioritization in Minotaur's workflow.

We study the following ordering schemes for error-injection prioritization to understand which error sites result in SDCs:

- **Bit position of registers (BitPos)**: Injecting into specific bits first (such as the MSB or LSB).
- **Dynamic instance of error site (DI)**: Error sites from an earlier dynamic instance may be more prone to SDCs than later dynamic instances.
- **Register type (RT) – integer vs. floating point**: Certain register types could be more susceptible to SDCs than others.
- **Operand kind (OP) – source vs. destination**: Prioritizing source vs. destination register may also show a pattern for SDC-causing instructions.
- **Equivalence class size (ECS)**: This ordering is specific to Approxilyzer and prioritizes injections in error sites of largest equivalence classes first, which is the default ordering used by Approxilyzer to maximize the number of error sites with predicted outcome for a given number of total error injections.
- **Random ordering**: Error sites are chosen at random.

3.3.4 Input Prioritization

Mission-critical applications with high resiliency requirements must undergo analysis using multiple inputs to build confidence that most SDC-PCs in the application have been identified. To that end, a naïve, but prohibitively expensive, scheme could analyze many inputs in their entirety to find all SDC-PCs in an application. Instead, we adapt test-case prioritization from software testing in the form of *input prioritization* to speed up resiliency analysis for multiple inputs.

In our scheme, an *Input Selector* (Box 4 in Figure 3.1) chooses inputs for resiliency analysis according to an order specified by an *input prioritization objective*. We choose to analyze the input with the shortest execution time, prioritizing faster analyses first (e.g., we choose Min before Ref). Input prioritization can lead to faster resiliency analysis speed for each subsequent input because the PCs already identified as SDC-PCs by prior inputs need not be (re)analyzed. Thus, we can leverage input-prioritization to find many of the SDC-PCs from one (faster) input, and carry this information onto another (slower but larger) input to avoid unnecessary error injections. Minotaur's Input Selector can successively select inputs for resiliency analysis until it meets an analysis target (e.g., a coverage or resource target).

3.4 EVALUATION METHODOLOGY

Our error-injection infrastructure builds on Approxilyzer, based on simulation using Wind River Simics [71] and GEMS [81] running OpenSolaris. Our workloads are compiled to the SPARC V9 ISA with all optimizations enabled. Approxilyzer's error model uses single-bit architecture-level errors (Section 2.4), which are a limited but effective [101] and realistic subset of hardware errors [102]. With resiliency becoming a first-class software design objective [103], techniques with different speed, precision, and error models are needed at different stages of software development. Evaluating Minotaur with tools that use different error models (lower-level, multi-bit, etc.) is part of our future work.

To evaluate Minotaur, we use seven workloads from three benchmark suites [79, 80, 83] spanning multiple application domains, summarized in Table 3.1. Column 3 lists the reference (Ref) input parameters used in our study. For five of the benchmarks—Blackscholes, Swaptions, LU, Water, and FFT—we use the same inputs as Approxilyzer [44] for the reference inputs. For Streamcluster, prior evaluations [78, 104] showed that the benchmark benefits from realistic datasets (as opposed to data points generated internally by the application); hence, we use a dataset from the UCI Machine-Learning Repository [105, 106, 107] as its Ref input. For Sobel, we use the bird image from the iACT [108] repository as input. We chose relatively small Ref inputs for almost all applications to be conservative and not over-estimate the benefits of input minimization. To evaluate the quality of the outputs, we use the same metrics as Approxilyzer [44] for Blackscholes, Swaptions, LU, Water, and FFT; for Streamcluster and Sobel, we use maximum relative error (*max-rel-err* from Approxilyzer [44]).

Evaluating Minotaur using the above workloads involved performing over 8.4 million error-

Application	Domain	Ref Input	Min Input	PC (%)	Branch (%)	Def-Use (%)
Blackscholes	Financial Modeling	64K options	21 options	100	100	99.38
Swaptions		16 options 5000 simulations	1 option 1 simulation	99.91	99.23	98.42
Streamcluster	Data Mining	centers = [10,20] num iterations = 3	centers = [4,5] num iterations = 1	99.97	99.77	98.67
LU	Scientific Computing	512x512 matrix 16x16 block size	16x16 matrix 8x8 block size	100	100	95.56
Water		512 molecules	216 molecules	99.89	99.36	99.85
FFT	Signal Processing	2^{20} data points	2^{8} data points	100	100	99.59
Sobel	Image Processing	100% image size (321x481 pixels)	25.25% image size (81x121 pixels)	100	100	100

Table 3.1: Applications studied and key input parameters (the ones that changed during minimization) for Ref and Min. The last three columns show the coverage of Min relative to Ref for different input quality criteria.

injection experiments spanning approximately seven weeks of simulation time on a 200-node cluster of 2.4GHz Intel Xeon processors.

3.4.1 Input-Quality Criteria

Since no available tool can easily measure test coverage at the object-code level, we developed our own tools using dynamic traces from Simics [71] for PC, branch, and def-use coverage for the object code. For PC coverage, we simply track the PCs executed by the input. For branch coverage, we store the unique branch-target PC pairs that represent control edges exercised by the input. For def-use coverage, we analyze the definition and use of operand registers exercised by the input, and store unique PC pairs that represent a def-use edge. For all criteria, we measure Min's coverage relative to Ref.

3.4.2 Input Minimization

Minotaur uses application run time as the minimization objective and targets 100% PC coverage (relative to Ref) as the input quality target when possible. We measure PC, branch, and def-use coverage for each Min *relative* to its corresponding Ref; e.g., if Min executes all PCs executed by its Ref, we consider it to have 100% PC coverage. Similarly, if Min exercises all branch-target and def-use pairs exercised by Ref, we consider it to have 100% branch and def-use coverage, respectively.

We choose PC coverage as our quality criterion because it is simple and fast to compute and it is the analog of the widely used statement coverage criterion for software testing (Section 3.2.1). We find that the Min inputs generated using PC coverage are surprisingly effective, and also exhibit high (but not perfect) branch and def-use coverage.

3.4.3 Accuracy Analysis

Minotaur uses input minimization to generate a Min that is a good representative of a Ref. We quantify Minotaur's accuracy for a given input as the fraction of SDC-PCs found by the input (either Min or Ref) relative to the total number of SDC-PCs found by the union of both inputs.

To understand the sources of inaccuracy, we analyze the SDC-PCs identified by Min and Ref by grouping them into categories based on whether they were found by Ref, Min, or both. We further distinguish the cases where certain PCs are explored (i.e., analyzed for resiliency) by one input but not both inputs. The difference occurs when the targeted error-site coverage is less than 100% and Minotaur chooses different PCs to meet that coverage for the two different inputs. We use the term *explore* to convey that at least one error site for a PC was analyzed (for a given input) by Minotaur. If no error site for a PC was analyzed (for a given input), we say that the PC was *not explored* by the input. Note that *not explored* does not mean *not executed* by the input; it simply means that the PCs were not analyzed for resiliency.

We group the SDC-PCs into five categories:

1. **Common**: Both Min and Ref classify the PCs as SDC, which are considered accurately classified by both.

2. **MinSDC**: Min classifies these as SDC-PCs and Ref explores them but does not classify them as SDC-PCs. Although Ref did not find these SDC-PCs, they are still candidates for hardening because they were found by a realistic Min input. Hence, these PCs are considered accurately classified by Min, but not by Ref.

3. **MinSDC+**: Min classifies these as SDC-PCs and Ref does not explore them. For similar reasons as MinSDC, this category is also considered accurately classified by Min, but not by Ref.

4. **RefSDC**: Ref classifies these as SDC-PCs and Min explores them but does not classify them as SDC-PCs. These PCs are inaccurately classified by Min because relying only on Min's analysis would leave these PCs unprotected.

5. **RefSDC+**: Ref classifies these as SDC-PCs and Min does not explore them. This category is also considered inaccurately classified by Min.

The error-injection prioritization scheme (Section 3.4.4) does not affect accuracy because it finds the same set of SDC-PCs for an input as without the optimization, albeit faster. Employing the input-prioritization scheme for all inputs (Section 3.4.5) will result in 100% accuracy since input-prioritization obtains the union of SDC-PCs found by analyzing all inputs (while optimizing resiliency analysis speed).

3.4.4 Error-Injection Prioritization

We explore 38 different error-injection prioritizations using combinations of the schemes from Section 3.3.3. For BitPos, DI, and ECS schemes, we test both ascending (A) and descending (D) ordering. We also explore compositional schemes. For example, BitPos_A + ECS_D first orders error injections by bit positions in ascending order (i.e., starting with the LSB), followed by ordering in descending equivalence class size. For RT and OP schemes, we simply pick the type/kind of register (e.g., OP_{Src} or OP_{Dest}) to prioritize.

To understand the bounds on the error-injection prioritization gains, we also run an Oracle best and worst case. The best case assumes that the Oracle identifies an SDC-PC with a single injection. For the worst case, the Oracle picks (for each PC) all injections that are not SDC-causing before picking an SDC-causing injection, reducing the benefit of early termination.

3.4.5 Input Prioritization

Our Input Selector prioritizes (faster) Min over Ref. Section 3.5 shows that while Min exhibits high accuracy (Section 3.4.3), it misses a small number of SDC-PCs found only by Ref. To achieve 100% accuracy, resiliency analysis on Ref is run after resiliency analysis on Min completes, but *only* for PCs that Min did not find as SDCs (Section 3.3.4).

3.4.6 Runtime Analysis of Minotaur

We evaluate the time that Minotaur takes to perform resiliency analysis on a single input. The Input Quality Checker, Minimizer, and Input Selector (Boxes 1, 2, and 4 in Figure 3.1) take negligible time compared to the resiliency analysis (Approxilyzer) time (Box 3); therefore, we focus on the resiliency analysis component.

Ideally, the runtime performance would be measured directly by measuring all components of Approxilyzer and every error injection. However, this cannot be done precisely on a busy cluster which introduces variability between runs. We estimate the total runtime by measuring statistically sampled error injections and using formulas as follows.

The time for resiliency analysis for a given application and input (Ref or Min) depends on: (1) equivalence class generation time ($t_{equiv\_class\_gen}$) [42, 44], (2) total injections of each outcome category ($I_{masked}, I_{det}, I_{OC}$) for a target error site coverage, and (3) the average error-injection runtime of each outcome category ($t_{masked}, t_{det}, t_{OC}$). We measure the runtime for each category separately because it can be quite different; e.g., an OC error requires additional post-processing (compared to Masked) to quantify the error quality into Good/Maybe/Bad categories, while Detected outcomes involve simulator and OS overhead to report outcomes such as SegFaults.

We measure the runtime by sampling 1,000 error-injection experiments for each of masked, detected, and OC outcomes per application and input, excluding outliers in the top and bottom 2.5% of runs. The total samples correspond to a 99.8% confidence level with 5% error margin in timing measurements [109]. The time for resiliency analysis is calculated as:

$$TotalRuntime = t_{equiv\_class\_gen} + \textstyle\sum_n (I_n \times t_n) \tag{3.1}$$

where each outcome type $n \in \{masked, detected, OC\}$ is weighted by the number of injections with that outcome and average injection runtime for that outcome.

In practice, error injections (the second term of Equation 3.1) dominate the total runtime of resiliency analysis. Thus, even though $t_{equiv\_class\_gen}$ is much shorter for Min (order of minutes) compared to Ref (order of hours), it is negligible compared to the total time of injection experiments.

All runs for Ref and Min begin with a checkpoint at the start of the region of interest (ROI), generally provided by the benchmarks, to avoid simulator startup cost and application initialization overhead. We break down the measurements into two components: the application runtime only inside the ROI, and the remaining runtime from the end of the ROI to the injection outcome. The latter runtime includes simulation overheads, various file I/O, and analysis of the application output.

3.5 RESULTS

We evaluate Minotaur's impact on a resiliency analysis tool, Approxilyzer [44], by analyzing (1) the speedup and accuracy from a minimized input (Min) for resiliency analysis (Section 3.5.1); (2) the speedup from error-injection prioritization with early termination (Section 3.5.2); (3) the combined speedup from minimization and error-injection prioritization (Section 3.5.3); and (4) the speedup from applying input prioritization across multiple inputs (Section 3.5.4).

3.5.1 Input Minimization

Min Quality: Table 3.1 shows the Min generated by applying Algorithm 3.1 to each Ref, using PC coverage as the input quality criterion. Most applications show a large reduction of input parameter values in Min (Column 4), which translates to faster application runtimes relative to Ref.[1] Additionally, Min maintains very high PC coverage relative to Ref (Column 5), which translates to high accuracy in finding SDC-PCs.

Not all workloads achieve a significant application speedup with the input quality threshold set to 100%. Slightly reducing the threshold by less than a percent, however, results in substantially higher minimization for Swaptions, Streamcluster, and Water. We show that the PC coverage reduction does not impact Min's accuracy significantly, while allowing Minotaur to benefit from running the faster Min.

The last two columns of Table 3.1 show the branch and def-use coverage of the generated Min (relative to Ref) and are discussed further at the end of this section.

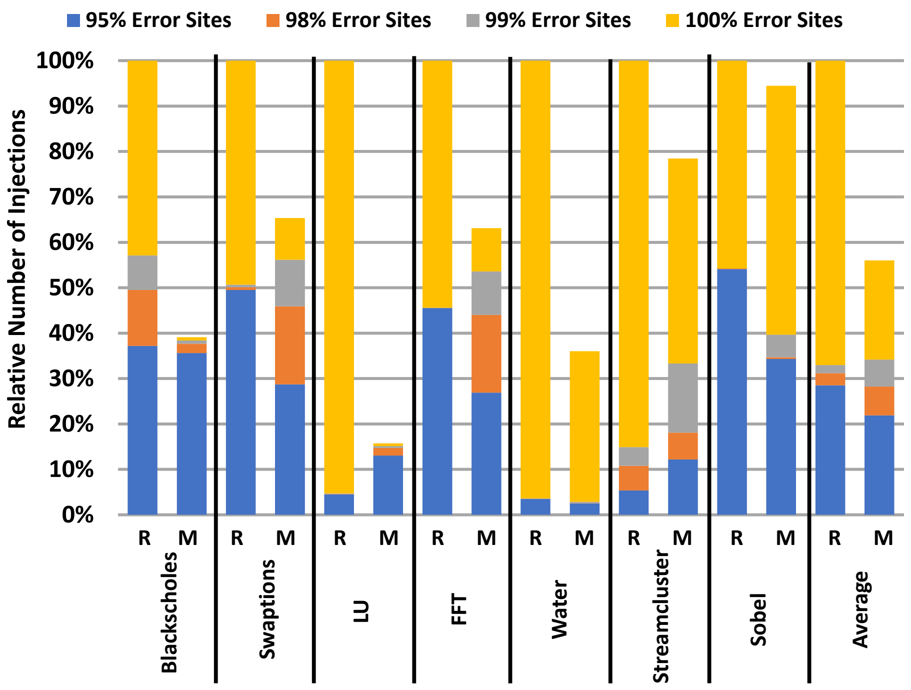


Figure 3.3: Number of error injections for different error-site coverage targets for each benchmark, relative to 100% error-site coverage for Ref (Ref100). R=Ref, M=Min.

[1]Many of our Ref inputs are themselves relatively small; higher benefits are likely with larger Ref inputs.

Minimization Speedup: Min typically runs faster than Ref because it has fewer dynamic instructions, resulting in fewer error injections and a shorter runtime per injection.

Figure 3.3 shows the total number of error injections needed for resiliency analysis for an application, relative to analyzing 100% of Ref's error sites (Ref100). Past studies found that targeting 100% error-site coverage was too expensive and so targeted just the top 99% of error sites (Ref99). By using input minimization, achieving 100% error-site coverage is no longer elusive for many applications. Figure 3.3 shows that for the Min inputs of Blackscholes, Swaptions, LU, and FFT, the number of error injections required for 100% error site coverage (Min100) is comparable to the number of error injections for Ref99 Thus, for these applications, it becomes tractable to run resiliency analysis with Min100. The other applications (Water, Streamcluster, and Sobel) also reduce the number of error injections from Ref100 to Min100, but the total number is still very large, presenting a trade-off between resiliency-analysis runtime and error-site coverage. We choose to favor runtime and use 99% error-site coverage for these applications. Henceforth, we use the umbrella term Min (unless otherwise stated) to encompass Min100 for Blackscholes, Swaptions, LU, and FFT, and Min99 for Water, Streamcluster, and Sobel. We use Ref to refer to Ref99 for all applications.

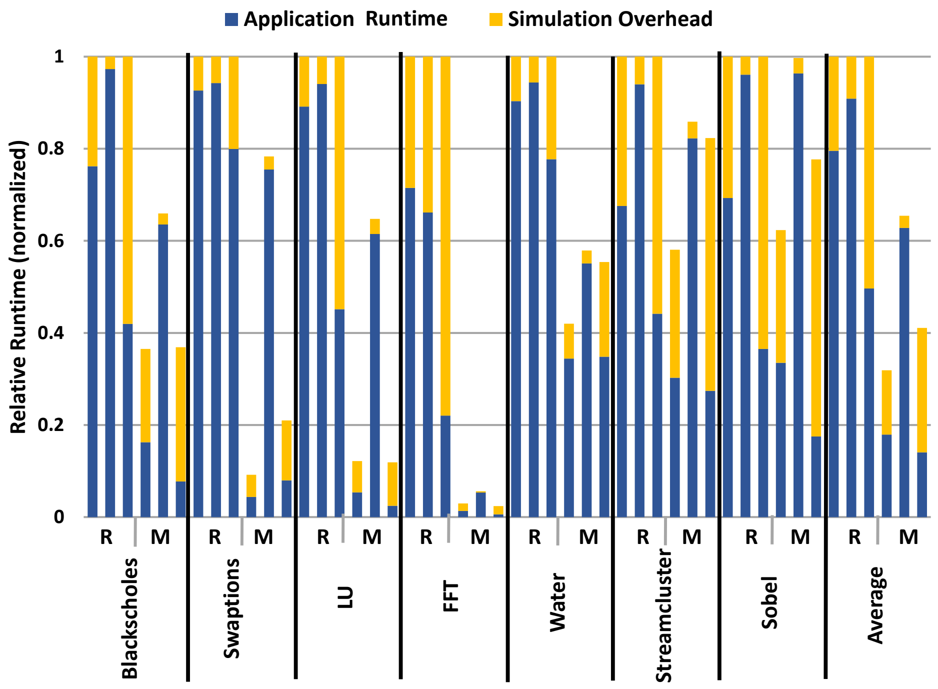


Figure 3.4: Average runtime per injection, normalized to Ref. Each set of three bars represents (from left to right) Masked, Detected, OC runtime (Section 3.4.6), divided into application runtime and simulation overhead. R = Ref and M = Min.

Not only does Min require fewer error injections for most of our workloads, each individual injection runs faster compared to Ref. Figure 3.4 shows the average runtime per injection for Ref and Min for different outcome types (Masked, Detected, and OC). Each bar is divided into the application runtime during the ROI (which begins after an application's initialization phase) and the simulation overhead (Section 3.4.6).

Min injections run 2.1× faster on average[2] than Ref for all outcome types for two primary reasons. First, the application runtime itself is faster (2.3× on average across outcome types) due to the smaller input. Second, for some applications, the I/O and other simulation environment overhead is significantly reduced for Min (1.8× on average). This is most notable for LU and FFT, where a large output matrix is generated for Ref but not for Min. The output matrix needs to be extracted for comparison and error classification (Figure 2.2). Min's smaller output matrices allow for faster post-processing, further speeding up the resiliency analysis relative to Ref for these applications.

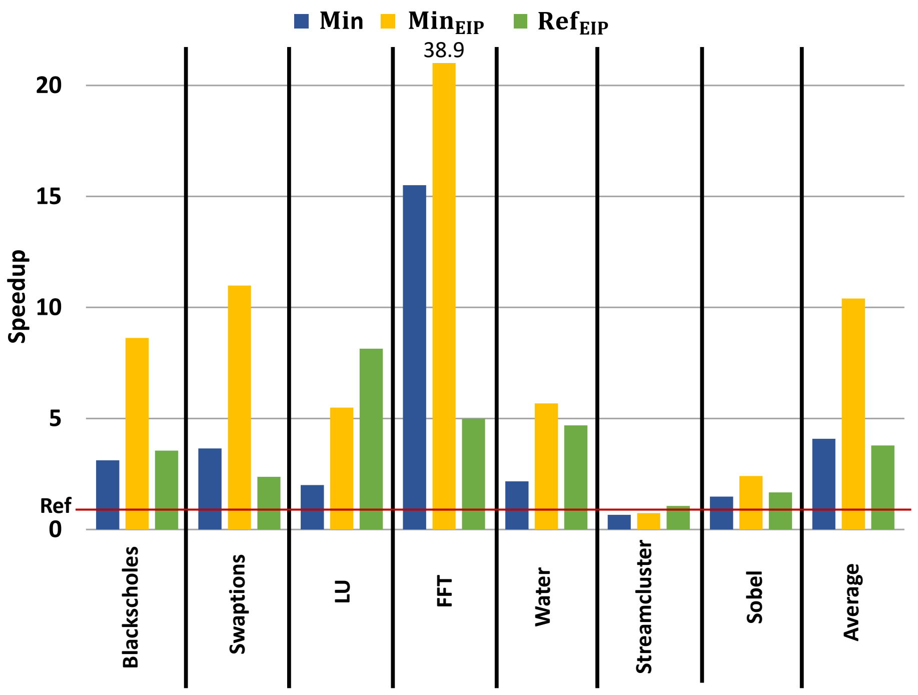


Figure 3.5: Min, Min$_{EIP}$, and Ref$_{EIP}$ speedup relative to Ref.

Figure 3.5 shows the total speedup obtained for Min (and the Minotaur optimizations discussed in the next sections). The first bar for each application shows the speedup from using a Min input relative to Ref. Overall, the combination of having fewer error sites and faster runtime per injection results in a 4.1× speedup for Min over Ref on average (up to

[2]All averages in this chapter refer to the arithmetic mean.

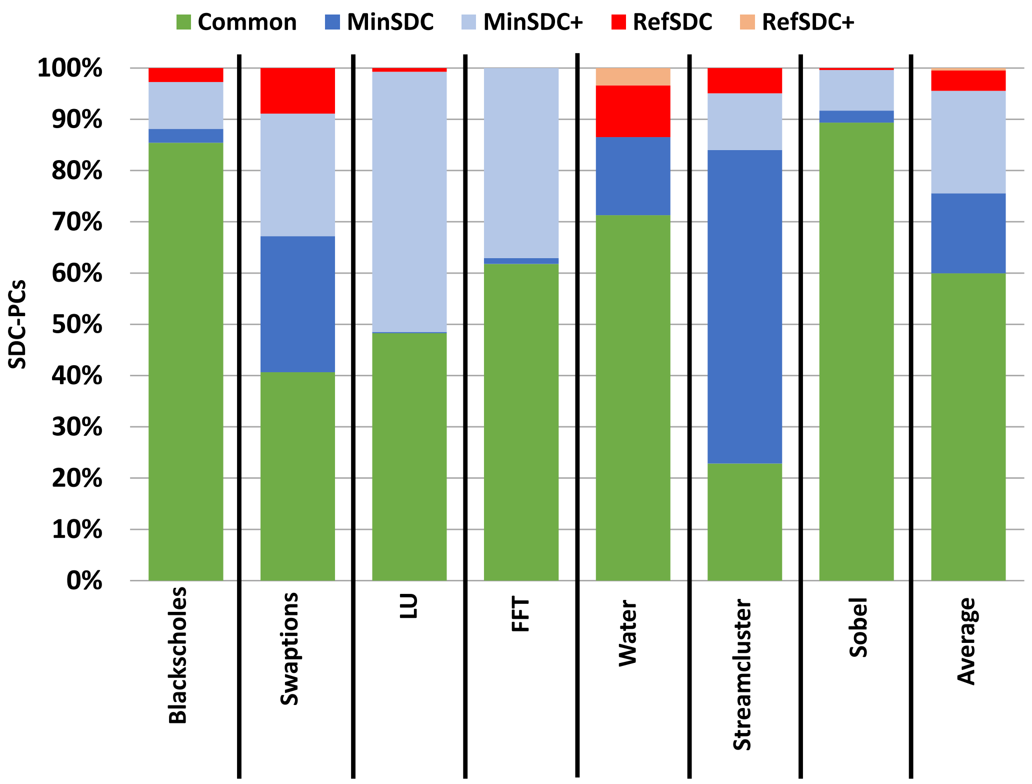


Figure 3.6: Min and Ref accuracy. The Y-axis represents all SDC-PCs found by Min or Ref in an application.

15.5× for FFT), with nearly all applications showing speedup. Even for the applications that do not show much speedup (Streamcluster and Sobel), the Min inputs are more accurate than Ref inputs (they identify more SDC-PCs) and benefit from error-injection prioritization, as discussed in the next sections.

Minimization Accuracy: Figure 3.6 shows the accuracy of Ref and Min for each application. The Y-axis corresponds to the union of SDC-PCs found by Ref or Min, distributed into the five accuracy categories (Section 3.4.3). The results show that a majority of SDC-PCs are categorized in the same way by both Ref and Min (60% on average are *Common*). Further, a large number of PCs fall in the MinSDC and MinSDC+ categories (35% on average). These are SDC-PCs that Min finds that Ref misses – either due to misclassification by Ref (MinSDC) or due to the lack of exploration of that PC by Ref altogether (MinSDC+).

Figure 3.7 explains the surprising result of finding additional SDC-PCs over Ref in the MinSDC+ category. The Y-axis corresponds to the total number of static PCs explored for different error site coverage targets. Ref error sites, although much more than Min error sites, generally explore fewer distinct PCs than Min at lower error site coverage targets. Figure 3.7 shows that, on average, for 99% error-site coverage (sorted by equivalence class

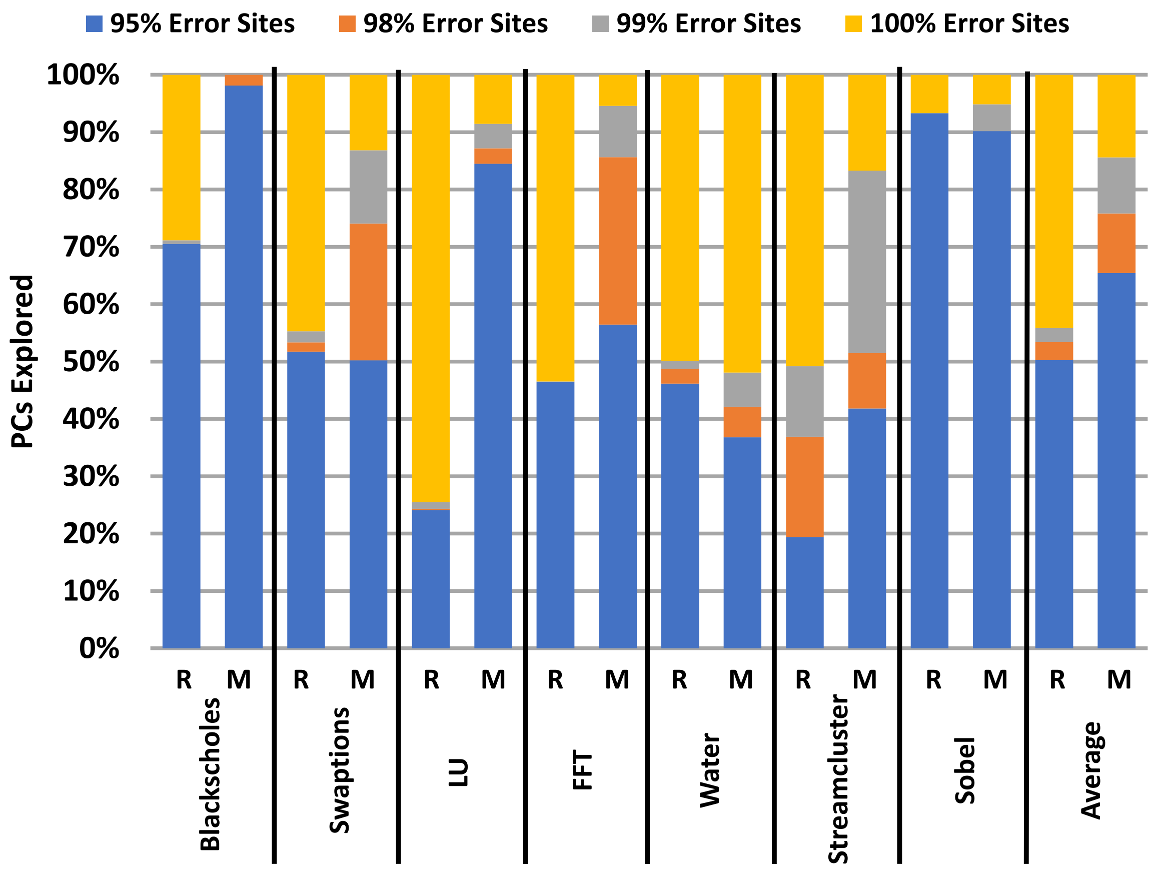


Figure 3.7: Percentage of PCs explored for different error-site coverage targets. R = Ref, M = Min.

size), Ref explores 55% of the static PCs explored by the union of Ref and Min, while Min explores 85%. Thus, it can still be advantageous to run resiliency analysis with Min for workloads such as Streamcluster and Sobel, even though the total analysis time is similar to that of running with Ref.

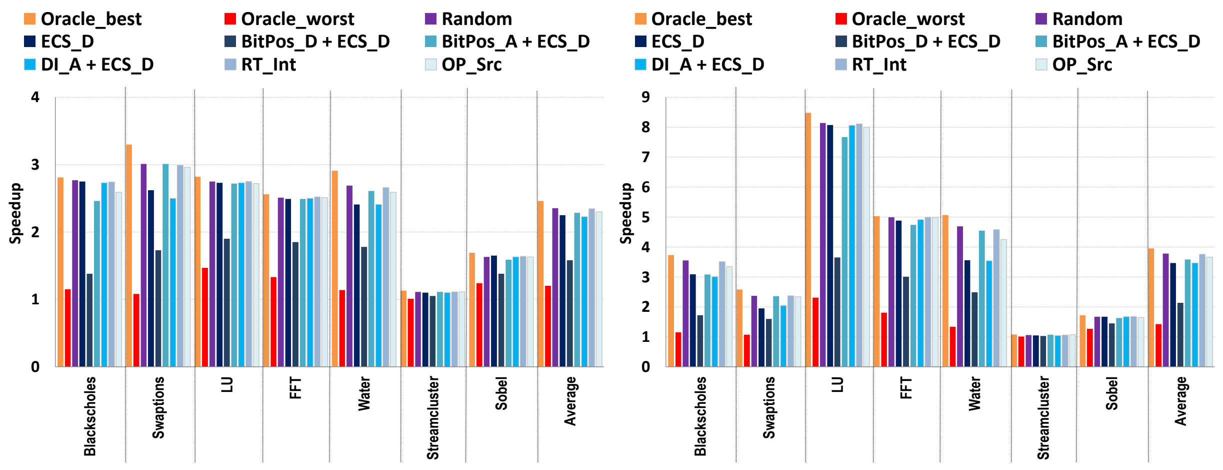
The remaining two categories, RefSDC and RefSDC+, reflect a loss of accuracy for Min. For many workloads, there are no RefSDC+ because Min explores all the PCs explored by Ref. The RefSDC category is also small, but not insignificant (4% on average). Upon further study of the misclassified PCs, we found that a majority of the mismatches occur at the boundary of SDC categories that distinguish if protection is needed or not. For example, in many cases Ref identifies a PC as SDC-Maybe, but Min identifies it as SDC-Good. Often the difference in output quality between these is less than 1%. Similarly, on the other end of the protection spectrum, there are many PCs that mismatch because Ref classified the PC as SDC-Bad but Min classified it as DDC.

Overall, Min shows significantly higher accuracy than Ref. Of the total SDC-PCs discovered, on average, Min finds 96% (the sum of Common, MinSDC, and MinSDC+ categories) while Ref finds only 64% (the sum of Common, RefSDC, and RefSDC+) of these SDC-PCs.

Improving Min Selection Criteria: We studied branch and def-use coverage of Min (relative to Ref) to understand if these stronger criteria could have been used to generate an alternate Min that provides higher accuracy than PC coverage. Table 3.1 shows that the Min inputs generated using PC coverage already have very high branch and def-use coverage of 99.76% and 98.78%, respectively, relative to Ref. Further, as discussed, Min already finds 96% of the SDC-PCs discovered by the union of Ref and Min. Thus, the potential improvement from using the more complex criteria is limited.

Nevertheless, we isolated the branch-target and def-use pairs that were in Ref but not in Min to determine if they were responsible for the RefSDCs in Figure 3.6. We found that none of the RefSDC PCs intersect with the isolated branch-target pairs and only four intersect with the def-use pairs (one each for Blackscholes and Swaptions and two for LU). A more comprehensive analysis would explore the entire control and data flow paths rooted at the isolated branch-target and def-use PCs in Ref to conclusively confirm whether the stronger criteria would add further accuracy. We leave such an analysis and exploration of even more complex input quality criteria (e.g., path coverage) to future work, given that our results already show that PC coverage provides an excellent sweet spot for simplicity, performance, and accuracy.

3.5.2 Error-Injection Prioritization



(a) Min Speedup

(b) Ref Speedup

Figure 3.8: Speedup with error-injection prioritization for Min and Ref.

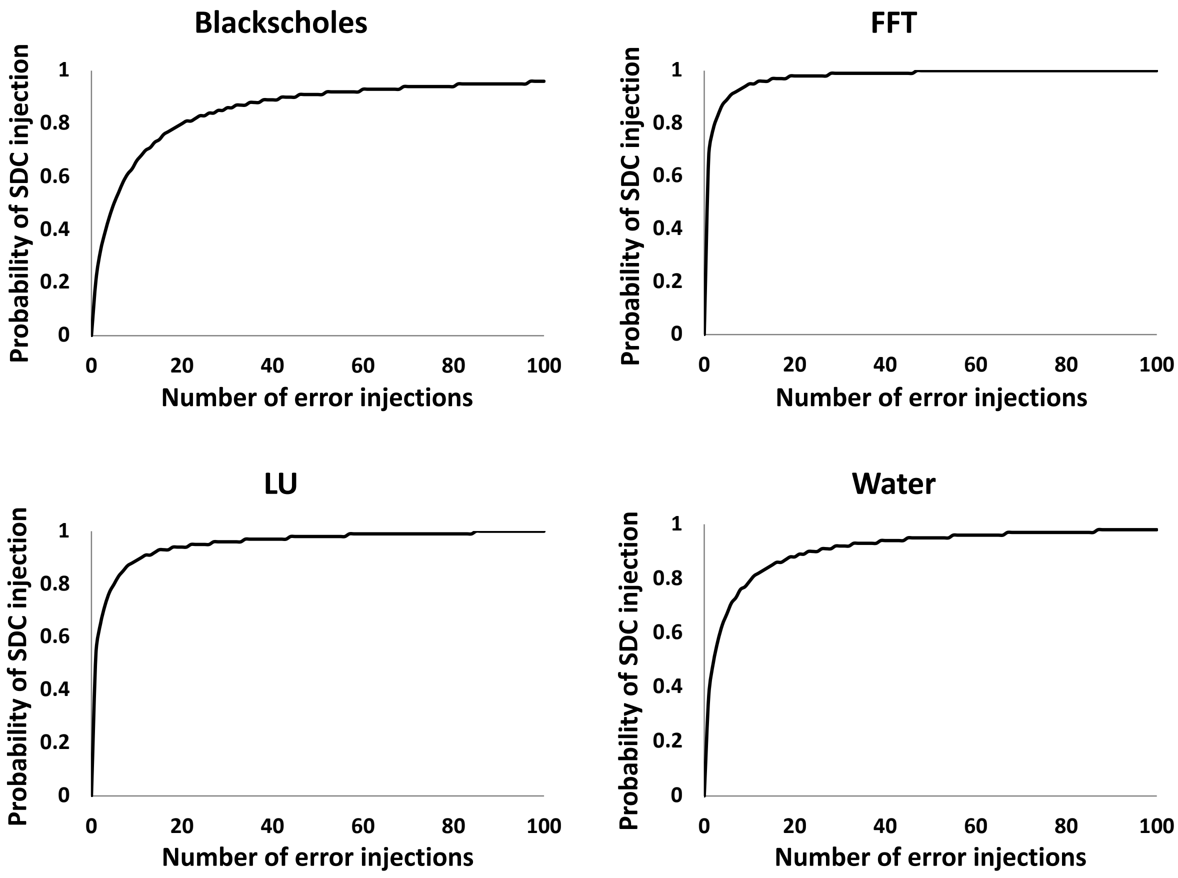


Figure 3.9: Cumulative probability (Y-axis) of picking an SDC-causing error injection within the first n injections (X-axis) for an SDC-causing PC.

We study 38 different error injection prioritization schemes (Section 3.4.4). For brevity, we show results only for the 7 most effective schemes, in addition to the oracle best-case and oracle worst-case schemes.

Figures 3.8a and 3.8b show the speedup results for Min and Ref, respectively, for different error injection prioritization schemes with early termination enabled. The figures show a noticeable speedup for most cases for both Min and Ref. Random prioritization gains the best average speedup of 2.4× and 3.8× for Min and Ref (upto 3× and 8.1×), respectively, while also being very close to the oracle best-case.

To understand the surprising result that Random performs the best, Figure 3.9 plots the cumulative probability (averaged over all SDC-PCs) of choosing an SDC-causing error injection after n error injections in an SDC-causing PC. Figure 3.9 shows only four applications using Ref input, but the trends are representative across the workloads and inputs. The figure shows that the probability of finding an SDC injection shoots up within the first few injections. Upon investigation, we uncover an interesting insight – when a PC is SDC-causing, a large fraction of the injections in that PC result in an SDC outcome. Randomly choosing an injection therefore tends to quickly find an SDC for that instruction. Thus, we choose the Random error injection prioritization scheme for the remainder of the evaluations in this chapter.

3.5.3 Minimization With Injection Prioritization

This section discusses the benefits of combining input minimization with error injection prioritization. Figure 3.5 shows the speedup in resiliency analysis, relative to Ref, from (1) using Min, (2) using Min with error injection prioritization (referred to as Min_{EIP}), and (3) using Ref with error injection prioritization (Ref_{EIP}). As discussed previously, using only Minotaur's input minimization optimization for resiliency analysis provides a 4.1× average speedup (up to 15.5×) compared to Ref (first bar for each application in Figure 3.5). Combining Minotaur's input minimization optimization with error injection prioritization results in an average speedup of 10.3× (up to 38.9× for FFT), relative to Ref. In contrast, Ref_{EIP} observes only a 3.8× average speedup (up to 8.14× for LU) relative to Ref (third bar for each application in Figure 3.5 and also discussed in Section 3.5.2).

Recall that the accuracy of Min_{EIP} is the same as that of Min as described in the previous section. Thus, in addition to Min_{EIP} significantly outperforming Ref and Ref_{EIP} on average, Min_{EIP} has the added benefit of finding many SDC-PCs that were not found by Ref (and Ref_{EIP}) – Min finds 96% of the total SDC-PCs while Ref finds 64%.

3.5.4 Input Prioritization

For safety-critical systems which may require even higher accuracy, Minotaur provides the additional optimization of *input prioritization*. This optimization can speed up the analysis of multiple inputs in an attempt to further improve SDC-PC identification without taking the performance hit of running resiliency analysis for each individual input in its entirety. Figure 3.10 shows the runtime of analyzing both Min_{EIP} and Ref_{EIP}, without and with input prioritization, normalized to the runtime of Min_{EIP}.

The first bar for each application shows the runtime of employing a naive input prioritization scheme, by simply running Min_{EIP} followed by Ref_{EIP} analyses in their entirety (Min_{EIP} + Ref_{EIP} in the figure). The second bar shows the runtime of running Min_{EIP} and Ref_{EIP} with input prioritization enabled. That is, Min_{EIP} is first run in its entirety (which is relatively fast, as discussed in Section 3.5.3), followed by Ref_{EIP} *but only for PCs not identified as SDC-PCs* by Min_{EIP}. Thus, input prioritization requires the second input

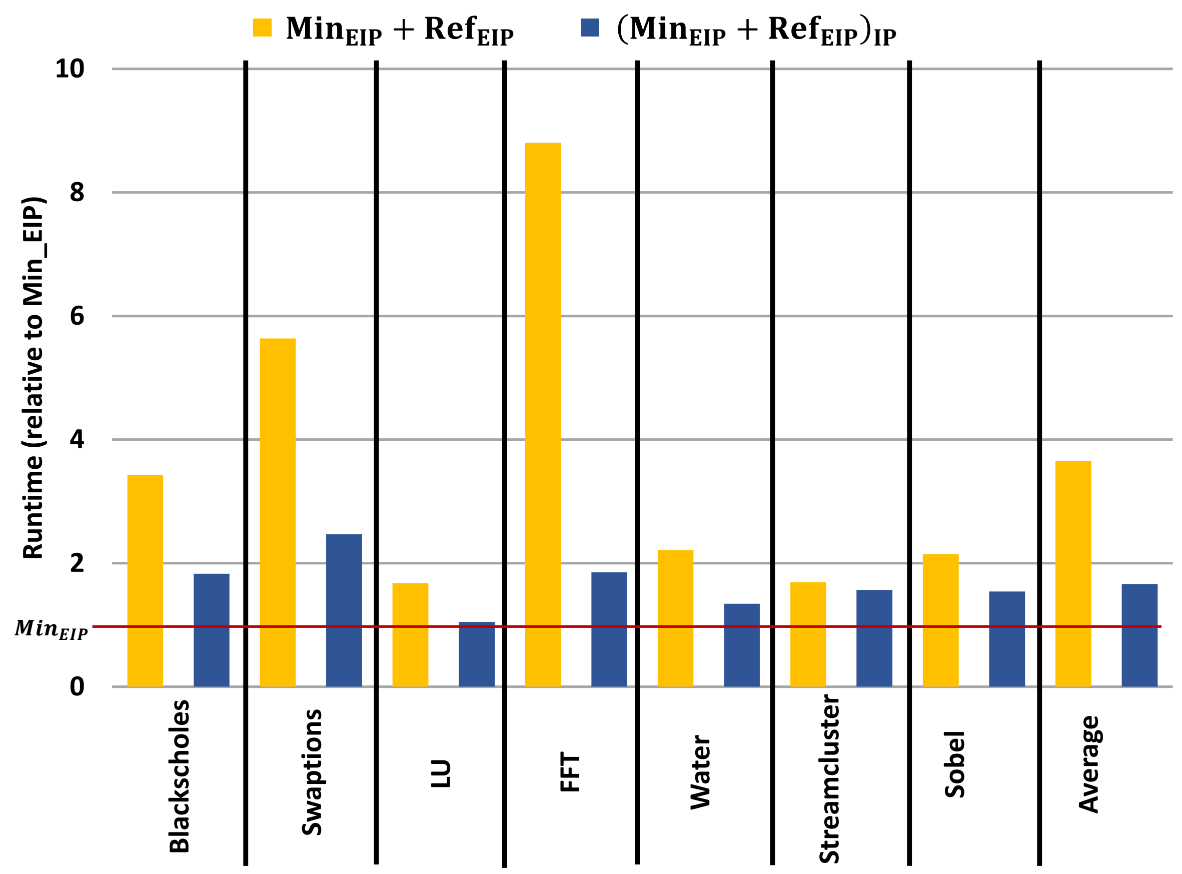


Figure 3.10: Resiliency analysis time for analyzing both Min_{EIP} and Ref_{EIP}, without and with input prioritization, normalized to analysis time for only Min_{EIP}.

(Ref_{EIP} in our study) to run for only a fraction of the original resiliency analysis time.

Figure 3.10 shows that without input prioritization, Min_{EIP} + Ref_{EIP} runs 3.7× slower than Min_{EIP}. Using input prioritization ((Min_{EIP} + $\text{Ref}_{EIP})_{IP}$ in the figure) brings the analysis time to only 1.6× slower than Min_{EIP}. Thus, leveraging input prioritization allows Minotaur to analyze both inputs 2.3× faster on average than analyzing each input alone in its entirety. By carrying over information from one input analysis to the next, Minotaur is capable of achieving 100% accuracy while running much quicker than previous techniques.

3.6 MINOTAUR EXTENSIONS

Minotaur's techniques can be used to benefit analyses beyond those discussed so far. This section demonstrates Minotaur's generality by discussing and evaluating two extensions.

3.6.1 Extension to Approximate Computing

The resiliency analyzer we chose (Approxilyzer [44]) can also be used for approximate computing. Approxilyzer can identify approximable instructions by grouping error sites differently. Whereas for resiliency we focus on SDC-Maybe and SDC-Bad outcomes,Approxilyzer classifies an instruction as approximable if no egregious errors – Detected, DDC, or OC above a user-defined threshold – are observed for any dynamic instance of that instruction. We use the following user-defined thresholds: 1) for financial applications, errors in individual outputs that are smaller than a cent are tolerable and 2) for other applications, relative errors up to 5% in individual outputs are tolerable. We use the same Min and Ref inputs as in Table 3.1, and apply random error injection prioritization with early termination (we observe the same trend that randomized error injection ordering performs close to oracle best). For approximate computing, early termination is triggered when an error-injection reveals a PC as non-approximable, indicating that no further injections are required for that instruction.

For approximate computing, Minotaur's analysis time without error injection prioritization is the same as that for resiliency since we use the same Min and Ref inputs. That is, Min observes an average 4.1× speedup compared to Ref, due to Min's smaller size. Applying error injection prioritization for approximate computing analysis (where early termination differs compared to resiliency, as described above), Min analysis can be sped up by 4.4× on average, while Ref shows an average speedup of 5.53×. Combining the two optimizations, Min_{EIP} shows an average speedup of 18× compared to Ref for approximate computing analysis.

We use an accuracy metric similar to that in Section 3.4.3, adapted from SDC-PC to Approximable-PC. Min shows very high accuracy – of all the approximable-PCs identified by both Min and Ref, on average, Min identifies 96% while Ref identifies 81%.

3.6.2 Selective Instruction Analysis

Minotaur can speed up analysis for any desired subset of PCs. For example, a user may desire to analyze the "hot" PCs that account for X% of the dynamic execution. The user can identify the "hot" PCs by first profiling Ref and then switching to Min to run the resiliency analysis. For instance, by targeting the PCs for the top 25% of the dynamic execution in Blackscholes, Min_{EIP} speeds up the analysis by 6.8× over Ref for the same PCs and with 100% accuracy.

3.7 SUMMARY

This chapter presents Minotaur, a toolkit to improve the analysis of software vulnerability to hardware errors by leveraging concepts from software testing. Minotaur adapts several concepts from software testing for software bug detection to resiliency analysis for hardware error detection: 1) identifying test-case quality criteria, 2) test-case minimization, and 3) two adaptations of test-case prioritization. We evaluate Minotaur on Approxilyzer, the resiliency analysis tool introduced in Chapter 2. Minotaur's single-input techniques speed up Approxilyzer's resiliency analysis by 10.3× on average while significantly improving SDC-PC detection accuracy (96% vs. 64% on average) for the workloads studied. Further, Minotaur presents a technique, *input prioritization*, which enables finding SDC-PCs across multiple inputs at a speed 2.3× faster (on average) than analyzing each input independently.

Although Minotaur is already very effective, there are many avenues of future work to improve both Minotaur's effectiveness and its applicability. For example, it is possible to explore more input quality criteria (such as path coverage, loop coverage, or state coverage [87]) as well as develop new quality criteria geared specifically towards resiliency (e.g., criteria derived from ACE bits [35] or PVF [27]) or towards approximate computing (e.g., using parameter range coverage). Further, more sophisticated optimizers to improve the speed and scalability of the Minimizer along with custom minimization objectives (e.g., number of error-sites analyzed) for faster Mins. The Input Selector can also be improved, by tuning analysis speed versus accuracy for multiple Refs and Mins with variable input quality thresholds.

To widen the applicability of Minotaur, we can apply it to other resiliency and approximation analysis techniques proposed in the literature, using a broader range of error models abstracted at lower and higher layers of the system stack than studied here. Our end goal is a seamless integration of resiliency analysis (and hardening) into the standard software development and testing workflow. Minotaur opens up many avenues for further research towards this ambitious end goal. Modern software development practices such as continuous integration encourage developers to continuously commit their code, which would be ideally checked for resiliency, making fast and accurate resiliency analysis techniques such as Minotaur even more important.

Chapter 4: Optimizing General-Purpose, Software-Directed Instruction-Level Protection

4.1 MOTIVATION

Resiliency analysis techniques, such as those proposed by Approxilyzer and accelerated by Minotaur, can help inform better resiliency hardening schemes by identifying erroneous error sites and providing a general understanding of error propagation in applications and systems. Such an analysis is advantageous in reducing the scope of protection to only the most critical locations in an application. At the same time, the specific hardening scheme used to implement the protection can also benefit from an optimized analysis and implementation, in order to reduce the high overheads of protection.

State-of-the-art systems typically employ ECC or parity protection for major storage structures such as main memory, caches, and the register file [110, 111, 112, 113]. However, prior work indicates that datapath errors originating from unprotected latches scattered across the processors will contribute significantly to the total SDC rate [114, 115]. Without datapath reliability mechanisms, such systems may not be able to maintain high reliability at future error rates and system scales. Traditional hardware-only solutions that duplicate the entire processor can provide datapath reliability [50, 51]. However, processor duplication is expensive and excessive for workloads or sections of code that are inherently resilient. Software-based redundancy overcomes these issues and can provide the flexibility of protecting just the vulnerable parts of the program without incurring the high design, debug, and testing costs attributed to hardware-only schemes.

In this chapter, we target software-directed instruction replication for GPU error detection. Software instruction-level duplication has been studied extensively for CPUs and has been shown to provide runtime overheads that are significantly lower than 100% by exploiting under-utilized hardware resources (approximately 60% using a 4-way issue super-scalar processor and 40% for Intel Itanium CPUs) [116, 117, 118]. Duplication at this level has never been explored for GPUs. Prior research has shown that many GPU workloads under-utilize GPU cores [119, 120], indicating potential for a low-overhead instruction-level duplication solution.

For GPUs, software-based redundancy can alternatively be introduced at various granularities such as the process, kernel, and thread. However, only instruction-level duplication can be applied seamlessly to workloads that produce non-deterministic results at a coarse granularity (e.g., at the function, GPU kernel, or application output level) without requiring spare hardware resources to be reserved solely for redundancy purposes. Higher level duplication techniques often suffer from limitations in one of these aspects. Section 7.3 discusses these trade-offs in detail.

This chapter introduces SInRG (pronounced "synergy"), **S**oftware-managed **In**struction **R**eplication for **G**PUs, which is a family of techniques that optimize software-based instruction duplication for GPUs. Our work is the first to establish a practical approach to software-directed instruction duplication for GPU-based systems, identify GPU-specific opportunities for overhead reduction, and explore software and hardware performance optimizations to lower the overheads significantly.

SInRG first implements a commonly-studied CPU instruction duplication algorithm in NVIDIA's production compiler and evaluates it on real GPUs. This algorithm duplicates the data-flow chains leading to non-duplicated instructions and maintains two register spaces such that the original and duplicate instructions operate on the original and shadow register spaces, respectively [117]. Whenever a non-duplicated instruction is executed, the source register values are verified, and a higher layer in the system is notified if verification fails. Our results show an average runtime overhead of 69%.

Main contributors to the overhead stem from two sources. (1) Doubling the number of required registers per thread can adversely affect performance for some workloads because the register file is a shared resource and its inefficient use can limit the number of concurrent threads (and performance). (2) The total number of executed instructions increase with the introduction of additional verification and notification instructions. These new instructions also introduce new dependencies, which can limit the ability to software-pipeline instructions, further increasing runtime overheads.

SInRG addresses the first issue by employing an instruction duplication algorithm that trades off the per-thread register requirement for more verification and notification instructions. We propose a set of solutions to reduce the overheads caused by executing verification and notification instructions. (1) SInRG removes direct dependencies between the verification and notification instructions, using a per-thread flag to defer error notification to the end of a thread. It trades off error containment for performance, which can be mitigated through a small ISA extension that provides error notification capability to the verification instructions. (2) We discover that the verification and deferred notification can be implemented using just a single, high-throughput assembly instruction supported by current GPUs. (3) With the aim of eliminating the verification and notification instructions altogether, SInRG accelerates the first solution above through hardware support. Some of these solutions defer error detection until the end of the kernel, which may affect fine-grain recoverability. The increase in detection latency is however not a concern for coarse grain coordinated checkpointing solutions [121, 122, 123, 124].

Results show that SInRG reduces the average runtime overhead to 36% (1.94× lower than the naïve implementation) with software-only techniques. Simple hardware extensions that eliminate verification and notification instructions reduce the average overhead to 30%. We compare SInRG to a prior state-of-the-art GPU software-based approach – thread-level duplication (TLD) [125, 126, 127] – and further optimize it for comparison. Results show that SInRG is faster than TLD for a majority of the workloads studied despite it not needing/utilizing spare thread resources.

We evaluate the error detection capabilities of SInRG by quantifying the dynamic instruction coverage (counting the executed instructions that are protected by SInRG). Results show that on average, 87% of dynamic instructions are covered. We also conduct architecture-level error injection experiments to show that SInRG is effective in reducing SDCs. Results show that the percentage of injected errors that result in SDCs is always lower than the percentage of uncovered dynamic instructions. Lastly, we evaluate the effect on the true failure rate (measured as Failure In Time or FIT, where 1 FIT = 1 failure in 1 billion hours of operation) reduction by conducting accelerated high-energy particle testing. Results show that SInRG can reduce the SDC FIT rate by an order of magnitude.

4.2 BACKGROUND AND CHALLENGES

4.2.1 GPU Background

This section reviews basic GPU architecture terminology and the NVIDIA GPU compilation flow because SInRG is implemented using NVIDIA's technology. However, the ideas presented in this chapter can be applied to other GPU architectures.

GPU programming models consider thousands of threads that each execute the same code. Threads are grouped into 32-element vectors called *warps* to improve efficiency. The threads in each warp execute in a single instruction, multiple thread (SIMT) fashion. Many warps are assigned to execute concurrently on a single GPU streaming multiprocessor (SM). An SM offers resources that are shared by all the executing threads, such as the register file and shared memory (or scratchpad). A GPU consists of many SMs attached to a memory hierarchy that includes SM-local scratchpad memories, L1 caches, a shared L2 cache, and multiple DRAM channels.

A user can write parallel programs using high-level programming languages such as CUDA [128] or OpenCL [129], and use a front-end compiler to generate intermediate code in a virtual ISA called parallel thread execution (PTX) [130]. A backend compiler optimizes and translates PTX instructions into machine code that can run on the device. NVIDIA's native ISA is called SASS [131]. The backend compiler can be invoked in two ways: (1) ahead-of-time compilation of compute kernels via a PTX assembler (ptxas) or (2) just-in-time compilation by the GPU driver (if the PTX code is part of the binary).

4.2.2 Challenges with GPU Instruction Duplication

Overheads of an instruction duplication algorithm arise from the introduction of the three types of instructions: redundant, verification, and notification instructions. Prior optimizations target leveraging under-utilized resources and reduce the number of the added instructions. GPUs, however, present several new challenges in developing a cost-effective solution.

Limited shared resources: Since SMs provide resources that are shared among all the executing threads, inefficient per-thread usage of these shared resources may limit the number of warps that can simultaneously run on the SM (also known as warp occupancy). The register file is one such resource; doubling the per-thread register requirement can limit the warp occupancy and increase the overall runtime.

Additional dependencies: The verification and notification instructions added by an instruction duplication algorithm introduce read-after-write dependencies between themselves, which may limit instruction level parallelism (ILP) and the instruction scheduler's ability to pipeline instructions. Such limitations can significantly increase the runtime overheads if the workload is not capable of executing enough concurrent threads. Prior research has also noted the importance of ILP for GPUs [132].

Extra instructions: Verification and notification instructions increase the dynamic instruction count. Moreover, throughput offered by the assembly instructions used for them can be low. For example, compare operations have half of the maximum throughput offered by some instructions on NVIDIA GPUs [128, 131].

4.3 SINRG: SOFTWARE-MANAGED INSTRUCTION REPLICATION FOR GPUS

4.3.1 Sphere of Replication (SoR)

GPUs used in HPC and safety-critical systems protect major memory structures such as DRAM, caches, and the register file using ECC/parity. However, unprotected execution units and pipeline registers remain susceptible to soft-errors. Since GPUs are designed to maintain high throughput for arithmetic intensive workloads, the datapaths constitute a significant fraction of the chip area, unlike CPUs which devote most of their non-cache logic to instruction delivery and control speculation.

To protect the execution units and pipeline stages, we employ assembly instruction-level duplication without duplicating values in memory. We use the term sphere-of-replication (SoR) to identify at a high-level what is duplicated and which hardware structures we expect to be protected by this approach. This technique can detect errors that affect program text (instructions) and computation (instruction execution). Since not all instructions are duplicated (e.g., branch instructions and atomic operations remain unduplicated), the coverage is high but not complete. We quantify coverage in later sections.

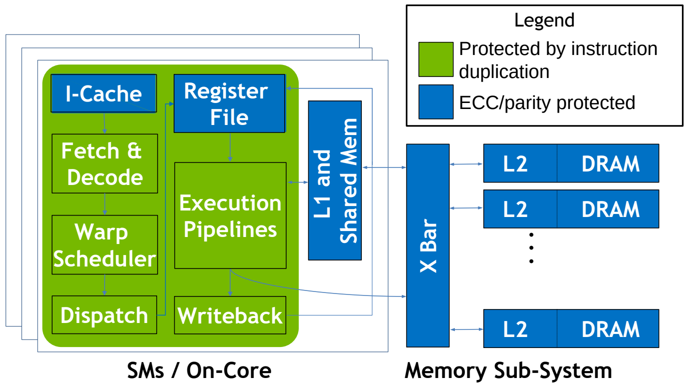


Figure 4.1: The GPU hardware structures in the SoR. We focus on protecting the GPU datapath against transient errors.

Figure 4.1 shows the hardware structures that SInRG protects. Almost all of the SM units used to execute an instruction (from instruction fetch to write-back) receive protection from single-event errors. SInRG also delivers additional protection to some of the structures that are protected by hardware ECC/parity(e.g., I-cache for all SInRG schemes and register file for the schemes that duplicate registers). This additional protection comes for free.

4.3.2 Instruction Duplication Algorithms

SInRG duplicates all instructions that (1) produce deterministic values, (2) do not directly modify the control flow, and (3) do not write to memory. We call such instructions *duplication eligible*.

SInRG performs duplication using two main base algorithms. The first algorithm, inspired by Reis et al. [117], duplicates all the instructions in a data-flow chain leading to a non-duplicated instruction and verifies the values only at the end of the chain. Duplicated instructions operate on a *virtual shadow register space*. For instructions that are not duplication eligible and write to a register (e.g., atomic operations and special registers), we copy the result of the original instruction to the shadow register space to maintain the functionality of shadow execution. We call this algorithm *DRDV*, because it **d**oubles the virtual **r**egister space and **d**elays **v**erification until the end of a data-flow chain.

The second base algorithm duplicates all duplication eligible instructions and places them just before the original instruction. The duplicated instruction reads the same source registers used by the original instruction and writes to a new virtual register. We immediately verify the value in the new register with the destination register value of the original instruction, and notify the runtime layer for appropriate event handling if the verification fails. This scheme adds verification and notification instructions for every duplication eligible instruction, increasing the total number of dynamic instructions significantly (and hence is often ignored by CPU implementations). We call this algorithm *SRIV* because it uses a **s**ingle **r**egister space and **i**mmediately **v**erifies each instruction's result.

4.3.3 SInRG Optimizations

This section presents techniques we propose to address the challenges mentioned in Section 4.2.2. Table 4.1 summarizes the trade-offs offered by these optimizations. Each SInRG duplication technique is listed with a qualitative comparison of its attributes (relative to an uninstrumented workload), which are discussed in more detail below.

Trading off additional dynamic instructions to reduce register requirements: The DRDV algorithm doubles the virtual registers required per thread. Running the NVIDIA compiler's production-quality register allocator after the instruction duplication pass can reduce the real register usage per thread. Despite this optimization, DRDV often observes a significant increase in the number of registers used per thread. For workloads where the register file is a critical resource, this approach can either reduce the number of threads that can run in parallel or increase the number of register spill/fill instructions. The SRIV algorithm naturally provides an interesting trade-off because it does not alter the original application's register requirement by much, but instead executes more dynamic instructions. This trade-off can benefit some workloads, especially when the register file is a critical resource, which we analyze in more detail in Section 4.6.

Deferring error notification: The code to notify the upper layers of the system (e.g., a trap instruction and the control flow instructions to skip the trap in fault-free executions) is typically added after every verification instruction for error containment. The added dependency between the two instructions can contribute significantly to performance overheads, as

Table 4.1: Summary of the SInRG techniques

	SInRG Technique	Attributes: ✓= low, O = medium, ✗= high		Attributes: ✓= yes, ✗= no	
		# Verification Instructions	Register requirement per thread	Error Containment	Error Masking Potential
DRDV	Base	✗✗	✗	✓	✓
	FastSig	✓	✗	✗	✓
	HW-Notify	✓	✗	✓	✓
	HW-Sig	✓✓	✗	✗	✗
SRIV	Base	✗✗	✗	✓	✗
	FastSig	O	✓	✗	✗
	HW-Notify	O	✓	✓	✗
	HW-Sig	✓✓	✓	✗	✗

Abbreviations:	
DRDV:	Double register space, delayed verification
SRIV:	Single register space, immediate verification
FastSig:	Software-only, fast signature-based checking
HW-Notify:	Hardware instruction to compare-then-trap
HW-Sig:	Signature-based checking in hardware
TLD:	Thread Level Duplication [126]
TLD-Sig:	TLD [126] with delayed notification using signatures

mentioned in Section 4.2.2. To reduce the overheads, we investigate deferring the notification until the end of the function. The results of all verification instructions are accumulated to produce a single flag (signature), which is then used by a single error notification instruction at the end of the function. Similar approaches have been explored and shown to be effective in the context of software testing [133, 134, 135]. This optimization drastically reduces the number of error notifications (and associated control flow instructions) and enables better instruction scheduling. However, it allows some erroneous values to propagate to memory before the error is detected and notified. While this optimization may violate the error containment assumptions of some recovery schemes, it works fine for coarse-grain coordinated checkpoint systems that discard memory values in the event of a detected error to roll back to a previous checkpoint [121, 122, 123, 124].

Verification and accumulation of the result must be implemented efficiently for a low overhead solution. This can be accomplished using one or two high-throughput assembly instructions on current GPUs. This approach also addresses the third challenge mentioned in Section 4.2.2 and explained in Section 4.4.3. We call this combined software optimization *FastSig* and it applies to both the DRDV and SRIV algorithms.

Eliminating notification instructions: To eliminate explicit error notification instructions and provide high error containment, we propose a simple extension to an existing GPU instruction that is used to compare two values. This extension raises an exception in hardware if the values mismatch. We call this technique *HW-Notify*.

Eliminating verification and notification instructions: We eliminate the verification and notification instructions by proposing *HW-Sig*, which uses the same principles as FastSig and provides hardware support to maintain the signature register. This register is initialized at kernel launch time. It is updated by each of the original duplication-eligible and duplicate instructions such that it will have the same initialized value at the end of a fault-free kernel execution. Maintaining one register per thread can be expensive in GPUs because SMs support thousands of threads. We overcome this challenge by maintaining just one signature register per hardware lane, which would be used by all threads (from different warps) that execute on the lane. HW-Sig improves performance without sacrificing error coverage. Since it defers the error notification, similar to FastSig, the trade-offs are also similar.

We also considered a scheme that extends the ISA such that each duplicate instruction automatically verifies the result produced by the original instruction and notifies upper layers upon failure. The source operands of the original instruction should not be updated before the duplicate instruction executes, which introduces new instruction scheduling constraints. As our evaluation showed lackluster performance, we do not discuss it further in this work.

4.4 SINRG IMPLEMENTATION

4.4.1 GPU Compilation Flow

GPU instruction duplication can be implemented at several places in the compiler tool chain. While performing it early in the flow before PTX code generation is perhaps easiest to implement, later compiler optimization passes may transform the program and eliminate the resilience-oriented instructions. Inserting the replicated and checking instructions directly into the SASS code ensures tight control over the final program binary, but requires re-implementation of instruction scheduling and register allocation which are already in the back-end compiler.

Avoiding these limitations, we implement SInRG within the back-end compiler (ptxas), applying our transformations on the intermediate representation there. The duplication algorithm runs after all back-end optimizations are performed, but before the instruction

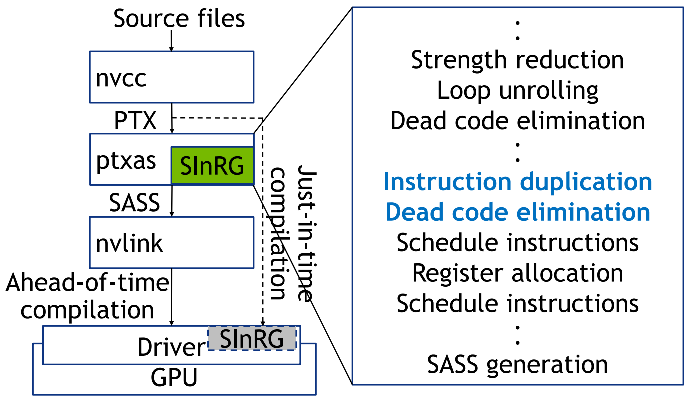


Figure 4.2: The GPU compiler flow with SInRG.

scheduling or register allocation passes. This approach leverages the production-quality instruction scheduler already implemented in the back-end compiler, which helps to lower the performance overheads of the duplication and verification code. It also enables instruction duplication on programs for which only the PTX code (rather than the CUDA or OpenCL source code) is available. Figure 4.2 summarizes the compilation flow for NVIDIA GPU programs, including the SInRG instruction duplication pass. We evaluate SInRG using the ahead-of-time compilation flow, but the just-in-time compiler can employ the same instruction duplication algorithms.

4.4.2 Instruction Duplication Compiler Pass

In SInRG, we duplicate every duplication-eligible instruction once, using a data-structure to track already-protected instructions so as not to duplicate them multiple times. We then create the data structure to track the shadow register mapping. In our implementation, we duplicate all major register classes—general-purpose registers, predicate registers, and condition codes—except for the predefined registers such as zero-value register and thread-id. Next, for each original duplication-eligible instruction, we duplicate the instruction and map the duplicate into the shadow register space.

Instructions that are *not* eligible for duplication include memory writes, control-flow instructions, instructions that produce non-deterministic values, barrier spill/fill instructions, and instructions that write to pre-assigned physical registers. Non-deterministic instructions—those where the replica and the original instruction can produce different values—include S2R instructions that read special registers whose values change over time (e.g., the clock value), atomic operations, and volatile and non-cached memory reads [131].

Algorithm 4.1: The *DRDV* back-end compiler instruction duplication algorithm, run once per function.

```
1  create list of original instructions
2  clear original to shadow register mapping
3  for each instruction in the function do
4      if instruction is duplication-eligible and original then
5          duplicate the original instruction
6          for all operands in the duplicate instruction do
7              if shadow register does not exist then
8                  create a shadow register for the source
9              end
10             replace original register to shadow register
11         end
12     else if instruction is copy eligible and original then
13         insert a move instruction to copy the destination register value to the shadow
             space
14     end
15 end
16 for each instruction in the function do
17     if instruction is not duplication eligible and is original then
18         for all sources in this instruction do
19             verify original and shadow registers have same value
20             if values are different then
21                 notify error to higher level (trap)
22             end
23         end
24 end
```

A load can be non-deterministic if there is a data race in the program. While we would ideally only mark the race-vulnerable loads as non-deterministic, identifying only this subset of loads is not feasible. Instead, we conservatively mark all generic, global, shared, texture, and surface loads as non-deterministic. We mark local and constant loads as deterministic because they cannot partake in data races. Local memory offers per thread storage (which cannot be accessed by other threads) and constant memory is read-only (and cannot be written to). We do not apply SInRG passes to built-in CUDA Runtime API calls.

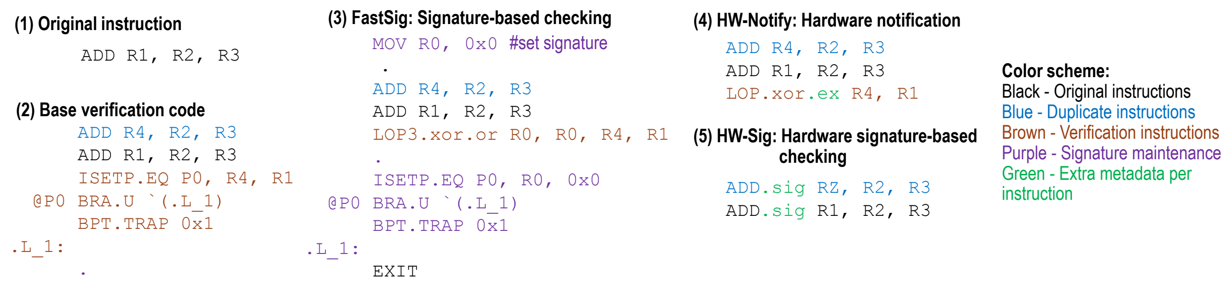
For DRDV, verifying the inputs of a non-duplicated instruction requires adding a set of verification and notification instructions, one for each source operand. We implement an optimization where we insert only one error notification instruction per non-duplicated instruction (as opposed to one per source operand) by chaining multiple verification instructions. We place the duplicate instruction after the original instruction and map the registers used by it into a shadow register space. For all non-duplicated *copy eligible* instructions, we insert a move instruction to copy the destination register value into the shadow register space so that duplicated instructions can use it. Finally, we insert verification instructions to check original and shadow register values for all inputs to non-duplicated instructions. This approach reduces the verification overhead (compared to SRIV) by chaining multiple replicated instructions on the path to a single verification. Algorithm 4.1 describes our implementation of the DRDV instruction duplication algorithm.

For SRIV, we place the duplicate *before* the original instruction because the original instruction may overwrite a source operand, and we want the duplicate to generate the same result as the original instruction using the same source operands. We do not duplicate the original move operations because they naturally duplicate the source register value into the destination register. We verify them by comparing the source and destination registers of the original operation. Verification and notification consist of a comparison operation, a conditional branch instruction, and a trap instruction (`BPT`). Algorithm 4.2 describes our implementation of the SRIV instruction duplication algorithm.

Algorithm 4.2: *SRIV* back-end compiler instruction duplication algorithm, run on each function.

```
1  create list of original instructions
2  for each instruction in the function do
3      if instruction is duplication eligible and original then
4          duplicate and place it before the original instruction
5          for all destination registers in the duplicate instruction do
6              replace original register with a new virtual register
7              verify the original and new registers have same value
8              if values are different then
9                  notify error to higher level (trap)
10             end
11         end
12 end
```

Figures 4.3(1) and 4.3(2) show an example of how we duplicate and verify an add instruction using SRIV. Base verification includes two additional instructions in the critical path for each duplicated instruction and creates sequential dependencies which can affect performance. The branch and trap instructions also limit the instruction scheduler, which does not efficiently schedule instructions across trap instructions or basic blocks.



(1) Original instruction

```
        ADD R1, R2, R3
```

(2) Base verification code

```
        ADD R4, R2, R3
        ADD R1, R2, R3
        ISETP.EQ P0, R4, R1
    @P0 BRA.U `(.L_1)
        BPT.TRAP 0x1
.L_1:
        .
```

(3) FastSig: Signature-based checking

```
        MOV R0, 0x0  #set signature
        .
        ADD R4, R2, R3
        ADD R1, R2, R3
        LOP3.xor.or R0, R0, R4, R1
        .
        ISETP.EQ P0, R0, 0x0
    @P0 BRA.U `(.L_1)
        BPT.TRAP 0x1
.L_1:
        EXIT
```

(4) HW-Notify: Hardware notification

```
    ADD R4, R2, R3
    ADD R1, R2, R3
    LOP.xor.ex R4, R1
```

(5) HW-Sig: Hardware signature-based checking

```
    ADD.sig RZ, R2, R3
    ADD.sig R1, R2, R3
```

Color scheme:
Black - Original instructions
Blue - Duplicate instructions
Brown - Verification instructions
Purple - Signature maintenance
Green - Extra metadata per instruction

Figure 4.3: Optimizing the verification and error notification code.

4.4.3 SInRG Optimizations

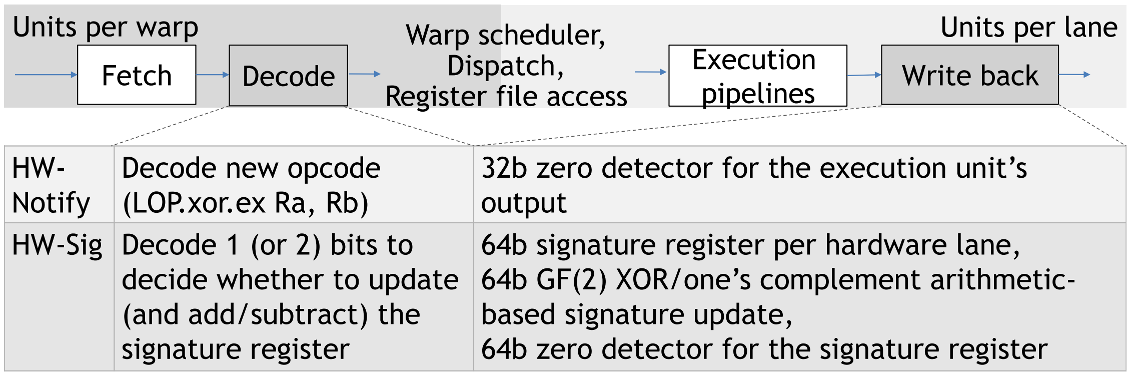
FastSig: This technique accumulates the results of the verification instructions into a signature register and uses it at the end of the function for deferred error notification. This signature (flag) register is initialized to zero at the beginning of a function. On every register verification, the values produced by the original and duplicate instructions are added to and subtracted from the signature register, respectively. If the signature register is not equal to zero at the end of the function, an error has occurred.

Using simple add and subtract operations may miss some errors due to over/under-flow. Instead, we compute bit-wise difference between the destination registers of the original and duplicate instructions using *XOR*, and then *OR* the result with the signature register to update it. During a fault-free execution, the signature register will remain zero. We discovered that the `LOP3` operation supported by the current NVIDIA GPUs can create any arbitrary logical function using three source operands and is well suited for the signature accumulation [130, 131]. Moreover, it offers the highest throughput among all supported instructions. We maintain a separate predicate signature register and use the `PSETP` instruction to perform a similar accumulation operation in one instruction.

Figure 4.3(3) shows an example of how this optimization reduces the number of static verification instructions from three to one. Furthermore, this optimization allows us to predicate the verification instructions if the original store instruction is predicated, providing added benefit. In the base approach, the verification (`ISETP`) instruction cannot be predicated because it must generate a correct predicate register for the subsequent branch instruction.

Since FastSig relaxes error containment, an error can propagate to memory and in some cases result in a crash/hang before the notification instruction is executed. If a function has many conditional return instructions, the number of error notification instructions will also be high, increasing overheads.

HW-Notify: We propose a new branch-free instruction that compares two values and raises an exception on a mismatch to provide low-latency error detection with full error



HW-Notify	Decode new opcode (LOP.xor.ex Ra, Rb)	32b zero detector for the execution unit's output
HW-Sig	Decode 1 (or 2) bits to decide whether to update (and add/subtract) the signature register	64b signature register per hardware lane, 64b GF(2) XOR/one's complement arithmetic-based signature update, 64b zero detector for the signature register

Figure 4.4: Summary of the two hardware techniques.

containment. This instruction replaces the signature update operation used by FastSig and avoids the need to maintain a signature register. It is similar to a logical (LOP) operation except that it does not need a destination register. Hardware changes, as summarized in Figure 4.4, include instruction decoder support for the new operation and some logic in the register write-back stage to raise an exception based on the results of a bit-wise equality check. Since current GPUs support exception reporting and handling [136], HW-Notify can leverage this existing framework for traps. Figure 4.3(4) illustrates how the instruction is used.

HW-Sig: This technique eliminates all verification and notification instructions. As mentioned in Section 4.3.3, maintaining one register per thread requires significant on-chip storage (10s of kB/SM) because an SM supports thousands of threads. We propose using just one signature register per hardware lane (not per thread) per context. Since instructions can write to one or two 32-bit registers, we propose using a 64b signature register. We initialize the signature register to zero at the kernel launch time (using a synchronous reset signal) and ensure that it is zero at the end of the kernel. As each instruction executes, it updates the signature register by adding or subtracting its destination register values based on whether the instruction is original or duplicate, respectively. Operations that are commutative, easy to design in hardware, and require low area overhead are good candidates for signature updates. For example, binary Galois Field arithmetic (GF(2)) that uses *XOR* operations can be used for signature accumulation and subtraction [137]. We need one extra meta-data bit in the instruction to indicate whether the signature register should be updated by the instruction. When HW-Sig is employed with SRIV, the duplicate instruction only updates the signature register; its destination is replaced with *RZ*. The signature update logic need not be in the critical path and can be performed in parallel with the write back stage while the result of the instruction is being written back to the register file. Figure 4.4 summarizes the hardware changes.

At the end of the kernel, we activate the register checking logic using a global signal. If the value is non-zero, an exception is raised. This approach trades off the ability to detect the error until the end of the kernel and diagnose which thread is corrupted, which is not a concern for existing coarse grained checkpointing solutions [121, 122, 123, 124].

Storage overhead can be reduced by accumulating the ECC bits of each result, instead of the result itself. Hence the signature register only needs to be as wide as the error code (e.g., 7-bit single-error correction, double error detection (SEC-DED) is used for the 32-bit GPU registers [110, 112]). The signature update can take place in a pipeline stage following ECC encoding without performance concerns because this logic is not in the critical path of the datapath.

A hardware error can be missed if (1) both the original and duplicate instructions see the same corruption in their results, which is not possible for single instruction error model, or if (2) a single error propagates to an even number of instructions in the same thread and these affected instructions update the signature register such that the observed errors happen to cancel out. The second scenario is impossible for SRIV (duplicate instructions do not write to registers), and highly improbable for DRDV because the conditions are challenging to meet. The likelihood of this scenario can be reduced by choosing a signature update function that is less likely to cancel errors (e.g., one's complement add as opposed to GF(2) XOR).

4.5 EVALUATION METHODOLOGY

Our experimental flow targets NVIDIA Pascal (Titan-Xp) architecture-based GPUs with Compute Capability 6.1 [112]. We modify NVIDIA's production back-end compiler and use it with the CUDA 8.0 toolkit. The host system has an Intel i7-3930K CPU (3.2 GHz) and 32 GB of system memory. We evaluate SInRG using 16 workloads, 15 of which are from the Rodinia benchmark suite (version 3.0) [138]. The last workload is the matrix multiplication program (referred as *mm*) provided as a sample in the CUDA 8.0 toolkit.

4.5.1 Performance Metrics

We measure runtime overheads by running workloads directly on the system with the GPU. For the application-level runtime, we take the average time from five consecutive runs after a warm-up run. We obtain the GPU kernel-level runtime by analyzing a GPU execution trace that contains the times when the kernels are launched and their duration.

The *--print-gpu-trace* option for the *nvprof* tool prints this trace. We exclude the time spent copying data between the GPU and host memory.

To understand the source of slowdowns, we collect the total number of dynamic instructions, number of spill/fills, increase in register usage per thread, warp occupancy, warp execution efficiency, and stall reasons using *nvprof*. We measure the increase in the binary file size as a secondary overhead metric.

Evaluating the optimizations that require hardware support: We implement the modifications needed for HW-Notify and HW-Sig in NVIDIA's production compiler and measure the performance overhead using real GPUs. Since these techniques propose using new ISA extensions that are not available on current GPUs, we generate instructions that are closest in term of performance and functionality to measure expected runtime overheads. For example, we generate `LOP` with a dummy destination register (RZ) in place of the HW-Notify compare-and-trap instruction. For HW-Notify, we remove the notification instructions and the control-flow to branch around them. For HW-Sig, we remove all the signature update instructions such that there are no verification and notification instructions.

Comparison to Thread-Level Duplication: We quantitatively compare SInRG to a prior competitive GPU software-based solution – thread-level duplication (TLD) [125, 126, 127]. We implemented a TLD algorithm that is similar to the *Intra-Group-LDS FAST* configuration from [126] and the *Intra-Permute* configuration from [127], which is the most aggressive organization that they consider. On every memory write, TLD communicates the address and value to the neighboring redundant thread using a `SHFL` instruction, compares them with the local values, and notifies higher layers on an error (where only the redundant thread performs this last task). We also implemented an optimization called *TLD-Sig* that defers the notification until the end of the function using a predicate signature register (not explored by prior work). A thorough exploration of TLD optimizations is beyond the scope of this work.

4.5.2 Coverage Metrics

Dynamic instruction coverage: We measure the fraction of dynamic instructions in a program that are assumed to be protected by SInRG. We measure this by first categorizing instructions at compile-time by modifying the back-end compiler as follows. (1) All duplication-eligible instructions are categorized as *covered original.* (2) All duplicate instructions are categorized as *covered duplicated.* (3) We categorize verification and notification instructions as *verification.* An error in these instructions will likely result in a verification failure, assuming only a single fault occurs during a program run. (4) Instructions that are not duplicated are categorized as *uncovered.* We conservatively mark instructions as uncovered if we cannot identify an instruction that covers the instruction following compiler transformations that are performed after the duplication pass. (5) Remaining instructions, mostly consisting of register spills and fills, are categorized as *others.* Since most of the registers are duplicated in DRDV, a corruption during a spill or fill will likely be detected by the code that verifies the register value once it is filled. An error in a spilled register that is never filled has no consequence.

We next obtain dynamic instruction counts per static instruction using a binary instrumentation tool, which is similar to SASSI [139]. Combining this data with the above instruction categories, we obtain the dynamic instruction coverage. This metric assumes that all instructions have equal vulnerability.

SDC reduction: We conducted architecture-level error injections for all of our workloads using a modified version of the SASSIFI tool [140]. We injected single-bit flips into the destination registers of randomly selected SASS instructions (one error per run). This methodology, unlike dynamic instruction coverage, accounts for architecture-level propagation. We observe no error detections during error-free runs, which confirms that SInRG's false-positive rate is zero. We calculate 95% confidence intervals using the Wilson score interval and find that all intervals are less than 5% of the estimated mean. We modified some of the workloads such that SDC identification is feasible, which include printing the final result and fixing the random number generator's seed for deterministic runs.

We also conducted accelerated high-energy particle beam experiments to quantify the effect of employing SInRG on the true SDC rate at the full GPU level. Accelerated particle beam testing is one of the most accurate and widely-accepted methods of measuring FIT. We conducted the experiments at a proton facility with particle energy >200MeV. We used a Volta-based GPU [111] with ECC enabled and targeted the entire GPU package. We used our modified back-end compiler with the CUDA 9.0 toolkit and recompiled the workloads for Compute Capability 7.0 without any technical challenges, which demonstrates that SInRG algorithms are portable across toolkits and applicable to different architectures. Due to the statistical nature of the experiments and limited availability of beam time, we studied the FIT rate reduction for only the matrix multiplication workload. We used two DRDV versions: one based on the Section 4.4.2 (DRDV) and a second similar version that duplicates loads using a function-level heuristic. This heuristic marks all loads as deterministic based on the non-existence of an atomic operations in the function, which was appropriate for this workload. These two versions, referred to in Section 4.6 as *DRDV* and *DRDV with LD dup*, respectively, allow us to understand the effect of not duplicating most loads.

4.5.3 Area Costs and Effectiveness Analysis

We implement Verilog models of the structures needed for HW-Sig to estimate their hardware costs. The circuits are synthesized with the Synopsys toolchain using a 16nm industrial technology library [141]. We estimate circuit area using a NAND2 gate-equivalents metric. We conducted gate-level error injections to evaluate the efficacy of using a SEC-DED accumulator with HW-Sig, as a single error in the pipeline can propagate to many erroneous bits, and the SEC-DED code can alias if more than three bits are erroneous. We use the Hamartia framework [66] to flip the the output of a single gate or flip-flop per injection. This methodology is similar to that of Nedel et al. [142], though we use netlist rewriting to simulate errors without modifying the gate-level simulator. Our results are based on six unpipelined DesignWare components [143], using random inputs.

4.6 RESULTS

4.6.1 Software-Only Techniques

Performance: We begin our evaluation by measuring the performance overheads of the baseline SInRG versions. As the baseline SRIV incurs very high overheads (>100% for all our workloads), we do not analyze it further. The GPU kernel runtime for DRDV incurs an arithmetic average overhead of 69%, as shown in Figure 4.5. Employing the software-only FastSig optimization reduces the average runtime overheads to 39% and 49% for DRDV and SRIV, respectively. Workloads with low baseline IPC have more potential for improvement when employing SInRG. Underutilized resources (due to memory operations, hazards, poor code, etc.) can cause low IPC, which SInRG exploits by hiding duplication overhead. To understand the effect of SInRG's runtime overheads on different architectures, we evaluated FastSig-DRDV on a Volta-based GPU for a subset of workloads and observed similar overhead trends.

Our results show that the average application-level runtime overheads are only 6%, 4%, and 5% for baseline DRDV, FastSig-DRDV, and FastSig-SRIV, respectively. This is much lower than the GPU kernel-level overheads because these runtimes include time spent on host and copying memory.

FastSig-DRDV outperforms FastSig-SRIV for some workloads, such as *lud* (18% versus 59% overhead). This phenomenon can be explained by the difference in dynamic instruction count, which is 33% higher for FastSig-SRIV. FastSig-SRIV has better performance for some workloads, despite executing more dynamic instructions. For example, the runtime overheads

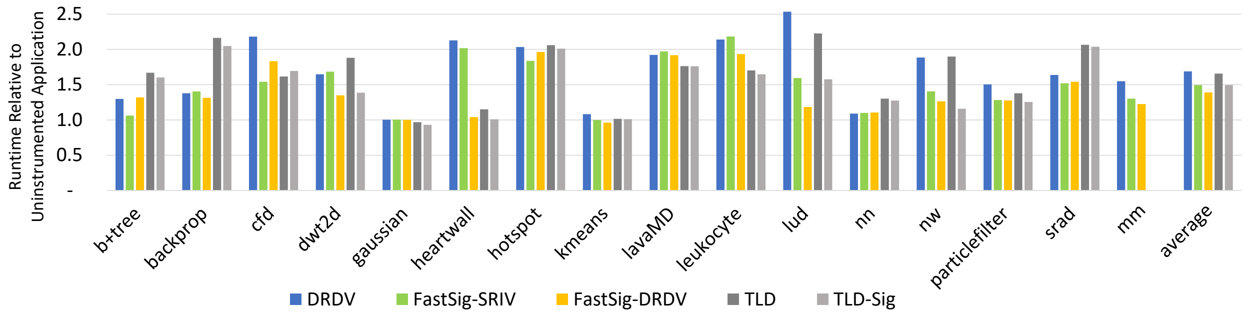


Figure 4.5: The runtime overheads of the base DRDV along with optimized software-only FastSig SInRG versions.

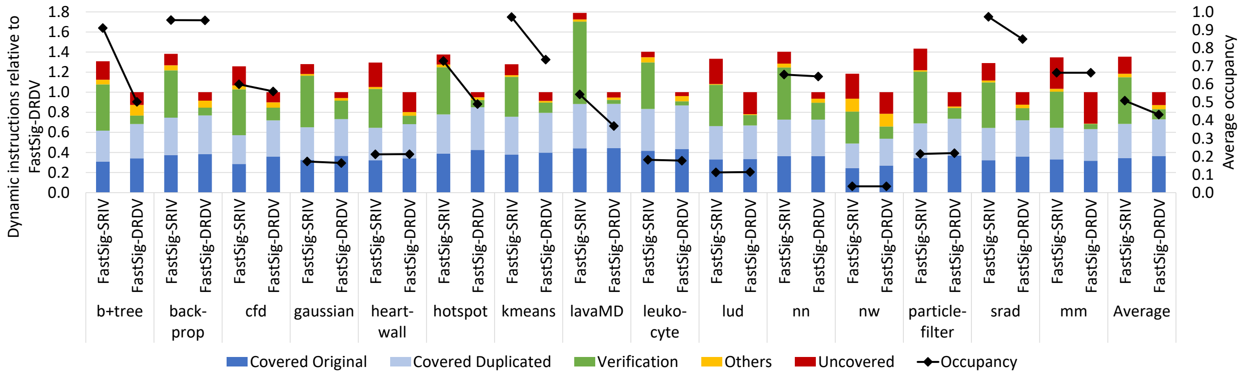


Figure 4.6: Dynamic instruction-class counts. The secondary y-axis shows the average warp occupancy of each workload. The fraction of uncovered instructions is generally small, as shown by the top segment of each bar.

for *b+tree* are 32% versus 6% for FastSig-DRDV and FastSig-SRIV, respectively. Although the SRIV version executes more dynamic instructions, the warp occupancy is 1.82× higher, resulting in better overall performance.

Figure 4.6 further explains the dynamic instruction count increase. The results are normalized per workload to the total instruction count of FastSig-DRDV and show that FastSig-SRIV always executes more instructions than FastSig-DRDV. As discussed above, this is not the only indicator of performance. Warp occupancy, which is also plotted in Figure 4.6 on the secondary axis, is almost always reduced by using FastSig-DRDV. This relative decrease correlates well with the relative runtime overhead increase (Figure 4.5) compared to FastSig-SRIV. In summary, selecting an instruction duplication algorithm that is aware of the GPU resource requirements for a workload can provide better performance.

Comparison to thread-level duplication: If a workload has spare thread and register resources, it can benefit from TLD beyond what SInRG offers because SInRG does not exploit spare thread resources. This is the case for *lavaMD* and *leukocyte*, as shown in Figure 4.5.

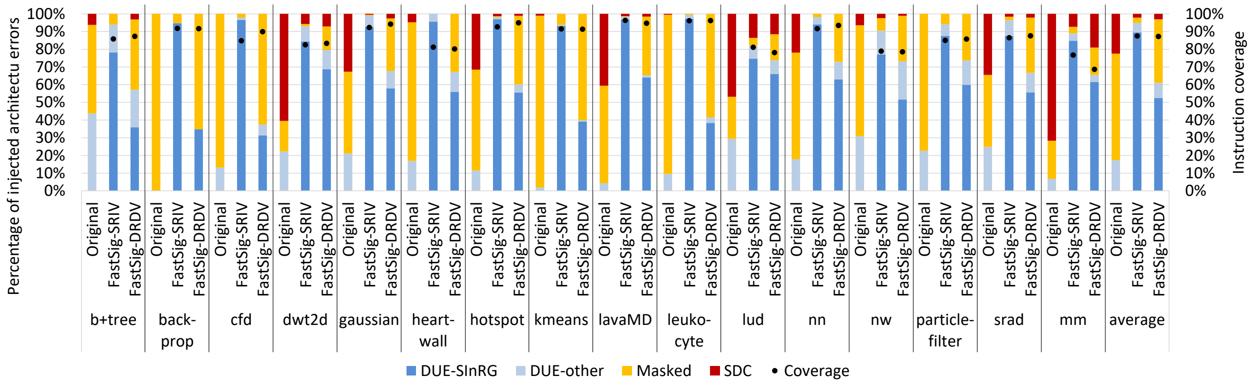


Figure 4.7: Architecture-level error injection results. Original refers to uninstrumented program.

These workloads exhibit low warp occupancy (Figure 4.6) for both FastSig versions, since they are not register resource limited.

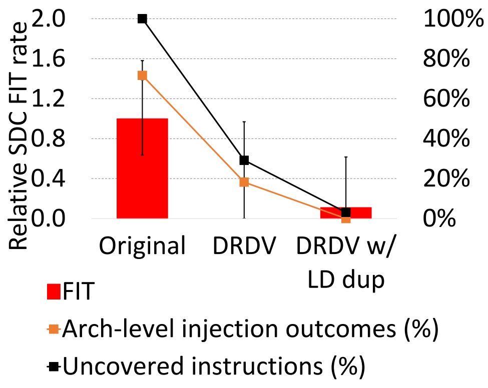
As mentioned in Section 4.5.1, TLD-Sig optimizes TLD by deferring error notification until the end of the function, which reduces the runtime overheads for most of the workloads. Results show that despite this optimization, one of the FastSig optimized SInRG versions outperforms TLD-Sig for a majority of the workloads.

Code bloat: The average increase in the program binary file size, which includes non-duplicated host code, was a modest 12% for FastSig-DRDV and 19% for FastSig-SRIV. SInRG's static instruction overhead ranges from 74%–115% for FastSig-DRDV and from 180%–227% for FastSig-SRIV. The overheads for FastSig-SRIV are relatively higher because it adds more verification instructions.

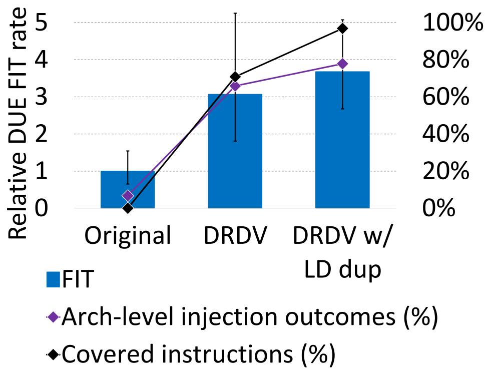
Dynamic instruction coverage: Results in Figure 4.6 show that the original programmer-defined instructions (other than compiler-inserted spill and fill code) account for an average of 36% and 25% of the total dynamic instructions for FastSig-DRDV and FastSig-SRIV, respectively. The duplicated instructions account for a similar fraction. The percentage of verification instructions varies significantly based on the workload and the algorithm. As expected, the average percentage is 2.4× more for FastSig-SRIV compared to FastSig-DRDV (35% versus 15%). While a small fraction of instructions are categorized as *others* for most workloads, spills and fills increase the prevalence of this instruction class for some register constrained workloads. We assume all the above instructions are covered by SInRG. Finally, the fraction of uncovered instructions also varies by workload and depends on the prevalence of control, global and shared memory reads, and atomics in the program. On average 88% and 87% of the dynamic instructions are considered covered by FastSig-SRIV and FastSig-DRDV, respectively.

SDC reduction through architecture-level error injections: Figure 4.7 shows architecture-level error injection results for the uninstrumented programs and the two SInRG versions. It shows that SInRG is effective in reducing the SDC percentage. We expect the dynamic instruction coverage, plotted on the secondary y-axis, to correlate well with FastSig-SRIV's DUE (Detected Unrecoverable Errors [144]) percentage because FastSig-SRIV performs immediate verification, providing no opportunity for error masking. Since the error notification is delayed, some of the errors may result in DUE-other (crashes/hangs) prior to being flagged by SInRG as DUE-SInRG (e.g., b+tree, lud). FastSig-DRDV provides opportunity for masking until the end of the data-flow chains, lowering the expected DUE rate. The results clearly show this trend — the SDC percentage is always lower than the percentage of uncovered instructions.

SDC reduction through accelerated particle testing: Figure 4.8a shows the effectiveness of SInRG in reducing the GPU SDC FIT rate while running the *mm* workload. The observed SDC rate for both SInRG versions is an order of magnitude lower than the uninstrumented program. Figure 4.8b shows that the DUE FIT rate increases with SInRG; we observed the evidence of several SInRG error detections in the system logs. These results establish that SInRG is effective in significantly improving the reliability of GPUs.



(a) SDC



(b) DUE

Figure 4.8: SDC and DUE FIT rates for the mm workload, normalized to the original mm. Architectural injection and instruction coverage results are plotted on the secondary y-axis.

We plot the dynamic instruction coverage and architecture-level error injection results in Figure 4.8 (on the secondary y-axes) to analyze the trends. These results, however, cannot be directly compared to FIT rates because these methods estimate the program-level error propagation probabilities once the error has manifested at the architecture level.

Figure 4.8a shows that the percentage of uncovered dynamic instructions reduces from 100% to 29% and 3% for DRDV without and with load duplication, respectively. The corresponding SDC percentage reduction from architecture-level error injection are 18% and 0%, down from 72%. These trends correlate strongly with each other and the FIT estimates. We noted similar strong correlations for the DUE results.

4.6.2 Optimizations Through Hardware Support

Performance: We apply HW-Notify to the two FastSig versions to overcome their limitation and provide perfect error containment. Figure 4.9 shows the average overheads for HW-Notify-DRDV and HW-Notify-SRIV are 37% and 46%, respectively, which are similar to the FastSig versions.

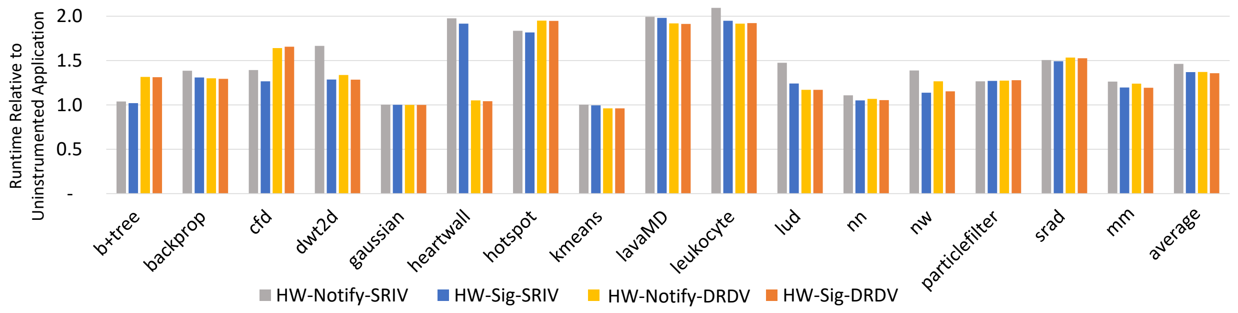


Figure 4.9: Runtime overheads for the hardware-based schemes.

HW-Sig eliminates the verification instructions altogether providing a faster solution. HW-Sig-SRIV is significantly faster than the HW-Notify-SRIV, with average overheads of 33% versus 44%. This improvement is expected because FastSig-SRIV has many verification instructions that HW-Sig eliminates. We did not observe such a high improvement for HW-Sig-DRDV over HW-Notify-DRDV (35% to 33%). Even though the average overheads between HW-Sig-SRIV and HW-Sig-DRDV are similar, we observe significant differences for different workloads. When HW-Sig is employed with DRDV, maintaining the shadow data-flow chain and the shadow register space provides additional instruction scheduling flexibility, which can be beneficial for workloads that are not register limited.

Hardware costs: Table 4.2 gives the circuit area estimates following synthesis, as well as a 32b adder and a SEC-DED encoder for reference. The HW-based SInRG schemes require modest amounts of new hardware per lane—the total area of the new structures is similar to or less than that of an adder, which itself represents a small fraction of the pipeline logic. The 64b HW-Sig accumulator is shown with two signature accumulation algorithms—GF(2) XOR and one's complement. The HW-Sig accumulator is the largest new structure, but it can be efficiently replaced by a SEC-DED accumulator at less than 1/6 (GF(2) XOR) or 1/8 (one's complement) the cost.

Table 4.2: Area costs per lane for hardware extensions.

Structure	Technique	# FFs	Area (NAND2)
32b Zero Detector	HW-Notify	1	19
64b XOR Accumulator	HW-Sig	65	496
64b One's Accumulator	HW-Sig	65	1044
7b SEC-DED XOR Accumulator	HW-Sig	8	73
7b SEC-DED One's Accumulator	HW-Sig	8	130
32b Adder	Reference	96	715
2x 32b SEC-DED Encoders	Reference	14	354

The SEC-DED accumulator potentially has imperfect error coverage because a single-event transient error in the pipeline can propagate to many erroneous bits, and the SEC-DED code can alias if more than three bits are in error. Our gate-level error injection campaign determined this risk to be minimal—over 3,862 logically unmasked errors injected into six fixed-point and floating-point arithmetic units, only one would remain uncaught by the SEC-DED accumulator. This leads to a 95% confidence interval of (0.0%, 0.2%) for the percentage of errors that the SEC-DED accumulator would miss.

We target a 2 GHz clock (assuming 50% margin for uncertainty and unmodeled control circuitry), which is an efficient operating point for more complex functional units such as the multiply-add unit. All of the considered circuits achieve this speed using automatic register retiming.

4.6.3 Automated Duplication Technique Selection

As explained earlier, either DRDV or SRIV can perform best for a specific GPU kernel, depending on its requirements and the available GPU resources. We explore heuristics to automatically select the SInRG algorithm for each dynamic kernel at kernel launch time. We obtain and pass the following information to a machine learning model to predict which algorithm to apply. We pass (1) an occupancy estimate using number kernel specific information such as registers needed per thread, shared memory usage, and thread block size and target GPU resource constraints, (2) the increase in the number of static instructions, and (3) the increase in static spill/fill instructions. Based on our initial study, we find that supervised learning methods such as Decision Tree and Random Forest perform well for this task. Auto-selected duplication algorithm for FastSig, HW-Notify, and HW-Sig reduces the average runtime overheads to 36%, 34%, and 30%, respectively. These are significantly lower when compared to the average overheads obtained by DRDV and SRIV individually for the respective techniques.

4.7 SUMMARY

Software-based instruction duplication is an attractive resiliency hardening technique because it can be employed on state-of-the-art systems and can be selectively applied to resilience-critical workloads. In this chapter, we implement intra-thread instruction duplication on GPUs (inspired by prior CPU work) and find the overheads to be high, averaging 69% over a variety of workloads. We propose several software-only and software-hardware optimizations to reduce the overheads and implement them in NVIDIA's production compiler. Our GPU-specific software optimizations trade off error containment for performance and reduce the average runtime overhead to 36%. We also propose new ISA extensions with limited hardware changes and area costs to further lower the average runtime overhead to just 30%.

While we show that reliability overheads can be reduced with various optimizations in software and hardware, the auto-tuner approach to select the optimal duplication strategy (discussed in Section 4.6.3) indicates that adapting the resiliency solution to the workload can have a major impact on runtime performance and overheads. In general, this is a recurring theme even for Approxilyzer (Chapter 2) and Minotaur (Chapter 3), where having more information about the application can provide more opportunity for optimized resiliency solutions. Thus, a deeper understanding of the application and error propagation at an algorithmic level could also help inform a software-directed resiliency approach to protect against soft errors. In the next two chapters, we perform a deep-dive into a single application domain, deep learning, and show that we can leverage domain knowledge to design scalable tools and techniques for specialized error analysis.

Chapter 5: PyTorchFI: A Runtime Perturbation Tool for DNNs

5.1 MOTIVATION

With the recent advances in machine learning (ML) alongside enabling hardware (such as GPUs [145] and custom ML processors [146, 147]), deep neural networks (DNNs) have quickly become a dominant player in the application space. Today, DNNs are heavily used across many application domains and hardware platforms, ranging from entertainment devices such as personal phones, to stringently safety-critical systems such as perception software in self-driving vehicles.

With the ubiquitous utilization of DNNs across many domains, it is crucial that DNNs operate reliably in the face of errors. There is mounting evidence that even tiny perturbations such as soft errors can cause a DNN to output an incorrect result at the software level [148, 149, 150, 151]. Additionally, recent work in adversarial machine learning has shown that malicious perturbations in the input (and more advanced attacks such as rowhammer within a network), can alter a DNNs execution [152, 153, 154, 155, 156, 157, 158, 159, 160]. On one hand, most of the time an error has a negligible impact on the computation because it either gets masked out entirely (e.g., due to activation functions such as rectified linear or $ReLU$ layers) or does not cause the DNN's decision to cross a decision boundary (a misclassification). On the other hand, there *are* errors which manifest into observable output corruption. It is therefore crucial that developers understand the dependability and reliability characteristics and limitations of their models before deployment in the field.

Developers are currently lacking the tools to study the impact of perturbations on DNNs. In order to detect and mitigate hardware errors which can propagate and affect DNN outcomes or malicious adversarial perturbations in the network, researchers and developers alike need accurate tools for assessing DNN reliability in the face of different error types. Such tools must be easy-to-use (for widespread adoption by researchers and developers), extensible (to keep up with the fast moving field of deep learning, while also allowing for the study of different perturbation models), and fast (since DNNs can become very large and there are many possible places within a network for an error to manifest).

In this chapter, we introduce **PyTorchFI** [48], a runtime perturbation tool for DNNs developed in the popular PyTorch [161] framework. PyTorchFI allows users to perform neural network perturbations in weights and/or neurons in convolutional operations of DNNs during execution. Therefore, it enables the study of the manifestation and propagation of different perturbations at the application level. PyTorchFI is designed to be programmer-friendly and easy-to-use: it minimizes the programmer overhead by streamlining the installation process through the `pip` package manager, and provides a simple and intuitive implementation for performing perturbations at runtime. In addition, PyTorchFI is extremely fast with negligible runtime overhead due to its native implementation. Further, PyTorchFI is very versatile: by abstracting the notion of an "error" to that of a "perturbation" and designing for the latter, PyTorchFI can enable many additional research applications beyond just reliability.

This chapter focuses on presenting and discussing the technical underpinnings and design decisions of PyTorchFI, which are key to making it an easy-to-use, extensible, fast, and versatile error analysis tool. Additionally, we showcase multiple research use cases for PyTorchFI, and subsequently perform a deep dive into convolutional neural network (CNN) resilience analysis and hardening in Chapter 6.

5.2 BACKGROUND

5.2.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a class of deep neural networks (DNNs) used to analyze visual imagery such as image recognition or object detection. We primarily consider CNN resiliency due to their prevalent use across many safety-critical applications [162, 163, 164, 165, 166].

A classification CNN takes as input an image which propagates through many computational layers until it arrives at a *softmax* layer. The softmax provides a probability for each class the network aims to predict, indicating the confidence of predicting a specific class. The class with the highest confidence (the *Top-1 confidence*) indicates the CNN's prediction for the image. During training, a cost function (such as the cross entropy loss) is computed from the softmax and backpropogated to update weight values to improve prediction accuracy. At deployment time, a classification CNN operates in feed-forward mode only to perform an *inference*.

A CNN is composed of various layers between the input and the softmax. The predominant layer type of a CNN is the convolutional (conv) layer, and typically constitutes more than 90% of a CNNs total computations [167]. A *neuron* (also called an activation value) is the fundamental component of a conv layer, computed as a dot product between a filter of weights and an equal-sized portion of the input. Each dot product is composed of many multiply-and-accumulate (MAC) operations. A plane of neurons is known as a feature map (or fmap for short). Each conv layer in the network may have a different number of filters, which map one-to-one with the number of output fmaps for that layer.

5.2.2 Other Tools

Two related error injection tools that offer similar capabilities are Ares [168] and TensorFI [169], designed to operate within the Keras [170] and TensorFlow [171] frameworks, respectively. These two tools allow modifying the state of the layers in DNNs as they are executing. However, Ares requires changes to the Keras inference computation to introduce dynamic perturbation, which is required for most dependable research studies. TensorFI requires the users to update a configuration file in addition to making modifications to the TensorFlow program. While Keras and TensorFlow are commonly used deep learning frameworks, PyTorch has emerged as a popular framework for DNN research for its ease-of-use [172] and its use of dynamic graphs for DNN computations, which is extremely powerful for understanding and debugging DNN models.

Apart from filling the gap for the PyTorch framework, we addressed the limitations of the prior tools and offer a tool that is fast and easy-to-use. Specifically, we address the issue of portability and longevity of the tool by implementing PyTorchFI in Python 3 rather than Python 2 [169] (Python 2 is no longer support as of January 1, 2020 [173]); we support injecting errors during both inference and training (we present a use case in Section 5.4.3); and we minimize the programmer overhead via a simplistic API which does not require model modifications. Additionally, PyTorchFI is still extremely fast as it operates at roughly the same native speed of PyTorch on silicon.

5.3 TOOL DESCRIPTION

At a system level, PyTorchFI is a lightweight tool built on top of PyTorch [161] which enables perturbations on weights and neurons of DNN models for perturbation analysis with use cases including hardware resiliency, adversarial attacks, robust DNN design, and interpretability. This section explains the design choices and implementation details which make PyTorchFI an easy-to-use, extensible, fast, and versatile tool for error perturbations in DNNs. An overview of PyTorchFI is illustrated in Figure 5.1.

5.3.1 Design Choices

Dynamic perturbations for neurons can be implemented in several ways within the PyTorch framework. The simplest implementation is to append an intermediate layer after every convolutional layer, and apply a transformation layer to perturb output values before proceeding to the next layer in the network. Studying the effects of different perturbation

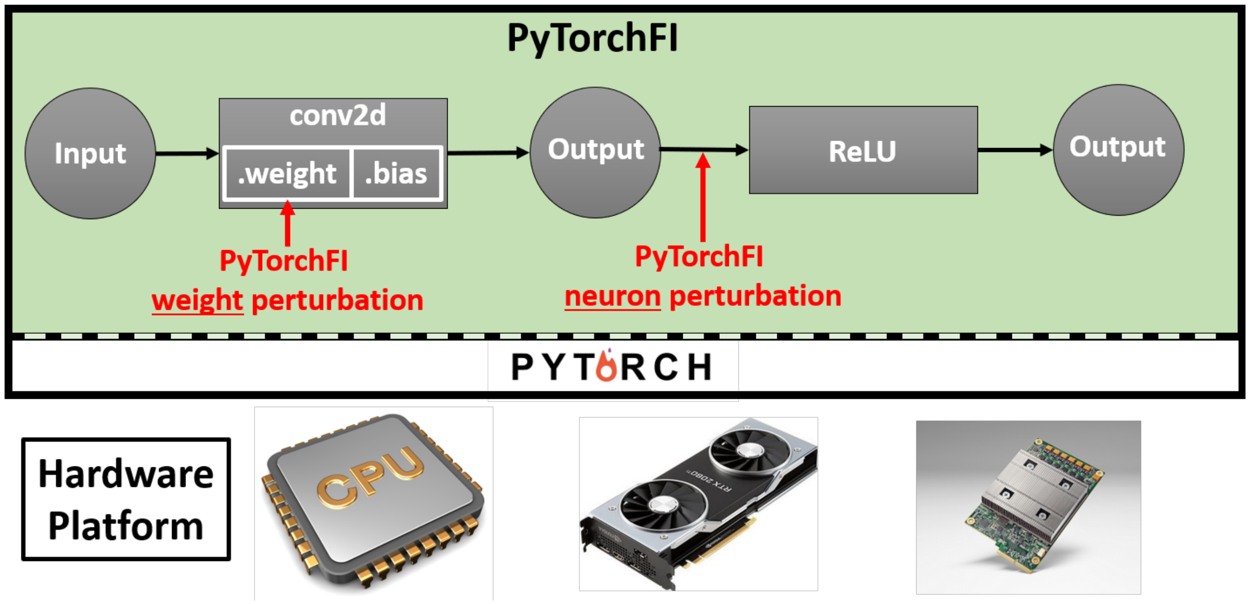


Figure 5.1: PytorchFI [48] is a lightweight tool built on top of PyTorch [161] that enables error perturbations for research into different domains of deep learning. Perturbations in weights are performed offline by modifying the weight tensor, while neuron perturbations are implemented using `hooks` on convolution operations.

models using this method would require major alterations to the network configuration. For deep networks with many layers, or networks with custom layers in-between convolutions, making the modifications to the model for this approach will require non-trivial effort for the user.

Another option is to modify the PyTorch source code to intercept the computation of the neuron to perturb it. This method suffers from a lack of portability because it may require separate implementations for convolutions on CPU, GPU, and other backends. It would require patching scripts and developer maintenance for future versions of PyTorch.

Rather than modifying the network topology or the PyTorch source code, we utilize PyTorch's `hook` functionality to perturb neuron values during the forward pass of a computational model. By leveraging the `hook` application programming interface (API) to instrument error, PyTorchFI avoids altering any source code of PyTorch while also enabling compatibility with future PyTorch versions. Furthermore, it allows the perturbation to run at the native speed of PyTorch, with minimal introduced overheads (overheads depend only on the code introduced for perturbation – the instrumentation methodology introduces nearly no runtime overhead).

5.3.2 Implementation

Identifying hooks as the best candidate for instrumenting neuron perturbations is an important step to ensure that PyTorchFI is fast, extendable, and easy to integrate with existing implementations. For weights, we further optimize PyTorchFI by providing wrapper functions that directly modify the weight tensor before an inference, effectively perturbing weights offline and away from the critical path (Figure 5.1). This optimization translates to no runtime overhead for weight perturbations.

PyTorchFI was designed from the ground up for minimal programming overhead for the programmer. As a result, a researcher can begin using PyTorchFI by following just three steps: (1) importing PyTorchFI, (2) initializing their model, and (3) performing a perturbation with a custom or provided default error model. The following are the steps to install and use the tool:

1. **Installing and importing PyTorchFI**: PyTorchFI has been published to the `pip` package manager of Python, an extremely popular method for managing libraries such as `numpy` and `scikit-tools`. This makes PyTorchFI easily accessible, and requires no compilation or configuration scripts. Importing the tool is as easy as using *import* in the beginning of the code.

2. **Initialization:** Initializing PyTorchFI takes the model for which perturbations will be performed. Other arguments include input image height and width, and optional parameters such as batch size, model data type (e.g., FP32 or FP16), and whether to run on the CPU or GPU. PyTorchFI then performs a single, dummy inference to profile the model and gathers all the hyperparameters of the network, such as the number of layers, filter sizes, and feature map sizes. This information is used for ensuring that perturbations are legal, and to provide detailed debugging messages to the end user.

3. **Perturbation:** The third step involves selecting a perturbation model and a perturbation location. We provide a default set of perturbation models for the user to select from, such as a random value, a single bit flip, or zero value. The user can also easily implement their own perturbation model.

   In addition to the perturbation model, the user needs to specify the location of the weight/neuron that will be perturbed. This can be a single location (specified by the layer, feature map, and neuron's coordinate position in the tensor) or multiple locations to incur multiple perturbations across the network. The user can also specify whether to have the same perturbation across all elements in a batch, or a different perturbation per element.

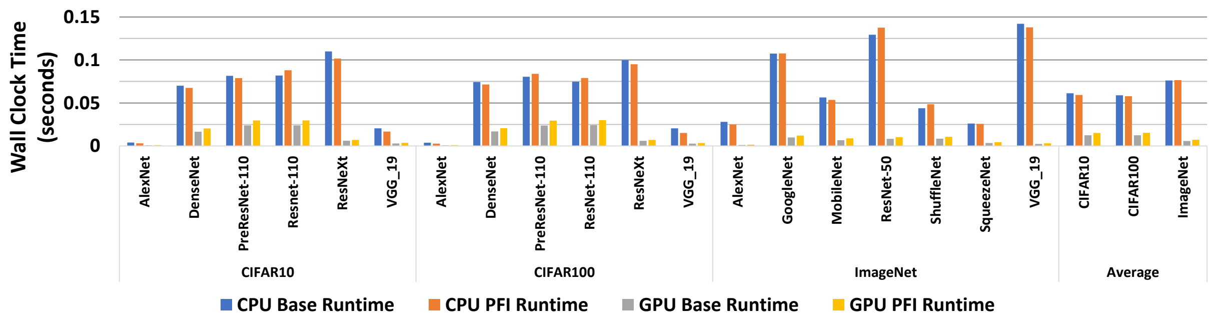


Figure 5.2: Average runtime for 19 networks across three datasets, with and without PyTorchFI (PFI), for a single neuron injection with batch size = 1. PyTorchFI effectively runs at the same native speed on both CPU and GPU with negligible overhead.

The actual perturbation occurs during runtime by taking in the location of the erroneous neuron/weight and appending it to a list of positions in the tensor to change. Then, on every layer, the forward hook will iterate through all of the locations and corrupt the corresponding value based on the selected perturbation model.

5.3.3 Performance Evaluation

PyTorchFI has been tested on PyTorch versions 1.0 through 1.6. We expect long term support for PyTorchFI, as hooks are becoming first class objects in the PyTorch environment: they have been explicitly mentioned in every PyTorch release [174] and are widely used within the PyTorch ecosystem.

PyTorchFI's implementation has extremely low overhead since there is only a single check on every layer. If there are no perturbations defined, then there is no overhead. It also scales very well, since the same hook can be used to inject single or multiple error within the same operation.

To evaluate the runtime overhead introduced by PyTorchFI, we measured the runtimes of pretrained DNNs with and without perturbations introduced by PyTorchFI. We ran our experiments on two hardware platforms – for CPU, we used an AMD EPYC 7401 processor with 1 TB of RAM and for GPU we ran on an NVIDIA Titan Xp with 12 GB of RAM. We used the default perturbation model provided by PyTorchFI (a uniform, random value between [-1,1]) on a random neuron location for random input images. We averaged the runtime across 1000 trials for each network.

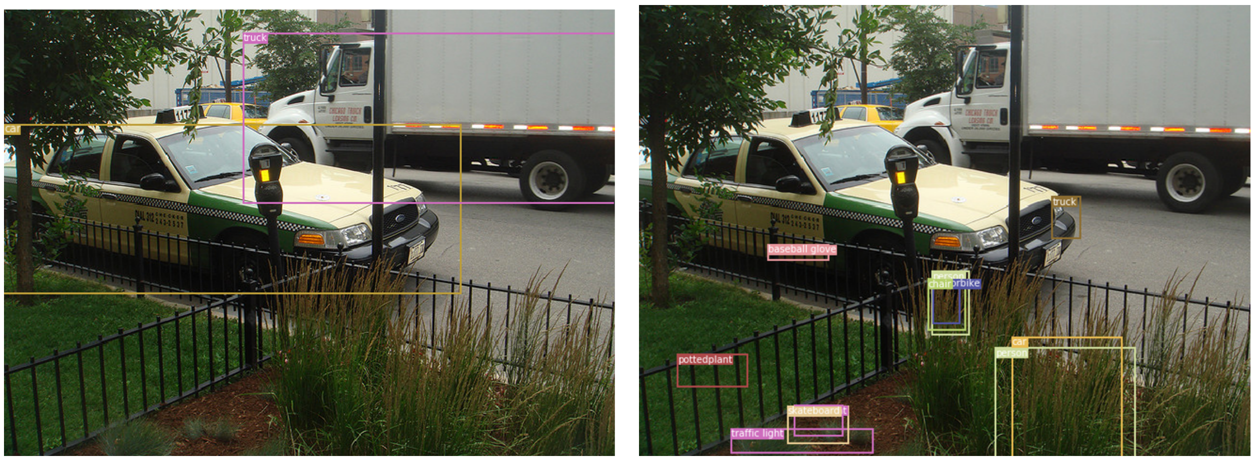
Figure 5.2 shows the runtime results. We see that all inferences (with and without PyTorchFI) typically take less than 0.2 seconds for both CPUs and GPUs. As GPUs are known to offer higher throughput for deep learning workloads compared to CPUs, the GPU runtimes we observed were a lot faster. More importantly, what we find is that the runtime with perturbations differs by less than 10 millisecond in wall-clock time across both platforms, all models, and datasets. Further, we also performed a study of PyTorchFI using inference batching (a common practice for some DNN inference applications). We swept the batch size from 1 to 512. We observed the same trend: the wall clock time overall went up (as batching takes longer to run than for a single inference), while the runtimes with and without PyTorchFI were comparable and within the error margins, indicating an amortized cost per model for instrumenting perturbations. Thus, PyTorchFI is extremely fast, effectively operating on the native speed of the underlying hardware platform.

5.3.4 Limitations

PyTorchFI operates at the application level of DNNs, which is useful for modeling high level perturbations and understanding their effect at the system level. Lower level perturbation models, such as register-level faults, cannot be captured at this level. However, we can still use PyTorchFI to model lower level faults by mapping them to either single- or multiple- bit flips (in single or multiple neurons) [175]. Recent studies have shown that high level models can be used to study the effect of errors at the system level [64, 176]. At the same time, higher level models can run 4-6 orders of magnitude faster [65] compared to low-level implementations [177]. We show that PytorchFI runs at the native speed of silicon, as it requires no code instrumentation for error modeling. This enables a faster exploration of the large state space which is crucial for understanding real-life aspects of errors in safety-critical applications.

5.4 PYTORCHFI USE CASES

With the use of a domain-specific perturbation tool such as PyTorchFI, we can go beyond just resiliency analysis for classification networks (the primary focus of Chapter 6 and a major contribution of this thesis), and explore other interesting research problems in the domain of deep learning. In this section, we demonstrate PyTorchFI's versatility as a research tool by showcasing four additional use cases: 1) object detection resiliency, 2) robustness of adversarial training techniques, 3) training error-resilient models, and 4) DNN interpretability. While these are not the only uses of PyTorchFI, we show these to illustrate the importance and generality of the tool. Our goal is to demonstrate the uses of the tool and not to fully address challenges in each of the areas covered by the use-cases.



(a) No perturbation (b) PyTorchFI perturbations

Figure 5.3: Perturbations on YOLOv3 object detection network

5.4.1 Resiliency Analysis of CNNs used for Object Detection:

We used PyTorchFI to study resilience of object detection networks, exploring another class of DNNs widely used in autonomous vehicle systems. Object detection is more complex than image classification: it combines both the object localization and classification problems. Thus, the definition of an output corruption in this context changes dramatically from a Top-1 misclassification for a classification network.

Using PyTorchFI, we perturb multiple neuron values (one neuron perturbation per layer, each with a uniformly chosen random FP32 value) and study the effect on the inference output. Figure 5.3 illustrates the observed differences. Figure 5.3a depicts a correct inference with the YOLOv3 network [178] on an image from the COCO dataset [179]. In this image, the network identifies two objects (a car and a truck) by placing a border around each object and classifies each of the detected objects. Figure 5.3b shows that the perturbed network can behave irrationally, identifying many phantom objects each of which are classified seemingly arbitrarily. This example illustrates that PyTorchFI can be used to perform perturbations on DNN tasks beyond classification networks and with a different error model than the one used in previous sections. More importantly, it illustrates that perturbations can cause egregious outputs which must be studied for building resilient object detection networks for many safety-critical applications.

Using PyTorchFI, researchers can study the effect of perturbations across different error models on emerging DNN tasks. As nearly all the perception tasks in autonomous system are being performed by DNNs, it is important to have a versatile tool which can be used to perform detailed resiliency studies.

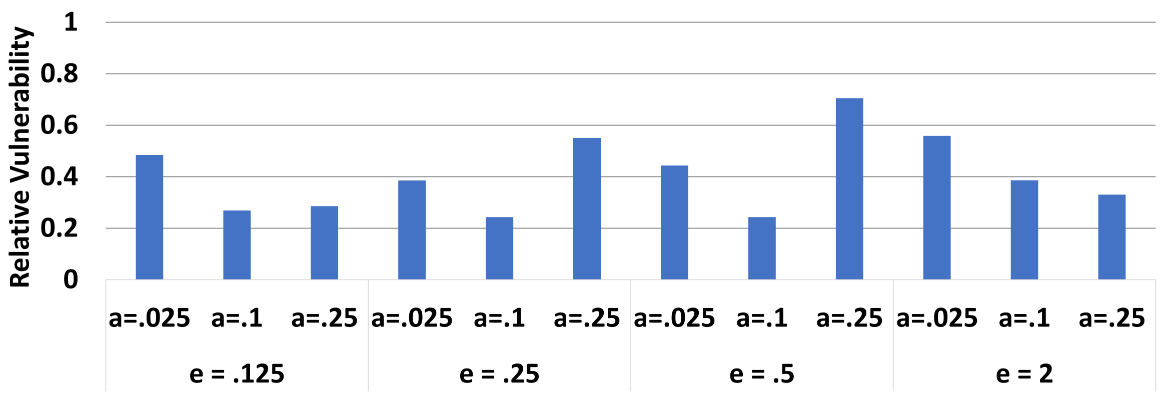


Figure 5.4: Relative vulnerability (compared to a baseline model without IBP) of the first two layers of AlexNet when trained with different IBP parameters.

5.4.2 Resilience Analysis of Models Robust to Adversarial Attacks

In a traditional adversarial attack setting for classification networks, small perturbations in the input layer of a DNN typically propagate through subsequent layers and eventually lead to an incorrect classification [180]. Some of the defense strategies developed to protect a DNN from adversarial attacks aim at limiting the propagation of the perturbation from one layer to the last one. To that end, PyTorchFI can be used to validate that protection against adversarial attacks should make a network inherently more resilient.

We consider the case of AlexNet on CIFAR-10 [181], and train a version of AlexNet through the Interval Bound Propagation (IBP) approach [182]. For a perturbation with a maximum $L_\infty = \epsilon$ norm in input, IBP computes the corresponding minimum and maximum probability of each class in output. Training is performed by minimizing the cost function

$$J = \sum (1-\alpha)log(p_{win}) + \alpha log(p_{win} - \delta p_{win}(\epsilon)), \tag{5.1}$$

where $\sum log(p_{win})$ is the traditional cross-entropy loss function, whereas $\sum log(p_{win} - \delta p_{win}(\epsilon))$ is the worst-case cross entropy, computed when the DNN is under attack and the magnitude of the attack is ϵ. For training, we follow the procedure for AlexNet in [183], but minimize the cost function in Equation 5.1. To guarantee stable convergence, we use curriculum learning as described in [182], and we scale linearly both α and ϵ from 0 to their respective maximum values from iteration 41 to iteration 123. We consider different values of $\alpha = \{0.025, 0.1, 0.25\}$ and $\epsilon = \{0.125, 0.25, 0.5, 2.0\}$ as these two parameters affect the robustness of the trained DNN in a different way: increasing ϵ leads to networks that are resistant to large input perturbations, while increasing α gives more importance to the worst case entropy, potentially penalizing the accuracy on clean data.

Table 5.1: Training ResNet18 with and without PyTorchFI for resiliency.

	Baseline	**PyTorchFI**
Training time	2h 8m 33s	2h 8m 57s
Test accuracy	95.50%	95.34%
Post-training output misclassifications (out of 24 million)	10,543	7,701

We used PyTorchFI to analyze the effect of IBP on the resiliency of the network. The results showed improvement in the total resilience after training with IBP. While performing the per-layer vulnerability analysis, we discovered that the first two layers of AlexNet developed higher resilience compared to the rest of the layers. Figure 5.4 summarizes this key finding. This figure shows the vulnerability of the first two layers (defined using Top1-misclassifications) relative to a baseline AlexNet that is not trained with IBP. The analysis with PyTorchFI shows that the IBP is capable of improving resilience by up to 4×: this is a positive side-effect of adversarial training that, on the other hand, decreases the accuracy on clean data by approximately 3%. Our results also show that not all models trained to be robust to adversarial attacks are equally resilient. PyTorchFI enables us to investigate the reason for such differences and eventually develop a method that is robust to adversarial attacks and also highly resilient to hardware errors.

5.4.3 Training for Inherently Error-Resilient Models

Most use cases presented so far assume a trained model, which is then vetted using PyTorchFI for robustness to errors. A different approach towards DNN resiliency is to attempt to build reliability inherently into the network *while* training.

We propose a training procedure where we inject errors during training using PyTorchFI to increase the robustness of the network to errors once deployed. Injecting errors/noise during training can reduce the converged accuracy of the model and increase the training time. Models trained to be robust to traditional adversarial attacks commonly observe such a behavior. In contrast, our initial experiments show that some resilience can be built into the models via an injection-based training method with nearly no change in the model accuracy and training time.

Training, as described in Section 5.2.1, consists of many forward and back-propagation passes. PyTorchFI can be used to inject errors during forward passes during training, where the error model selection can be part of the training protocol.

Incorporating PyTorchFI into training requires minimal modifications — three additional lines of code as described in Section 5.3.2. We integrate one of the built in error models for training, namely, a random neuron per layer is changed to a uniformly random value between [-1, 1] during the forward pass. Evaluations are presented on ResNet18 [184] trained on CIFAR10 [185]. Two models are trained for comparison: a baseline without PyTorchFI, and one with it. Both models are trained from the same initialization conditions for a clean comparison, and no other hyper-parameters are varied.

Table 5.1 summarizes some of the key elements between the two models, which were trained on an Nvidia Titan V. We find that training with PyTorchFI has a negligible impact on training time, where both models completed the same number of iterations on the dataset in the same amount of time. Importantly, integrating PyTorchFI into training does not adversely affect convergence. We find that training with PyTorchFI reduces the accuracy of the final model by 0.16%. Note that convergence time, unlike training time, describes the number of epochs required to reach the final accuracy; training with PyTorchFI does not affect convergence time either.

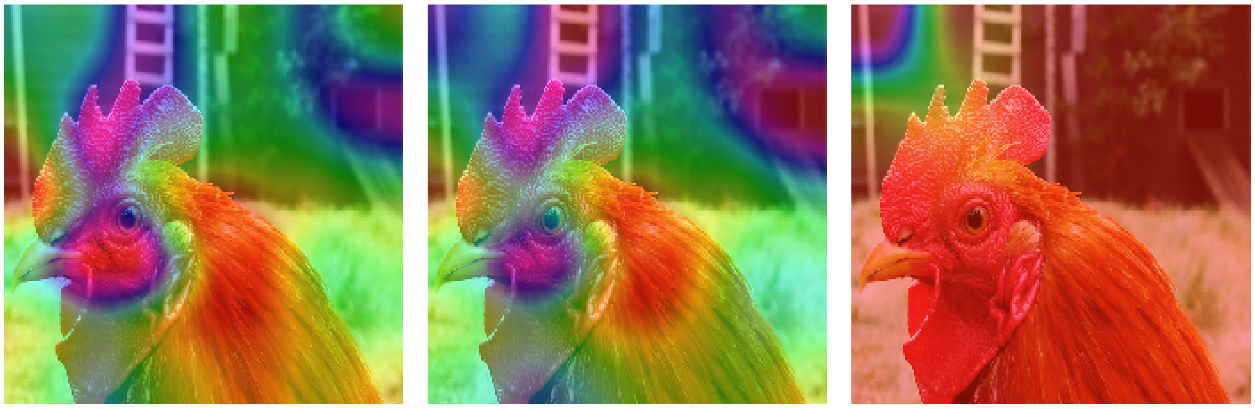
After training, we performed error injections on a separate test set to compare the resiliency of both networks. We measured the number of Top1-misclassifications due to perturbations, and found that the number of misclassifications are *reduced* for the ResNet18 model trained with PyTorchFI.

While these encouraging results show that some robustness can be introduced with no practical change in training time and model accuracy, selecting a different error model and the frequency with which we injection errors during the forward pass (during training) may likely provide different robustness, accuracy, and training time trade-offs. Studying this trade-off space is an interesting future research direction.

5.4.4 Interpretability

One important research question which can provide insight into the reliability and dependability of neural networks is to interpret *how* a DNN works. While prior research has looked into DNN interpretability [186, 187, 188], the field is still evolving and state-of-the-art techniques cannot fully explain the predictions made by the models. We propose a technique which can work alongside the state-of-the-art techniques to assist in DNN interpretability.

One popular technique for visualizing the important input pixels which contributed to a DNN inference is Guided-GradCAM [189, 190]. Guided-GradCAM performs backpropagations starting at different layers to generate gradients for the input, which are then aggregated and visualized based on magnitude. We perform error injections using PyTorchFI in the forward pass of GradCam on specific feature maps, to highlight the effect of a neuron



(a) No perturbation (b) Low sensitivity (c) High sensitivity

Figure 5.5: Visualization of error injections in DenseNet using Grad-CAM [189]. a) shows the original visualization with no perturbation, b) shows a perturbation in the least sensitive feature map, and c) shows a perturbation in the most sensitive feature map.

firing and the affect that a specific feature has on classification. Figure 5.5a shows the superimposed heatmap generated by the Guided-GradCam technique on a correct inference using DenseNet [191]. Figure 5.5b shows the effect of injecting an egregiously large value of 10,000 in a feature map which has little impact on the classification as defined by the gradient values of the feature map. As shown, although the neuron value in this feature map is extreme, the visualization technique shows little difference in the output; this is also corroborated in the softmax where the Top-1 class does not change. On the other hand, perturbing a neuron of a "highly vulnerable" feature map skews the heatmap as portrayed in Figure 5.5c. Thus, an error-injection technique can be tuned to shed insight into the mapping between important input pixels and important feature maps. This simple experiment can perhaps guide a more rigorous iterative algorithm: perturbing a network at different feature maps and observing the effect on the heatmap along with the Top-1 network classification to extract which regions of the input pixels are picked up during inference to arrive at the correct classification of the image. This is an interesting future research direction.

5.5 SUMMARY

PyTorchFI is a runtime perturbation tool for deep neural networks (DNNs), implemented for the popular PyTorch deep learning platform. PyTorchFI enables users to perform perturbations on weights or neurons of DNNs at runtime. It is designed with the programmer in mind, providing a simple and easy-to-use API, requiring as little as three lines of code for use. It also provides an extensible interface, enabling researchers to choose from various perturbation models (or design their own custom models), which allows for the study of hardware error (or general perturbation) propagation to the software layer of the DNN output. Additionally, PyTorchFI is extremely versatile: we demonstrate how it can be applied to multiple different use cases for dependability and reliability research. With the use of PyTorchFI to emulate soft errors at runtime, Chapter 6 presents multiple resiliency analysis and hardening solutions for CNNs, focusing on this domain due its common utilization in many safety-critical applications.

Chapter 6: Domain-Specific Resiliency Analysis Techniques for Deep Learning

6.1 MOTIVATION

General purpose resiliency analysis and hardening techniques, as described in Chapters 2, 3, and 4, provide the important aspects of flexibility and generality in addressing the challenge of hardware errors. The previous chapters have shown that focusing efforts at the instruction granularity for analysis and hardening provides a suitable abstraction level for software-directed hardware resiliency. At the same time, however, a general purpose resiliency solution can underperform (and introduce higher overheads) if it does not take into consideration the executing application, as expounded upon in Chapter 4. While SInRG aimed to use compiled program artifacts (such as register utilization and total instruction count) to select the appropriate duplication strategy for an application, a deeper understanding of the executing application can provide additional insights into how soft errors propagate and corrupt the output of an application. These results pave the way for *specialized* resiliency solutions, which aim to better understand error propagation at the software level, as well as leverage domain-specific knowledge to develop low-overhead solutions. This chapter shows how such an application error analysis can be performed in a scalable way, with a focus on deep learning and convolutional neural networks (CNNs) due to their increased adoption in many high-performance and safety-critical applications.

As described in previous chapters, modern-day techniques to protect against soft errors in deep learning applications use duplication-based approaches, such as dual modular redundancy (DMR) [50, 51, 52]. For example, Tesla's recent Fully Self-Driving (FSD) system deploys two fully redundant DNN accelerator chips along with accompanying redundant control logic, power, and peripheral packaging on the board for reliability [53]. With the increasing computational demands of the perception and planning tasks performed by such systems [192], paying the high overheads of duplication is undesirable, especially if an alternative method exists with similar error coverage but at lower overheads. Moreover, the all-or-nothing protection offered by full duplication may result in over-protection and inefficient use of resources.

In this work, we advocate for software-directed, *selective protection* of CNN models to avoid the high overheads of indiscriminate redundancy. In order to effectively and efficiently perform selective protection, we ask three important questions:

(Q1) At what granularity should the selective protection be performed?

(Q2) Which components at this granularity should be selected for protection?

(Q3) How should the selective protection be implemented?

Answering these questions requires understanding the resilience characteristics of CNNs. Furthermore, by understanding the effect of soft errors on the outcome of CNNs, we can potentially leverage domain-specific knowledge to develop low-overhead reliability solutions and avoid the heavy hammer DMR solution.

Using PyTorchFI [48] (introduced and described in Chapter 5) for analysis and soft error emulation, we introduce two distinct, yet complementary, techniques for selective protection in CNNs. The first technique we propose targets *feature-map level resilience*, which we call FLR (Section 6.3). FLR selectively protects the vulnerable feature maps (fmaps for short) of a CNN before deployment for static, "built-in" resiliency. We find that not all fmaps of a CNN have the same vulnerability to soft errors. By identifying and selectively protecting the highly vulnerable fmaps, FLR helps avoid the uninformed and hefty hammer of full duplication by honing protection efforts on the most important sub-components of the network. However, addressing the second question (*which fmaps to protect?*) requires a typically expensive resiliency analysis of the CNN that involves simulating many error injections and evaluating the outcomes. Traditional approaches measure a binary outcome for an output corruption: either the error caused an output misclassification, or it did not. To accelerate this analysis, we propose a novel, domain specific metric called ΔLoss which converts the binary metric for output corruptions into a continuous metric. ΔLoss speeds up analysis by 3.2$\times$ on average (up to 9.4$\times$) by gathering vulnerability information even when an output misclassification does not occur (Section 6.3.2). Addressing Q3, FLR protects the vulnerable fmap computations by duplicating the corresponding filters in the network. We show that FLR exhibits a sublinear error coverage versus runtime overhead tradeoff. For example, our results show that SqueezeNet [193] can attain 90% error coverage with less than 30% overhead.

The second novel technique we present targets the granularity of each *inference* a CNN performs (Section 6.4). Inference-level resilience, or ILR, is a dynamic, confidence-based resilience technique, which selectively reruns inferences deemed vulnerable to soft errors using only the output confidence values. ILR leverages a key property we discovered between an inference output and the probability a soft error will corrupt the output. Specifically, we find that there is a strong inverse correlation (Spearman coefficient of -0.93) between the difference of the top 2 confidences (called Top2Diff) reported by a CNN's inference output classification and the occurrence of a misclassification (due to a soft error) (Section 6.4.2). We perform resiliency analysis on CNNs to identify network-specific thresholds for Top2Diff (addressing Q2), and perform a logic check after each inference to rerun if the Top2Diff is below the threshold (addressing Q3). Our results show that for the networks studied, we can get 90% error coverage on average with only 17% overhead (as low as 9% overhead for ResNet50).

ILR does surprisingly well: for a given coverage, our results show that its overhead is lower than either DMR or FLR. Even though FLR duplicates a fraction of computations, the duplication decision is static and it happens all the time. In contrast, ILR duplicates a full CNN computation, but is invoked dynamically and less frequently, resulting in lower overall overhead. We consider a novel combination of the two where FLR selectively protects fmaps that complement the selective inference protection by ILR for a target error coverage. Our results show that the combined technique, called FILR (Section 6.5), can obtain 99.78% error coverage with only 48% overhead on average (as lows as 20% for ResNet50, or 5× less overhead compared to full duplication).

6.2 CNN ERROR MODEL

In this work, we focus on transient errors in the hardware that occur randomly during the inference phase of a CNN. We focus on inference since CNNs are usually trained offline once, the model's correctness is verified by measuring model accuracy, and then the model is deployed where the inference task is performed repeatedly with no in-field verification.

As described in Chapter 4, processors deployed in exascale and safety-critical systems employ ECC/parity to protect large storage structures (such as those storing weights and intermediate data) [47]. The level of protection offered by that alone without logic protection will likely not be sufficient, particularly as deep learning continues to grow and delivers high performance computational power to many critical application domains. In this work, we focus on transient computational errors during inference, or simply errors which manifest at a neuron's output. We employ a single-bit flip error model at the neuron level, which is in line with many other studies [69, 148, 194, 195, 196].

6.3 FLR: FEATURE-MAP LEVEL RESILIENCE

As described in Section 5.2.1, a CNN is composed hierarchically: groups of *neurons* form feature maps (*fmaps*) which are grouped into *layers* to comprise a full network. Understanding and quantifying the vulnerability at finer granularities can help avoid full network duplication by enabling selective protection of the most vulnerable components only and reduce overheads compared to traditional DMR techniques.

This section introduces a resiliency analysis and hardening technique called FLR. Given a pretrained network, FLR targets the computational component of fmaps for fine grained analysis and protection (Section 6.3.1), quantitatively estimates the vulnerability of each fmap using a new, domain-specific metric called ΔLoss (Section 6.3.2), then selectively protects the most vulnerable fmaps via filter duplication (Section 6.3.3). FLR is a software-driven technique, enabling a flexible analysis which can subsequently be deployed on various hardware platform backends. Figure 6.1 shows an overview of FLR.

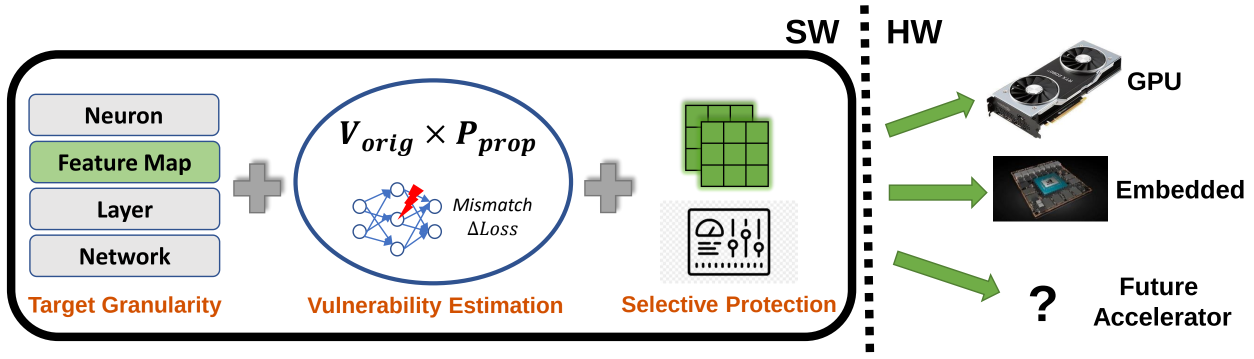


Figure 6.1: FLR design overview. Given a pretrained network, FLR (1) targets fmaps for selective analysis and protection, (2) estimates vulnerability of each fmap, and (3) selectively protects the most vulnerable fmaps in software before deployment.

6.3.1 Target Granularity: Feature Maps

Of the three CNN sub-components (i.e., neuron, fmap, layer), we target feature maps as the sweet-spot for resiliency analysis and hardening for a couple of reasons. First, neuron-level analysis may be *too* fine-grained, with many millions of neurons per CNN (e.g., ResNet50 and VGG19 trained for ImageNet have over 11 and 14 million neurons, respectively). These numbers increase with input size. Evaluating vulnerability of each neuron via error injection will be extremely time consuming. Additionally, neurons are not immune to all translational effects in input images (e.g., rotation, zoom), making this granularity less robust for reliability analysis.

Fmaps and layers, on the other hand, are much more tractable in terms of total components (ResNet50 and VGG19 have 26,560 and 5,504 fmaps across 53 and 16 convolutional layers, respectively), and they are typically trained to have the same behavior across similar images [197, 198]. Performing fmap analysis has the additional benefit that the results can be composed to perform layer- and network-level vulnerability analysis. To the best of our knowledge, this is the first work to target fmaps for vulnerability analysis and selective protection with no retraining and no loss in original pre-trained network accuracy in the absence of hardware errors.

6.3.2 Vulnerability Estimation

FLR quantifies the vulnerability of each fmap in a CNN due to an error by computing the likelihood that an error manifests and propagates to the output and produces an SDC. We compute the likelihood as the product of two components: (1) the *origination vulnerability* (V_{orig}), which captures the likelihood a transient hardware error corrupts the output of an fmap, and (2) the *propagation probability* (P_{prop}), which is the probability the fmap-level manifestation propagates to and corrupts the CNN output. We compute the vulnerability of each fmap, $V_{fmap}[i]$, as:

$$V_{fmap}[i] = V_{orig}[i] \times P_{prop}[i] \tag{6.1}$$

We define the vulnerability of the CNN, V_{CNN}, as the probability that the CNN produces an SDC due to a transient hardware error that occurs during inference. This vulnerability can be computed as the sum of vulnerabilities of each of the N fmaps in the CNN:

$$V_{CNN} = \sum_{i}^{N} V_{fmap}[i] \tag{6.2}$$

Using Equations 6.1 and 6.2, FLR measures the *relative vulnerability* of each fmap in the CNN, or intuitively the contribution of an fmap towards the total CNN vulnerability:

$$Vrel_{fmap}[i] = V_{fmap}[i]/V_{CNN}. \tag{6.3}$$

Error Origination Vulnerability (V_{orig}):

V_{orig} depends on the implementation of the architecture on which the CNN is being run and the computation that generates a feature map (e.g., convolution). Assuming that the major storage structures (e.g., DRAM, caches, and register files) are ECC/parity protected in the target hardware platform [47], most of the errors originate from the unprotected computations. V_{orig} can be computed using the hardware details, the numerical precision of the computation, raw failure rates of the logic and storage structures, and the computation structure.

Given that MAC operations are used to perform a convolution and produce an fmap, we assume that V_{orig} is directly proportional to the number of MACs in a convolution, without loss of generality. In this work, we compute V_{orig} for an fmap as the fraction of the number of MACs used to compute the fmap to the number of MACs in the entire CNN. Our formulation can be extended to compare V_{CNN} *across* different networks, in which case V_{orig} should not be normalized to allow a network-level vulnerability assessment based on the total number of computations performed by different networks. We leave this interesting direction as future work.

Error Propagation Probability (P_{prop}):

P_{prop} is the fraction of the fmap-level error manifestations that propagate to the CNN output, producing SDCs. While the true P_{prop} values for fmaps may not be known, we can estimate them using statistical error injection. P_{prop} can also be calculated using heuristics; however, we explored multiple heuristics and none had significantly high accuracy. We discuss the heuristics explored briefly in Section 6.8.1, and focus our analysis here using error injection metrics.

Number of Mismatches: Counting mismatch from error injections may require many observations for statistical convergence. This metric (although commonly used for its accuracy), suffers from two issues. (1) It is a binary metric, which means that only error injections that change the Top-1 class can affect the P_{prop} measurements. Injection experiments where the softmax changes but not the Top-1 class are not captured by this metric. As a result, estimating an accurate SDC probability requires many injection experiments. (2) The Top-1 mismatch-based SDC metric does not extend naturally to other, non-classification CNNs. This is one of the open problems expressed in the recent survey of DNN resiliency [164]. We address these issues with a new metric to estimate P_{prop} called $\Delta Loss$.

Average Delta Cross Entropy Loss ($\Delta Loss$): Cross entropy loss is traditionally used during CNN training to measure how different the predicted result is from the expected (known) result to improve the prediction accuracy of the network. More generally, it is used in information theory to measure the entropy between two distributions – the true distribution and the estimated distribution. Adapting this metric to resiliency, we can calculate the average absolute difference between the cross entropy loss values observed during an error-free inference and an error-injected inference. This can be expressed as:

$$\Delta Loss_{fmap} = \frac{\sum_i^N \mid (\mathcal{L}_{golden} - \mathcal{L}_i) \mid}{N} \tag{6.4}$$

where $\mathcal{L}_{golden}$ is the cross-entropy loss for an error-free inference and $\mathcal{L}_i$ is the cross-entropy loss for the i^{th} error-injected inference across N total error injections. We use the absolute difference to capture the magnitude of the change in cross entropy loss observed due to an error injection. The larger the average $\Delta \text{Loss}_{fmap}$, the more vulnerable the fmap. Since this method does not predict the SDC percentage, it can be used only to estimate the relative P_{prop}. Figure 6.2 provides an example to illustrate the advantage of using this new metric.

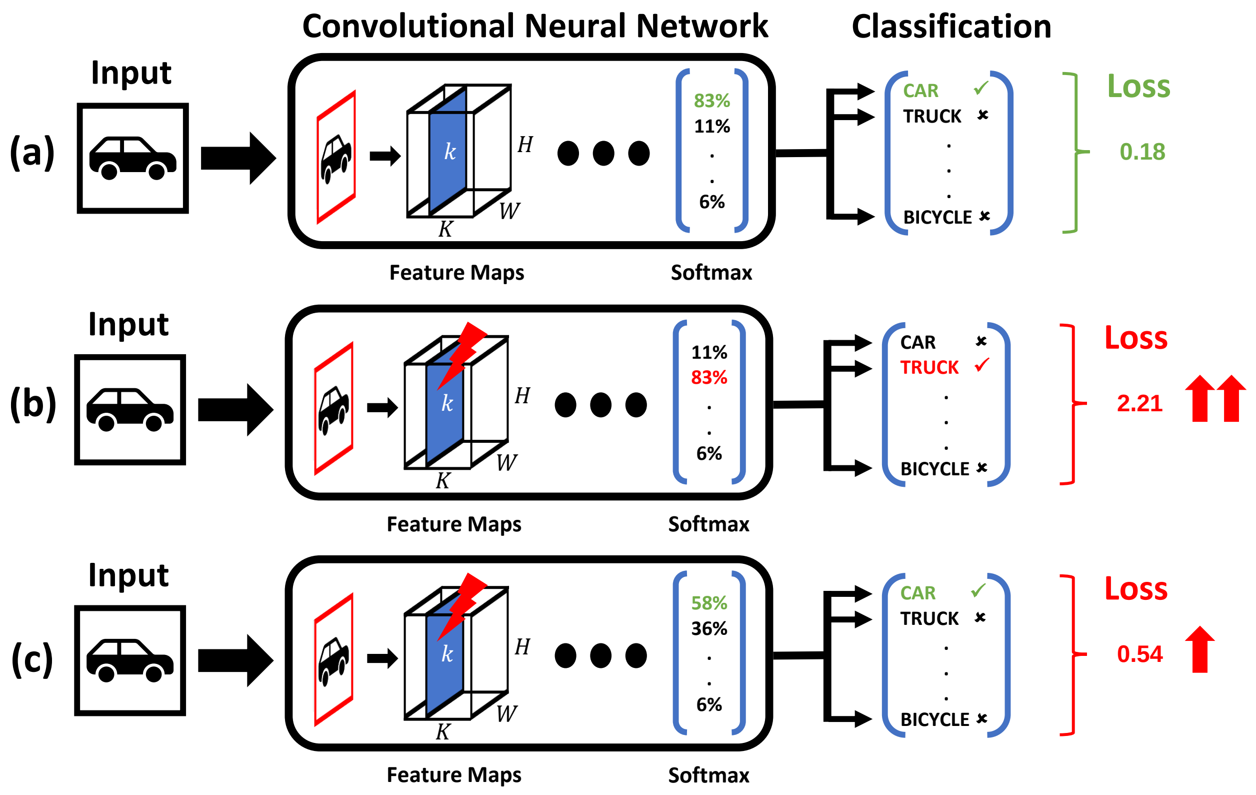


Figure 6.2: ΔLoss example where (a) shows an error-free inference classifying the car correctly, (b) shows an example of a mismatch where an error causes the network to select truck instead of car, (c) shows an example where an error causes a drop in confidence for car that does not lead to a mismatch; however, the drop can be captured by measuring ΔLoss.

6.3.3 Selective Protection

Once the fmap vulnerabilities are quantified, FLR selects the most vulnerable fmaps to harden them from SDCs. Individual fmap computations can be protected by duplicating the filters that correspond to them. Filter duplication results in two copies of the same logical fmap, where any mismatches between the two copies are used to detect errors during inference and trigger a higher-level system response. The duplicated fmaps need to be dropped before execution of the subsequent layer. The comparison of the two duplicate feature maps can be performed lazily to remove it from the critical path. Although this is not the only possible implementation for fmap hardening (e.g., algorithm-based fault tolerant or ABFT techniques [196]), it extends naturally from the relationship between filters and feature maps. Overall, FLR is a highly tunable software-directed selective protection approach, allowing the designer to control the error coverage versus computational overhead trade-off based on the resiliency requirements of the system. We study this trade-off in Section 6.7.

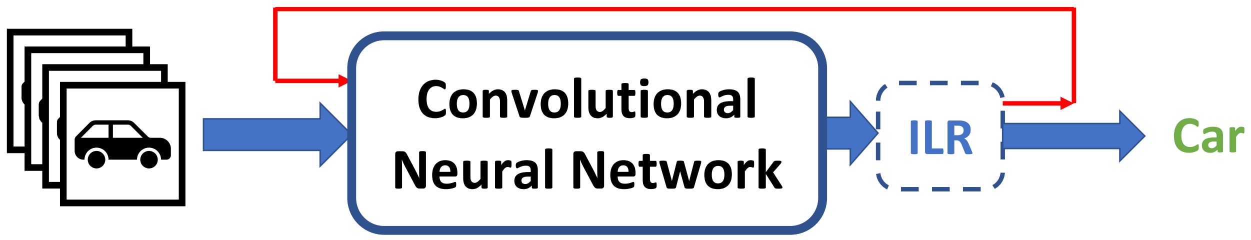


Figure 6.3: ILR design overview. ILR selectively reruns inferences deemed vulnerable to SDCs based on the output confidence values.

6.4 ILR: INFERENCE LEVEL RESILIENCE

Another granularity which can be targeted for CNN resilience is an individual *inference* for an image. (Section 6.4.1). In this section, we introduce ILR, a novel, per-image inference confidence-based CNN resiliency technique. ILR selectively reruns images for inferences that are vulnerable to SDCs by using only information provided after an inference is complete, namely the confidences in the softmax. We study two confidence-based criteria, Top1Conf and Top2Diff (Section 6.4.2), and identify a confidence threshold to trigger reruns during deployment (Section 6.4.3). Figure 6.3 depicts the high level design of ILR.

6.4.1 Target Granularity: Inference

The target granularity for CNN resiliency chosen for ILR is an individual *inference*. While FLR targets static, structural duplication of select fmaps before network deployment, ILR uses dynamic information to perform selective, full network reruns. The motivational insight behind ILR is that the classification confidence of a CNN for an inference is related to the probability that a soft error can cause a classification mismatch. Furthermore, despite their importance, SDCs should be an exception and not the norm; thus, to avoid incurring static overheads to have high resilience, a dynamic anomaly detector can significantly reduce overheads while maintaining high error coverage.

6.4.2 Vulnerability Estimation

To selectively identify which inferences are vulnerable and need to be reexecuted, we explore two decision functions which operate on the softmax (summarized in Algorithm 6.1). The first function assesses vulnerability of an inference based on the highest confidence value observed from the softmax (the Top1-Conf). We select this metric to examine if an inference with high confidence in prediction is more robust to soft errors. In this scenario, if the Top1-

Algorithm 6.1: ILR Implementation

```
1 Input: Softmax layer confidence values
2 if $F(Softmax)$ then
3     Re-run CNN for image
4 else
5     No-op
6 Version 1: $F(Softmax)$ returns Top1Conf < THRESHOLD
7 Version 2: $F(Softmax)$ returns Top2Diff < THRESHOLD
```

Conf lies above a certain threshold, the inference is deemed less vulnerable to perturbations, while inferences with Top1-Conf below the threshold should be conservatively rerun to avoid a possible SDC.

The second criterion we explore is the difference between the top two classes in the softmax, called Top2Diff. The intuition behind this choice is that a transient error needs only do enough computational damage to the network to cause the CNN to classify the image as the second highest class, rather than the (originally correct) top class. Thus, a smaller Top2Diff is akin to a smaller catalyst for the soft error to overcome to cause a mismatch, compared to a large Top2Diff which is more robust to mismatches from soft errors. In Section 6.7, we show that a small Top2Diff is highly correlated with SDC occurrences, and thus a good selection for the ILR conditional function.

For both decision functions studied, we perform resiliency analysis using error injections to identify the operational threshold for a target error coverage. More broadly, ILR can employ a complex, higher dimensional function which uses inference-specific artifacts to perform selective re-execution. Such a generalization is useful for extending ILR to additional, non-classification CNNs. We leave this exploration as future work.

6.4.3 Selective Protection

ILR protects against SDCs by rerunning vulnerable inferences, and verifying the output between the two runs. The ILR logic check is a highly attractive solution from an implementation standpoint due to its simplicity, and can be performed either in hardware or software. In this work, we consider the ILR logic overhead as negligible and focus on the reexecution overhead incurred as a result of ILR. In practice, rerunning an inference is not the only method for duplication. For example, the inference can be subsequently run on a separate, optimized model, rerun on hardened hardware, use lower precision during verification, or optimized in other ways to avoid incurring “full” overhead from a duplicate rerun.

6.5 FILR: ILR + FLR

ILR and FLR can be employed independently for CNN resiliency, as each targets a different axis for selective resiliency analysis and hardening. In this work, we also explore a combination of the two techniques, to evaluate the benefits of dynamic, selective inference duplication with static, selective fmap duplication.

To combine the two techniques, we first performs ILR analysis on the dataset to identify an optimal `THRESHOLD` for coverage versus overhead as determined by the decision function, Top2Diff. We then run FLR analysis, *but only on the subset of inferences not protected by ILR*. The main idea is to focus the FLR analysis (which will result in a flat, built-in resiliency overhead by duplicating fmaps before deployment) on the SDCs which ILR does not protect against.

6.6 EVALUATION METHODOLOGY

Benchmarks: We perform our evaluation of FLR, ILR, and FILR on 7 popular CNNs pre-trained on the ImageNet dataset [199], each listed in Table 6.1 with a count of topological parameters. As described in Section 5.2.1, we assume that a transient error during a MAC operation of a convolution will corrupt a single bit in a neuron. We use INT8 neuron quantization during inference, as highly optimized systems typically employ quantization prior to deploying CNNs. Such models run significantly faster with hardware support for reduced-precision operations, which is prevalent in GPUs and CPUs. These benefits come with a small but acceptable loss in classification accuracy (reported in Table 6.1). Additionally, prior work has shown that limiting the numerical range of a neuron can significantly reduce the SDC rate in a network [148], which we incorporate by operating in a quantized regime.

Infrastructure: We use the PyTorch framework v1.1 [161], and obtain pretrained models for CNNs from the PyTorch TorchVision repository [205]. We use PyTorchFI [48] (Section 5) to perform error injections on the CNNs. All experiments in this section are run on an Amazon EC2 p3.2xlarge instance [206], which has an Intel Xeon E5-2686 v4 server processor, 64GB of system memory, and an NVIDIA V100 GPU with 16GB of device memory [207].

Data Partitioning: ImageNet [199] provides a test set of 50,000 images, which we randomly split into an analysis set (AS) and a deployment set (DS) using an 80/20 ratio for evaluation.[1] Since this work focuses on pretrained networks, we do not use the images

[1]We use "AS/DS" to disambiguate from the common "train/test" nomenclature in ML workloads, although the evaluation concept is the same.

Table 6.1: CNNs studied with key topological parameters.

Neural Network	**Conv Layers**	**Total Fmaps**	**Total Neurons**	**Average Neurons/ Fmap**	**INT8 Quantized Accuracy**
AlexNet [145]	5	1,152	484,992	421	56.04%
GoogleNet [200]	57	7,280	3,226,160	443	69.43%
MobileNet [201]	52	17,056	6,678,112	391	62.18%
ShuffleNet [202]	56	8,090	1,950,200	418	67.01%
SqueezeNet [193]	26	3,944	2,589,352	241	57.39%
ResNet50 [203]	53	26,560	11,113,984	656	75.79%
VGG19 [204]	16	5,504	14,852,096	2698	72.20%

from the ImageNet training set as they would already have been used for training. As in a real-life scenario, we assume the developer only has access to the AS for reliability analysis of the CNNs, and we validate our results on the DS. The 40,000 images in the AS are the same across all networks and techniques explored, and similarly for the 10,000 DS images.

During reliability analysis, we are primarily interested in identifying SDCs on images that are defined as *originally correct* during an error-free inference, and which result in a mismatch due to a transient error. Thus, for error coverage analysis, we perform error injections only on images which are *originally correct* (meaning the inference resulted in the same class as the dataset label) during an error-free execution [148, 150, 208]. When measuring runtime overhead, we include *all* images for evaluation, since at runtime it is unknown which ones give correct or incorrect outcomes.

While the AS remains the same throughout analysis, the resiliency analysis methodology differs for FLR and ILR since they are different techniques (we elaborate in Section 6.6.1 and Section 6.6.2). However, for validation, we perform a single, unified error injection campaign on the DS. For the DS, we perform 10 million random error injections per network, where each error injection is performed on a random bit of a random neuron for a random image. In total, we perform 70 million error injection experiments across all networks for the DS.

6.6.1 FLR

We partition the evaluation of FLR into two parts: 1) comparing the accuracy and speed of the two metrics, mismatch and ΔLoss; 2) evaluating the coverage versus overhead tradeoff provided by FLR's selective fmap duplication.

To compare mismatch and ΔLoss, we first generate a statistical oracle for fmap vulnerability by performing 12,288 injections per fmap (inj/famp) for each network, which corresponds to at a least 99% confidence level with less than 0.23% confidence intervals. We define our statistical oracle using the number of mismatches obtained at 12,288 inj/fmap (shorthand: Mismatch-12288). For each individual error injection experiment, we flip a random bit of a random neuron in the fmap for a random image. As described in previous chapters, a true oracle is computationally elusive, due to the exponentially large population size of all possible error sites. In total, we performed a total of 855 million unique error injections across all CNNs studied for FLR.

We generate a cumulative vulnerability distribution based on a greedy selection algorithm for which fmap order to protect. We sort all fmaps in descending order of vulnerability (based on the metric being considered) and choose the first several fmaps whose relative vulnerability adds up to the targeted coverage. The error coverage is always extracted from the oracle mismatches of each fmap. We model the expected computational overhead as the total number of MAC operations in those selected fmaps as a fraction of the total MAC operations in all fmaps. We use MACs as a reasonable proxy to the actual overhead, while providing some abstraction for the actual hardware used. Our technique is designed to make FLR platform agnostic, providing a portable analysis for CNN reliability on any hardware backend.

To compare the accuracy of the metrics, we measure the average Manhattan distance between each cumulative distribution and the oracle cumulative distribution. We perform a sweep from 64 inj/fmap to 12,288 inj/fmap for each metric, using the Manhattan distance as a measure for how similar the vulnerability estimations are (zero Manhattan distance implies same vulnerability estimations). To compare the speed of each metric in performing the FLR vulnerability analysis, we identify the number of inj/fmap required to attain 99% similarity to the oracle. By ensuring that all infrastructure is held constant during error injection experiments (i.e., the hardware used, the number of images in a batch for parallelizing error injections, and the runtime of an inference), the only differentiating factor for analysis runtime is the total number of error injections performed, which we use to calculate speedup. Finally, we analyze the coverage versus overhead tradeoff for different networks, and show that the expected coverage (as indicated by the AS) is very accurate compared to the actual coverage (as indicated by the DS).

6.6.2 ILR

For ILR evaluation, we perform 1000 error injection experiments per image in the AS for each CNN studied. For each error injection experiment, we flip a random bit of a random neuron in the network at runtime. In total, we perform 184 million unique error injections across all CNNs studied for ILR. We conservatively select 1000 inj/image, which corresponds to a 99% statistical confidence level with less than 0.81% confidence intervals. Our strong validation results of ILR on the DS support our evaluation choices (Section 6.7.2).

We evaluate the two logical conditionals for ILR, Top1Conf and Top2Diff, sweeping the threshold values from 0.0 to 1.0 in increments of 0.01, and measuring the provided error coverage and associated overhead. The error coverage indicates how many SDCs are protected against the given Top2Diff/Top1Conf threshold for rerun, while the overhead is the additional number of inferences performed due to ILR reruns.

6.6.3 FILR

We evaluate FILR using the same error injection infrastructure as ILR described in Section 6.6.2. For the ILR component of FILR, we select Top2Diff as the decision criterion. Given a target coverage, we run ILR analysis to generate threshold values which provide less coverage than the target. We then run FLR on the subset of inferences not covered by ILR at each threshold value, and selectively duplicate the most vulnerable feature maps which bridge the gap to the target coverage. We calculate the associated overhead as the "built-in" redundancy provided by fmap duplication from FLR for each inference, in addition to the total number of inferences performed by ILR in deployment, i.e., both the original inference and the selectively re-run inferences, where each individual inference runtime is based on the uninstrumented CNN inference plus the additional fmap duplication overhead from FLR. We report results at the target coverage of 100%, and validate the results by measuring the coverage and overhead on the DS using the Top2Diff threshold values and selected duplicate fmaps from the AS.

6.7 RESULTS FOR EFFECTIVENESS OF TECHNIQUES

This section focuses on evaluating the effectiveness of FLR, ILR, and FILR by highlighting the high-level insights and results. Sections 6.8 provides additional results and in-depth analysis into the techniques.

6.7.1 FLR

Mismatch versus ΔLoss Convergence: We begin by analyzing the two metrics we use to quantify fmap vulnerability. Figure 6.4 provides empirical evidence for the convergence of Mismatch-based analysis and ΔLoss-based analysis as the number of injections per fmap

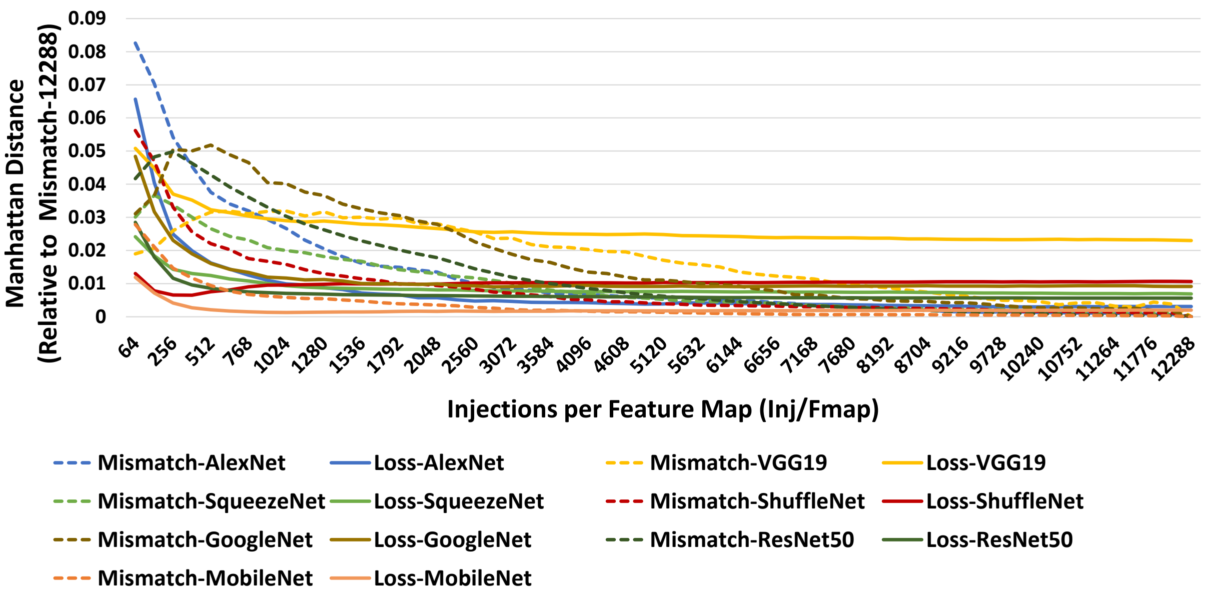


Figure 6.4: ΔLoss and Mismatch converge as inj/fmap increase.

increases. The X-axis in the figure shows the number of injections per fmap used for each analysis, and the Y-axis shows the Manhattan distance between the vulnerability ranking of fmaps obtained at each point relative to the statistical oracle of Mismatch-12288.

Our first observation is the scale on the Y-axis, which indicates that even at 64 inj/fmap, ΔLoss differs from the oracle by less than 7% on average. Second, the results show that ΔLoss quickly asymptotes to its final ordering of fmaps, and it does so sooner than Mismatch. Table 6.2 lists the number of inj/fmap required for Mismatch and ΔLoss to arrive within 1% of the oracle. For Mismatch, the number of inj/fmap ranges from 640-5632 while for ΔLoss it is lower going from 128-1536 inj/fmap. Additionally, both Mismatch and ΔLoss converge without requiring a full 12288 inj/fmap. To attain a very high accuracy fmap vulnerability ordering, our results show that Mismatch requires 5.2× fewer inj/fmap on average than the Oracle, while ΔLoss requires 16.7× fewer inj/fmap on average, resulting in an average speedup for ΔLoss of 3.2× over Mismatch (up to 9.4×) .

VGG19 does not attain 99% similarly from ΔLoss and asymptotically approaches the 97.5% mark instead. While still relatively high, we believe this is attributed to the average size of fmaps in VGG19, listed in column 5 of Table 6.1. For this network, the statistical error in the large injection campaign of Mismatch-12288 might not be small enough. However, as we show in the following section, this difference is minute when considering the coverage versus overhead trade-off, since the precise ranking of fmaps is less important so long as it is approximately well-ordered. Thus, while the Manhattan distance provides us with

Table 6.2: Comparison of Mismatch and ΔLoss

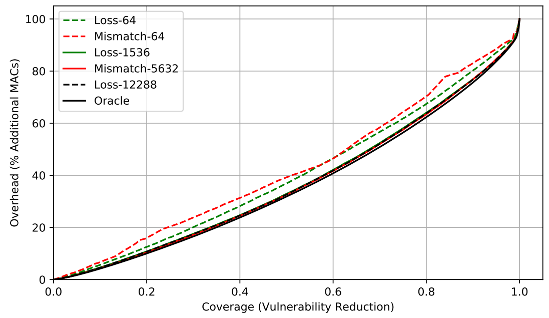
Network Neural	**Inj/Fmap for 99% Oracle Mismatch**	**ΔLoss**	**Speedup from Oracle Mismatch**	**ΔLoss**	**Loss Speedup**
AlexNet	2560	896	4.8×	13.7×	**2.9**×
GoogleNet	5632	1536	2.2×	8.0×	**3.7**×
MobileNet	640	128	19.2×	96.0×	**5.0**×
ResNet50	3584	384	3.4×	32.0×	**9.4**×
ShuffleNet	1664	1280	7.4×	9.6×	**1.3**×
SqueezeNet	3072	896	4.0×	13.7×	**3.4**×
VGG19*	2560	1536	4.8×	8.0×	**1.7**×
Geomean	2382	738	5.2×	16.7×	**3.2**×

* For 97.5% similarity to Oracle

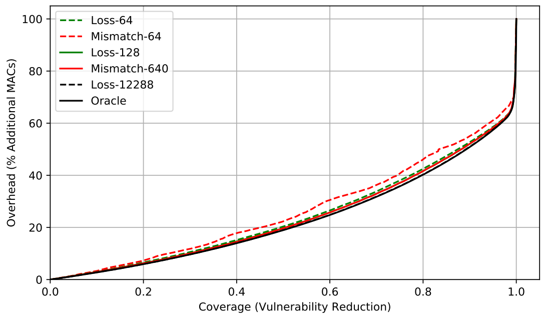
empirical evidence for the convergence of Mismatch and ΔLoss, FLR does not suffer from small imprecisions in the ordering. Attaining a good ordering quickly is advantageous for faster offline resiliency analysis, and this can be performed with ΔLoss for all networks studied.

Coverage versus Overhead: Figure 6.5 shows the performance of FLR as a selective resiliency technique, measured by the coverage versus overhead trade-off for selective fmap duplication. The X-axis shows the cumulative coverage by selectively protecting fmaps based on a vulnerability ordering, and the Y-axis shows the corresponding overhead as a percentage of additional MAC operations. We plot the trade-off for 6 vulnerability orderings: the Oracle (Mismatch-12288), Loss-12288, mismatch and loss at the 99% convergence points (Table 6.2), and Mismatch and Loss at 64 inj/fmap. The inclusion of Mismatch-64 and Loss-64 helps further illustrate the the faster convergence of ΔLoss relative to Mismatch, as explained in the prior section.

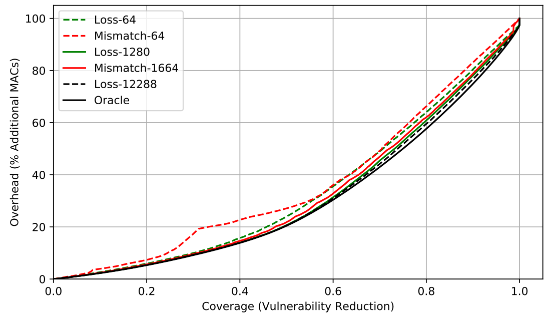
First, the results show that the computational overhead is always sublinear to coverage, indicating that selective protection is in fact advantageous to full duplication and can even provide large benefits. For example, covering 90% of errors in SqueezeNet incurs less than 30% overhead for the network, emphasizing that only a fraction of fmaps possess most of the vulnerability for the network. MobileNet is another interesting network, reaching nearly 98% coverage (for 64% overhead) before a sudden, vertical rise in overhead for the last 2%. In this case, we find that MobileNet has a unique feature: an imbalance of fmap sizes (captured by V_{orig}), such that the vulnerability of larger fmaps dominate, while a tail of smaller fmaps can be relegated in protection. For VGG19, despite the small difference in convergence between ΔLoss and Mismatch in fmap ordering (Table 6.2), using selective protection shows that an informed, approximate ordering (as employed by FLR at fewer inj/fmap) still provides an



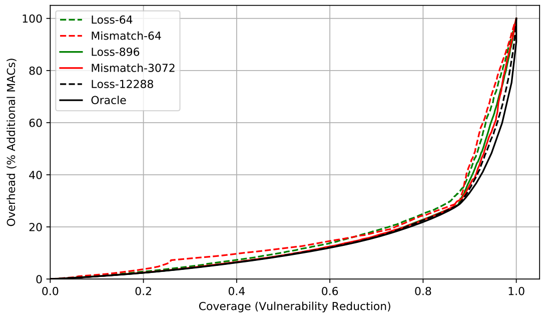
(a) GoogleNet-ImageNet



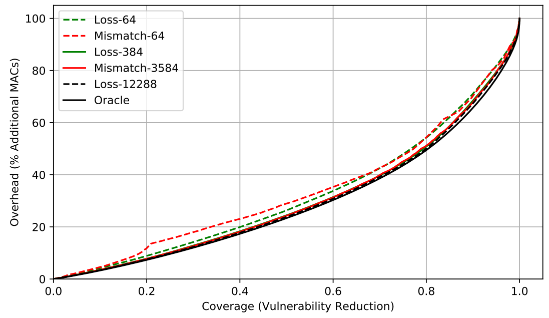
(b) MobileNet-ImageNet



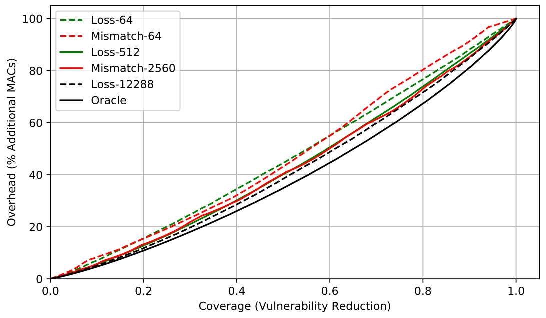
(c) ShuffleNet-ImageNet



(d) SqueezeNet-ImageNet



(e) ResNet50-ImageNet



(f) VGG19-ImageNet

Figure 6.5: FLR vulnerability reduction versus computational overhead.

opportunistic coverage versus overhead tradeoff for CNN resiliency.

Validation: Figure 6.6 validates the use of ΔLoss as a metric for vulnerability analysis, where we show the estimated coverage predicted by FLR using ΔLoss on the AS (X-axis), and comparing it to the actual coverage as measured by the number of SDCs protected against on the DS (Y-axis). The results show that ΔLoss is representative of the actual vulnerability as measured by mismatches in the DS. Thus, the prediction provided by ΔLoss is an excellent alternative for system developers for error analysis compared to Mismatch.

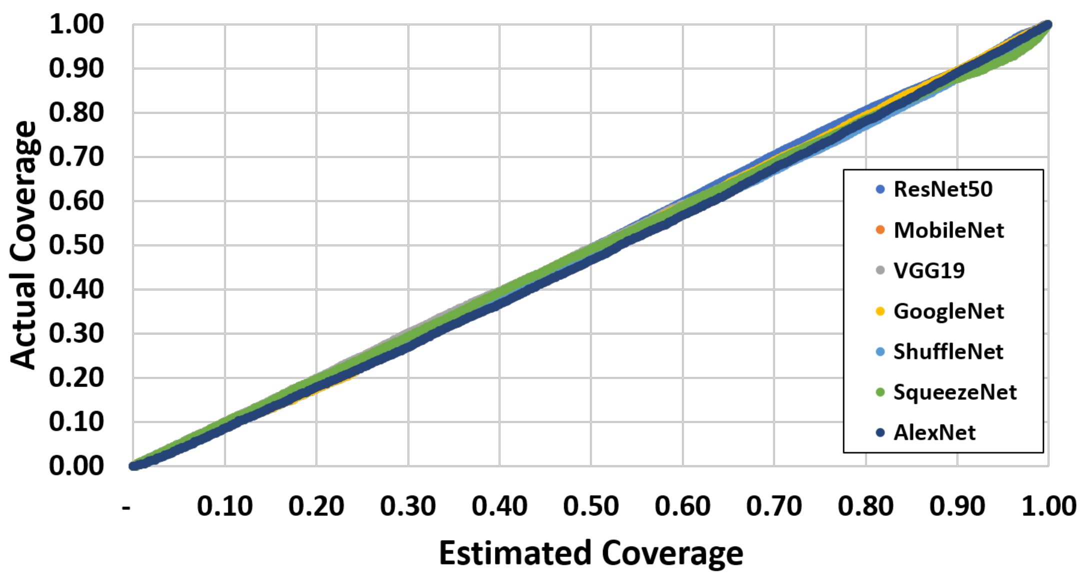


Figure 6.6: FLR validation of predicted versus actual coverage.

6.7.2 ILR

Correlation Between Inference Confidence and SDCs: We discovered a strong correlation between inference output and the vulnerability of the inference. Figure 6.7 illustrates this correlation. We plot the number of SDCs for 1000 randomly selected images run on AlexNet on the primary Y-axis. The secondary Y-axis shows the image's error-free Top1Conf and Top2Diff values. We measure the Spearman correlation between the per-images SDC rate and the two confidence metrics we extract. For AlexNet, the Spearman correlations are -0.87 for Top1Conf and -0.93 for Top2Diff, where -1.0 indicates a perfect inverse relationship. Both metrics exhibit a very high correlation relationship between the number of SDCs observed for an image and the image's confidence, which we can leverage for resiliency analysis and hardening.

Coverage versus Overhead: While the two metrics for ILR seemingly have very high correlations, we find that Top2Diff performs significantly better as a criterion for detecting SDCs. Figure 6.8 illustrates this difference. Each point shows the coverage and overhead obtained by setting the threshold for reruns by ILR, swept from 0.01 to 1.0 with 0.01 increments. Figures 6.8a and 6.8b show the coverage versus overhead tradeoff for using Top1Conf and Top2Diff in ILR, respectively. For both the metrics, we find that ILR provides a favorable trade-off in terms of obtaining high coverage and incurring low overhead, signified by all points being below the x=y line. More importantly, the pareto-optimal threshold values for Top2Diff are strictly better for Top2Diff, illustrated by a lower "knee" for each

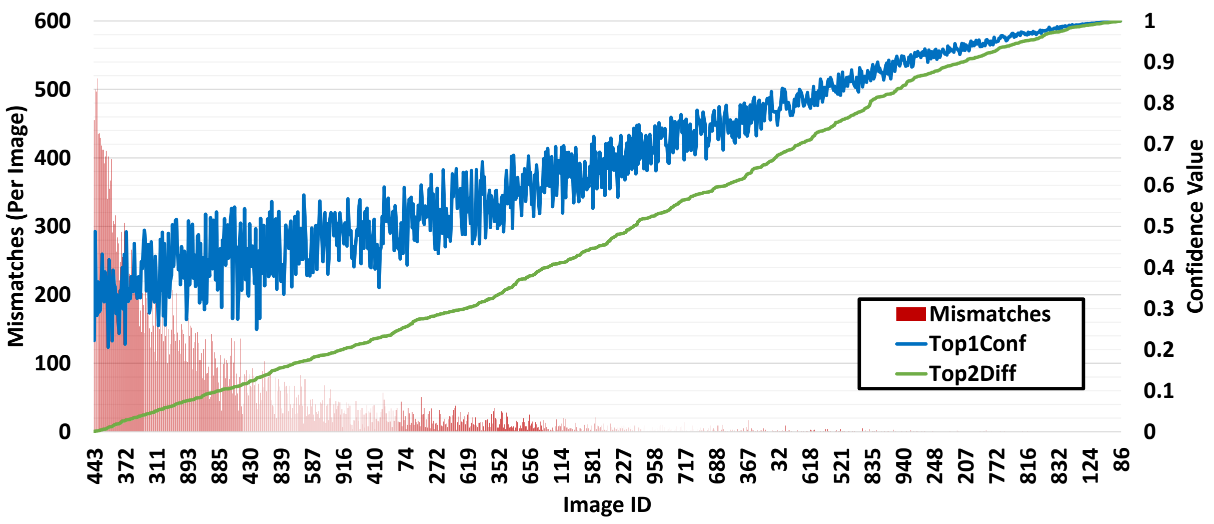
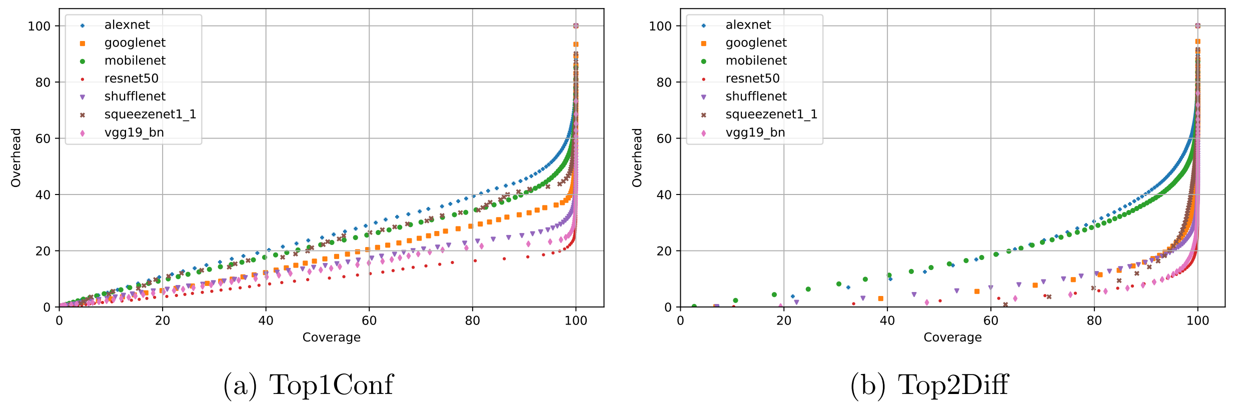


Figure 6.7: Correlation between an image's output confidence and the number of SDCs (AlexNet).



(a) Top1Conf (b) Top2Diff

Figure 6.8: ILR: Coverage versus overhead at different thresholds.

network. ResNet50, for example, can achieve 90% coverage at only 9% overhead using ILR with Top2Diff, while incurring 18% overhead with Top1Conf. Figure 6.9 summarizes this result for all networks, showing that on average, we can obtain 90% coverage with only 17% overhead using Top2Diff, compared to 31% overhead on average with Top1Conf. We focus on Top2Diff as the decision criterion for ILR moving forward as it performs better.

Validation: We validate ILR by measuring the coverage versus overhead trade-off on the DS. Figure 6.10 shows the tradeoff plot for ILR using Top1Conf and Top2Diff, showing similar trends as analyzed on the AS (Figure 6.8b). That the AS and DS contain exclusively different images reinforces the use of Top2Diff as a confidence-based metric for CNN resiliency by quantitatively showing a strong correlation between the confidence of an inference and SDCs.

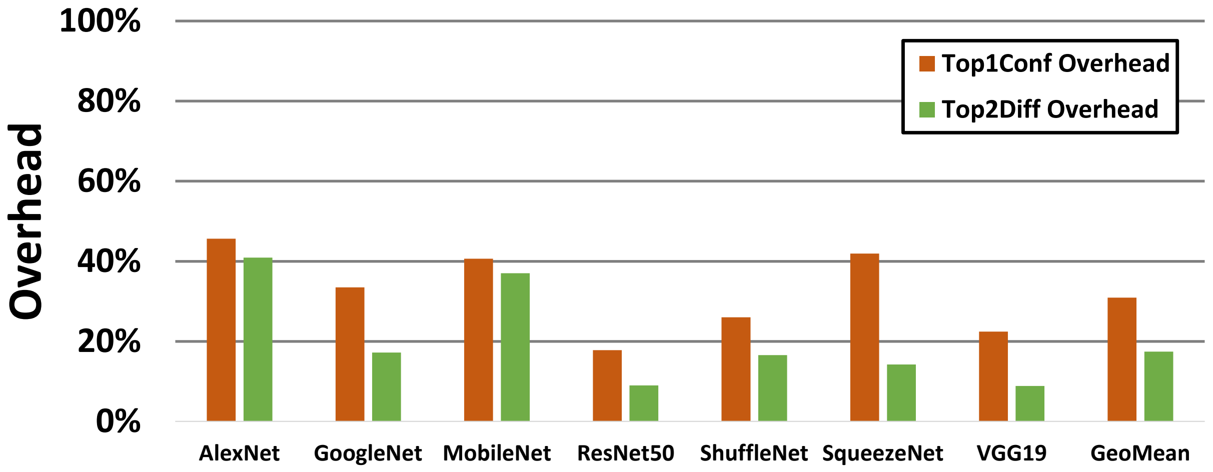
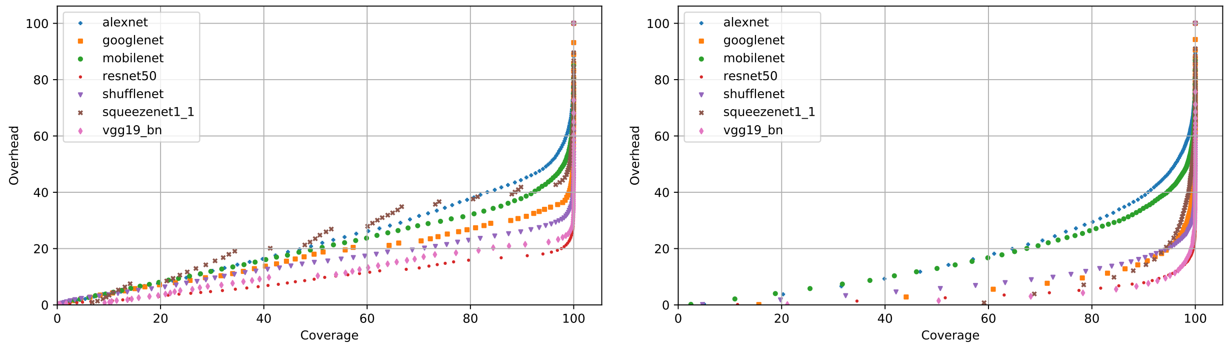


Figure 6.9: ILR overhead at 90% coverage for Top1Conf & Top2Diff.



(a) Top1Conf

(b) Top2Diff

Figure 6.10: ILR validation results on DS.

Analysis: Our results show that confidence-based metrics for SDC detection are highly effective. Furthermore, Top2Diff specifically helps explain the phenomenon of a mismatch, showing that a soft error has a higher probability of causing a mismatch if the margin between the top two classes is small, which translates to a higher probability of an SDC appearing.

While ILR using Top2Diff shows impressive gains compared to DMR by performing selective inference duplication, we observe that the overhead can still be considered rather high for resource constrained systems, since despite their importance SDC events are typically rare. We looked more into this issue, and found that the primary cause of this bloated overhead was due to the many false positive inferences ILR chooses to re-run, specifically for images which are originally *incorrectly* classified and have a small Top2Diff. Moreover, we found that networks that exhibit a high classification accuracy on average such as ResNet50

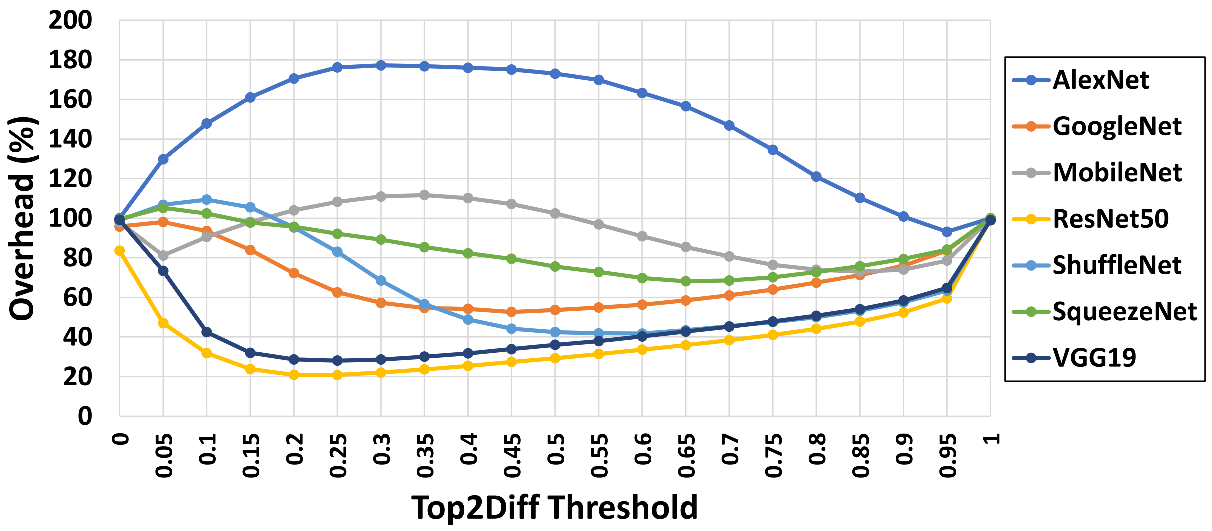


Figure 6.11: FILR coverage versus overhead.

(column 6 in Table 6.1) suffer less from this phenomenon than other networks, such as AlexNet. This encourages additional research on improving network prediction accuracy in general (which is already a ripe area of ML research). At the same time, we can explore using ILR with a secondary metric to attempt to differentiate between correct/incorrect inferences after the primary metric of Top2Diff. We leave this avenue as future work.

6.7.3 FILR

Combination of Techniques Results: Figure 6.11 shows the results for FILR, which combines the two resiliency techniques of FLR and ILR as an ensemble resiliency solution. All points in the figure represent 100% coverage. The X-axis shows the Top2Diff threshold used by ILR, which also influences the FLR analysis as described in Section 6.6.3. The Y-axis shows the overhead for FILR, which is a result of both the static overhead from fmap duplication and dynamic overhead from inference reruns.

The 0.0 Top2Diff threshold (on X-axis of the figure) corresponds to no coverage or overhead contribution from ILR and only selective fmap protection. The 1.0 Top2Diff threshold corresponds to only using ILR (with FLR not needing to protect any fmaps). We find that there exists an optimal point for each network below the 1.0 Top2Diff threshold, where ILR and FLR collaborate to achieve high, 100% coverage with an overhead that is far lower than 100%. On average, the overhead via FILR is 48.07% across networks, and as low as 20.78% for ResNet50.

Validation: Figure 6.12 shows the validation on the optimal points from FILR. Our

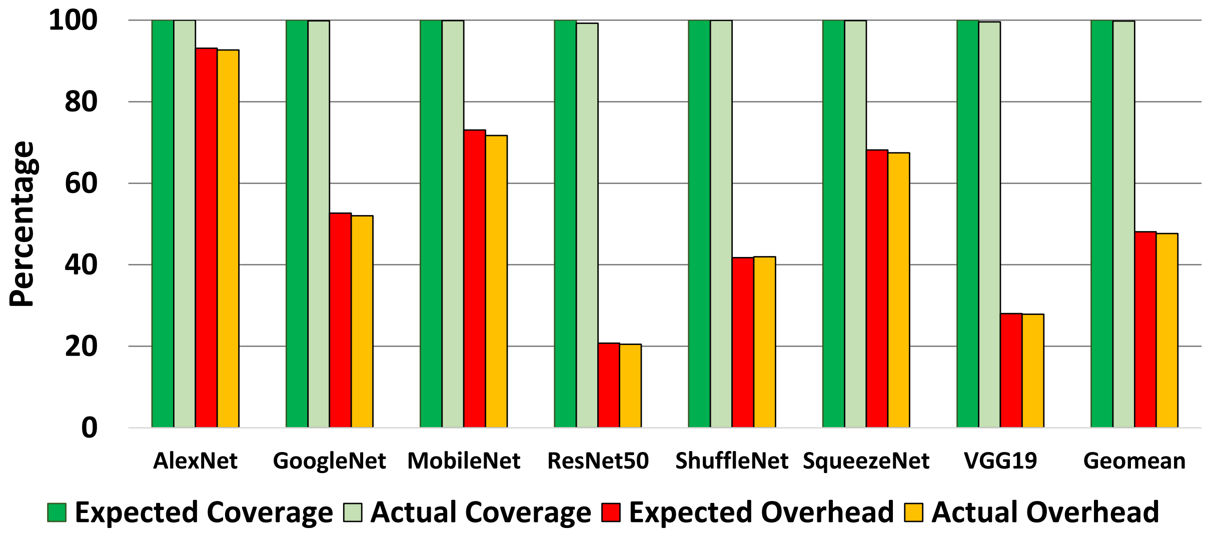
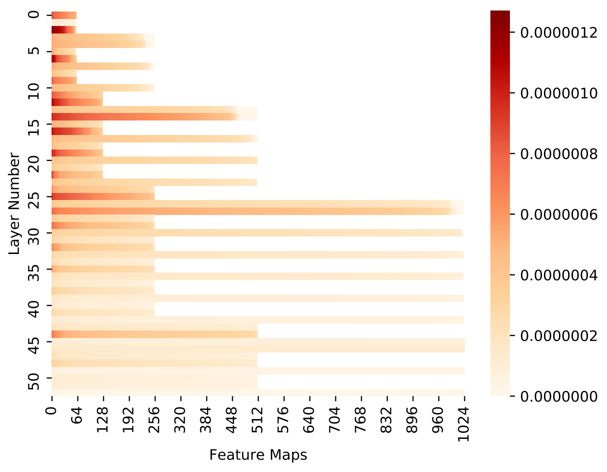


Figure 6.12: FILR validation at optimal Top2Diff thresholds.

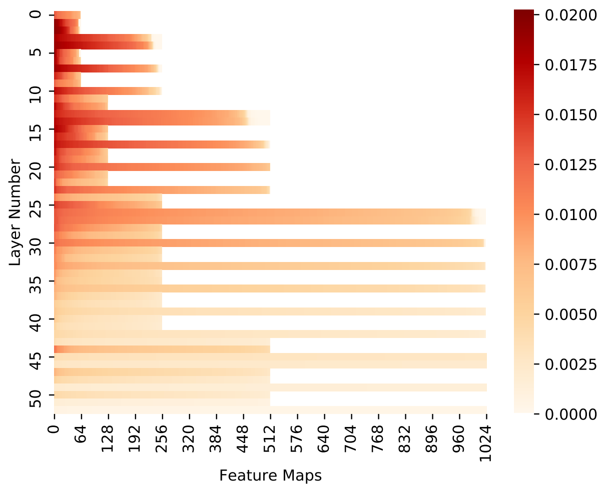
results show very strong validation, with 99.78% coverage at an average of 47.66% overhead (as low as 20.47% for ResNet50). These results show that the FILR technique is better than the sum of the parts, where each technique individually required near 100% overhead to obtain near 100% coverage.

Analysis: FILR is efficient at identifying a balance between FLR and ILR to obtain less than 100% overhead. For ResNet50, VGG19, and ShuffleNet, nearly all points provide sub-100% overhead, benefiting from ILR's initial low cost at high coverage. Other networks such as GoogleNet, SqueezeNet, and MobileNet show less overall improvement, but FILR is still able to identify multiple sub-100% overhead points for efficient resiliency. AlexNet contains one such point at 0.95 Top2Diff threshold, but as it is a smaller and low-accuracy network, there seems to be less opportunity for lower overheads at such high coverage. Nevertheless, there still exists a point below 100% overhead which is identified by FILR, further validating the use of selective, granular resiliency over DMR.

The results show that FILR is effective at flattening the sharp rise observed by ILR alone (in Figure 6.8b) at higher coverage points. More generally, FILR combines the benefits of each technique, by providing a low overhead starting point via ILR, followed by a shallow growth for higher error coverage via FLR's selective feature duplication. The key behind this symbiotic relationship is using FLR to focus selective protection of fmaps for the subset of SDCs missed by ILR. Furthermore, using ΔLoss as the resiliency metric for FLR has a subtle yet important contribution in the combined analysis, since it can help distinguish fmap vulnerabilities at fine granularity with fewer samples.



(a) V_{fmap} ranking



(b) P_{prop} ranking

Figure 6.13: Layer level vulnerability analysis with ΔLoss on ResNet50-ImageNet (cutoff at 1024 fmaps).

6.8 DETAILED ANALYSIS AND EXTENSIONS

We performed additional analyses and experiments for FLR, ILR, and FILR, to shed more light on various aspects of the techniques. We provide additional details into layer-level and network level resilience of CNNs, the effect of different error models on analysis, multiple heuristics for calculating vulnerability of fmaps, and implementation measurements on a Jetson Xavier platform.

6.8.1 FLR

Layer Level Analysis: As layers in a CNN consist of many fmaps, we study whether the vulnerable fmaps of FLR are clustered in certain layers or not. Figure 6.13a shows a heatmap of ResNet50's fmap vulnerabilities (V_{fmap}), which are computed using ΔLoss. Fmaps per layer are sorted based on V_{fmap} values. The darker the color, the more vulnerable the fmap. We find that on average, a small fraction of fmaps (<33%) account for a large percentage (>76%) of a CNN's vulnerability. The figure also show that the highly vulnerable fmaps are distributed across layers. A layer-level analysis and protection may provide an inefficient solution as it will likely protect more fmaps than required to meet the resilience requirements. A fine-grained analysis at the fmap level provided by FLR informs which fmaps to target duplication for an optimal solutions.

We also study the importance of incorporating V_{orig} in the vulnerability formulation by

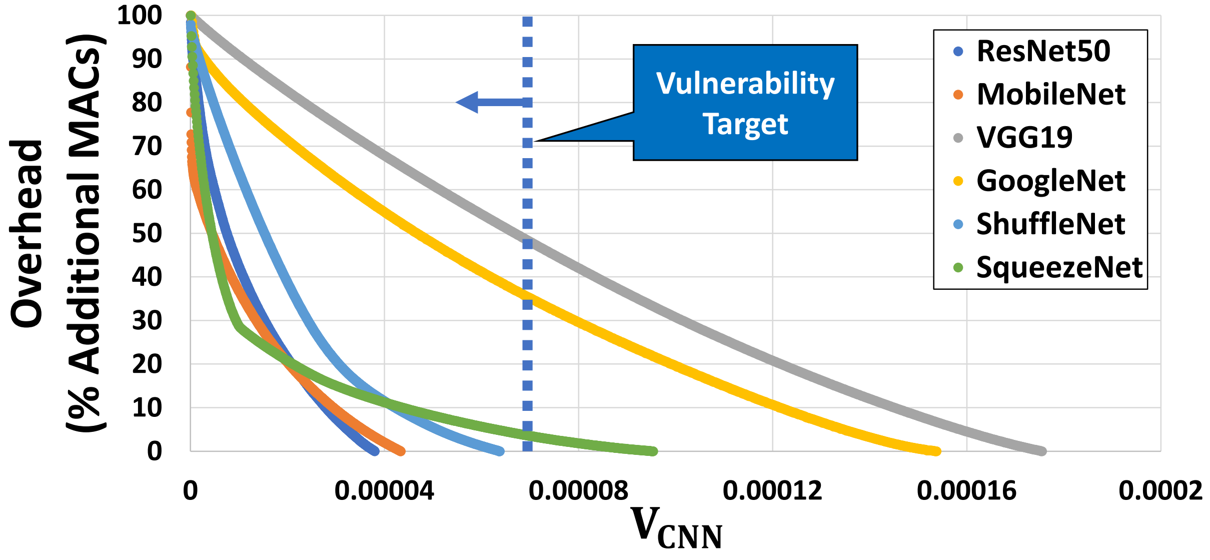


Figure 6.14: Network level analysis for ImageNet networks.

comparing the P_{prop} and V_{fmap} values. Figure 6.13b shows the updated per-fmap profile combined at the layer level using just P_{prop} (i.e., not including V_{orig}). Prior work used similar quantities to determine what and how much to protect [150, 168, 209]. This figure (when compared to Figure 6.13a) shows that the fmap and layer vulnerabilities could vastly differ, and could lead to protecting significantly different set of fmaps or layers if the origination vulnerability is ignored.

Network Level Analysis: FLR further allows a developer to compare total vulnerability values, V_{CNN}, of different CNNs, allowing them to make an informed decision about selecting a CNN that meets the resilience, performance, and accuracy targets. Results for six different CNNs trained on the ImageNet dataset to solve the same classification problem are shown in Figure 6.14. For a given vulnerability target, this analysis informs the developer how many fmaps need protection for a selected CNN and estimates the associated overheads. Since we use ΔLoss to predict P_{prop}, the X-axis values are in an arbitrary units, but allow for relative comparison and selection. A separate small experiment can be performed to calibrate the scale to real probabilities. For example, based on a vulnerability target of 0.00007 (as exemplified in Figure 6.14), selecting ResNet50 would satisfy the reliability need without additional protection. Selecting SqueezeNet and GoogleNet would require duplicating a fraction of fmaps which would amount to approximately 5% and 35% overheads, respectively.

Different Error Models: As described in Section 6.6, we assume that a transient error on a flip-flop during a MAC operation of a convolution will corrupt a single neuron's value.

While this chapter has focused primarily on this error model, prior work has shown that low-level errors can manifest as single or multiple bit flips [210]. To that end, we evaluate P_{prop} using a few additional error models, and compare the resiliency analysis results.

In each model we explore, an error is injected into a neuron that is randomly chosen from an fmap, followed by substituting the original value with the erroneous value. We explore the follow three error models: (1) *FP-Rand* represents a random, multi-bit error in a neuron storing a floating-point value. A random value between [-max, max] is selected as the injected error, where max is the maximum observed neuron value in the fmap across the training set. FP-Rand limits the the error by bounding it between a range. Previous work found that inference is highly sensitive to errors in the sign and exponent bits and a simple output fmap-level range detector can mitigate many of the most severe corruptions [148], which we incorporate in this error model. We use 16-bit floating point (FP16) precision for our evaluation. (2) *FxP-Rand* considers 8-bit integer (INT8) *fixed-point quantization* (a commonly used scheme in many commercial products [211, 212]), which quantizes the CNN based on the range of neuron values observed during training. The erroneous value is a randomly selected INT8 value. (3) *FxP-Flip* represents a random single bit-flip on a fixed-point quantized 8-bit integer neuron. This third model is the same model that was used previously throughout this chapter.

The different computational precisions used for FP-* and FxP-* impact the FIT rate of the MAC operation and the error origination vulnerability, V_{orig}. Two factors affect the FIT rate of a MAC – the number of unprotected bits in the MAC and the logical error propagation probabilities through the multiplier and accumulator implementations. We assume V_{orig} for FxP-Rand to be 0.75× the V_{orig} for FP-Rand, primarily based on the reduction in the number of unprotected bits[2]. We assume V_{orig} for FxP-Rand and FxP-Flip are identical.

Figure 6.15 shows the cumulative relative vulnerability ($Vrel_{fmap}$) of the fmaps in AlexNet-ImageNet, where the X-axis is sorted in descending order of $Vrel_{fmap}$, which are measured using 12,288 inj/fmap and mismatch as the SDC determination criterion. For the comparison, we use the same fmap order on the X-axis, which is obtained based on FxP-Flip $Vrel_{fmap}$.

Results show that an fmap's contribution towards the total network vulnerability is practically the same for the different error models, with less than .0001% difference. We, however, found that the absolute total vulnerability, V_{CNN}, changes with the models. FP-Rand and FxP-Rand exhibit similar propagation probabilities (P_{prop}), as both models have the same dynamic range and multi-bit perturbation error model. However, V_{CNN} is lower for FxP-

[2]Since each MAC unit typically has two input registers (8-bits for INT8 and 16-bits for FP16) and one accumulator (of 32 bits), we use $(8 + 8 + 32)/(16 + 16 + 32) = 0.75$.

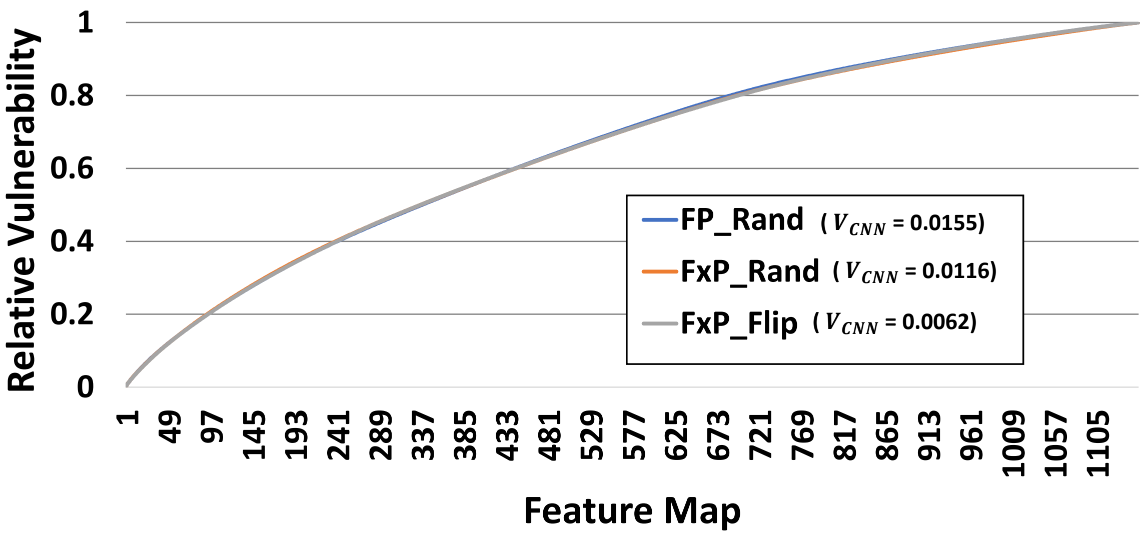


Figure 6.15: $Vrel_{fmap}$ is similar across error models, even with different V_{CNN} (AlexNet-ImageNet).

Rand due to the influence of the numerical precision of the MACs (INT8) on V_{orig}. FxP-Flip shows lower V_{CNN}, which we attribute to the less egregious error model, i.e., a single-bit perturbation, compared to a random value from the entire range.

Heuristic Exploration for P_{prop}: Our primary results and evaluation for FLR focused on error injection analysis for P_{prop}. We also explored other heuristic based approaches for P_{prop}, discussed briefly here. Despite the heuristics running quickly relative to an error injection campaign, the resulting accuracy for fmap-level analysis was not as high as desired for the heuristics explored [49]. The heuristics rely on information from a set of error-free inferences to estimate the vulnerability of an fmap, and fall under two general categories: (1) obtaining fmap-level information using observations from the forward pass during an inference, and (2) performing an additional backward pass (a back-propagation) to provide additional information via differentiation for vulnerability estimation. We studied a total of 6 non-injection based heuristics, and include a summary in Table 6.3.

- *Max Neuron Value*: This simple forward-pass technique assigns an fmap the value of the maximal observed neuron value across many sample inputs. Thus, effectively, it assumes that errors in feature maps where the activation values can be high are more likely to affect the outcome.

- *Feature Map Range*: This technique assigns each fmap the value computed by finding the difference between the largest and smallest activation value across many sample inputs. This ranking scheme takes into consideration that networks will typically be

Table 6.3: Summary of estimation techniques for FLR

Vulnerability estimation based on error injections

Method Name	**Description**
Mismatch	Top-1 Misclassification rate in fmap due to error
ΔLoss	Average delta cross entropy loss of fmap due to error

Vulnerability estimation based on heuristics, with no error injections

Method Name	**Description**
MaxNeuron	Max neuron value observed for fmap
FmapRange	Range of neuron values observed for fmap
L2	Average L2-norm value of fmap
Gradient	Average magnitude of gradients for fmap
Gain	Analytical model of Top-1 class change for variation in fmap [213]
ModGain	Alternative formulation of Gain analytical model (Equation 6.5)

quantized before deployment [213, 214, 215], constraining their dynamic range and, in effect, also reducing the possible observable corruptions in the neurons in the feature maps. Thus, it models the maximal range of error values which may be observed during inference.

- *Average L2*: The L2-norm calculates the distance of the vector coordinate from the origin of the vector space. Specifically, it is calculated as the square root of the sum of the squared vector values. We compute the L2-norm of an fmap (vector) averaged across multiple input samples to assign this value to the fmap as an estimate of relative vulnerability.

- *Gradient*: One of the key components of CNN training is the gradient descent algorithm used to update a network's weights. During training, a CNN performs back-propagation to adjust weights in order to minimize a loss function (a typical loss function is cross-entropy loss, discussed above). This is done by obtaining the gradient value at each weight, and adjusting the weights incrementally during each training epoch by using the gradient value. We use a similar technique but adapted to neurons (rather than weights). As neurons are differentiable, they too have gradient values which can be used to predict vulnerability. For this technique, we perform a backward pass which *only* computes the gradients for each neuron and does not modify the network parameters, unlike the backward pass used in the training phase. We compute the gradients with respect to the cross-entropy loss at the output. We use the absolute value of neuron gradients obtained, and average them per fmap across many samples from the dataset.

- *Gain*: Recent work by Sakr et al. [213, 215] proposed an analytical model which bounds mismatch probability in the context of network quantization. We implement this technique, called Gain, which intuitively models the expectations of a class change for a CNN.

- *ModGain*: We adapt the Gain formulation for reliability analysis. While Gain utilizes a model of independent noise neuron corruption, we propose to study the effect of *replacing* a neuron by a random scalar belonging the the fmap's dynamic range. Under such setup, it can be shown that the Gain approach may be re-purposed to obtain *ModGain*, where the gain of fmap F is:

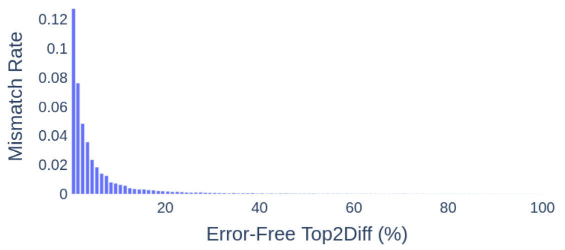
$$E_F = \mathbf{E}\left[\sum_{i=1, i\neq\hat{y}}^{M} \frac{\sum_{a\in F} a^2 \left|\frac{\partial\left(z_i - z_{\hat{y}}\right)}{\partial(a)}\right|^2}{\left|z_i - z_{\hat{y}}\right|^2}\right] \tag{6.5}$$

  where F's neurons $\{a\}_{a\in F}$ are combined with the M soft outputs $\{z_i\}_{i=1}^{M}$ given a predicted label of $\hat{y}$.

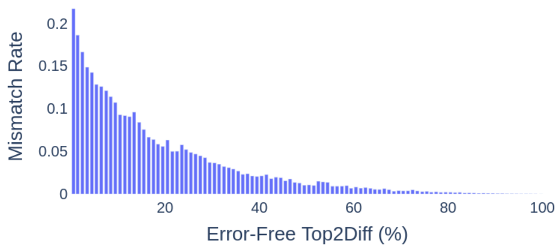
Compared to the forward-pass only techniques (the first three), the techniques which leverage back-propagation (the latter three) are slower in estimating P_{prop} due to the extra operations. However, by leveraging framework optimizations, all heuristics are expected to run faster than error injections campaign due to the limited number of total inferences compared to error injection experiments. While advantageous from a runtime perspective, these heuristics did not perform well (relative to the Oracle described in Section 6.6.1), because the network topology was not taken into consideration (whereas the topology *is* incorporated by injecting errors and emulating their propagation to the output). We leave the exploration of better heuristics for estimating P_{prop} as future work.

6.8.2 ILR

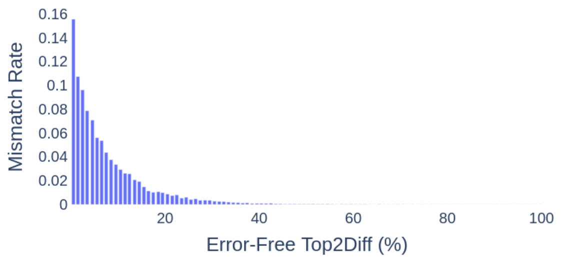
As described previously in Section 6.6.2, the offline analysis for threshold selection of ILR may require many experimental error injections to increase the statistical confidence in the final threshold. However, empirically, we find that a reduced number of sampling can



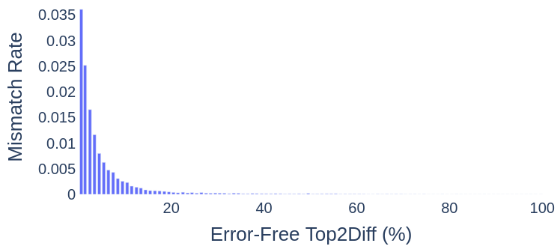
(a) GoogleNet-ImageNet



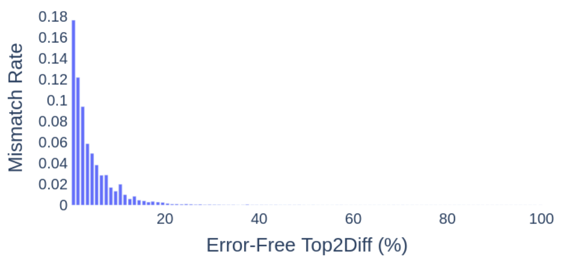
(b) MobileNet-ImageNet



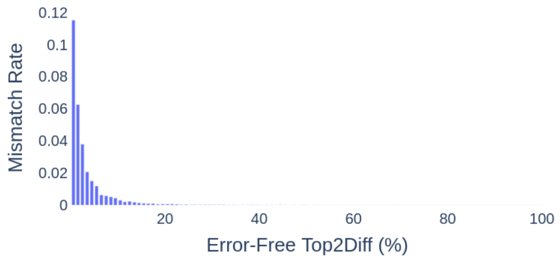
(c) ShuffleNet-ImageNet



(d) SqueezeNet-ImageNet



(e) ResNet50-ImageNet



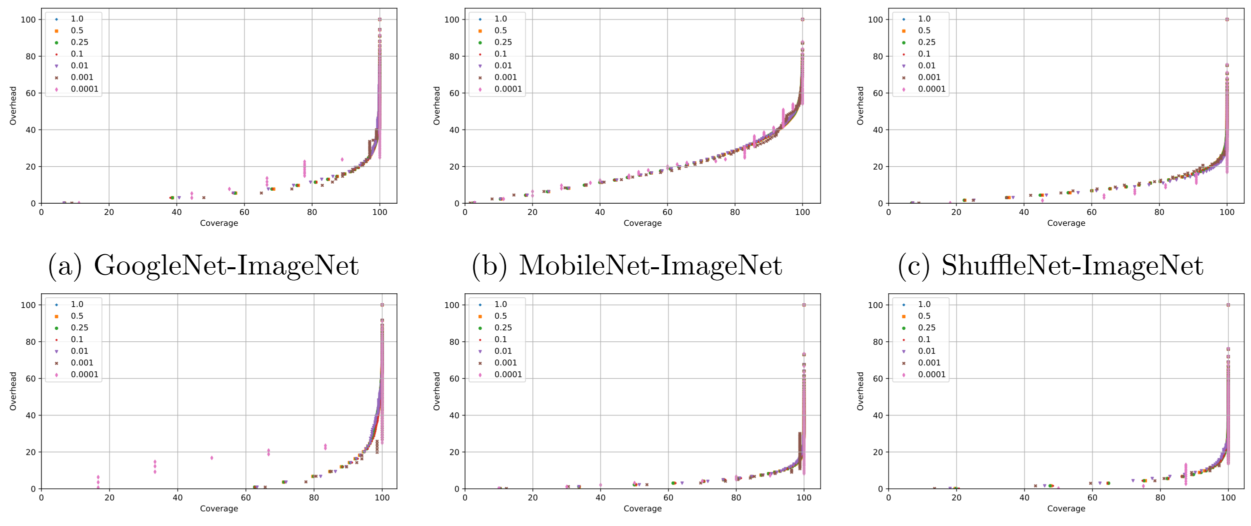
(f) VGG19-ImageNet

Figure 6.16: Error rate of images for a given fault-free Top2Diff threshold.

also suffice for quickly determining an ILR threshold while maintaining the accuracy of the selected threshold.

In order to understand this more, we grouped images together from the AS based on their original (error-free) Top2Diff values for evaluation. Loosely, each group can be considered an *equivalence class* of images, since the hypothesis is that images with similar top2diff will behave similarly in terms of SDC possibilities. We performed 1000 error injections per image, and plotted the number of output mismatches that occurred within each equivalence class. The results are illustrated in Figure 6.16.

We find that our intuition regarding Top2Diff holds - the class of images with lower Top2Diff produce more errors on average than classes of images with larger Top2Diff. Further, we find that the approximate location where Top2Diff of a network has a very low



(a) GoogleNet-ImageNet (b) MobileNet-ImageNet (c) ShuffleNet-ImageNet

(d) SqueezeNet-ImageNet (e) ResNet50-ImageNet (f) VGG19-ImageNet

Figure 6.17: Reduced sampling rates for ILR can accelerate Top2Diff threshold selection. The legend indicates the fraction of samples relative to our baseline injection campaign (e.g., 1.0 in the legend corresponds with 40 million samples).

mismatch rate is analogous to the optimal Top2Diff threshold selected by ILR, as explained in Section 6.7.2.

The main takeaway from these results indicates that images with lower Top2Diff have a higher mismatch rate, and this insight can be used to bias the sampling of images when looking for the optimal ILR threshold. Figure 6.17 provides emperical evidence for such a reduced sampling concept. The figure shows the overhead versus coverage plots for a different number of samples, based on the original 40 million error injections used to perform the ILR experiments in Section 6.6.2. The legend indicates the percentage of samples used, relative to the total 40 million injections from our baseline experiments (1.0 indicates 40 million total samples; 0.5 indicates 20 million total samples, etc). We find that the number of samples can be as few as 40,000 (.01% of our baseline) before loosing the fine-grained thresholding results of the baseline. Effectively, this is a relative speedup of 3 orders of magnitude. In terms of actual wall-clock time, that is the difference between an error injection campaign requiring multiple hours to complete versus less than a minute for threshold selection (when running error injection experiments on a high-end GPU such as an NVIDIA V100).

Overall, by understanding the correlation between a transient error on an inference's output confidences, we are able to further accelerate our error analysis campaign to run quickly without the loss of accuracy in identifying a good threshold for ILR.

6.8.3 FILR

We implemented FLR at the optimal point indicated by FILR (Figure 6.11) and measured the static model runtime overhead. We used a Jetson AGX Xavier as our platform, which has a 8-core ARM v8.2 CPU and 512-core Volta GPU with Tensor Cores and 32GB of shared memory. We ran the models on each of the CPU and GPU with a batch size of 1, and performed 100 runs and measured the average runtime overhead relative to an unhardened model. Figure 6.18 shows the results.

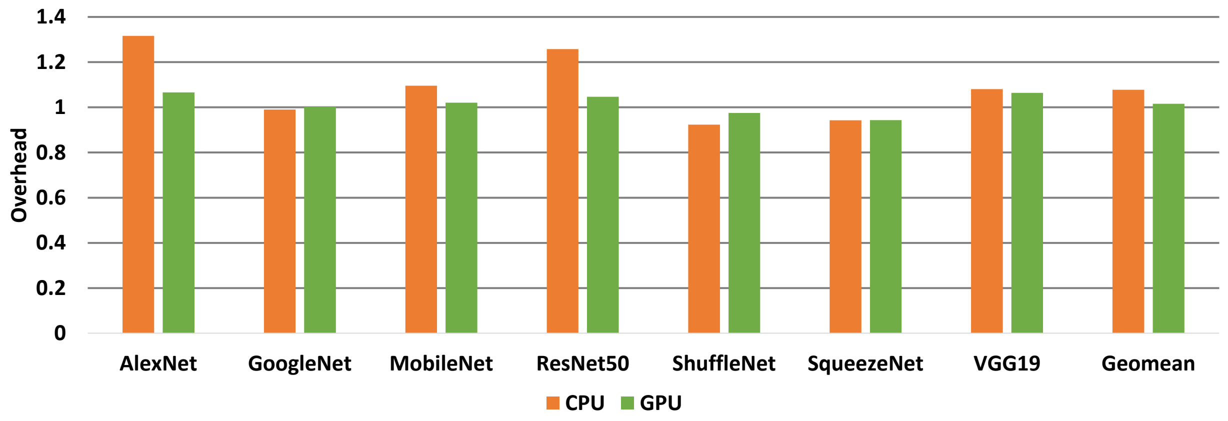


Figure 6.18: Model runtime overhead at optimal FILR design points, measured on a Jetson AGX Xavier.

We find that the measured runtime overheads are indeed platform-specific. Our measured CPU runtime overheads are higher on average, increasing the model runtime by approximately 7.7% on average, while GPU runtime overhead is much less at 1.6% on average. Many factors can influence the exact runtime of the models, including compiler optimizations, library implementations (in this case, cuDNN on CPU versus GPU), the hardware implementation which may accelerate certain computation (tensor cores on the GPU), as well as the possible availability of spare resources such that duplicate computations are run in parallel to effectively hide the latency of protection. While exact runtime overheads would require a more thorough analysis, this study shows the portability of implementing our techniques on different hardware platforms, as well as show the diverging results which may be accompanied by the platform of choice.

6.9 SUMMARY

As CNNs are increasingly employed in high performance and safety-critical applications, ensuring they are resilient to transient hardware errors is important. Full duplication provides high error coverage, but the overheads are prohibitively high for resource-constrained systems. Instead, selective protection targeting the most vulnerable work or components can provide a low-cost solution compared to employing indiscriminate redundancy across the board.

In this chapter, we develop and evaluate two selective protection techniques for CNNs that target different granularities. First, we develop a feature-map level resilience technique (FLR), which identifies and statically protects the most vulnerable feature maps in a CNN. Second, we develop an inference level resilience technique (ILR), which selectively reruns vulnerable inferences by analyzing their output. Finally, we show that the combination of both techniques (called FILR) is highly efficient. Our results show that the combination can achieve very high error coverage of 99.78%, while incurring only 48% overhead on average (and as low as 20% for ResNet50, or 5× less overhead compared to full duplication).

Chapter 7: Related Work

7.1 GENERAL PURPOSE RESILIENCY

Many successful analysis techniques have been proposed to address soft errors in both hardware and software. Here, we compare these approaches to our work.

Hardware resiliency analysis techniques: As described in Chapter 1, hardware resiliency analysis techniques can generally be categorized into two groups:

1) Techniques that rely purely on static/dynamic program analysis of error-free execution to model error propagation. ACE [35] analysis is often used to measure the Architectural Vulnerability Factors (AVF) [33, 35, 216, 217, 218] of hardware structures. PVF [27] isolates purely (program or software dependent) architecture-level vulnerabilities in the AVF; ePVF [36] further isolates bits that may lead to crashes and achieves a more accurate estimation of the program's SDC vulnerability. Many cross-layer resiliency solutions have been proposed using these techniques [219, 220]. Shoestring [30] uses a compiler analysis to identify vulnerable program locations. While fast, these techniques do not precisely model an error's impact on the execution because they use information from only an error-free execution.

2) Techniques that employ error injections. While typically slower than the previous group, these techniques employ error injections at different hardware and software abstractions [37, 38, 39, 40, 41, 102, 221, 222, 223, 224]. Some rely predominantly on statistical error injections for vulnerability analysis [32, 225, 226, 227, 228]. Others employ a hybrid approach combining combine program analysis with selected error-injection campaigns [41, 42, 44, 62, 229, 230]. For example, MeRLiN [41] applies ACE-like analysis and error pruning to accelerate statistical micro-architectural error injections. It can provide fine-grained reliability estimates for hardware structures and SDC vulnerability estimates for software. VTrident [62] uses error injections in static instructions to build an input-dependent model on top of Trident's [61] error propagation analysis to predict the instruction's SDC vulnerability. Rezlyer analysis (which is leveraged heavily in Approxilyzer as described in Chapter 2), is also a hybrid technique, but the primary goal is not a statistical average or probability—it is to determine precisely if/how an error in any specific instruction will impact the final output.

Concepts similar to Minotaur: We discuss the most directly related works from other domains with similarities to different concepts in Minotaur in Chapter 3. IRA [99] uses statistical techniques to generate reduced canary inputs that are used to explore different approximation techniques; once an appropriate technique is found, it is applied to the larger input. In Minotaur, the Min input is used not just for exploration, but also for the final resiliency analysis. The Ref input is analyzed only if additional accuracy is desired from multiple inputs and even so, only a subset of Ref needs analysis. A key difference is that IRA targets online production time analysis whereas Minotaur is motivated by offline development time analysis.

DeepXplore [231] proposes the criterion of neuron coverage for quantifying the fraction of a deep learning system's logic exercised by a set of test inputs based on the number of neurons activated by the inputs. Neuron coverage is an orthogonal application-specific input quality criterion that could be employed by Minotaur for appropriate domains.

There are several (static and runtime) approaches in other contexts that share the same goal as Minotaur's early termination technique, namely, cutting the computation short without sacrificing accuracy [229, 232, 233, 234]. A recent example is SnaPEA [232] where convolution operations are terminated early if their output is predicted to be zero.

MinneSPEC [235] aims to provide reduced input workloads to improve performance (usually runtime of applications), which differs from our objective of uncovering SDC-PCs.

Minotaur is an orthogonal technique that can be used to improve many of the above techniques. In general, the concepts of measuring input quality and input minimization are broadly applicable to all techniques that use application inputs. PC coverage as an input quality criterion can conceptually apply to many of the above techniques, but it needs experimental verification. Error injection prioritization can be directly applied to all techniques that use error injections. Input prioritization is also a general concept that can be applied in cases where multiple inputs are used.

7.2 APPROXIMATE COMPUTING

Many techniques have been proposed that leverage approximate computing for improved performance, energy or reliability. Loop perforation [78], voltage scaling [236, 237], approximate ALU computations [238, 239], approximate kernels [240, 241, 242], neural hardware [243], and memory system approximations [244, 245] are all possible techniques that trade off accuracy for system benefits.

Programming language support, such as that in [83, 239, 246, 247, 248] helps programmers abstractly express approximations and check program correctness at the cost of increased programmer burden. Recent frameworks [247, 249, 250] build on these languages to automatically identify approximate regions of the code while providing some statistical [250] or probabilistic [247] guarantees on the final end-to-end error. While these frameworks advance the state-of-the-art to greatly reduce programmer burden, they still require the programmer to adopt a new programming language and/or modify their source code. Thus, they cannot be used for very large multi-kernel programs (static analysis may be complicated and under-estimate the approximation potential) or for programs where the source code is not available (such as legacy code). We believe that Approxilyzer is a complementary technique and can be used as a front end plugin to these frameworks. A concurrent work [251] provides statistical guarantees on final output quality given an approximate kernel and accelerator configuration using compiler support and hardware binary classifiers. While this work focuses on coarse-gain approximation with accelerators, Approxilyzer is a general framework which studies approximation at the fine granularity of single instructions.

SAGE [241] automatically generates approximate kernels for GPUs but like other methods [240, 252] uses an online mechanism to catch unacceptable quality degradation in a reactive fashion. Approxilyzer on the other hand provides offline output quality information. Techniques such as [253] control output quality constraints by tuning various knobs in an approximate program. Approxilyzer solves the problem of identifying approximate code and as such is an orthogonal technique.

The idea of identifying unacceptable output corruptions for selective reduced-cost resiliency protection has previously been explored in the realm of soft computations. A combination of error injections and static analysis is used in [254] to identify Egregious Data Corruption (EDC) prone code and data segments in soft computations that can then be protected by detector placements. IPAS [255] uses machine learning to identify and protect only those Silent Output Corruptions (SOC) instructions that alter the output of scientific codes. Khudia et al. [256] use compiler analysis to identify critical variables in the application that are likely to generate Unacceptable Silent Data Corruptions (USDCs) in the presence of errors and only protect those using strategic expected value checks. Approxilyzer classifies error outcomes into categories based on approximation potential and predicts the impact errors in individual instructions with high accuracy. This allows for very fine tuning of resiliency protection schemes for different quality and overhead requirements.

Application of approximate computing to hardware resiliency has also been demonstrated in specialized domains such as bio-medical applications. Sabry et al. [257] study Electrocardiogram (ECG) monitoring wireless body sensor nodes and tradeoff inaccuracies inherent to the domain to achieve resiliency overhead savings. Approxilyzer also exploits accuracy loss for resiliency overhead savings but does so in a manner that can be used by any general-purpose program.

Many techniques have been proposed that leverage approximate computing at the level of software [75, 78, 240, 241, 242, 251, 253, 258, 259, 260, 261, 262], programming languages [83, 239, 246, 248, 249, 250, 263] and hardware [243, 244, 261, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275] for improved performance, energy, or reliability. Criticality-testing [104, 254, 276, 277, 278, 279, 280] of approximate computations is important for many domains. Minotaur is an orthogonal set of techniques that can be used to improve many of these analyses that use application input(s).

7.3 GPU RESILIENCY

Software introduced redundancy: Prior techniques have introduced redundancy at multiple granularities including the process, GPU kernel, thread, and assembly instruction level. Process-level redundancy replicates the process and compares results at system call boundaries [52, 281]. This approach suffers from limitations for multi-threaded workloads. Kernels or thread blocks can be re-executed and their outputs then compared to ensure correctness [125]. This approach is challenging for workloads where the kernel or block outputs are non-deterministic, which can arise from rounding errors and reading clock values during execution. Thread-level duplication (TLD) has been employed for CPUs [281, 282, 283, 284] and GPUs [125, 126, 127] and requires spare hardware resources. Wadden et al. [126] and Gupta et al. [127] each proposed a compiler-based approach for GPUs that duplicates thread-blocks and threads, and observed high overheads for block-level duplication due to inter-block communication and synchronization. We quantitatively compare SInRG to TLD in Chapter 4. One drawback for TLD is that programmer intervention may be required to ensure spare hardware resources are available. Intra-warp communication constructs such as warp vote and shuffle operations, for example, must be handled accordingly for proper TLD operation.

Software instruction-level duplication does not have these limitations and has been explored for CPUs [116, 117, 118]. Oh et al. [116] proposed a technique to duplicate instructions at the assembly level and insert checking instructions to validate the results. The average runtime overheads were approximately 60% on a 4-way issue superscalar processor. SWIFT [117] proposed a compiler-based approach and exploited wide, underutilized processors by scheduling both original and duplicated instructions in the same execution thread, and reported overheads of about 40% on Intel Itanium CPUs. The applicability of such techniques for GPUs has not been studied previously, and SInRG addresses this gap. To reduce the overheads further, Shoestring developed a compiler technique to selectively duplicate instructions by trading off coverage for performance [67]. Combining such a technique with SInRG is an interesting future direction.

Hardware introduced redundancy: Traditional business-class systems (e.g., IBM Z

Series machines [51]) employ expensive hardware-managed dual- or triple-modular redundancy schemes at prohibitively high cost for commercial use. In safety-critical systems, similar techniques are being employed to meet the safety integrity requirements [50]. Recent server and business-class processors (e.g., IBM System Z machines [285]) adopt fine-grained hardware checkers to detect errors in individual processor components, presumably with substantive design effort. Such an approach has also been explored for GPUs [286].

Warped-DMR [115] and RISE [287] proposed hardware mechanisms to exploit underutilized parallelism in GPUs for error detection. Specifically, Warped-DMR uses the idle single-instruction, multiple thread (SIMT) lanes to redundantly execute some of the threads within the warp and achieve intra-warp DMR execution. RISE proposed mechanisms to predict and use idle SM cycles and SIMT lanes to execute redundant work [287]. Warped-RE extended these approaches and introduced redundancy to verify every warp instruction [288]. It re-executes the instruction to correct any detected errors. Each of these techniques requires complex hardware changes and they are not directly comparable to SInRG techniques.

7.4 CNN RESILIENCY

With the increasing use of CNNs in HPC and safety-critical applications, there has been a recent surge in research on CNN resilience. Many proposed methods for CNN resilience require network retraining. This has been performed for redistributing vulnerability across a network [208, 289] and introducing additional components that require fine-tuning [209]. One goal of our work is to avoid training altogether due to its high associated costs. Retraining is also often not an option for proprietary models, as the training recipe and datasets may be unavailable.

Many works have explored error injection analysis for CNNs [148, 150, 208, 209, 290, 291, 292, 293]. Most of these works, however, have limited resiliency analysis studies by either focusing on only a few small networks (using MNIST and CIFAR10 datasets), performing relatively few injection experiments (a few 1000 injections per *network*), or cap the number of images studied. To the best of our knowledge, we perform the first large scale CNN reliability study across many networks and datasets. Further, we introduce a new method, ΔLoss, which opens up several new research directions including allowing other techniques to validate their results for accuracy without limitations of very large error injection campaigns.

Prior work has explored performing selective duplication at finer granularities, such as kernel-level duplication in GPUs [290], layer level duplication [291], feature map level duplication [208], and neuron level duplication [209]. Further, recent work has shown that software level TMR hardening can provide similar error protection compared to hardware duplication, while costing less area [291]. In this work, we target fmap-level resilience in software with FLR and inference-level resilience for ILR. While this work is not the first to target fmaps level granularity, we are the first to evaluate protection of fmaps without retraining and while performing a large scale study and analysis. To the best of our knowledge, this is the first work to introduce ILR, as well as a combination of two granularities for holistic CNN resiliency.

CNN model pruning techniques aim to remove redundant and less-useful parameters from a model to improve execution efficiency [294], which can be considered an inverse operation of hardening. These techniques often reduce accuracy by a few small factors. FLR focuses on identifying vulnerable feature maps, which it then proceeds to duplicate to improve reliability, with no effect on classification accuracy. There are many similarities between pruning and hardening. (1) Pruning is typically a two-phased process. The first phase identifies a filter to remove, and the second phase (called a fine-tuning phase) removes the filter and retrains the network. (2) Recent work found that pruning full filters (rather than individual weights in a filter) can have minimal effects on accuracy, while improving the pruning speed [295]. This is analogous to our fmap target granularity and protection strategy, where FLR duplicates feature maps instead of neurons. (3) Pruning techniques rank filters using heuristics to identify candidates to prune [295, 296]. We also explore similar heuristics to estimate fmaps based on vulnerability, as described in Section 6.7.1. The objective of a pruning technique is to zero-out a filter, removing it from the model. In contrast, for the resiliency analysis, we assume various error models which change a single neuron.

Recent work in this domain has also explored mapping lower-level error models to higher level models [175], as well as the nearby work of approximations for accelerating CNN computations via quantization [297]. Both these works can potentially be analyzed by FILR for resiliency analysis, and may provide good future directions for exploration between quantization, different error models, and CNN resilience. Adversarial attacks [152, 153, 154, 155, 156, 157, 158, 159, 160, 298] are another set of CNN perturbations similar in spirit to soft errors (in that they can affect the output of a classification), with the subtle difference that adversarial attacks focus on the input into the CNN, while transient errors in computations are typically modeled on the internal CNN computations [69, 148, 175, 194, 195, 196].

Chapter 8: Conclusion and Future Work

8.1 CONCLUSIONS

Modern systems at scale are increasingly susceptible to transient hardware errors at current technology sizes. With commodity hardware used across various systems with a range of reliability requirements, it is important to design tunable, low-cost resiliency solutions to address the issue of hardware errors. In this thesis, we promote software-directed, selective duplication as a means to avoid the high overheads of traditional full duplication methods, while maintaining high error coverage.

Approxilyzer [44] and gem5-Approxilyzer [45] are general purpose tools, which allow the developer to understand the impact of an error on the applications output. Using the novel pruning techniques introduced by Relyzer [42], Approxilyzer takes as input a quality metric for an application and automatically (without user directives) provides a resiliency profile at the instruction granularity for a single-bit flip error model. Effectively, it allows the user to understand the impact of virtually any bit-flip on the outcome of the application, and allows the user to subsequently tune their resiliency versus overhead versus output quality based on the system requirements. This key insight explicitly identifies the duality between resilience and approximate computing, and we explore both opportunities given an application's error profile.

While very powerful as a resiliency analysis tool, Approxilyzer need to be accelerated and more scalable for general use. We introduce Minotaur [46], a software-testing inspired toolkit to speed up the runtime of resiliency analysis tools such as Approxilyzer. While Approxilyzer's analysis is orders of magnitude better than a naive error analysis, it can still be slow due to the large number of possible error sites requiring detailed study. Minotaur bridges between software testing and hardware resiliency by adapting four techniques from the software engineering domain to make hardware error analysis faster and thus more scalable. We show that Minotaur can significantly improve the runtime of Approxilyzer, while *simultaneously* improving its accuracy in identifying hardware errors as well.

The third contribution of this thesis focuses on reducing the implementation overhead of instruction-level duplication, by taking into consideration the hardware platform and unique opportunities provided by the backend architecture. We develop a family of techniques called SInRG [47], which is the first practical approach to software-directed instruction duplication for GPU-based systems. With SInRG, we identify GPU-specific opportunities of overhead reduction, explore software and hardware performance optimizations to lower replication overhead, and implement an auto-tuner to selectively choose the best duplication technique for a give workload.

We find a common theme across the general-purpose tools and techniques discussed in this thesis. Specifically, tuning techniques and analyses to the application on hand can provide additional opportunity for reducing overheads and identifying errors. The fourth contribution expands this observation, by performing domain-specific resiliency analysis on convolutional neural networks (CNNs), due to their prevalence in safety-critical tasks. We developed an open-source CNN perturbation tool, PyTorchFI [48], and use it to introduce two selective protection schemes at different granularities: feature maps level resilience (FLR) and inference level resilience (ILR). Moreover, we show that the combination of the two resilience schemes is better than the sum of their parts (called FILR). On average, we find that FILR can achieve very high coverage (99.78%) while incurring only 48% overhead on average (as low as 20% for ResNet50, or 5× less overhead compared to full duplication).

Overall, the goal of this thesis makes fundamental contributions in the space of soft error resilience, by using principled and scalable software directed techniques for error analysis. Furthermore, this thesis promotes specialized resiliency analysis (by introducing novel techniques in the space of CNN resilience) as a means for low-cost resiliency techniques by understanding how hardware errors propagate at the software level to impact the outcome of an application. This opens the door for many new domain-specific resiliency analysis tools and techniques, with use cases beyond just resilience, such as the fertile domains of approximate computing and deep neural network design.

8.2 LIMITATIONS AND FUTURE DIRECTIONS

8.2.1 Error Model Exploration

One of the challenges associated with hardware resilience is fully capturing how a natural phenomenon (such as an alpha particle strike) maps to an architectural error. In this work, we assume a single-bit flip error at the register level (in Chapters 2, 3, and 4) and a single-bit flip error at the neuron level (in Chapter 6). While these are commonly used error models and represent real errors, single-bit flip models are not representative of all real errors in hardware [101, 175, 210], and a challenging aspect of hardware resilience is identifying the correct error model to use to study an application's error tolerance. Additionally, other error types in hardware can also be important, such as permanent errors or intermittent errors, each of which may provide more insight into an application's resilience.

For example, multiple bit flips within the same register (or neuron) or across multiple registers (or neurons) can be useful to analyze or discover whether an application can be approximated in face of more than one error. Additionally, a more precise error modeling is a more attractive solution for industry practitioners and/or certification standards for hardware. Thus, while useful to study and design tools around, a single-bit flip error model is not the only model, and exploration of other error models can be useful in many capacities.

Many of the tools and techniques described here can be modified to work with other error models. In particular, PyTorchFI was designed ground-up with this feature in place, and we provide a few simple examples in Section 5 on how different error models can be utilized in different capacities.

8.2.2 System Level Resilience

Studying the impact of an error within a single building block of a system (as we did in Chapter 6 for CNNs) can provide insights into the error propagation and inherent error tolerance of the building block. However, at a system level, it is important to understand where and how such a building block fits into the overall design and workflow of the system, especially since it can provide opportunities to further reduce resilience overheads.

For example, in an autonomous vehicles (AV) setting, understanding how a CNN classification is used in the next stage of the system can indicate whether a certain CNN needs hardening or not. Many systems (including AV systems) have many safe-guards in the software already, in order to tolerate errors from a machine learning model. Understanding the software-introduced redundancy can help direct the developer to the most vulnerable units in the system, and can also inform developers where to focus resiliency analysis efforts to optimize hardening solutions in the system.

Incorporating an entire system is a challenging task – however, it provides many opportunities for reducing overheads by either hiding the costs (as illustrated with SInRG in Chapter 4) or avoiding protection altogether if it is unnecessary (as explored independently with Approxilyzer and FILR). By generalizing error analyses and tools to explore error propagation *across* system components, additional insights can be gathered to optimize system-level resilience.

8.2.3 Transfer Resilience

A fascinating concept in the domain of deep learning is transfer learning. The main premise is that one can train a network on one dataset, and then transfer the weights to another dataset to "jump-start" the training with a high accuracy. We find that this is very common in the medical field, where most networks which perform well are transferred from the Inception-Net neural network which is training on ImageNet.

From a reliability standpoint, an interesting future step is exploring whether the reliability properties can also be transferred in the process. Further, extrapolating beyond neural networks, do certain programming patterns exhibit similar reliability characteristics? Such "code primitives" can be used to understand a program's expected robustness while running on unreliable hardware by analyzing the source code, which would be much faster than current methods while also providing hints to the developer on how to structure their code. From a hardening perspective, a similar avenue would be to identify applications and code patterns which behave similarly from a reliability standpoint, and target reliability solutions in the hardware to take advantage of the identified pattern. This would enable a more tightly coupled hardware-software co-design for reliability, as well as bring the software developer into the picture as they know their code the best and further extending what Minotaur (Chapter 3) began for combining software testing and hardware resilience.

8.2.4 A Unifying Theory of Errors for DNNs

The concept of an "error" in the domain of deep learning is severely overloaded. An error in the context of resilience, for example, denotes an output corruption due to a natural occurrence such as a transient bit flip during computation. However, in the context of an adversarial input to the network, an error occurs at the *input* of the network, fooling the network into a misprediction. Moreover, a validation error in DNNs relates to the training accuracy of the model, and is predicated on the statistical nature of the machine learning model. While all these "errors" are seemingly different, they all share a common theme: an undesirable and incorrect outcome.

Many concepts may be adapted across these sub-domains for better accuracy or (analogously) fewer errors. For example, training with errors as exemplified in Section 5.1 can make a network more resilient, with minimal effect on validation accuracy. Another possible research direction is to incorporate Top2Diff in the cost function during training, to maximize the distance between the top two classes for strong confidence in predictions.

Overall, finding overlaps between the different "error models" in the domain of DNNs is a tantalizing opportunity for improving deep learning models. Theoretically formulating the relationship is also grounds for future work.

8.2.5 Bridging Between General Purpose Analysis and Specialized Error Analysis

Another interesting direction from this thesis is the exploration of the efficacy of general purpose techniques versus specialized error analysis. Studying the benefits and trade-offs of both techniques can help inform when a general-purpose tool can be enough to analyze an application versus when it would be advantageous to develop specialized domain-specific tools for application error analysis. In other words, similar to the approach taken by FILR to combine two techniques at different granularities, it would be interesting to study how an instruction-level analysis intersects with a specialized granularity of analysis such as identifying vulnerable feature maps in CNN. Furthermore, are certain domains more amenable to general-purpose solutions, while others benefit more from a specialized approach to resilience? A taxonomy and categorization of the two quantitatively is both current limitation and a promising opportunity for additional research.

8.2.6 A Software Testing Framework for Error Analysis

Chapter 3 introduced the concept of bridging between software testing and resiliency analysis. More generally, it paves the way for a plethora of future research that can create a software engineering discipline for hardware errors. It should be possible to adapt many more techniques from the rich literature on software testing [85, 90, 299, 300, 301, 302] and program analysis [303, 304, 305, 306] to provide principled and scalable approaches for software resiliency to hardware errors.

For example, a critical requirement for scalable analysis is to keep up with an evolving code base and its implications in the face of hardware errors (*incremental analysis*). Even subtle programmatic changes can alter an application's resiliency profile, and affect which parts of the program are more susceptible to hardware errors. Furthermore, current resiliency analysis techniques generally require the entire program to be available, and any change in this program requires the expensive analysis to be rerun from scratch. In a real software development workflow, the entire program is rarely available and tested at once; rather, individual program units are tested and then composed together (*compositional analysis*). Practical tools and workflows must support both incremental *and* compositional resiliency analysis and hardening.

The software engineering discipline has architected scalable solutions for the similar obstacle of finding bugs in code as the code evolves, with *incremental* analysis techniques such as delta testing, mutation testing, and automated test generation for extensive code coverage. Various techniques also exist for *compositional* analysis of code, leveraging symbolic testing and black-box testing techniques to analyze large systems.

This work opens the door to adaptation and adoption of the above (and many other) software engineering techniques for addressing hardware errors. A long-term goal is to enable research to incorporate resiliency analysis and hardening seamlessly within the software development workflow; i.e., to create a software engineering discipline for hardware errors. Minotaur enables this goal by taking the first steps at identifying and bridging the gap between these two domains.

References

[1] S. Borkar, “Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation,” *IEEE Micro*, vol. 25, no. 6, 2005.

[2] N. DeBardeleben, J. Laros, J. T. Daly, S. L. Scott, C. Engelmann, and B. Harrod, “High-end Computing Resilience: Analysis of Issues Facing the HEC Community and Path-forward for Research and Development,” *Whitepaper*, 2009.

[3] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, and M. Snir, “Toward Exascale Resilience: 2014 Update,” *Supercomput. Front. Innov.: Int. J.*, 2014.

[4] P. Ricoux, “European Exascale Software Initiative EESI2-Towards Exascale Roadmap Implementation,” *2nd IS-ENES workshop on high-performance computing for climate models*, 2013.

[5] 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*,” *International Journal of High Performance Computing*, 2014.

[6] J. F. Ziegler and H. Puchner, *SER–history, Trends and Challenges: A Guide for Designing with Memory ICs.* Cypress, 2004.

[7] P. Kogge, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, and R. Lucas, “Exascale computing study: Technology challenges in achieving exascale systems,” *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Techinal Representative*, vol. 15, 01 2008.

[8] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, “Toward exascale resilience,” *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, p. 374–388, Nov. 2009. [Online]. Available: https://doi.org/10.1177/1094342009347767

[9] J. Yoshida, “Toyota case: Single bit flip that killed,” https://www.eetimes.com/document.asp?doc_id=1319903#, 2013.

[10] Safety Research and Strategies, Inc., “Toyota unintended acceleration and the big bowl of ’spaghetti’ code,” ”http://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-%E2%80%9Cspaghetti%E2%80%9D-code”, 2013.

[11] S. Levy, K. B. Ferreira, N. DeBardeleben, T. Siddiqua, V. Sridharan, and E. Baseman, "Lessons learned from memory errors observed over the lifetime of cielo," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 554–565.

[12] S. Di, H. Guo, R. Gupta, E. R. Pershey, M. Snir, and F. Cappello, "Exploring properties and correlations of fatal events in a large-scale hpc system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 2, pp. 361–374, 2019.

[13] L. Tan and N. DeBardeleben, "Failure analysis and quantification for contemporary and future supercomputers," *ArXiv*, vol. abs/1911.02118, 2019.

[14] M. Dimitrov and H. Zhou, "Unified Architectural Support for Soft-Error Protection or Software Bug Detection," in *Proc. of International Conference on Parallel Archtectures and Compilation Techniques (PACT)*, 2007.

[15] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-Error Detection Using Control Flow Assertions," in *Proc. of International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003.

[16] S. K. S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve, "mSWAT: Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems," in *Proc. of International Symposium on Microarchitecture (MICRO)*, 2009.

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

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

[19] S. Sahoo, M.-L. Li, P. Ramchandran, S. V. Adve, V. Adve, and Y. Zhou, "Using Likely Program Invariants to Detect Hardware Errors," in *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2008.

[20] N. J. Wang and S. J. Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, July-Sept 2006.

[21] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost Program-level Detectors for Reducing Silent Data Corruptions," in *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2012.

[22] M. Dimitrov and H. Zhou, "Anomaly-based Bug Prediction, Isolation, and Validation: An Automated Approach for Software Debugging," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[23] G. Lyle, S. Chen, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "An End-to-end Approach for the Automatic Derivation of Application-Aware Error Detectors," in *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2009.

[24] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *Proc. of International Symposium on Microarchitecture (MICRO)*, 2007.

[25] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based Fault Screening," in *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[26] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults," in *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2008.

[27] V. Sridharan and D. R. Kaeli, "Eliminating Microarchitectural Dependency from Architectural Vulnerability," in *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[28] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLFIED: Symbolic Program-level Fault Injection and Error Detection Framework," in *International Conference on Dependable Systems and Networks*, 2008.

[29] A. Benso, S. D. Carlo, G. D. Natale, P. Prinetto, and L. Tagliaferri, "Data Criticality Estimation in Software Applications," in *Proc. of International Test Conference (ITC)*, 2003.

[30] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic Soft Error Reliability on the Cheap," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[31] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra, "CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16, 2016.

[32] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding Error Propagation in Deep-Learning Neural Networks (DNN) Accelerators and Applications," in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[33] X. Li, S. Adve, P. Bose, and J. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors," in *Proc. of International Symposium on Computer Architecture (ISCA)*, 2007, pp. 341–352.

[34] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "Measuring Architectural Vulnerability Factors," *IEEE Micro*, vol. 23, no. 6, pp. 70–75, Nov. 2003.

[35] 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 *Proc. of International Symposium on Microarchitecture (MICRO)*, 2003.

[36] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2016, pp. 168–179.

[37] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," in *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2014, pp. 375–382.

[38] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance," in *European Dependable Computing Conference (EDCC)*, 2015, pp. 245–255.

[39] J. Calhoun, L. Olson, and M. Snir, "FlipIt: An LLVM based fault injector for HPC," in *European Conference on Parallel Processing*. Springer, 2014, pp. 547–558.

[40] J. Li and Q. Tan, "SmartInjector: Exploiting Intelligent Fault Injection for SDC Rate Analysis," in *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, 2013, pp. 236–242.

[41] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "MeRLiN: Exploiting Dynamic Instruction Behavior for Fast and Accurate Microarchitecture Level Reliability Assessment," in *Proc. of International Symposium on Computer Architecture (ISCA)*, 2017.

[42] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[43] S. K. Sastry Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "GangES: Gang Error Simulation for Hardware Resiliency Evaluation," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 61–72, June 2014.

[44] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a Systematic Framework for Instruction-level Approximate Computing and its Application to Hardware Resiliency," in *Proc. of International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–14.

[45] R. Venkatagiri, K. Ahmed, A. Mahmoud, S. Misailovic, D. Marinov, C. Fletcher, and S. Adve, "gem5-approxilyzer: An open-source tool for application-level soft error analysis," *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 214–221, 2019.

[46] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve, "Minotaur: Adapting software testing techniques for hardware errors," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304050 p. 1087–1103.

[47] 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, 2018.

[48] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "Pytorchfi: A runtime perturbation tool for dnns," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2020, pp. 25–31.

[49] A. Mahmoud, S. Hari, C. W. Fletcher, S. Adve, C. Sakr, N. R. Shanbhag, P. Molchanov, M. Sullivan, T. Tsai, and S. W. Keckler, "HarDNN: Feature Map Vulnerability Evaluation in CNNs," *ArXiv*, vol. abs/2002.09786, 2020.

[50] X. Iturbe, B. Venu, E. Ozer, and S. Das, "A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, June 2016, pp. 246–249.

[51] W. Bartlett and L. Spainhower, "Commercial Fault Tolerance: A Tale of Two Systems," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, pp. 87–96, January 2004.

[52] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 6, no. 2, pp. 135–148, April 2009.

[53] Sean Hollister, "Tesla's new self-driving chip is here, and this is your best look yet," 2019. [Online]. Available: https://www.theverge.com/2019/4/22/18511594/tesla-new-self-driving-chip-is-here-and-this-is-your-best-look-yet

[54] "John Hennessy and David Patterson Deliver Turing Lecture at ISCA 2018," June 2018, https://www.acm.org/hennessy-patterson-turing-lecture.

[55] E. Cheng, S. Mirkhani, L. G. Szafaryn, C. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra, "Tolerating soft errors in processor cores using clear (cross-layer exploration for architecting resilience)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1839–1852, Sep. 2018.

[56] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," in *Proc. of International Symposium on Microarchitecture (MICRO)*, dec 2003.

[57] A. DeHon, H. M. Quinn, and N. P. Carter, "Vision for cross-layer optimization to address the dual challenges of energy and reliability," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 1017–1022.

[58] N. P. Carter, H. Naeimi, and D. S. Gardner, "Design techniques for cross-layer resilience," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 1023–1028.

[59] M. S. Gupta, J. A. Rivers, L. Wang, and P. Bose, "Cross-layer system resilience at affordable power," in *2014 IEEE International Reliability Physics Symposium*, June 2014, pp. 2B.1.1–2B.1.8.

[60] E. Rotenberg, "Ar-smt: a microarchitectural approach to fault tolerance in microprocessors," in *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, June 1999, pp. 84–91.

[61] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling Soft-Error Propagation in Programs," in *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2018.

[62] G. Li and K. Pattabiraman, "Modeling Input-Dependent Error Propagation in Programs," in *Proc. of International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 279–290.

[63] Q. Lu, G. Li, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Configurable detection of sdc-causing errors in programs," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 3, pp. 88:1–88:25, Mar. 2017. [Online]. Available: http://doi.acm.org/10.1145/3014586

[64] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "Llfi: An intermediate code-level fault injection tool for hardware faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 11–16.

[65] 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 *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 249–258.

[66] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Hamartia: A fast and accurate error injection framework," in *Proceedings of the International Conference on Dependable Systems and Networks Workshops (DSN-W).* IEEE, 2018, pp. 101–108.

[67] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proceedings of the the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 385–396.

[68] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: Intelligent protection against silent output corruption in scientific applications," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO).* IEEE, 2016, pp. 227–238.

[69] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2018.

[70] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer code repository," Website, 2016. [Online]. Available: https://cs.illinois.edu/approxilyzer

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

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

[73] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[74] I. Akturk, K. Khatamifard, and U. R. Karpuzcu, "On quantification of accuracy loss in approximate computing," in *Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, 2015.

[75] J. Park, E. Amaro, D. Mahajan, B. Thwaites, and H. Esmaeilzadeh, "AxGames: Towards Crowdsourcing Quality Target Determination in Approximate Computing," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 623–636.

[76] R. Venkatagiri, A. Mahmoud, and S. Adve, "Towards more precision in approximate computing," in *Workshop on Approximate Computing Across the Stack (WAX)*, 2016.

[77] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proceedings of the 20th Annual International Conference on Supercomputing*, New York, NY, USA, 2006, pp. 324–334.

[78] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard, "Managing Performance vs. Accuracy Trade-offs with Loop Perforation," in *SIGSOFT FSE*, 2011, pp. 124–134.

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

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

[81] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Computer Architecture News*, vol. 33, no. 4, 2005.

[82] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[83] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A Programmer-guided Compiler Framework for Practical Approximate Computing," in *Technical Report UW-CSE-15-01-01, University of Washington*, 2015.

[84] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *DATE*, 2009.

[85] G. J. Myers, *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 1979.

[86] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 367–375, 1985.

[87] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[88] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, Oct 1988.

[89] International Organization for Standardization, "Road vehicles – Functional safety," https://www.iso.org/standard/43464.html, 2011.

[90] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *STVR*, vol. 22, no. 2, pp. 67–120, 2012.

[91] C. Zhang, A. Groce, and M. A. Alipour, "Using Test Case Reduction and Prioritization to Improve Symbolic Execution," in *ISSTA*, 2014, pp. 160–170.

[92] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case Reduction for C Compiler Bugs," in *Proc. of International Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[93] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction: Delta debugging, even without bugs," *STVR*, vol. 26, no. 1, pp. 40–68, 2015.

[94] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause Reduction for Quick Testing," in *ICST*, 2014, pp. 243–252.

[95] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce, "Evaluating Non-Adequate Test-Case Reduction," in *Proc. of International Conference on Automated Software Engineering (ASE)*, 2016.

[96] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *TSE*, vol. 28, no. 2, pp. 183–200, 2002.

[97] J. Brauer, M. Dahlweid, T. Pankrath, and J. Peleska, "Source-Code-to-Object-Code Traceability Analysis for Avionics Software: Don'T Trust Your Compiler," in *Proceedings of the 34th International Conference on Computer Safety, Reliability, and Security - Volume 9337*, ser. SAFECOMP 2015, 2015.

[98] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, and T. Quinot, "Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework," in *Embedded Real Time Software and Systems (ERTSS)*, 2010.

[99] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation," in *Proc. of International Conference on Programming Language Design and Implementation (PLDI)*, 2016, pp. 161–176.

[100] I. S. LLC, "Image Magick," Website, 2018, https://www.imagemagick.org/.

[101] 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 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[102] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra, "Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design," in *Proc. of International Design Automation Conference (DAC)*, 2013, pp. 1–10.

[103] D. E. Bernholdt, A. Geist, and B. Maccabe, "Resilience is a Software Engineering Issue," *Software Productivity for Extreme-Scale Science (SWP4XS) Workshop, Oak Ridge National Laboratory*, 2014.

[104] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of Service Profiling," in *Proc. of International Conference on Software Engineering (ICSE)*, 2010, pp. 25–34.

[105] M. Isenkul, B. Sakar, and O. Kursun, "Improved Spiral Test Using Digitized Graphics Tablet for Monitoring Parkinson's Disease," in *The 2nd International Conference on e-Health and Telemedicine (ICEHTM-2014)*, 05 2014.

[106] B. E. Sakar, M. E. Isenkul, C. O. Sakar, A. Sertbas, F. S. Gürgen, S. Delil, H. Apaydin, and O. Kursun, "Collection and Analysis of a Parkinson Speech Dataset With Multiple Types of Sound Recordings," *IEEE Journal of Biomedical and Health Informatics*, vol. 17, pp. 828–834, 2013.

[107] D. Dheeru and E. Karra Taniskidou, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[108] A. K. Mishra, R. Barik, and S. Paul, "iACT: A Software-hardware Framework for Understanding the Scope of Approximate Computing," in *WACAS*, 2014.

[109] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical Fault Injection: Quantified Error and Confidence," in *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*, April 2009.

[110] D. A. Oliveira, P. Rech, L. L. Pilla, P. O. Navaux, and L. Carro, "GPGPUs ECC Efficiency and Efficacy," in *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, October 2014, pp. 209–215.

[111] NVIDIA, "NVIDIA Tesla V100 GPU Architecture, The World's Most Advanced Datacenter GPU," http://www.nvidia.com/object/volta-architecture-whitepaper.html, 2017.

[112] NVIDIA, "NVIDIA Tesla P100, The Most Advanced Datacenter Accelerator Ever Built," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[113] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[114] 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," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, May 2014.

[115] H. Jeon and M. Annavaram, "Warped-DMR: Light-weight Error Detection for GPGPU," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2012, pp. 37–47.

[116] N. Oh, P. Shirvani, and E. J. McCluskey, “Error Detection by Duplicated Instructions in Super-scalar Processors,” *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, March 2002.

[117] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, “SWIFT: Software Implemented Fault Tolerance,” in *International Symposium on Code Generation and Optimization (CGO)*, March 2005, pp. 243–254.

[118] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, “Software-controlled Fault Tolerance,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 4, pp. 366–396, December 2005.

[119] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, “A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads,” in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, December 2010, pp. 1–11.

[120] M. Burtscher, R. Nasre, and K. Pingali, “A Quantitative Study of Irregular Programs on GPUs,” in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, November 2012, pp. 141–151.

[121] A. Nukada, H. Takizawa, and S. Matsuoka, “NVCR: A transparent checkpoint-restart library for NVIDIA CUDA,” in *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 104–113.

[122] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, “CheCUDA: A checkpoint/restart tool for CUDA applications,” in *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, December 2009, pp. 408–413.

[123] N. Toan, H. Jitsumoto, N. Maruyama, T. Nomura, T. Endo, and S. Matsuoka, “MPI-CUDA applications checkpointing,” in *IPSJ SIG Technical Report*, August 2010, pp. 1–7.

[124] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, “CRUM: Checkpoint-Restart Support for CUDA’s Unified Memory,” in *Proceedings of International Conference on Cluster Computing (CLUSTER)*, September 2018.

[125] M. Dimitrov, M. Mantor, and H. Zhou, “Understanding Software Approaches for GPGPU Reliability,” in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, March 2009, pp. 94–104.

[126] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, “Real-World Design and Evaluation of Compiler-Managed GPU Redundant Multithreading,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 73–84.

[127] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, and R. Gupta, "Compiler Techniques to Reduce the Synchronization Overhead of GPU Redundant Multithreading," in *Proceedings of the Design Automation Conference (DAC)*, June 2017, pp. 1–6.

[128] NVIDIA, "CUDA C Programming Guide :: CUDA Toolkit Documentation," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, September 2017.

[129] Khronos Group, "The Open Standard for Parallel Programming of Heterogeneous Systems," March 2017. [Online]. Available: https://www.khronos.org/opencl/

[130] NVIDIA, "PTX ISA :: CUDA Toolkit Documentation," http://docs.nvidia.com/cuda/parallel-thread-execution/, September 2017.

[131] NVIDIA, "CUDA Binary Utilities :: CUDA Toolkit Documentation," http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html, September 2017.

[132] P. Xiang, Y. Yang, M. Mantor, N. Rubin, and H. Zhou, "Revisiting ILP designs for throughput-oriented GPGPU architecture," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2015, pp. 121–130.

[133] A. Nistor, D. Marinov, and J. Torrellas, "Instantcheck: Checking the determinism of parallel programs using on-the-fly incremental hashing," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2010, pp. 251–262.

[134] J. Ohlsson and M. Rimen, "Implicit Signature Checking," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, June 1995, pp. 218–227.

[135] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow Checking by Software Signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, March 2002.

[136] NVIDIA, "XID Errors," http://docs.nvidia.com/deploy/xid-errors/index.html, October 2017.

[137] P. Koopman, K. Driscoll, and B. Hall, "Selection of cyclic redundancy code and checksum algorithms to ensure critical data integrity," Carnegie Mellon University, Tech. Rep. 3-2015, March 2015. [Online]. Available: http://repository.cmu.edu/ece/293

[138] 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 *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[139] 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 *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 185–197.

[140] S. K. S. Hari, T. Tsai, M. Stephenson, S. Keckler, and J. Emer, "SASSIFI: An Architecture-level Fault Injection Tool for GPU Application Resilience Evaluation," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 249–258.

[141] Synopsys Inc., "Design Compiler J-2014.09," August 2014.

[142] W. Nedel, F. Kastensmidt, and J. Azambuja, "Implementation and experimental evaluation of a CUDA core under single event effects," in *2014 15th Latin American Test Workshop (LATW)*, March 2014, pp. 1–4.

[143] Synopsys Inc., "Designware IP library." [Online]. Available: http://www.synopsys.com/products/designware/

[144] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," in *11th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2005, pp. 243–247.

[145] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014. [Online]. Available: http://arxiv.org/abs/1404.5997

[146] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss Project," http://eyeriss.mit.edu.

[147] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and B. Dally, "Deep compression and EIE: Efficient inference engine on compressed deep neural network," in *The International Symposium on Computer Architecture (ISCA)*, 2016.

[148] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126964 pp. 8:1–8:12.

[149] S. Jha, S. S. Banerjee, T. Tsai, S. Hari, M. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "ML-based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection ," in *International Conference on Dependable Systems and Networks (DSN)*, 2019.

[150] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardelenben, "Binfi: An efficient fault injector for safety-critical machine learning systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, 2019.

[151] A. H. M. Rubaiyat, Y. Qin, and H. Alemzadeh, “Experimental Resilience Assessment of an Open-Source Driving Agent,” *Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2018.

[152] B. Nelson, M. Barreno, F. J. Chi, A. D. Joseph, B. I. P. Rubinstein, U. Saini, C. A. Sutton, J. D. Tygar, and K. Xia, “Exploiting Machine Learning to Subvert Your Spam Filter,” in *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[153] H. Xiao, B. Biggio, G. Brown, G. Fumera, C. Eckert, and F. Roli, “Is feature selection secure against training data poisoning?” in *The International Conference on Machine Learning (ICML)*, 2015.

[154] R. Perdisci, D. Dagon, W. Lee, P. Foglat, and M. Sharif, “Misleading worm signature generators using deliberate noise injection,” in *IEEE Symposium on Security and Privacy (S&P)*, 2006.

[155] A. Newell, R. Potharaju, L. Xiang, and C. Nita-Rotaru, “On the Practicality of Integrity Attacks on Document-Level Sentiment Analysis,” in *Artificial Intelligent and Security Workshop (AISec)*, 2014.

[156] J. Newsome, B. Karp, and D. X. Song, “Paragraph: Thwarting Signature Learning by Training Maliciously,” in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2006.

[157] B. Biggio, B. Nelson, and P. Laskov, “Poisoning Attacks against Support Vector Machines,” in *The International Conference on Machine Learning (ICML)*, 2012.

[158] A. Nguyen, J. Yosinski, and J. Clune, “Deep neural networks are easily fooled: High confidence predictions for unrecognizable images,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[159] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” in *The International Conference on Learning Representations (ICLR)*, 2015.

[160] N. Carlini and D. Wagner, “Towards Evaluating the Robustness of Neural Networks,” in *IEEE Symposium on Security and Privacy (SP)*, 2017.

[161] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” in *NIPS-W*, 2017.

[162] D. A. Reed and J. Dongarra, “Exascale computing and big data,” *Commun. ACM*, vol. 58, no. 7, p. 56–68, June 2015. [Online]. Available: https://doi.org/10.1145/2699414

[163] H.-J. Yoon, A. Ramanathan, and G. D. Tourassi, “Multi-task deep neural networks for automated extraction of primary site and laterality information from cancer pathology reports,” in *INNS Conference on Big Data*, 2016.

[164] C. Torres-Huitzil and B. Girau, “Fault and error tolerance in neural networks: A review,” *IEEE Access*, 2017.

[165] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, “Applied machine learning at facebook: A datacenter infrastructure perspective,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[166] V. Sze, Y. Chen, T. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[167] S. R. Li, J. Park, and P. T. P. Tang, “Enabling sparse winograd convolution by native pruning,” *ArXiv*, vol. abs/1702.08597, 2017.

[168] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, “Ares: A framework for quantifying the resilience of deep neural networks.” ACM, 2018.

[169] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, “TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications,” *ArXiv*, vol. abs/2004.01743, 2020.

[170] F. Chollet et al., “Keras,” https://keras.io, 2015.

[171] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015. [Online]. Available: http://tensorflow.org/

[172] H. He, “The State of Machine Learning Frameworks,” https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/, 2019.

[173] Python, “Sunsetting Python 2,” https://www.python.org/doc/sunset-python-2/, 2020.

[174] PyTorch, “Releases pytorch/pytorch,” https://github.com/pytorch/pytorch/releases, 2020.

[175] Y. He, P. Balaprakash, and Y. Li, “Fidelity: Efficient resilience analysis framework for deep learning accelerators,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 270–281.

[176] C.-K. Chang, G. Li, and M. Erez, "Evaluating Compiler IR-Level Selective Instruction Duplication with Realistic Hardware Errors," *The 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, 2019.

[177] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[178] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *ArXiv*, vol. abs/1804.02767, 2018.

[179] T.-Y. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," *ArXiv*, vol. abs/1405.0312, 2014.

[180] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. J. Goodfellow, A. Madry, and A. Kurakin, "On Evaluating Adversarial Robustness," *ArXiv*, vol. abs/1902.06705, 2019.

[181] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[182] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. A. Mann, and P. Kohli, "On the Effectiveness of Interval Bound Propagation for Training Verifiably Robust Models," *ArXiv*, vol. abs/1810.12715, 2018.

[183] Wei Yang, "Pytorch-classification," "https://github.com/bearpaw/pytorch-classification", 2017.

[184] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[185] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," *Computer Science Department, University of Toronto, Tech.*, 2009.

[186] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps," *CoRR*, vol. abs/1312.6034, 2014.

[187] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller, "Striving for Simplicity: The All Convolutional Net," *CoRR*, vol. abs/1412.6806, 2014.

[188] A. Mahendran and A. Vedaldi, "Salient Deconvolutional Networks," in *European Conference on Computer Visio (ECCV)*, 2016.

[189] R. R. Selvaraju, A. Das, R. Vedantam, M. Cogswell, D. Parikh, and D. Batra, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization," in *International Conference of Computer Vision (ICCV)*, 2017.

[190] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[191] G. Huang, Z. Liu, and K. Q. Weinberger, "Densely connected convolutional networks," *CoRR*, vol. abs/1608.06993, 2016. [Online]. Available: http://arxiv.org/abs/1608.06993

[192] NVIDIA, "Self-Driving Car Hardware — NVIDIA DRIVE," https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/.

[193] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[194] H. Guan, L. Ning, Z. Lin, X. Shen, H. Zhou, and S.-H. Lim, "In-place zero-space memory protection for cnn," *ArXiv*, vol. abs/1910.14479, 2019.

[195] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C. Cher, and P. Bose, "Understanding the propagation of transient errors in hpc applications," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[196] S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," *ArXiv*, vol. abs/2006.04984, 2020.

[197] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," *CoRR*, vol. abs/1712.04621, 2017. [Online]. Available: http://arxiv.org/abs/1712.04621

[198] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, pp. 1–48, 2019.

[199] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[200] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[201] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018. [Online]. Available: http://arxiv.org/abs/1801.04381

[202] N. Ma, X. Zhang, H. Zheng, and J. Sun, "Shufflenet V2: practical guidelines for efficient CNN architecture design," *CoRR*, vol. abs/1807.11164, 2018. [Online]. Available: http://arxiv.org/abs/1807.11164

[203] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[204] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[205] PyTorch, "Pytorch classification models," "https://pytorch.org/docs/stable/torchvision/models.html", 2019.

[206] F. P. Miller, A. F. Vandome, and J. McBrewster, *Amazon Web Services*. Alpha Press, 2010.

[207] NVIDIA, "NVIDIA Tesla V100 GPU Accelerator," https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf, 2018.

[208] C. Schorn, A. Guntoro, and G. Ascheid, "Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 979–984.

[209] L. Liu and J. Deng, "Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution," *CoRR*, vol. abs/1701.00299, 2017. [Online]. Available: http://arxiv.org/abs/1701.00299

[210] N. Chandramoorthy, K. Swaminathan, M. Cochet, A. Paidimarri, S. Eldridge, R. V. Joshi, M. M. Ziegler, A. Buyuktosunoglu, and P. Bose, "Resilient low voltage accelerators for high energy efficiency," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[211] Z. Luo, A. F. R. Perez, P. G, G. Ramamurthy, S. Kola, E. M. Fomenko, R. Poornachandran, L. Y. Guo, K. Puttannaiah, N. Sundaram, and D. Samoilov, ""Accelerate INT8 Inference Performance for Recommender Systems with Intel Deep Learning Boost (Intel DL Boost)"," https://software.intel.com/content/www/us/en/develop/articles/accelerate-int8-inference-performance-for-recommender-systems-with-intel-deep-learning.html, Oct 2019.

[212] NVIDIA, ""NVIDIA T4 Tensor Core GPU Datasheet"," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf, Mar 2019.

[213] C. Sakr and N. R. Shanbhag, “An analytical method to determine minimum per-layer precision of deep neural networks,” *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1090–1094, 2018.

[214] C. Sakr, J. Choi, Z. Wang, K. Gopalakrishnan, and N. Shanbhag, “True gradient-based training of deep binary activated neural networks via continuous binarization,” in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 04 2018, pp. 2346–2350.

[215] C. Sakr, Y. Kim, and N. Shanbhag, “Analytical guarantees on numerical precision of deep neural networks,” in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017. [Online]. Available: http://proceedings.mlr.press/v70/sakr17a.html pp. 3007–3016.

[216] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta, “Reliability-Aware Data Placement for Heterogeneous Memory Architecture,” in *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 583–595.

[217] 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 *Proc. of International Symposium on Microarchitecture (MICRO)*, 2003, pp. 29–40.

[218] A. Naithani, S. Eyerman, and L. Eeckhout, “Reliability-Aware Scheduling on Heterogeneous Multicore Processors,” in *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 397–408.

[219] B. Wibowo, A. Agrawal, T. Stanton, and J. Tuck, “An Accurate Cross-Layer Approach for Online Architectural Vulnerability Estimation,” *ACM Trans. Archit. Code Optim.*, pp. 30:1–30:27, 2016.

[220] A. Agrawal, B. Wibowo, and J. Tuck, “Software Marking for Cross-Layer Architectural Vulnerability Estimation Model,” in *Proc. of IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2017.

[221] A. Pellegrini, R. Smolinski, L. Chen, X. Fu, S. K. S. Hari, J. Jiang, S. V. Adve, T. Austin, and V. Bertacco, “CrashTesting SWAT: Accurate, Gate-Level Evaluation of Symptom-Based Resiliency Solutions,” in *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012.

[222] M.-L. Li, P. Ramachandran, R. U. Karpuzcu, S. K. S. Hari, and S. V. Adve, “Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults,” in *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[223] W. Dweik, M. Annavaram, and M. Dubois, "Reliability-Aware Exceptions: Tolerating Intermittent Faults in Microprocessor Array Structures," in *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–6.

[224] M. S. Gupta, J. A. Rivers, L. Wang, and P. Bose, "Cross-layer System Resilience at Affordable Power," in *2014 IEEE International Reliability Physics Symposium*, 2014, pp. 2B.1.1–2B.1.8.

[225] J. J. Cook and C. B. Zilles, "A Characterization of Instruction-level Error Derating and its Implications for Error Detection," in *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2008.

[226] W. Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Z. Yang, "Characterization of Linux Kernel Behavior Under Errors," in *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2003.

[227] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding Error Propagation in GPGPU Applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016, pp. 240–251.

[228] R. Venkatagiri, K. Swaminathan, C.-C. Lin, L. Wang, A. Buyuktosunoglu, P. Bose, and S. Adve, "Impact of Software Approximations on the Resiliency of a Video Summarization System," in *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2018.

[229] S. K. Sastry Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "Ganges: Gang error simulation for hardware resiliency evaluation," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. IEEE Press, 2014, pp. 61–72.

[230] K. Ahmed, "Relyzer+: an Open Source Tool for Application-Level Soft Error Resiliency Analysis," July 2018.

[231] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," in *Proc. of Symposium on Operating Systems Principles (SOP)*, 2017, pp. 1–18.

[232] A. Yazdanbakhsh, K. Samadi, and H. Esmaeilzadeh, "SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks," in *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018.

[233] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, 2015.

[234] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," *CoRR*, 2017.

[235] A. J. KleinOsowski and D. J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *IEEE Comput. Archit. Lett.*, vol. 1, no. 1, p. 7, Jan. 2002.

[236] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Designing a processor from the ground up to allow voltage/reliability tradeoffs," in *Proc. of International Symposium on High Performance Computer Architecture*, 2010.

[237] J. Sartori and R. Kumar, "Architecting Processors to Allow Voltage/Reliability Tradeoffs," in *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2038698.2038718 pp. 115–124.

[238] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design." in *Proc. of IEEE European Test Symposium*, 2013.

[239] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation," in *Proc. of International Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 164–174.

[240] W. Baek and T. M. Chilimbi, "Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation," in *Proc. of International Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 198–209.

[241] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning Approximation for Graphics Engines," in *Proc. of International Symposium on Microarchitecture (MICRO)*, 2013, pp. 13–24.

[242] J. Sartori and R. Kumar, "Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications," *Multimedia, IEEE Transactions on*, vol. 15, no. 2, pp. 279–290, Feb 2013.

[243] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *Proc. of International Symposium on Microarchitecture (MICRO)*, 2012, pp. 449–460.

[244] J. San Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelganger: A Cache for Approximate Computing," in *Proc. of International Symposium on Microarchitecture (MICRO)*, 2015.

[245] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," *SIGPLAN Not.*, vol. 46, no. 3, pp. 213–224, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1961296.1950391

[246] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware," in *Proc. of International Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2013, pp. 33–52.

[247] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[248] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, "FlexJava: Language Support for Safe and Modular Approximate Programming," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 745–757.

[249] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability Type Inference for Flexible Approximate Programming," in *Proc. of International Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2015, pp. 470–487.

[250] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik, "ExpAX: A Framework for Automating Approximate Programming," in *Technical Report, Georgia Institute of Technology*, 2014.

[251] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration," in *Proc. of International Symposium on Computer Architecture (ISCA)*, 2016.

[252] D. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, June 2015, pp. 554–566.

[253] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive Control of Approximate Programs," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 607–621.

[254] A. Thomas and K. Pattabiraman, "Error Detector Placement for Soft Computation," in *International Conference on Dependable Systems and Networks*, 2013, pp. 1–12.

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

[256] D. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, Dec 2014, pp. 319–330.

[257] M. M. Sabry, G. Karakonstantis, D. Atienza, and A. Burg, "Design of energy efficient and dependable health monitoring systems under unreliable nanometer technologies," in *Proceedings of the 7th International Conference on Body Area Networks*, ser. BodyNets '12. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2442691.2442706 pp. 52–58.

[258] T. Wang, Q. Zhang, and Q. Xu, "ApproxQA: A Unified Quality Assurance Framework for Approximate Computing," in *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*, 2017, pp. 254–257.

[259] H.-J. Wunderlich, C. Braun, and A. Schöll, "Pushing the Limits: How Fault Tolerance Extends the Scope of Approximate Computing," in *Proc. of International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2016, pp. 133–136.

[260] T.-W. Chin, C.-L. Yu, M. Halpern, H. Genc, S.-L. Tsao, and V. J. Reddi, "Domain-Specific Approximation for Object Detection," *IEEE Micro*, vol. 38, no. 1, pp. 31–40, January 2018.

[261] F. Betzel, K. Khatamifard, H. Suresh, D. J. Lilja, J. Sartori, and U. Karpuzcu, "Approximate Communication: Techniques for Reducing Communication Bottlenecks in Large-Scale Parallel Systems," *ACM Comput. Surv.*, vol. 51, no. 1, Jan. 2018.

[262] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, S. Misailovic, and S. Bagchi, "Videochef: efficient approximation for streaming video processing pipelines," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 43–56.

[263] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels," *SIGPLAN Not.*, vol. 49, no. 10, pp. 309–328, Oct. 2014.

[264] J. Sartori and R. Kumar, "Architecting processors to allow voltage/reliability tradeoffs," in *Proc. of International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011, pp. 115–124.

[265] J. Han and M. Orshansky, "Approximate computing: An Emerging Paradigm for Energy-efficient Design," in *ETS*. IEEE Computer Society, 2013, pp. 1–6.

[266] R. Boyapati, J. Huang, P. Majumder, K. H. Yum, and E. J. Kim, "APPROX-NoC: A Data Approximation Framework for Network-On-Chip Architectures," in *Proc. of International Symposium on Computer Architecture (ISCA)*, 2017, pp. 666–677.

[267] D. Jevdjic, K. Strauss, L. Ceze, and H. S. Malvar, "Approximate Storage of Compressed and Encrypted Videos," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 361–373.

[268] P. Guo and W. Hu, "Potluck: Cross-Application Approximate Deduplication for Computation-Intensive Mobile Applications," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 271–284.

[269] S. Xu and B. C. Schafer, "Exposing Approximate Computing Optimizations at Different Levels: From Behavioral to Gate-Level," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 11, pp. 3077–3088, 2017.

[270] H. Song, X. Song, T. Li, H. Dong, N. Jing, X. Liang, and L. Jiang, "A FPGA Friendly Approximate Computing Framework with Hybrid Neural Networks," in *Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018, pp. 286–286.

[271] C. Xu, X. Wu, W. Yin, Q. Xu, N. Jing, X. Liang, and L. Jiang, "On Quality Trade-off Control for Approximate Computing using Iterative Training," in *Proc. of International Design Automation Conference (DAC)*, 2017, pp. 1–6.

[272] T. Moreau, J. S. Miguel, M. Wyse, J. Bornholt, A. Alaghi, L. Ceze, N. E. Jerger, and A. Sampson, "A Taxonomy of General Purpose Approximate Computing Techniques," *IEEE Embedded Systems Letters*, vol. 10, no. 1, pp. 2–5, March 2018.

[273] H. Zhao, L. Xue, P. Chi, and J. Zhao, "Approximate Image Storage with Multi-level Cell STT-MRAM Main Memory," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 268–275.

[274] S. K. Khatamifard, I. Akturk, and U. R. Karpuzcu, "On Approximate Speculative Lock Elision," *IEEE Transactions on Multiscale Computing Systems, Special Issue on Emerging Technologies and Architectures for Manycore Computing*, 2017.

[275] I. Akturk, N. S. Kim, and U. R. Karpuzcu, "Decoupled Control and Data Processing for Approximate Near-threshold Voltage Computing," *IEEE Micro Special Issue on Heterogeneous Computing*, pp. 70–78, 2015.

[276] P. Roy, R. Ray, C. Wang, and W. F. Wong, "ASAC: Automatic Sensitivity Analysis for Approximate Computing," in *Proc. of Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2014, pp. 95–104.

[277] M. Carbin and M. C. Rinard, "Automatically Identifying Critical Input Regions and Code in Applications," in *Proc. of International Symposium on Software Testing and Analysis (ISSTA)*, 2010, pp. 37–48.

[278] B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee, "AutoSense: A Framework for Automated Sensitivity Analysis of Program Data," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[279] R. Akram and A. Muzahid, "Approximeter: Automatically finding and quantifying code sections for approximation," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, 2017.

[280] R. Akram, "Performance and accuracy analysis of programs using approximation techniques," Ph.D. dissertation, 2017.

[281] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August, "Runtime Asynchronous Fault Tolerance via Speculation," in *International Symposium on Code Generation and Optimization (CGO)*, April 2012, pp. 145–154.

[282] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors, "Using Process-level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2007, pp. 297–306.

[283] 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)*, March 2007, pp. 244–258.

[284] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2000, pp. 25–36.

[285] J. Rivers, M. Gupta, J. Shin, P. Kudva, and P. Bose, "Error Tolerance in Server Class Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 7, pp. 945–959, July 2011.

[286] R. Nathan and D. J. Sorin, "Argus-G: Comprehensive, Low-Cost Error Detection for GPGPU Cores," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 13–16, January 2015.

[287] J. Tan and X. Fu, "RISE: Improving the Streaming Processors Reliability Against Soft Errors in GPGPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2012, pp. 191–200.

[288] M. Abdel-Majeed, W. Dweik, and M. Annavaram, "Warped-RE: Low Cost Error Detection and Correction in GPUs," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2015, pp. 331–342.

[289] C. Schorn, A. Guntoro, and G. Ascheid, "An efficient bit-flip resilience optimization method for deep neural networks," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, 2019. [Online]. Available: https://doi.org/10.23919/DATE.2019.8714885 pp. 1507–1512.

[290] Y. Ibrahim, H. Wang, M. Bai, Z. Liu, J. Wang, Z. Yang, and Z. Chen, "Soft error resilience of deep residual networks for object recognition," *IEEE Access*, 2020.

[291] F. Libano, B. Wilson, J. Anderson, M. J. Wirthlin, C. Cazzaniga, C. Frost, and P. Rech, "Selective hardening for neural networks in fpgas," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 216–222, 2019.

[292] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitras, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *28th USENIX Security Symposium (USENIX Security 19).* Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/hong pp. 497–514.

[293] C. Schorn, T. Elsken, S. Vogel, A. Runge, A. Guntoro, and G. Ascheid, "Automated design of error-resilient and hardware-efficient deep neural networks," *ArXiv*, vol. abs/1909.13844, 2019.

[294] Y. L. Cun, J. S. Denker, and S. A. Solla, "Advances in neural information processing systems 2," D. S. Touretzky, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ch. Optimal Brain Damage, pp. 598–605. [Online]. Available: http://dl.acm.org/citation.cfm?id=109230.109298

[295] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *CoRR*, vol. abs/1608.08710, 2016. [Online]. Available: http://arxiv.org/abs/1608.08710

[296] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient transfer learning," *CoRR*, vol. abs/1611.06440, 2016. [Online]. Available: http://arxiv.org/abs/1611.06440

[297] Z. Song, B. Fu, F. Wu, Z. Jiang, L. Jiang, N. Jing, and X. Liang, "Drq: Dynamic region-based quantization for deep neural network acceleration," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00086 p. 1010–1021.

[298] L. Song, R. Shokri, and P. Mittal, "Membership inference attacks against adversarially robust deep learning models," 05 2019, pp. 50–56.

[299] M. Pezzè and M. Young, *Software testing and analysis - process, principles and techniques.* Wiley, 2007.

[300] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[301] B. Beizer, *Software testing techniques (2. ed.).* Van Nostrand Reinhold, 1990.

[302] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

[303] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis.* Springer Publishing Company, Incorporated, 2010.

[304] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977, pp. 238–252.

[305] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[306] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.