DEMOCRATIZING ERROR-EFFICIENT COMPUTING

BY

RADHA VENKATAGIRI

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

Doctoral Committee:

Professor Sarita V. Adve, Chair
Professor Darko Marinov
Assistant Professor Sasa Misailovic
Assistant Professor Christopher W. Fletcher
Professor David Brooks, Harvard University
Dr. Pradip Bose, IBM Research

ABSTRACT

We live in a world where errors in computing are becoming ubiquitous and come from a wide variety of sources – from unintentional bit flips in devices to deliberate approximations and malicious attacks. Future systems must be built to extract maximum computational efficiency while operating with heterogeneous sources of errors. The paradigm of *Error-Efficient Computing* offers a promising solution by designing efficient computing systems that conserve resources (e.g., time, energy, cost) by allowing the system to make controlled errors.

Despite its promise, the widespread adoption of error-efficiency has been thwarted by (1) a lack of principled and unified methodologies to assess and exploit error-efficiency opportunities in a given system and (2) excessive programmer burden. This dissertation addresses these limitations by *developing methodologies that enable systematic, principled and scalable application of error-efficiency with minimal programmer input.*

Starting from first principles, the first contribution of this work is the development of *automated application-level error analysis* tools and techniques that can automatically determine, with high speed and accuracy, the output that a given program will produce for each of the billions of errors that the program might encounter in its computation and data. Using automated error analysis, a comprehensive view of the application's error characteristics, or in other words its *error profile*, is derived without the need for programmer expertise. To demonstrate the versatility of this approach, the next contribution of this work is to show how the automatically generated *application error profiles can be used to devise different (hardware- and software-based) error efficiency solutions* – from low-cost resiliency to approximate computing – that can be customized to the application and/or user requirements and output quality targets. Finally, using the novel insight that analyzing a piece of software for (hardware) errors should be similar to testing it for software bugs, *concepts from software testing are systematically adapted to significantly improve the speed and scalability of automated error analyses*; while the improvements in speed and scalability makes error analysis more practical within a computing stack, the methodology used lays the foundation for a principled integration of (hardware) error analysis into the software development work-flow.

Overall, the contributions of this dissertation further the goal of enabling the adoption of error-efficiency as a first-class design principal by developing systematic methodologies that allow a principled, unified, and yet, customizable way of exploiting error-efficiency.

To Appa and Amma, for teaching me to fly.
To Sibin, for being the wind beneath my wings.

ACKNOWLEDGMENTS

A Ph.D. is a journey. When one starts on it, there is an idea of what a destination may look like, but rarely is the path visible. I am eternally grateful to my advisor, Prof. Sarita Adve, for shining the light on the path, mentoring me on how to walk it, enabling me with the tools and skills to discover parts of it on my own and cheering me on as I reach the destination. The number of hours spent in technical discussions with her have been enlightening and I am lucky to have had the chance to learn closely from a world class researcher and educator. In addition, Sarita has been extremely supportive and understanding when I faced difficult times and I am grateful to have her as a mentor who has gone to bat for me on so many occasions.

The work described in this document was not conducted in isolation. I want to thank the many collaborators and co-authors on this work whose support and intellectual contributions have helped realize and improve the work presented here. I want to thank my committee (and frequent collaborators) Prof. Darko Marinov, Prof. Sasa Misailovic, Prof. Christopher W. Fletcher, Dr. Pradip Bose and Prof. David Brooks. Their advice and feedback has made this work what it is and, in the process, I have had the opportunity to learn much from them.

I consider myself lucky to have had a chance to work closely with the following amazing researchers over the years: Siva Kumar Hari Sastry (NVIDIA), Abdulrahman Mahmoud (UIUC), Khalique Ahmed (AMD), Alper Buyuktosunoglu (IBM) and Karthik Swaminathan (IBM). Thank you for your patience and for teaching me so much. I am also grateful for the friendship and support of my wonderful lab-mates – Rakesh, Hyojin, Matt, John, Huzaifa, Gio, Adel, Sam and Wes.

The work presented in this thesis has been supported by the generous grants from the following funding agencies: (1) National Science Foundation (under Grants Nos. CCF-0811693, CCF-1320941, CCF-1421503, CCF-1703637, and CCF-1725734), (2) Defense Advanced Research Projects Agency (DARPA), (3) Center for Future Architectures Research (C-FAR), one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA and (4) Applications Driving Architectures (ADA) center, a JUMP center co-sponsored by SRC and DARPA.

I am lucky to have made many friends during my years in Urbana-Champaign, all of whom have enriched my life in multiple ways. Without the many evenings spent in coffee shops, playing board-games, cooking meals, discussing books/movies and planning commu-

nity theatre, this thesis would have been completed earlier but the journey would have been infinitely less fun.

I am also grateful to my extended family, both in the U.S and in India, for being a constant source of love and support. A special thank-you to my in-laws, B.M.C Kumar and Shobhana for instantly making me a part of their family. Their warm affection and encouragement has been invaluable in this journey.

Words are insufficient to express the debt of gratitude I owe to my parents for ensuring I did not want for anything and encouraging me to march to the beat of my own drum. I am beholden to my father, K. S. Venkatagiri, for encouraging me to pursue my passions and teaching me the value of curiosity and perseverance. My sister and I are extremely lucky to have had such an exemplary role model. He was my biggest cheerleader on this journey and although he is not here to see me complete it, I know that he is looking down on me with a big smile. My mother, Usha Venkatagiri, has helped shaped my life in so many positive ways with her unwavering love and dedication. Her warmth, childlike enthusiasm, grace in the face of adversities and dedication to family inspire me everyday. Thank you for teaching me what is important in life and for always celebrating my achievements as if they were your own personal victories.

Rupa Venkatagiri, my sister, has been my partner in crime and getaway car driver from the time I was born. My happiest memories are those that involve her. Thank you for constantly being a sounding board and for your unique perspectives that have made me wiser over the years. I am forever grateful for having known and spent many laughter-filled hours in the company of my wonderful brother-in-law Manu Prakash Cherubal, who was taken from us too soon. Their son, Reyansh, is the bundle of joy who has brightened our lives and brought magic to it.

I have often been told that choosing a spouse is the most important decision I will make in life. I am extremely fortunate and blessed to have chosen so very well. Sibin Mohan, my husband, is the sunshine that instantly brightens even my darkest days. Thank you for your love, friendship, humour and wisdom. Thank you for always pushing me to do better and for believing in me on days when I didn't believe in myself. This Ph.D. is a testament to your love and your support of my dreams.

TABLE OF CONTENTS

Chapter 1: INTRODUCTION

1.1 MOTIVATION AND RESEARCH VISION

We face an urgent need to improve compute efficiency due to the following trends: (1) diminishing benefits (in performance and power) of CMOS technology scaling, (2) increased computational demand due to the explosive growth of data and (3) increasing susceptibility of hardware to errors – from faulty devices to malicious attacks. The intersection of these trends makes it extremely expensive to guarantee perfect functionality across a range of computing systems – from high performance computing systems to low-power edge devices. Fortunately, the workloads that are driving this demand for computing (learning, recognition, mining and search, among others) also provide opportunities since their primary goal is not a precise, numerically correct answer. Rather, their 'correctness' lies in producing results that are good enough, or of sufficient quality, to produce an acceptable user experience.

The paradigm of *Error-Efficient Computing* [1] exploits this forgiving nature of applications by *allowing the computing system to make controlled errors.* Such systems can be considered as being error-efficient: they only prevent as many errors as they need to for an acceptable user experience. The definition of what constitutes an error varies across system components and which errors are acceptable depends on the application and/or user expectations. Allowing the system to make errors can conserve resources. The resources conserved could be time, energy, bandwidth or more abstract quantities such as reliable operation, manufacturing costs, etc. Some examples of error-efficient techniques include (1) *approximate computing* [2, 3, 4, 5, 6, 7, 8, 9, 10, 11] which deliberately introduces errors in computation for improved performance or energy, and (2) *ultra-low cost hardware resiliency* [12, 13, 14, 15, 16] which allows some unintentional hardware errors to escape as user-tolerable output corruptions (rather than incurring high overheads to prevent all errors).

Error-efficient computing can revolutionize the way we design hardware and software to exploit significant new opportunities for compute efficiency. Error-efficiency has the potential to be a key enabler for many emerging application domains – Autonomous Vehicles, Industrial Robotics, Virtual Reality etc. – with strict power, performance and/or reliability requirements. Despite its promise, the widespread adoption of error-efficiency has been thwarted by (1) a lack of principled and unified methodologies and (2) excessive programmer burden.

The overarching vision of this thesis is to (1) enable the adoption of error-efficiency as a *first-class design principle by a variety of users, regardless of expertise, using (2) methodologies that allow a principled, unified and yet, customizable way of computing efficiently with heterogeneous sources of errors.*

1.2 SUMMARY OF CONTRIBUTIONS AND IMPACT

In order to fulfill the research vision outlined above, this work starts from first principles by asking one of the most fundamental questions for error-efficient computing – how do errors in a program's execution affect its output? Today we rely on program or domain experts to have a deep understanding of how errors (perturbations in program execution) affect program output and articulate them via error specifications [3, 4, 5, 17, 18, 19] which are then used to devise error-efficient solutions. The undue burden placed on the programmer is very limiting for the adoption of error-efficiency since such expertise is sparse and can take years to develop for a given domain [20, 21, 22].

This work addresses these limitations by **developing methodologies that enable systematic, principled and scalable application of error-efficiency without the need for programmer expertise**. This thesis makes the following key contributions:

(1) It demonstrates, through a motivational study, the effectiveness of error-efficient techniques in meeting the strict performance, energy and resiliency requirements of emerging edge computing domains. This is the first study (in collaboration with IBM research) to show that software approximations can be applied to improve the computational efficiency of a state-of-the-art vision analytics workflow on-board Unmanned Aerial Vehicles (UAVs) without degrading system resiliency [22, 23].

(2) It develops **automated error analysis** techniques and tools that determine the impact of billions of errors on program output with high speed and accuracy. The automated error analysis tools developed in this work – Approxilyzer [24], gem5-Approxilyzer [25] and DataApproxilyzer [26] – are the first-of-their-kind to quantify output quality for virtually all errors (for a given hardware error model) in program instruction and data. The application error profiles they generate provide a comprehensive view of the application's error characteristics with absolute minimal programmer intervention. Two tools have been publicly released [27, 28] with a third release planned shortly.

(3) It demonstrates the versatility of the automated error analysis approach, by showing how the automatically generated application error profiles can be used to devise different (hardware and software) error-efficiency solutions – from low-cost resiliency to approximate computing – that can be customized to the application and user requirements [24, 26].

(4) It builds a framework called *Minotaur* [29] that shows a principled adaptation of

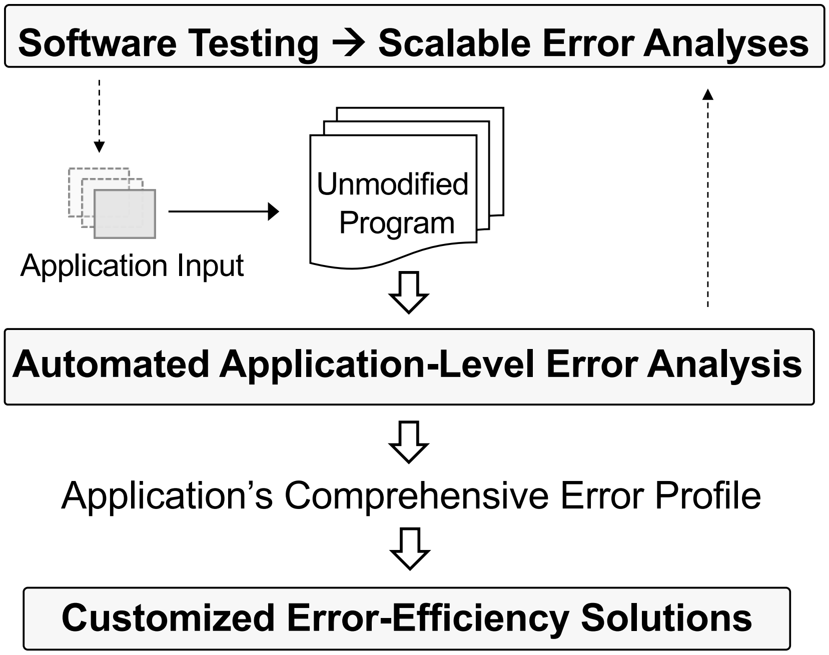

Software Testing → Scalable Error Analyses
Application Input
Unmodified Program
Automated Application-Level Error Analysis
Application's Comprehensive Error Profile
Customized Error-Efficiency Solutions

Figure 1.1: Summary of the contributions made by this work.

software testing techniques to significantly improve the speed and scalability of error analysis. Minotaur lays the foundation for systematically integrating (hardware) error analyses into the software development workflow.

Figure 1.1 shows a pictorial view of the contributions of this work along with the associated publications and open-source tool releases [22, 23, 24, 25, 26, 27, 28, 29, 30]. *Overall, this work democratizes error-efficient computing by broadening the reach of error-efficient computing to novice programmers and new domains.*

1.3 BRIEF OVERVIEW OF RESEARCH

1.3.1 Error-Efficiency for Emerging Domains

Error-efficient computing can enable emerging domains, such as edge-computing (UAVs, connected cars, industrial robotics, etc.), to meet strict performance and energy requirements. However, the strict reliability requirements of some mission-critical (high impact of failure/disruption) applications in this domain have traditionally not allowed inexact computations. For example, edge computing platforms on UAVs (Unmanned Aerial Vehicles), that are engaged in rescue and recovery operations, must not only meet strict real-time performance, energy and bandwidth requirements, they must also be resilient while operating in harsh environments subject to sharp variations in temperature, altitude and weather conditions, and tolerate glitches in input and output. Hence, any error-efficiency solutions applied to such systems (to improve, say, performance and energy) must not degrade the system resiliency.

In a collaborative study with IBM Research [22], we demonstrate the *resiliency and effectiveness of error-efficiency techniques for edge-computing applications* used in Unmanned Aerial Vehicles or UAVs (the concepts are relevant to other applications as well). The workload studied is a state-of-the-art Video Summarization (VS) application (developed at IBM Research) that constitutes key end-to-end video and image analytics performed aboard UAVs. *This first-of-a-kind work that studies the effect of approximations on system performance, energy and resiliency*, shows that software approximations yield significant energy and performance (up to 68%) benefits for the end-to-end VS work-flow without degrading the overall resiliency of the system. This study serves as motivation for the rest of the work in this thesis.

1.3.2 Tool-Suite for Automated Application-Level Error Analysis

For any error-efficiency solution, we first need a fundamental understanding of how errors in a program affect its resultant output quality. Techniques today rely on program/domain experts for this knowledge which is severely limiting.

To mitigate this limitation, this work develops a suite of automated application-level error analysis tools (henceforth referred to as the *AEA tools*) [25, 26, 31] that can *automatically extract the error characteristics of applications.* In this work, an error is defined as a perturbation in program state (computation or data) caused by an underlying fault in hardware. Using a hybrid technique [32, 33] of program analysis and some error injections, these tools *comprehensively* analyze billions of possible errors that can impact a program's execution with high *accuracy* (>95% on average) and at *low-cost* (requiring up to five orders of magnitude fewer error injections than naïve techniques). Furthermore, they can *precisely* (at fine granularity and within low error margins) quantify the output quality produced in the presence of each of the errors they analyze.

The AEA tools impose *minimum burden on the programmer* and only require the user to provide an unmodified program, a (domain-specific) quality metric and, optionally, a quality threshold. The error analyses techniques developed in this line of work are *general* and can be used to analyze different general-purpose computations and error models (single- and multi-bit error models spanning instructions and data are studied in this work). To the best of our knowledge, these are the first application-level error analysis tools that satisfy the requirements of automation (minimal programmer burden), accuracy, precision, comprehensiveness, low-cost and generality. The output of automated error analyses are *comprehensive application error profiles* that list the errors that can affect the program's execution along with the corresponding output quality expected for each of the errors. The application error profiles can then be used by programmers or systems to understand the application's error characteristics.

While Chapter 3 provides the details about the automated error analysis techniques, a brief description of the tools developed as part of it this work are provided below.

Automated Analysis of Errors in Program Instructions (Compute):

Approxilyzer [31] is the first automated error analysis tool to quantify the impact of virtually all errors (for a given error model) in program instructions on the program's output quality with high accuracy and speed. The error model used in Approxilyzer is single-bit transient errors in register operands of dynamic instructions. Approxilyzer's output is the application's comprehensive *instruction error profile.* In order to enable researchers to easily use and extend Approxilyzer concepts, we build a fully open-source error analysis toolkit called **gem5-Approxilyzer** [25]. gem5-Approxilyzer uses the open-source gem5 simulator and is designed to enable researchers to adapt and extend it to different instruction set architectures (ISAs).

Automated Analysis of Errors in Program Data (Storage):

Errors in instructions (compute) and data (storage) propagate differently through the program and, hence, require different analysis techniques. We build a tool called **DataApproxilyzer** [26] to automatically analyze and quantify the impact of errors in program data on output quality. DataApproxilyzer accurately and comprehensively analyzes many, and in some cases *all* errors (for a given error model) in program data. The error model used in DataApproxilyzer is multi-bit (1-bit, 2-bit, 4-bit and 8-bit) transient errors in system memory. The output of DataApproxilyzer is the application's comprehensive *data error profile.*

1.3.3 Automated Error Analysis to Customized Error-Efficiency

An application's error profile can be used to understand its error characteristics, which can inform customized error-efficiency. For example, this work demonstrates how an application's instruction error profile is used for error-efficiency solutions targeted towards *ultra-low cost resiliency to hardware errors* [31]. It shows that large resiliency overhead reductions (up to 55%) are possible if the user is willing to tolerate a very small loss in output accuracy (1%) while still providing high (99%) resiliency coverage [31]. In another example, the application's error profile is used to identify promising subsets of instructions and/or data for *approximate computing* across different quality thresholds and approximation targets. An illustrative end-to-end workflow demonstrates the automatic identification and mapping of non-critical (approximate) data (for given user quality targets) to be stored in approximate DRAM with low refresh rates (which saves energy but incurs modest errors), without programmer intervention [26]; prior techniques required the programmer to identify non-critical data with annotations.

The error profiles of applications are also used to gain *insights* that can motivate further research. For example, Chapter 4 shows that error profiles for the same application, compiled to different ISAs (SPARC vs. x86), can vary widely [25]; motivating the need for customized error-efficiency for different architectures.

1.3.4 Leveraging Software Testing Techniques for Efficient and Scalable Error Analyses

This work introduces a framework called *Minotaur* [29] that significantly improves the speed and scalability (across multiple inputs and workloads) of error analysis techniques. Minotaur uses the insight that *analyzing an application for (hardware) errors has many conceptual similarities to analyzing it for software bugs*; therefore, adapting techniques from the rich software testing literature can lead to principled and significant improvements in error analyses. Minotaur is the first work to suggest *leveraging software testing methodologies for comprehensive error analysis*; thereby, laying the foundation for a principled integration of hardware error analysis into the software development work-flow.

Minotaur identifies and adapts four concepts from software testing to: (a) introduce the concept of input quality criteria for error analysis and suggest a simple but effective criterion (statement coverage at the object-code level); (b) develop a methodology to create high quality (fast) minimized inputs from (slow) standard benchmark inputs; (c) prioritize the analysis of specific program locations for a given error analysis objective and terminate analysis early when the objective is met; and (d) show scalable error analysis over multiple inputs by progressively prioritizing analysis over fast (but potentially inaccurate) inputs first. Minotaur improves the speed of comprehensive error analysis by 4x, on average, over state-of-the-art tools like Approxilyzer. These gains go up to 39x (or 55x) when the error analysis is targeted to specific techniques like hardware resiliency (or approximate computing).

Chapter 2: ERROR-EFFICIENCY FOR EMERGING DOMAINS

Real time edge computing [34, 35] is a rapidly growing field where real time data processing and other compute services are pushed away from centralized points to the logical extremes or *edge* of a network. This reduces the communication bandwidth needed between edge devices (e.g., sensors) and the central data center by performing analytics and knowledge generation at or near the source of the data. One of the key enablers of this trend is the presence of simultaneously high-performance and energy-efficient embedded systems that can be used to do computing in devices that are at the edge of the network. Real time edge computing has many applications such as Unmanned Aerial Vehicles (UAV), connected cars, industrial robotics, etc.

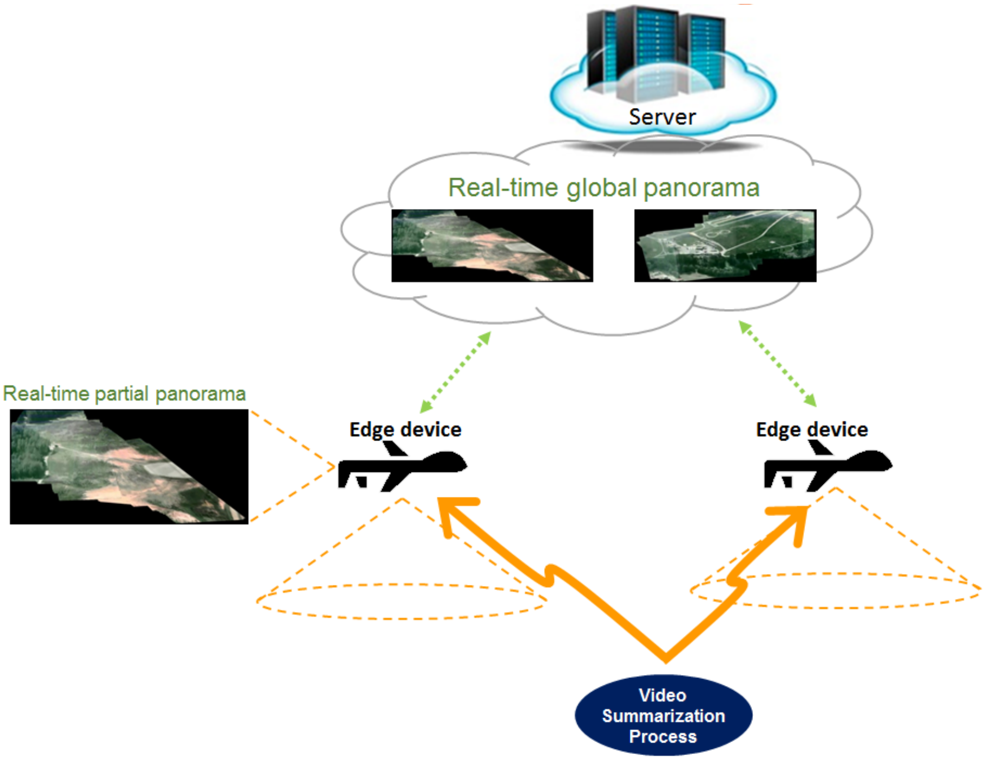

Server
Real-time global panorama
Real-time partial panorama
Edge device
Edge device
Video Summarization Process

Figure 2.1: Co-operative swarm of UAVs engaging in computation for real-time applications.

Figure 2.1 illustrates a real-life application of edge computing. Here, a swarm of UAVs, supported by a terrestrial server at the back-end, carry out tasks such as surveillance of hostile targets or rescue and recovery in the event of natural disasters. The UAVs communicate data and other analytics with the ground servers through wireless connections whose bandwidth, security and reliability might vary depending on physical and environmental factors. Hence, for real-time critical tasks, it is increasingly becoming essential that each UAV be equipped as a highly efficient mobile embedded system that can locally perform essential real-time computing tasks. One such task that is often performed locally aboard the UAV is *Video Summarization* [36]. This task involves extracting concrete context from the input video stream – captured by the many cameras on the moving UAV surveying a wide area – and summarizing it, usually in the form of a panoramic image. The panoramic image can then be transfered to a central ground server for further processing, for say, tracking or identifying rescue targets.

Edge computing platforms, such as that described above, are often deployed in rugged terrains with harsh environmental conditions and must satisfy the following requirements: a) ensure *high performance* to meet real time deadlines, particularly for mission-critical applications, b) be *energy efficient* to enable long range computing, and c) be *resilient* while operating in harsh environments subject to sharp variations in temperature, altitude and weather conditions, and tolerate glitches in input and output [37].

Error-efficient techniques such a *Approximate Computing* [7, 38] are increasingly gaining traction as a viable approach for high performance and energy efficiency. Approximate computing environments allow deliberate, but controlled, relaxation of correctness and trade-off computational accuracy for improvements in performance and energy. Many edge computing applications involve processing sensory signals (image, audio, etc.) which can inherently tolerate inaccuracies in data and/or computation without compromising overall mission targets and goals. This presents an opportunity to redesign these algorithms to incorporate approximate computing with the goal of meeting stringent performance and energy targets (requirements (a) and (b) from above) under specified constraints.

However, while most approximate computing techniques have in-built metrics and techniques to guarantee a certain output quality, it is not clear how they work in the face of sources of vulnerability in the processor, such as soft errors, voltage noise and aging phenomena. Further, these effects can be exacerbated (increased probability of radiation strikes at high altitudes in UAVs) when the approximate computing paradigm is adapted to the harsh conditions that these systems encounter during their operation. For successful deployment in edge computing environments, it is critical to ensure that the application of approximate computing techniques which yield performance and energy improvements not degrade the overall system resiliency (requirement (c) from above).

This work focuses on studying the interaction between software approximation and the application's resiliency to soft errors (henceforth referred to as *application resiliency* or simply *resiliency*) and to my knowledge is the first work to do so. To demonstrate this interaction a state-of-the-art end-to-end video summarization application [36] is analyzed which represents a typical and key vision analytics workflow executed by embedded systems on-board UAVs.

In particular, this work makes the following contributions:

- The application resiliency of an end-to-end video summarization application (henceforth termed as *VS* for brevity) is studied. The *VS* application serves as a representative workload for on-board UAV processing. Specifically, we study the application's resilience to radiation-induced soft (transient) errors, by performing runtime architectural fault injection experiments. All analyses are performed across two distinct inputs that realistically portray the different types of input video stream captured by cameras on the UAV.

- Performing resiliency analysis on a full, long-running end-to-end workflow is more expensive (in time and compute) than analyzing individual smaller kernels that together constitute the larger work-flow. This trade-off is examined by estimating the resiliency of individual representative kernels or *hot* functions in the *VS* application. It is shown that the hot functions are sub-optimal at capturing the behavior of the full application, thus motivating the need to develop and evaluate realistic applications with a full end-to-end workflow.

- This work characterizes the performance and energy of three different software approximation techniques applied to the VS algorithm. It shows that the approximations yield significant speedup and energy savings (up to 68%) without compromising the quality of the panoramic image output.

- The resiliency of approximate *VS* algorithms are examined. It is found that the approximations yield similar resiliency profiles to the baseline (precise) algorithm and in the worst case lead to a slight increase in Silent Data Corruption (SDC) rates (up to 2%). To the best of my knowledge, this is the first work that examines the effect of software approximations on application resiliency.

- Furthermore, the SDCs caused by the approximate algorithms are examined using a novel quality metric suitable to the domain of UAV image analytics. The results show that most of the SDCs generated by the applied approximations have small quality degradations and can potentially be tolerated by the application.

In summary, this work shows that error-efficiency techniques such a software approximation can be utilized to achieve significant gains in performance and energy without affecting application resiliency. This work does not claim to cover all possible types of error-efficiency techniques (or even the best ones) or comment on the general resilience of different techniques. Instead, the intent of this study is to encourage a comprehensive evaluation (performance, power, resilience) of system optimizations and show that *highly effective and resiliency-aware error-efficient techniques are possible in enabling emerging application domains such as edge computing.*

2.1 BACKGROUND

2.1.1 Video Summarization

UAVs are increasingly being used to perform tasks such as surveillance of hostile targets and rescue and recovery in the event of natural disasters. For decisive action in all these scenarios, it is essential to first extract and summarize the concrete context from the input video stream captured by the moving UAV. This operation is termed as *Video Summarization*. A sophisticated video summarization work-flow requires application of several Computer Vision techniques.

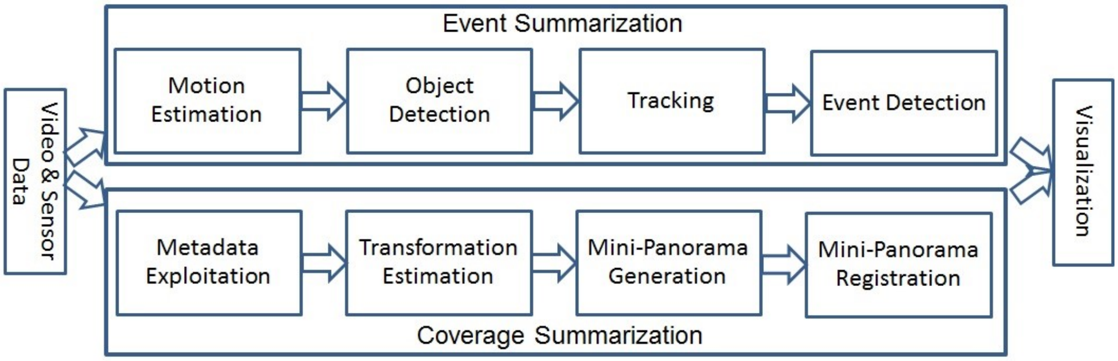


Figure 2.2: Video summarization for UAV videos.

One such example of an end-to-end application flow is described in Viguier *et al.* [36], where the UAV multimedia processing pipeline focuses on summarizing videos captured by the camera on-board the UAV. The framework for achieving this is shown in Figure 2.2. In order to achieve large data reduction without significant loss of events of interest, two types of summarizations are included: coverage and event summarization. Coverage summarization involves a complete panorama generation which describes the entire video with a single image that represents the entire spatial coverage area of the camera. The coverage summarization relies on spatially relating different video frames from the cameras, projecting them into a common view space, and stitching them together to build a single panorama. Event summarization comprises of tasks such as detection, recognition and tracking of moving objects such as vehicles and pedestrians. Finally, both intermediate results are integrated by overlaying the tracks (of moving objects) on the panorama to create a comprehensive and concise summarization of a whole UAV video.

This work focuses on *coverage summarization*. In particular, the focus is on algorithms that perform the task of generating video panoramas of the landscape covered by the cameras on a UAV and on energy-efficient and reliable implementations of the same.

2.1.2 Approximate Computing Techniques

Approximate computing is a fast growing trend that allows controlled relaxation of correctness for better performance and energy. Many techniques have been proposed that leverage approximate computing at the software [39, 40, 41, 42, 43, 44], programming language [2, 3, 4, 5, 17, 19, 45] and hardware [6, 7, 8, 9, 18, 46] level for improved performance, energy or reliability. Since application level resilience is measured in this work, the analysis is focused to software approximations. In particular, three broad classes of software approximations are studied.

(1) Input sampling: In this class of approximation, computation is only performed over a subset of the input. This class of approximation is especially popular in big data analytics [47] where the amount of data over which the computation needs to be performed is prohibitive in time and resources.

(2) Selective Computation: Another popular class of approximations are those in which only a fraction of the work is performed compared to the precise program. While the underlying algorithm remains unchanged, selective computations are simply skipped or dropped [39].

(3) Algorithmic Transformation: These approximations transform the code and replace precise but expensive computation with cheap but imprecise computation.

The selection of which approximations to apply depends on the application/domain and the end goal. For example, approximations skipping certain loop iterations [39] and approximations dropping some synchronizations [41] in the program both belong to the broad category of selective computation and either, both or neither might be an appropriate approximation for a given application. Section 2.2 describes approximations belonging to each of the categories described above that are specific to the application studied.

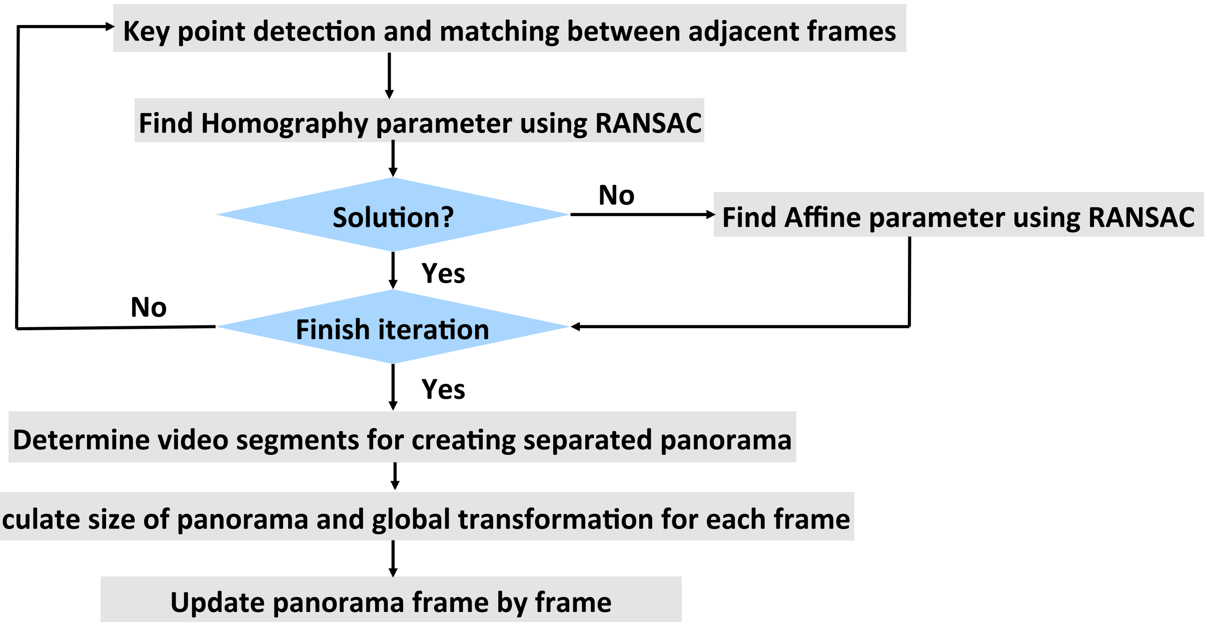

Key point detection and matching between adjacent frames
Find Homography parameter using RANSAC
Solution?
No
Find Affine parameter using RANSAC
Yes
No
Finish iteration
Yes
Determine video segments for creating separated panorama
Calculate size of panorama and global transformation for each frame
Update panorama frame by frame

Figure 2.3: Flowchart describing the key tasks that comprise the Video Summarization Algorithm.

2.2 VIDEO SUMMARIZATION ALGORITHM

The system architecture of interest in this work is a model where a swarm of UAVs is engaged in image scanning, analysis and stitching with the end goal of creating a global panorama of the observed landscape. Towards this goal, we describe in this section a video (image) stitching algorithm that has been developed and implemented in our experimental evaluation platforms (in collaboration with IBM research). This application (henceforth referred to as the *Video Summarization (VS)* algorithm) takes input videos captured by moving cameras and generates panoramas that provide a global view of the landscape. Since the input video is a concatenation of images captured by (various) moving cameras, it can contain various segments with dissimilar viewing angles and settings. Each of these segments are summarized by *mini-panoramas* and are stitched into a global panorama (simply referred to as *panorama*) at a later stage. In this work, we restrict the analysis to the generation of a panorama from video captured by a single camera on-board a single UAV.

2.2.1 Functional Overview

While a full detailed description of the algorithm is provided in [48], we describe the key capabilities of the algorithm [49] in the following paragraphs. A representative flow of the algorithm is shown in Figure 2.3.

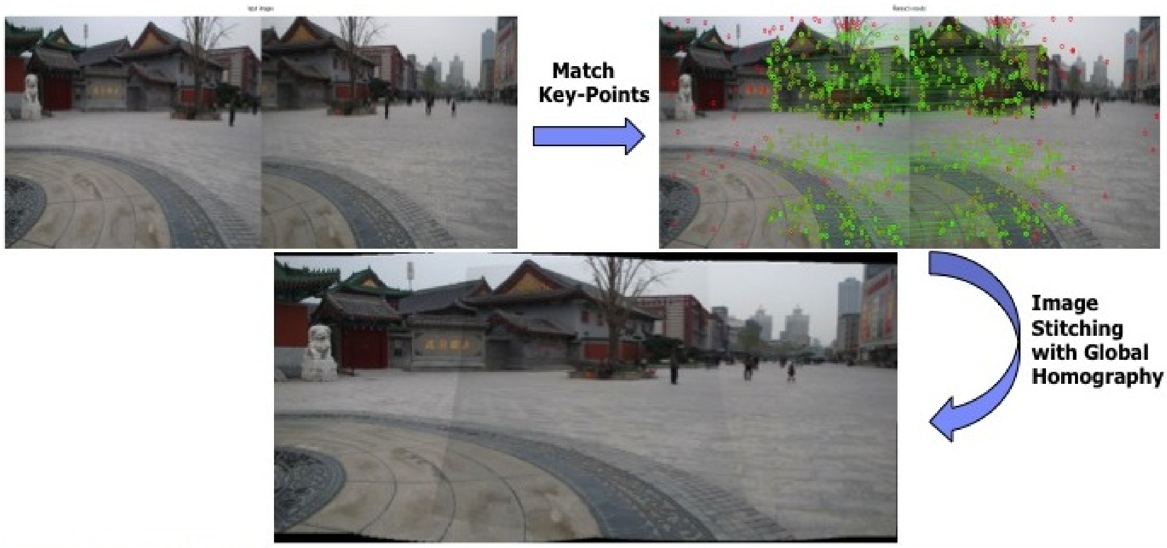


Figure 2.4: Simple example of stitching two images.

One of the fundamental functions performed by the *VS* algorithm is the comparison, transformation and stitching of two images from the input video. The algorithm first identifies key regions of interest (*key points*) within each image and then looks for matching key points within the images to identify potential common areas. It then applies transformations to the two images so that they are aligned correctly and have the same scale, lighting, perspective etc., before proceeding to stitch them together. Figure 2.4 shows this process using two sample images. We utilize FAST (Features from Accelerated Segment Test) detectors [50, 51] and ORB (Oriented FAST and Rotated BRIEF) descriptors [52] to achieve efficient and accurate feature point detection and matching. RANSAC (RANdom SAmple Consensus) [53] is used to compute the homography transformation between the two images.

Using the technique described above, successive frames of the input are pair-wise compared in the initial pass of the algorithm. However, not every pair of adjacent frames has enough matching *key points* to compute the homography transformation. In this case, we estimate a simpler affine transformation which requires fewer matching points. If sufficient number of matching points cannot be found even for the affine transformation, the corresponding frame is discarded. To generate the overall output panorama we align every frame to the first by transforming all the frames to have the same coordinate system as the first frame by using the homography transformations described above.

There are various other sophisticated elements of the stitching algorithm that are used to improve the rendered quality of the output panorama. The mathematical details of the transformations and corrective actions (e.g., to avoid blurs and distortions) are omitted here for brevity. Depending on the quality and number of the input video clips (collected by the moving cameras), the amount of computation performed by the video summarization procedure can vary.

2.2.2 Inputs to the Video Summarization Algorithm

I evaluate the *VS* application using two aerial videos from the VIRAT (Video and Image Retrieval and Analysis Tool) dataset [54] – *09152008flight2tape1_2* (hereby referred to as Input 1) and *09152008flight2tape2_4* (hereby referred to as Input 2). We use an input size of 1000 frames for both inputs.

The VIRAT dataset was chosen for the evaluation to represent realistic scenarios of videos captured during aerial surveillance with variations in resolution, diversity in scenes, changes in scale, focus and camera angles. The two inputs that we profiled vary significantly in these aspects as well as in the nature with which these parameters vary in the video stream. For instance, the number of changes that occur in Input 1 are much higher than Input 2, leading to a much larger number of mini-panoramas generated in the first input set. These videos were sampled at periodic intervals to yield around 3000 frames across the duration of the entire video. In addition, we further downsampled the video by a factor of 3 to enable a statistically significant number of error injection experiments to run within a reasonable amount of time without perceivable loss in information or image quality.

2.3 APPROXIMATE VIDEO SUMMARIZATION ALGORITHMS

As described in Section 2.2, the Video Summarization (*VS*) application is capable of effectively capturing several hours of video in single stitched image frames. However, given the constraints on power efficiency (of the on-board device) as well as real-time requirements of the mission, a complete and exact implementation of the algorithm may not be possible. In [37] and [55], the authors examine techniques to mitigate this limitation by dynamic adjustments of the link bandwidth and processor voltage/frequency.

In this paper, we consider software approximation, to the *VS* algorithm, as a means to realize performance and energy targets. Since computations involving images can be inherently tolerant to inaccuracies in data and/or compute, approximations to the the *VS* workflow have the potential to yield significant benefits without unduly compromising the quality of the final panorama image output.

We study three different approximations (belonging to the three broad classes described in Section 2.1.2 and presented in the same order). The details of the approximate algorithms are described below:

(1) Random Frame Dropping (*VS_RFD*): In this algorithm, we randomly drop frames from the input stream. Apart from improving the effective frame rate, this input approximation aims to leverage redundancies in consecutive images captured by a moving camera without substantial degradation in output image quality. In this work, we demonstrate results with up to 10% of the input frames being dropped.

(2) Key Point Down Sampling (*VS_KDS*): The *VS* algorithm described in Section 2.2 involves the computation of feature (key) points and attempts to match them across frames in order to be able to stitch the frames together. We propose an approximation in which matching is only performed on a fraction (one-third) of the key points as compared to the precise algorithm. This significantly reduces the computation time, which varies as $O(n^2)$ with the number of key points. In this algorithm, the source of error could be due to some frames being dropped on account of having insufficient matching key points. In such cases, it is hoped that the redundancy of the image will still enable complete coverage of the input video in our summarized output.

(3) Simple Matching (*VS_SM*): In the default algorithm, each key point in the current frame is compared with all key points in the incoming frame and the two nearest neighbors are determined for each key point. The key point is included in the list of *good* matches only if the ratio of the distance between the nearest and 2^{nd} nearest neighbor is above a certain threshold; i.e., the nearest match is sufficiently closer than the 2^{nd} nearest. This reduces the probability of a false positive, i.e., the probability that the key point in a frame incorrectly maps to a point in the subsequent frame, even if, in reality, there is no matching object. In case of *VS_SM*, the algorithm is altered to determine only the single nearest neighbor for each key point. In addition, we place an upper bound on the actual distance value and consider only those matches whose nearest neighbor is within a fixed distance of the key point. Hence, only those key points in the incoming frame which match almost perfectly with those in the original frame would be considered. Note that this technique still leaves room for some errors, for example, when there are two identical objects in the image. In such cases, both nearest neighbor distances could fall within the threshold and the mapping could happen to the incorrect object.

2.3.1 Effectiveness of the Approximate Implementations

An approximate algorithm has to produce acceptable quality outputs while enabling some system benefit (e.g. improved performance or energy efficiency). Hence, the three approximate algorithms are examined from the point of view of both system benefits and output quality to determine if they are good candidates for further study.

System benefits of approximation: We carried out an experimental evaluation of these algorithms on an IBM POWER-based server class machine. Figure 2.5 shows the Instructions Per Cycle (IPC), execution time and energy, normalized to the baseline *VS* algorithm. We observe that *VS_RFD* provides the maximum reduction in execution time (68%) for Input 1 by just dropping 10% of the total frames. On the other hand, *VS_KDS* yields the highest performance improvement of 18% in case of Input 2. Since the IPC (and hence, the power) remains relatively constant across the default and approximate implementations, the energy profile across the exact and approximate implementations varies similarly to that of the execution time.

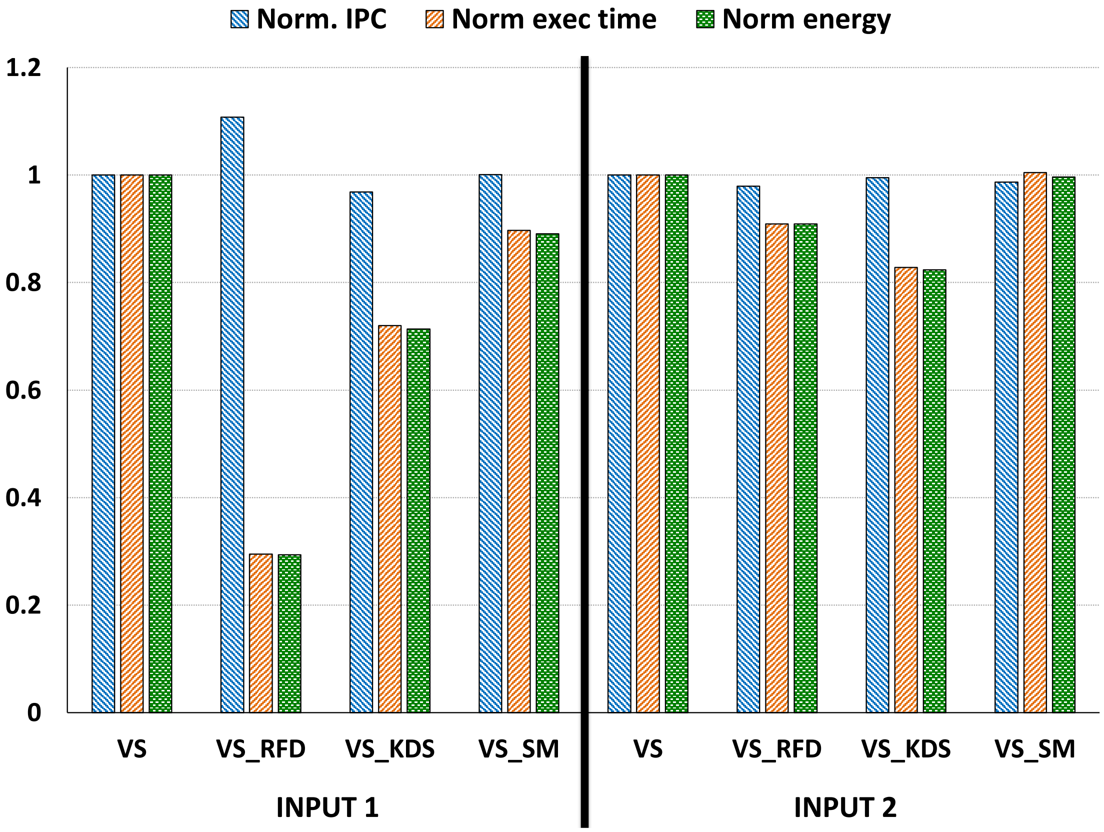


Figure 2.5: Comparison of IPC, execution time and energy of the proposed approximate algorithms (*VS_RFD, VS_KDS, VS_SM*) for Input 1 and Input 2, with the values normalized to the corresponding baseline (*VS*) for each respective input.

Comparison of output quality: Figure 2.6 compares the output images generated by the baseline *VS* algorithm and the three approximations described for the two inputs. Visual inspection shows that the approximate algorithms generate output images of acceptable quality. Even in the approximate output image with the worst quality (*VS_RFD* for Input

1), the quality degradation is due to image perspective and all the pertinent information in the final panorama is retained.

Tradeoffs between performance and output quality: The difference between the two inputs is evident from the tradeoffs between performance and output image quality for each approximation. For instance, a visual inspection of the output panoramas generated by the baseline *VS* algorithm and the three approximations (Figure 2.6) show that Input 2 is more robust to the proposed approximation techniques as compared to Input 1. On the other hand, the performance benefits due to approximation are clearly greater in case of Input 1.

This difference in the impact of approximation on the two inputs can be attributed to the fact that the variation between consecutive frames is much more pronounced in Input 1 than in Input 2. The execution time improvement is primarily due to the polynomial complexity of the algorithm in terms of number of frames that are processed. In addition to the frames dropped by the approximate implementation, the algorithm also discards additional frames without stitching them to the overall panorama, when sufficient matching points are not found. Consequently, the proposed approximation techniques result in several frames of Input 1 being discarded. While this reduces the number of computations, resulting in greater performance and energy benefits as compared to Input 2, it also adversely affects the output quality to a greater extent. The differences between the output images can be further analyzed quantitatively by means of our proposed metric, described in further detail in Section 2.4.4.

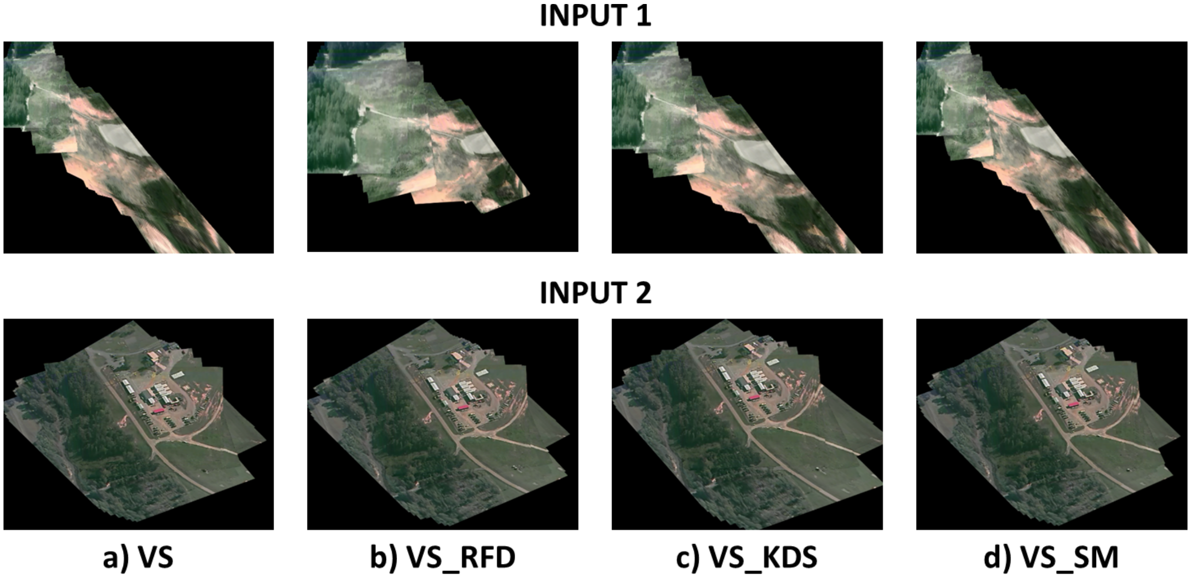


Figure 2.6: Comparison of the output panoramas obtained from baseline *VS* algorithm (a) and various approximation techniques (b,c,d) for the two input image sets.

2.4 METHODOLOGY

This section describes the design methodology and the evaluation environment used to measure the resiliency of the *VS* algorithm as well as its approximate versions. In section 2.4.3 a small case-study is described to understand the trade-offs of performing resiliency analysis on a full end-to-end application (such as the *VS* algorithm) vs. constituent small kernels. Section 2.4.4 defines a metric to calculate the quality of the corrupted output produced by the application when perturbed by errors. This metric is later used to analyze the quality of the corrupted outputs produced by the approximate *VS* algorithms.

2.4.1 Measuring Resiliency of Video Summarization Algorithm

Error injection is a widely used error analysis technique where an error is injected (typically one at a time) in a real or simulated machine and the outcome (impact of the error) is studied [56, 57, 58, 59, 60, 61]. My goal is to evaluate the application-level resiliency of the *VS* algorithm and its different approximate versions in the presence of hardware transient errors. The error model studied in this work assumes the occurrence of single bit errors in the architectural register file. The impact of an error on a program can be described by the following four outcomes:

(1) Mask: The error is masked by consecutive execution such that the application produces the correct output. This can happen if the error affects dead state or if the corrupted state is overwritten before being used.

(2) Crash: The error catastrophically affects the program state and results in the program crashing. For example, an error that leads to an out of bounds memory access.

(3) Silent Data Corruption (SDC): The error propagates through the program execution and corrupts the output. This is called a Silent Data Corruption because there is no obvious symptom of the error till the execution completes and the output is found to be corrupted.

(4) Hang: The error corrupts the internal state of the program such that neither completes nor crashes but hangs.

Comprehensively injecting errors in each potential error site in the program execution is prohibitively time consuming. For instance, in this study, each bit in every architectural register at every single execution cycle is a potential candidate for error injection. For most applications, the number of error sites is prohibitively large. Hence, we rely on statistical error injection in randomly selected error sites in the execution. This technique provides statistical summaries of the impact of errors on the application by estimating average rates for Mask, Crash, SDC and Hangs. Alternate, more comprehensive and higher precision techniques such as Relyzer [62] could be applied but are left to future work.

For accurately estimating the application resiliency, it is essential to perform error injections in a significant number of error sites that are uniformly distributed over the program execution. We use the term *error-site coverage* (or simply *coverage*) to indicate the relative robustness in the number and distribution of error sites picked for error injections.

We estimate the minimum number of error injection experiments needed to get an adequate statistical sample by observing the different rates of Mask, Crash, SDC and Hang over many error injections and the point at which these rates stabilize. In other words, the minimum number of error injections required are at the *knee* of the trend curves for the Mask, Crash, SDC and Hang rates. Beyond the *knee* of the curves, increasing the number of error injections should only change the outcome rates trivially.

2.4.2 Error Injection Environment

Error injection experiments are conducted on the IBM POWER-based machine, running Linux RHEL 6.5 operating system. The Application Fault Injection (AFI) [37] tool is used to perform error injections and evaluate the application's resiliency. Figure 2.7 shows the main components of AFI.

AFI is composed of two modules. The first module, *Fault Injector* takes the un-modified application binary, and injects error bits into the application's architectural state. Users can change the injection configurations to specify where to inject errors and how many errors to inject per run. For this study, AFI is configured to inject one single bit error (bit flip) in a general purpose register (GPR) or a floating point register (FPR). The execution cycle at which the error is injected is random. Once the program execution is continued after the error injection, the second module, the *Fault Monitor*, will check the application's running state and capture a potential hang or crash. If the application finishes normally, the Fault Monitor invokes a result checking procedure to determine if the outcome of the error injection is an SDC or Masked result.

Separate experiments are performed for error injections in GPRs and FPRs to enable separate examination of the vulnerability of these two register types to single bit flips.

2.4.3 Studying full end-to-end workflow vs. small (hot) kernels

An optimized statically compiled binary (using GCC 4.8.2 and OpenCV version 2.4.9) of the *VS* algorithm spans $\sim$1.5 million lines of assembly instructions. Error injection

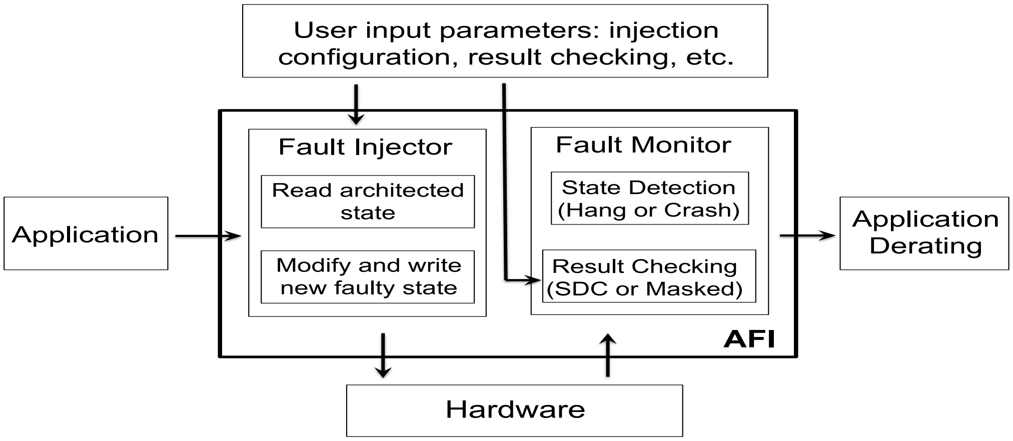

User input parameters: injection configuration, result checking, etc.
Fault Injector
Read architected state
Modify and write new faulty state
Fault Monitor
State Detection (Hang or Crash)
Result Checking (SDC or Masked)
AFI
Application
Application Derating
Hardware

Figure 2.7: Overview of the Application Fault Injection framework.

experiments on a large application like the *VS* algorithm, that run to completion, are time consuming and therefore limit the number of error injections that can be performed.

Since the application is a composition of many program and library functions, the question arises – can one simply carry out a resiliency study of some representative *hot* functions (functions that account for a significant fraction of the application execution time) and use the results to reason about the resiliency of the *VS* algorithm? If so, can one then study the resiliency of just those functions? This can lead to either reduced overhead (less number of error injections) or increased coverage (error injections take lesser time to run to completion and hence we can potentially do more of them).

To investigate further, we undertake a case study to perform resiliency analysis on a hot kernel taken from the *VS* application to see if the resiliency profile of the kernel is representative of the full end-to-end application. We show in Section 2.5.3 that this is not the case and the result of such an analysis is sub-optimal. This further motivates the need to develop and analyze full end-to-end applications that realistically simulate the full workflow, as opposed to studying just small kernel benchmarks that perform individual tasks.

Fig 2.8 shows the execution time distribution (by function) for the *VS* algorithm extracted using the Linux utility tool *Perf* [63]. Approximately 68% of the execution time is spent in OpenCV libraries [64]. 54.4% of the total execution time is consumed by just one OpenCV function – *WarpPerspectiveInvoker*, which is called from the *WarpPerspective* function, that applies a perspective transformation to an image according to a transformation matrix. Thus, we choose *WarpPerspective* as the *hot* function whose resiliency profile is studied.

We design a toy benchmark called *WP* that takes an image and a matrix as inputs and

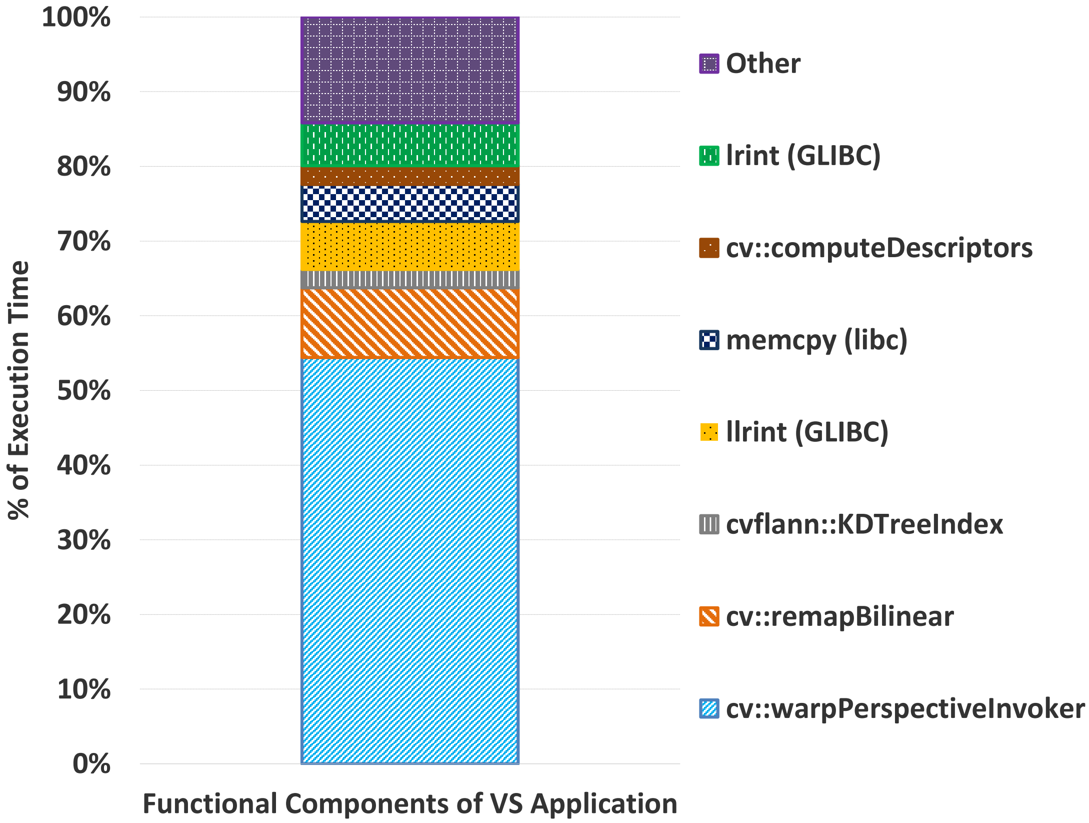


Figure 2.8: Execution profile of the *VS* application

calls the OpenCV function *WarpPerspective* on them and returns the transformed image as the output. Essentially, *WP* is equivalent to having a stand-alone *WarpPerspective* function and the output of *WP* is the return value of the function as seen by the *VS* application. The function *WarpPerspective* in turn calls two other functions: *warpPerspectiveInvoker* and *remapBilinear*. We study the outcomes from error injections in GPRs in these two functions for *VS* and *WP*. The error injection framework, AFI, affords the ability to control where the errors are injected and for this experiment we only consider the error injection experiments that inject errors in the functions of interest and observe the outcome at the end of the program (either *VS* or *WP*).

2.4.4 Defining SDC quality

As described in Section 2.4.1, any deviation in the application output due to an error is defined as a Silent Data Corruption or SDC. SDCs are the least desirable outcome of errors since they are very hard to detect until the application execution is completed and the corrupted output is generated. At that time it is too late for recovery techniques to correct the error. Crashes, on the other hand, can be detected using low cost symptom-based detectors [57] and hence protecting error sites that produce crashes incurs low overhead. Since SDCs do not produce any easily detectable software symptoms, protection against SDCs is normally done through techniques like redundancy that have high overhead. In order to reduce the resiliency overhead, we aim to quantify the *egregiousness* or *severity* of the SDCs produced so we can identify tolerable or benign SDC error sites that do not need to be protected.

Since the *VS* application produces an image (*mini panorama*) as the output, the check to determine if an SDC was produced is an image comparison between the error-free application's output (henceforth referred to as the *golden output*) and the corrupted output produced by the application execution injected with an error (henceforth referred to as the *faulty output*). To determine if there is an SDC, AFI's result checking procedure simply compares the error-free output, known *a priori*, with the output produced by the erroneous execution, and classifies the outcome as an SDC if there is any difference between the two images.

In addition to knowing how many error injection experiments result in SDCs, we am interested in quantifying the quality of the SDCs produced; i.e., the deviation between the golden and the faulty output. To do this, we define a quality metric which is calculated as follows:

Given a golden image *g_img* and a faulty image *f_img*, some global transformations are applied first to ensure that differences due to perspective, lighting, camera angle etc. are removed. This is done because, in the given system, the end purpose is to use the output image of the *VS* application for identification, tracking and/or surveillance. Hence the content of the image is of greater concern and minor cosmetic disturbances in the final image can be tolerated. The two transformed image matrices obtained after this corrective step are *g_img_tr* and *f_img_tr*. The pixel by pixel difference of these two images is given by the matrix *pixel_diff_img*, such that

$$pixel\_diff\_img = g\_img\_tr - f\_img\_tr \tag{2.1}$$

Since in the scenario of interest, the final panorama is going to be viewed by a human being, some errors in the color gradation of individual pixels can be tolerated. Thus, we only wish to capture those differences in the image where the pixel coloration is significantly modified. For this purpose, another matrix *pixel_128_diff_img* is defined where values from *pixel_diff_img* are stored only if the difference value is greater than 128, i.e. over half the range for an 8 bit pixel which can assume values between 0 and 255. Then, the *relative_l2_norm*, which estimates the deviation of the faulty output image from the golden output image (in percentage) is described as,

$$relative\_l2\_norm = \frac{||pixel\_128\_diff\_img||_2}{||g\_img\_tr||_2} * 100 \tag{2.2}$$

where, for an image X having n pixels $x_1, x_2, \ldots, x_n$

$$||X||_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} \tag{2.3}$$

Once the *relative_l2_norm* of a faulty image has been calculated, that SDC is assigned an integer number called the *Egregiousness Degree (ED)* which corresponds to the floor of its *relative_l2_norm* value. The higher the ED, the worst the quality of the corrupted output image. For example, if an SDC output has a *relative_l2_norm* of 10.25%, it is assigned an ED of 10. Any SDC that has a *relative_l2_norm* of greater than 100%, is not assigned an ED and is automatically categorized as an egregious SDC that must be protected.

2.5 RESULTS

2.5.1 Resiliency Profile of Video Summarization Algorithm

As described in Section 2.4.1, the minimum number of error injections needed are determined by studying the trend curves of the Mask, Crash, SDC and Hang rates with increasing number of error injections. As seen in Fig 2.9(a), the trend curves for the different rates start stabilizing after 1000 error injections and only vary slightly with increasing error injections. Thus, we conclude that a minimum of 1000 error injections is required to provide a statistical summary of the *VS* algorithm. Unless otherwise specified, all the experiments in this study use 1000 error injections (for each type of register; combined GPR and FPR is 2000 error injections).

The random error injections also provide good coverage in terms of the registers and bits in which the errors are injected. A representative histogram is presented in Fig 2.9(b). It shows that the errors are uniformly distributed among the 32 GPRs (for both inputs). We similarly confirm that the errors are uniformly distributed among 64 bits within the registers. For brevity, we have shown the coverage data (minimum error injections required and register coverage) for error injections in GPRs. Error injection experiments for all the different algorithms (GPR and FPR) show similar trends.

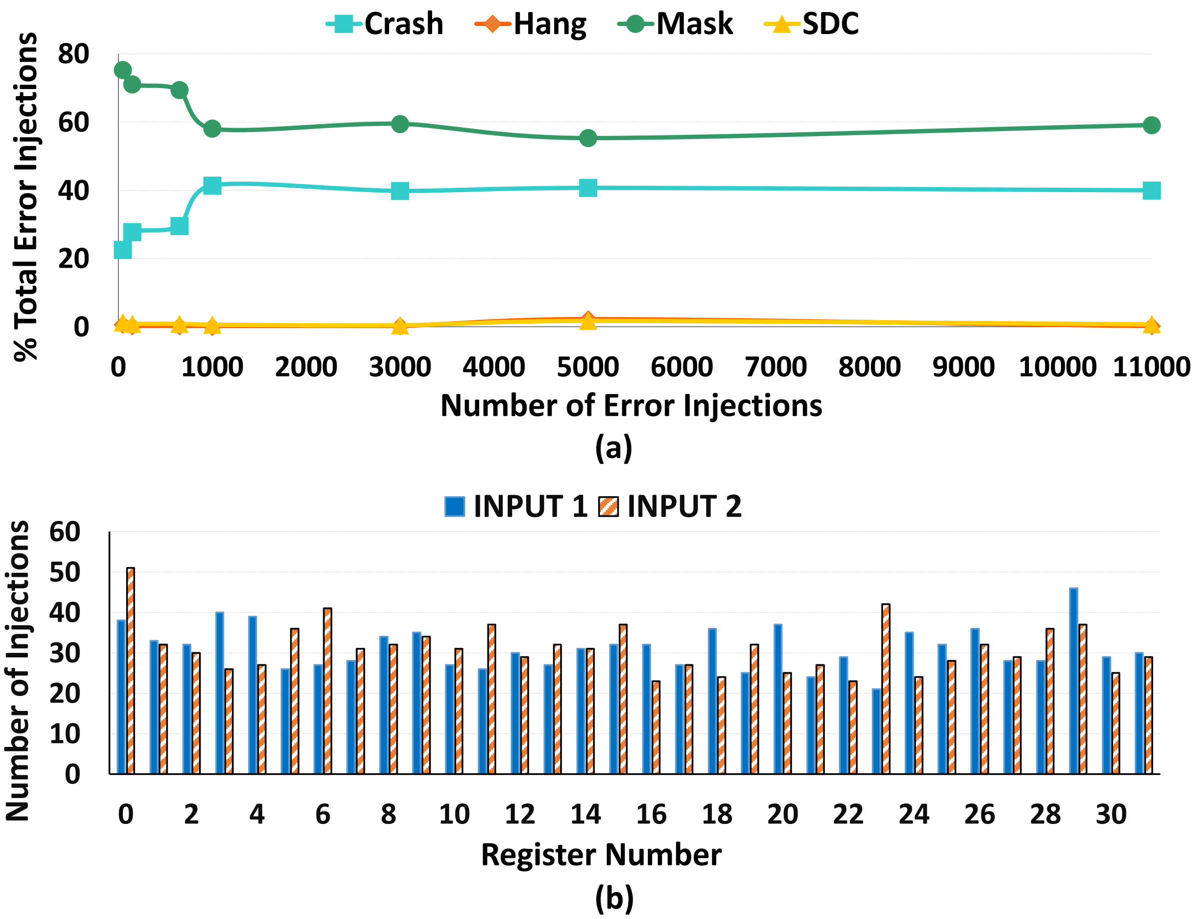


Figure 2.9: Graphs to show *Coverage* of error injection experiments. (a) Different error injection outcome rates with increasing number of error injection experiments for *VS* algorithm. The *knee of the curve* stabilizes at 1000 error injections. (b) Number of errors injected in different GPRs across 1000 experiments show a uniform distribution.

Figure 2.10 shows the different error injection outcome rates for 1000 error injections each in GPRs and FPRs for the *VS* algorithm. The resiliency profile looks very different for injections in the GPR and the FPR registers and we will explore these differences in the following paragraphs.

Error Injections in GPRs: Instructions that use GPRs form the bulk of the application and are heavily used in memory and control instructions and hence errors in them lead to the large Crash rate (40.16%). Analyzing the Crash outcomes further, the majority of the crashes can be attributed to the following two causes: 1) Segmentation Faults that generally occur due to memory access violations (92%), and 2) Abort signals raised by the application/library when it encounters internal constraint violations (8%). Analyzing the Crash causing error sites further, no clear trend is seen that corruption of certain registers or

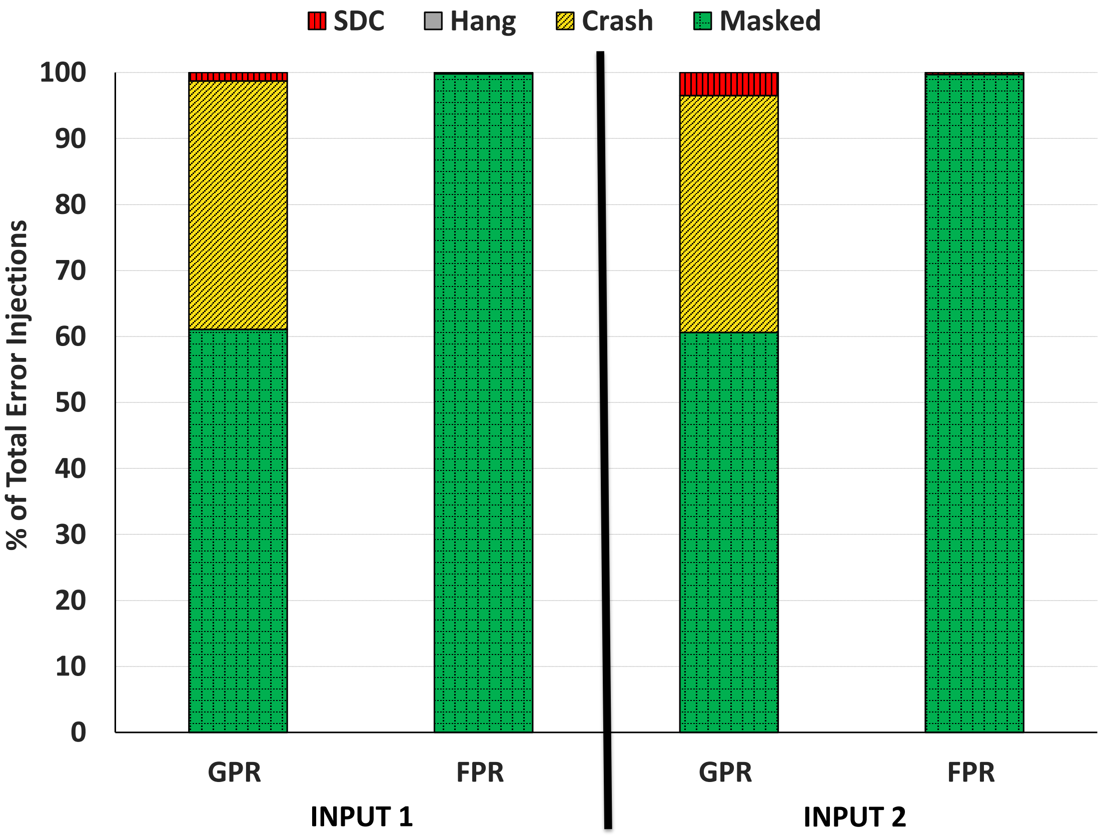


Figure 2.10: Resiliency Profile for the *VS* algorithm. Different error outcome rates for errors injected in GPR and FPR registers for the two different inputs are shown.

bit positions in the registers are more likely to result in a Crash. This is primarily because all the GPR registers are used heavily in control (corruption of any bit can cause a Crash) and memory (higher order bits more likely to cause a Crash) operations and, hence, are vulnerable to catastrophic outcomes when corrupted.

Error Injections in FPRs: Errors injected in the FPRs of the *VS* application are Masked 99.7% of the time. This is due to the way FPRs are used in the application. The *VS* algorithm operates on images which are stored as 8-bit integer pixels. Floating point operations are only used when some manipulation of the pixels is required. To do this, the integer pixels are converted to floating point, some transformation is applied and then they are converted back to integer using a saturation algorithm. The saturation algorithm causes many potentially SDC causing errors to become masked.

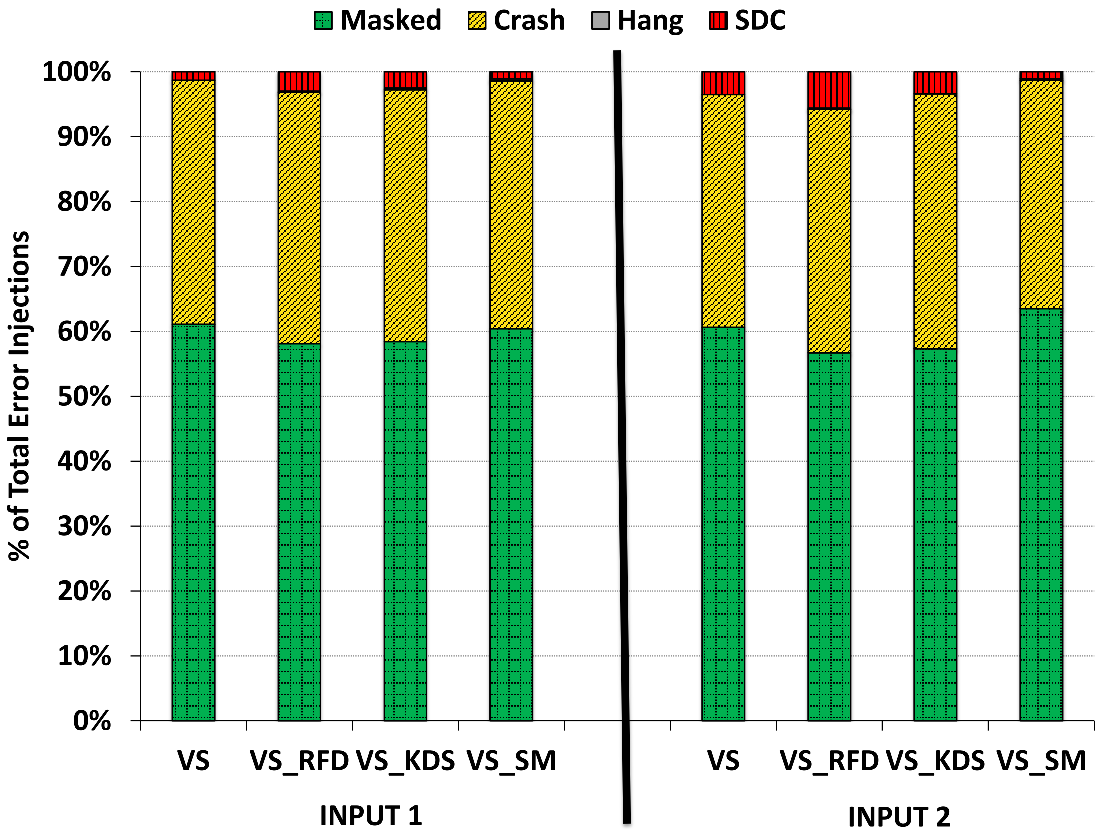


Figure 2.11: Resiliency Profile for the *VS* algorithm and its approximate versions. Different error outcome rates, for errors injected in GPR register for the two different inputs, are shown.

2.5.2 Resilience of Approximate VS Algorithms

Figure 2.11 shows the error injection results for 1000 error injection experiments in the GPRs of the different approximation algorithms compared with the baseline *VS* application for both the inputs. Similar to the baseline *VS* algorithm, FPR error injections in the approximate algorithms are masked $>$ 99.5% of the time and hence we do not show them here. The Crash, Mask and Hang rates of the approximate algorithms is very similar to the baseline *VS* algorithm. This is not surprising since the execution profiles of the approximate algorithms are very similar to the baseline *VS* algorithm. For Input1, the SDC rates increase from 1% (*VS*) to 3% and 2.5% for *VS_RFD* and *VS_KDS* respectively. In both these approximate algorithms, the errors that may have been masked in the final image (due to overlap by similar frames in the stitching process) are now exposed as SDC due to a reduction in redundancy; precipitated by dropping frames from input in *VS_RFD* or due to insufficient matching key-points in *VS_KDS*.

2.5.3 Trade-offs of studying an end-to-end workflow

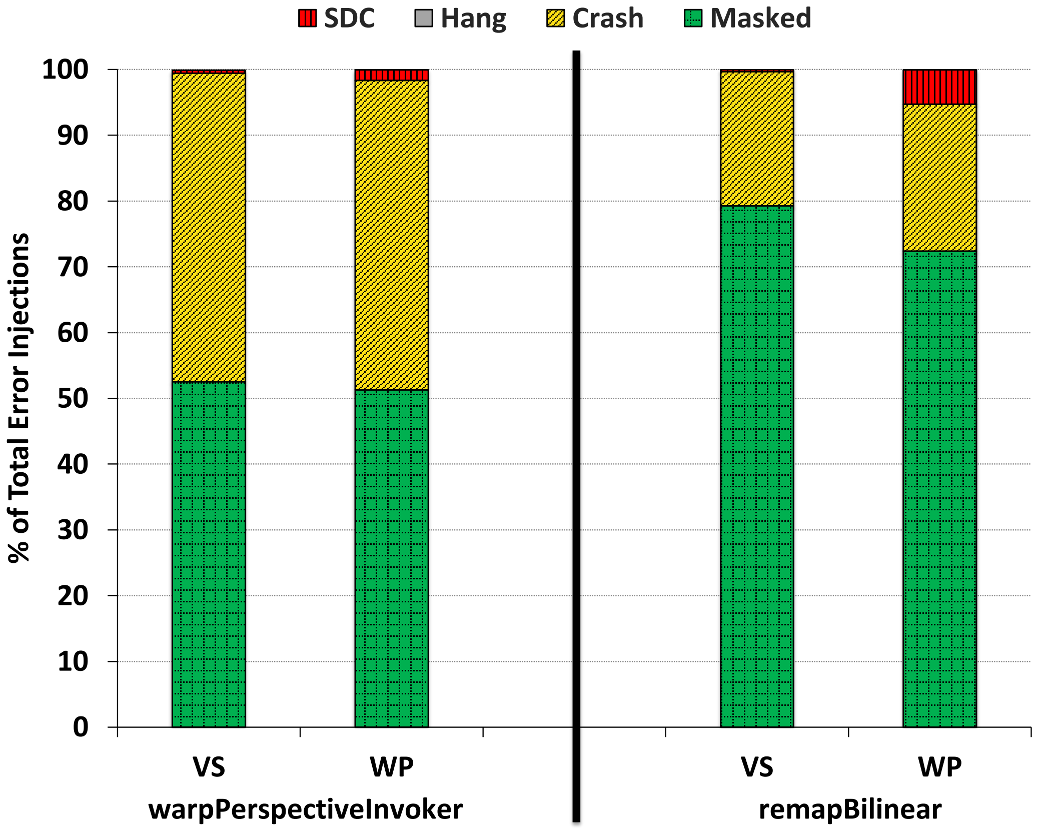


Figure 2.12: Comparison of the Masked, SDC and Crash rates for error injections in two *hot* functions for *VS* application and the stand-alone toy application *WP*.

As discussed in Section 2.4.3, we ask the following question: can one estimate the resiliency of the *VS* application by studying the resiliency of the representative stand-alone *WP* application? The results of the error injection experiments for both *VS* and *WP* are shown in Figure 2.12.

The Crash, Mask and SDC profiles of the standalone *WP* is different from that of an end-to-end workflow like the *VS*. In the *VS* application, the output of the *WarpPerspective* function would then be used to perform some other computation further down the workflow and, hence, there is a compositional effect where multiple computations flow into each other. This causes the effects of an error to manifest differently than if the workflow ended at the output of the *hot* function. In our case, the compositional effect leads to higher masking as the SDCs that are generated by errors in the *WarpPerspective* function are masked later in the workflow (for example, an adjacent image could later be stitched over the area corrupted by the function output).

Hence, we conclude that it is essential to analyze an entire end-to-end workflow, instead of just studying hot kernels/functions, to get an accurate understanding about the resiliency behavior of an application.

2.5.4 SDC quality

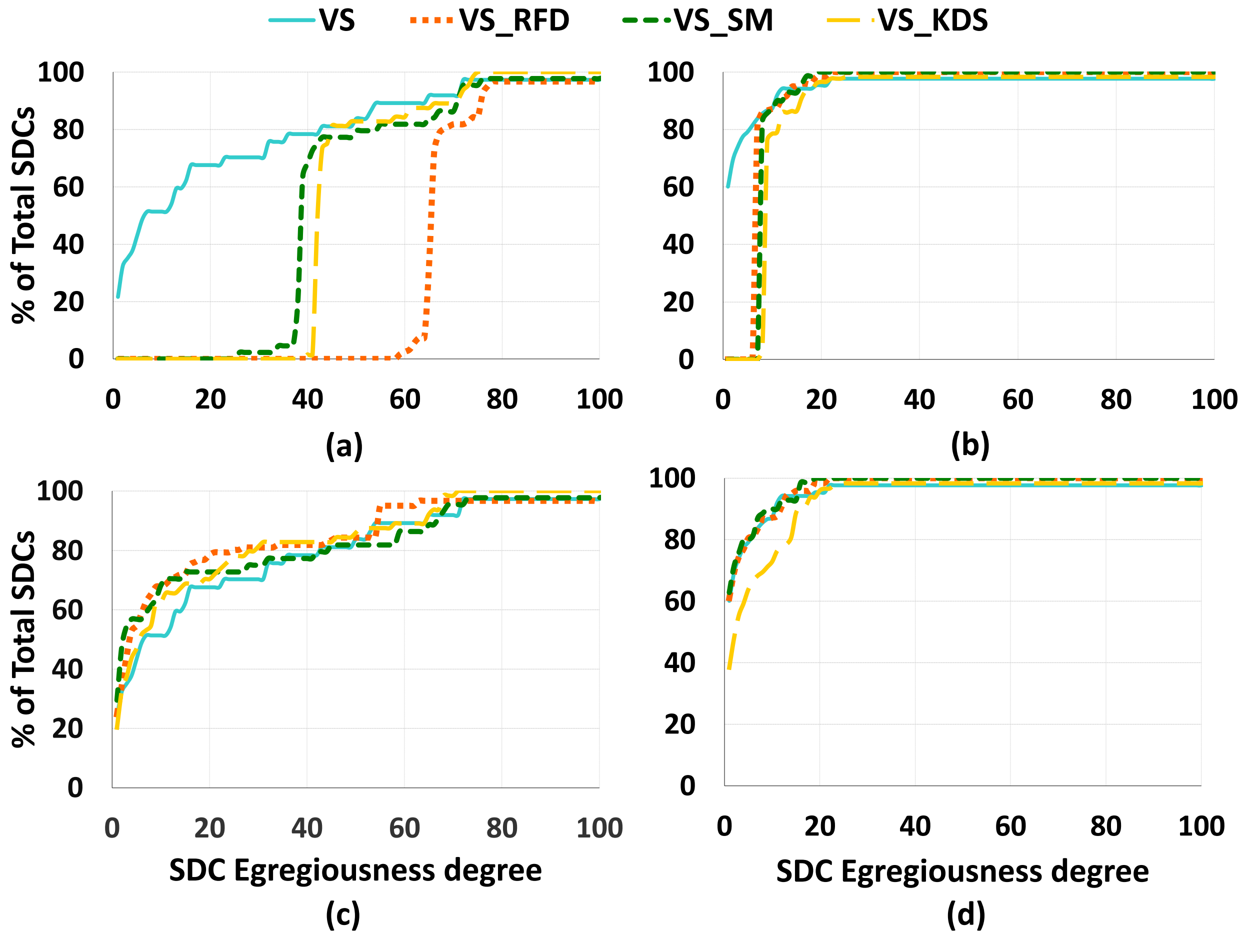
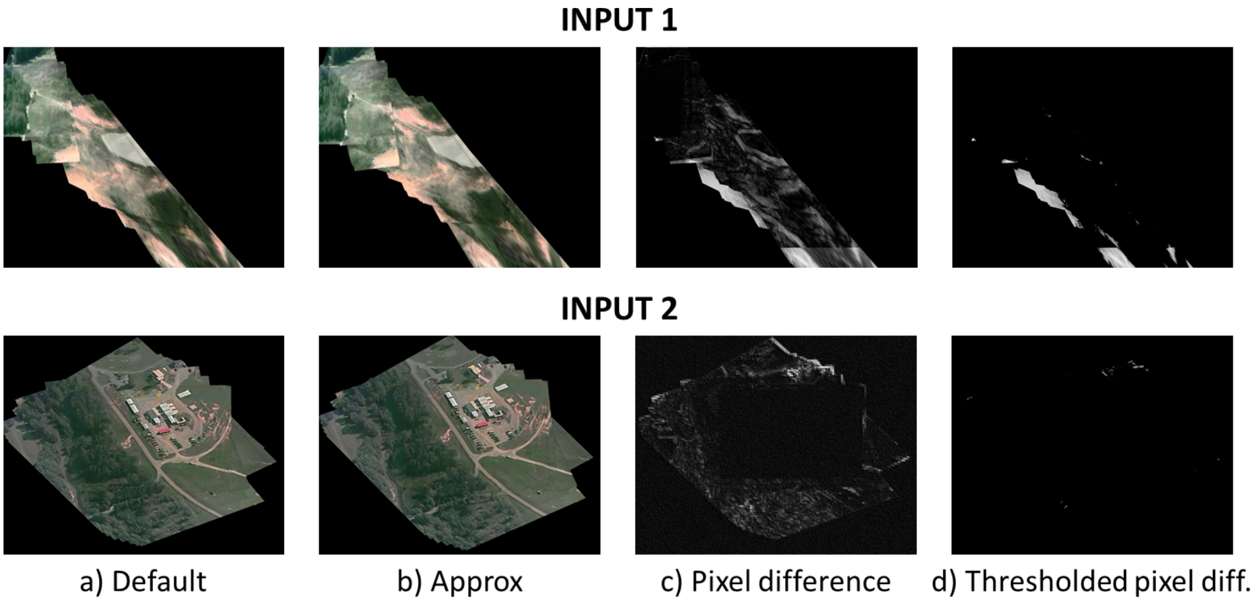


Figure 2.13: Quality of SDCs generated by GPR error injections in different video summarization algorithms. Each point on a given curve represents the percentage of SDCs (Y axis) generated that have an ED less than or equal to the ED represented on the X axis. (a) and (b) - The ED of the SDC is calculated by comparing against *VS_golden* for Input1 and Input2 respectively. (c) and (d) - The ED of the SDC is calculated by comparing against the corresponding *Approx_golden* for Input1 and Input2 respectively. Some of the curves do not reach 100% on the Y axis due to a very small fraction of SDCs that are classified as needing protection and not assigned an ED.

Figure 2.13 classifies SDCs according to their ED. In order to study a statistically significant number of SDCs, we perform 5000 GPR error injections per input and analyze the resulting SDCs. To calculate the ED of an SDC output image, it is compared to a baseline *golden* image. For the SDCs produced by the *VS* algorithm, this is straightforward as the golden image is the output of the error-free execution of the *VS* algorithm. For SDCs produced by the approximate algorithms, there are two potential golden images to compare against – the golden *VS* output (*VS_golden*) or the golden output of the corresponding approximate algorithm (*Approx_golden*). For example, ED of an SDC produced by error injection in *VS_RFD* can be calculated by either comparing it to *VS_golden* or by comparing it to *VS_RFD_golden*. We show the distribution of SDC egregiousness using both these methods.

Figure 2.13(a) and Figure 2.13(b) show the ED of the SDCs when compared against *VS_golden* for Input1 and Input2 respectively. The degradation in SDC quality in the approximate algorithms, as evidenced by the larger fraction of SDCs having higher EDs, is particularly sharp for Input1 (Figure 2.13(a)). On further analysis we observe that this is because the deviation between *Approx_golden* and *VS_golden* calculated using the metric specified in Section 2.4.4 is large. Even though it is verified by visual inspection that the approximate algorithm outputs are acceptable (Section 2.3.1), the metric assigns them a large ED. This may imply that the metric used is very conservative and we undertake a discussion about this in Section 2.6. For example, the ED of the *VS_SM_golden* for Input 1 when compared to *VS_golden* is 37. It thus follows that all subsequent SDCs produced by *VS_SM* will have an ED greater than or equal to 37. This is the reason for the shift in the ED curves of *VS_SM* with respect to the baseline *VS* in Figure 2.13(a).

Thus, to get a true understanding of the quality of the SDCs produced by the approximate algorithms, their egregiousness is estimated by comparing them to their corresponding *Approx_golden* output (Figure 2.13(c) and Figure 2.13(d)). The graphs show that the overall trend for the SDC quality for the *VS* and its approximate algorithms are very similar. The approximations do not fundamentally change the quality of the SDCs produced. For Input 2 (Figure 2.13(d)), the SDCs from *VS_KDS* have slightly worse quality (80% of SDCs produced by *VS* have ED less than 6 as opposed to ED of 14 for *VS_KDS*). This follows the trend seen in Section 2.3.1, where for Input 2, *VS_KDS* shows the most energy gains from less computation as a result of dropped frames. This in turn leads to a degradation of output quality. Another trend seen is that overall, the SDCs produced are relatively benign (even with our conservative metric). For example, for Input 2, 87%, 87%, 90% and 73% of the SDCs for *VS*, *VS_RFD*, *VS_SM* and *VS_KDS* respectively have an ED of less than 10. Thus, a large majority of the SDC causing error-sites need not be protected if an error of 10% is acceptable.



a) Default b) Approx c) Pixel difference d) Thresholded pixel diff.

Figure 2.14: a) Default output (*VS*) b) Approximate output (*VS_SM*) c) Absolute pixel difference between default and approximate outputs d) Thresholded difference between default and approximate outputs.

Hence, although approximating the *VS* application minimally changes its resiliency profile by slightly increasing the number of SDCs generated, this is offset by the fact that a large percentage of these SDCs may be tolerable and hence the cost of protecting them is low. *Thus, it is possible to realize safe, yet efficient approximations for this state of the art Video Summarization algorithm from the point of view of performance, power and reliability.*

2.6 DISCUSSION ON SDC QUALITY METRIC

Gauging if an approximate algorithm is good enough or if an SDC is tolerable in image processing applications like the *VS* algorithm is heavily dependent on the image comparison algorithm that calculates how closely the approximated image or the faulty image matches the original image. While manual inspection is still the best way to determine if the quality of an image is acceptable, this is impractical in cases where a large number of such images are generated or when an automated decision has to be made based on the error seen in the output. In Section 2.4.4, I outline an algorithm and metric to estimate the error in the output image, but this algorithm can produce false positives and can label some SDCs as more egregious than they actually are. For the outputs of the *VS_SM* algorithm the *relative_l2_norm* generated by the image comparison algorithm is approximately 37% and 8% for Input1 and Input2 respectively. This is because as can be seen in Figure 2.14(c), the pixel difference of the two images is considerable as the pixels in the faulty image have slightly shifted when compared to the default image. But to a human viewing these two images, there is no perceivable difference. Another factor to consider is that two images having the same *relative_l2_norm* may not be equally egregious depending on the final usage of the output. For example, even if 30% of a faulty image is blacked out, it may still be useful for surveillance or tracking if the remaining 70% had useful information that can be deciphered by a human being. Estimating an automated metric to compare images used for such domains remains an open problem.

2.7 CONCLUSION

In this work we study an end-to-end video summarization *VS* application that serves as a representative emerging workload for the domain of Real Time Edge Computing. We characterize the workflow of the application and examine three different approximation techniques to improve the power and performance efficiency of the workload while maintaining sufficient output integrity. We undertake a detailed resiliency study of the application as well as its approximate versions and show that the approximations do not degrade the resiliency of the baseline algorithm. We further introduce metrics to quantify the error introduced in an output image and use them to understand the behavior of SDCs in the different Video Summarization algorithms. We show that many of the SDCs produced by the application can be tolerable to the end user and hence can reduce the cost of protecting the application against transient faults. Thus, we conclude that error-efficient techniques such as software approximations are not only effective but also safe for emerging edge-computing applications.

Chapter 3: AUTOMATED APPLICATION-LEVEL ERROR ANALYSIS

Error-efficient computing techniques redefine "correctness" as producing results that are good enough or of sufficient quality to be acceptable to the user. Hence, one of the fundamental requirements for achieving safe and effective error-efficiency is to understand how errors (perturbations in computation and data) encountered during a program's execution affect the quality of the program output. With such knowledge, the system or end-user would have the ability to precisely make the desirable trade-offs in quality, resiliency, performance, and resource usage. The process of determining the output (quality) produced by a program in the presence of (given) errors is referred to as *error analysis*. The "holy grail" of error analysis is that it must be: (1) *Accurate*; it should provide the guaranteed impact of a given error on the end-to-end output quality, (2) *Precise*; it should determine the impact on output quality at fine granularity, (3) *Comprehensive*; it should provide the output quality impact for virtually all errors for a given error model, (4) *Automatic*; it should impose the absolute minimal programmer burden, (5) *General*; it should be applicable for general error models and applications, and (6) *Low-cost*; it should perform comprehensive error analysis across billions of errors in a program's execution in an inexpensive fashion.

Error analysis techniques that satisfy all six of the above requirements pose a significant research challenge and have thus-far remained elusive. Researchers have made significant progress by relaxing some of these requirements. For example, one class of techniques rely on empirically introducing a small (statistically determined) sample of errors during a program's execution (referred to as *error injection*) and observing the resultant behaviour [22, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74]. These techniques aim to provide statistical averages or probabilistic bounds of program behaviour under errors; they cannot comprehensively and accurately guarantee impact on output quality for the large majority of errors that are left out during sampling. Another class of techniques leverage program analysis to understand how errors in data may propagate to affect program output [15, 75, 76, 77, 78, 79, 80, 81, 82]. While fast (low-cost), such techniques cannot accurately and precisely model an error's impact on execution since they use information from an error-free execution.

In the absence of systematic and general-purpose error analysis methodologies, a large majority of error-efficient techniques today rely on program or domain experts to understand the error tolerance characteristics of their program and articulate it via custom annotations, data-types, error-bounds on function parameters etc. [3, 4, 5, 17, 18, 19] . This has been very limiting since such expertise is sparse and can take years to develop for specific domains [20, 21, 22]. To mitigate this limitation, this work develops a suite of automated application-level error analysis tools [25, 26, 31] that can automatically extract the error characteristics of applications.

The error analysis tools developed as part of this work can accurately (>95% on average, up to 99.9%) quantify the impact of virtually all errors (for a given error model) in a program's computation and data on its final output quality at very fine granularities (within a 2% error margin). Using a hybrid approach (built upon prior work [33]) of program analysis and relatively few (up to five orders of magnitude less) error injections, these tools can comprehensively analyze billions of possible errors that can impact a program's execution at low cost. They impose the absolute minimal programmer burden by only requiring the user to provide an unmodified program (along with relevant program inputs), a (domain-specific) quality metric and optionally a quality threshold. These tools are general and can be used to analyze any general-purpose application and have been evaluated for five different error models. To the best of our knowledge, *these are the first application-level error analysis tools that satisfy all six requirements of automation, accuracy, precision, comprehensiveness, (relatively) low-cost and generality.* The output of automated error analyses are *comprehensive application error profiles* that list the errors that can affect the program's execution along with the corresponding output quality expected for each of the errors. The application error profiles can be used by programmers or systems to understand the application's error characteristics.

The suite of tools described in this work have been systematically developed to enable errors analysis in both instructions (compute) and data (storage). In this work, an error is defined as a perturbation in program state (instructions or data) caused by an underlying fault in hardware. We focus on transient errors in this work. Specifically, we develop the following three automated error analysis tools:

1. **Approxilyzer**: Approxilyzer performs analysis of errors in program instructions (compute). Approxilyzer [31] is the first automated error analysis tool to quantify the impact, of virtually all errors (for a given error model) in program instructions, on the program's output quality with high accuracy, precision and (relatively) low-cost. The error model used in Approxilyzer is single-bit transient errors in register operands of dynamic instructions. Approxilyzer's output is the application's comprehensive *instruction error profile.* We have open-sourced and publicly released Approxilyzer [27].

2. **gem5-Approxilyzer**: To address the limitations in usability imposed by the proprietary simulator (Simics) and ISA (SPARC) used in the original implementation, we developed a fully open-source re-implementation of Approxilyzer, called gem5-Approxilyzer [25]. gem5-Approxilyzer has been built using the open-source gem5

simulator [83] and designed in a modular fashion to enable extensions to different ISAs, starting with X86 in this work. We have open-sourced and publicly released gem5-Approxilyzer [28].

3. **DataApproxilyzer**: Errors in instructions (compute) and data (storage) propagate differently through the program and, hence, require different analysis techniques. While Approxilyzer and gem5-Approxilyzer analyze errors in program instructions, we developed DataApproxilyzer to analyze errors in program data. DataApproxilyzer [26] is the first automated error analysis tool to quantify the impact, of virtually all errors (for a given error model) in program data, on the program's output quality with high accuracy, precision and (relatively) low-cost. The error model used in DataApproxilyzer is multi-bit (1-bit, 2-bit, 4-bit and 8-bit) transient errors in system memory. The output of DataApproxilyzer is the application's comprehensive *data error profile*.

The rest of this chapter details the design choices, analysis techniques, methodology and effectiveness of the automated application-level error analysis tool-suite. In the rest of the document the **A**utomated **E**rror **A**nalysis tools described above (Approxilyzer, gem5-Approxilyzer and DataApproxilyzer) are collectively referred to as *AEA tools*.

3.1 MOTIVATION FOR HIGH-LEVEL DESIGN CHOICES

This section briefly explains the motivation for making specific high-level design choices (which are orthogonal) for the error analysis techniques employed in this work.

3.1.1 Application-level Error Analysis

Different error-efficient techniques have been proposed at the level of software [10, 39, 40, 41, 42, 43, 44, 84, 85, 86, 87, 88] and hardware [6, 7, 8, 9, 10, 11, 89, 90, 91, 92, 93, 94, 95, 96, 97] with different trade-offs and targeted at different resource savings. We envision future systems to contain a heterogeneous mix of error-efficient computing components with control knobs that are exposed to software. The optimal combination of error-efficiency techniques for a given workload can then be customized or tuned based on the output quality requirements of the user and the error resilience characteristics of the workload. This methodology is analogous to computing systems today where selective computing elements (e.g., functional units, accelerators etc.) are activated at the direction of software (the workload). As will be shown in Section 4.2, different applications have widely varying error characteristics and a one-size-fits-all approach to error-efficiency is sub-optimal. Hence, this work focuses on developing systematic methodology to perform error analysis at the application-level.

3.1.2 Object Code Analysis

Application-level error analysis can be performed at source code [82, 98], compiler IR [99, 100] or object code (assembly) [13, 33] level. The advantages of source code and IR level analysis are twofold: (1) It is easier to reason about the results of the error analysis in relation to the source code and hence there is a simple 1:1 mapping to any programming language framework where the error characteristics can be specified and (2) The generality of error analysis is preserved across different architectures.

However the cost of preserving this generality is sacrificing customized "tight" solutions that may leave some computing efficiency untapped. Since the end goal is to enable maximum benefit from highly optimized and customized error-efficiency solutions, we choose to perform the error analysis on assembly level (object) code. We will show that this choice is validated (in Section 4.2.4) by undertaking a comparison (using gem5-Approxilyzer) of the same application compiled to two different ISAs. The error profiles of the same application are very different – leading to different static instructions being picked for resiliency and approximate computing solutions – under the two different architectures. This opportunity to customize architecture-specific error-efficiency solutions could not have been afforded by analysis of the source code or IR. Hence, the choice is made to perform error analysis at the object (assembly) code level.

3.1.3 Architectural Error Models

We choose architectural error models for our application-level error analysis. Architectural error models (i.e., errors that affect the architectural state of the workload) are commonly used in software-centric analysis [33, 82, 98, 99] as they are general and not hardware specific. Architectural error models are also useful for analyzing software in the early design phases when detailed lower level models are not available.

While architectural error models are important and realistic [101], they account for only a subset of hardware errors that can occur in a system. Accurately modeling how low-level (gate-level) hardware faults manifest at the architecture level is an open research problem today. Simulating all low-level faults in the system provides accuracy but is intractable in speed and cost. To bridge the gap between the accuracy of low-level fault models and the speed of high-level error models, researchers have suggested techniques such as hierarchical simulations [58] and fault dictionaries [102]. These techniques, however, are either applicable to a limited set of faults or still too slow for exploring resiliency profiles for complete applications. Additionally, their complexity and lack of generality further limits their use.

In the absence of high fidelity models that can translate the effects of low-level hardware faults on the system, analysis techniques (especially those at the early stage of software/hardware development) use high-level architectural models. It is important to note that the errors represented by architectural models are realistic for a class of hardware errors. For example, an error in the output latch of an adder can propagate and result in a single bit error in the architectural register file.

While progress is being made to improve the fidelity of error models, comprehensively analyzing high-level architectural models is an important research problem. Being able to comprehensively and effectively analyze large numbers of high-level errors can not only provide increased coverage (for modeling the effects) of hardware errors today [101] but can potentially play an important role in comprehensive testing of hardware and software in the future (when better translation is available between the low-level and high-level error models). This work significantly improves the state-of-art to show how high-level architectural errors can be analyzed accurately, comprehensively and at low-cost (a problem that was intractable before this work).

We note, however, that concepts and methodology developed as part of this work are general and can potentially be applied to other (lower-level) error models as well. For example, a recent tool called Merlin [103] takes inspiration from hybrid error analysis (specifically the concept of error equalization similar to that used in this work) and applies it to a set of lower-level micro-architectural errors. Extending the error analysis techniques from this work to comprehensively analyze other error models is part of our future work.

3.2 MOTIVATION FOR HYBRID ERROR ANALYSIS

Error analysis techniques in the literature can be classified into three main categories (focused largely on transient errors):

(1) Error injection: The most widely used evaluation technique is error injection, where an error is injected (typically one error at a time) in a given cycle in a real or simulated system and its impact studied [56, 57, 58, 59, 60, 61, 99, 104, 105, 106]. This technique can predict the impact of an error with high accuracy. Unfortunately, comprehensively injecting each error of interest at each cycle in an execution is prohibitively time consuming. Most work therefore performs statistical error injection which injects selected error types in a randomly selected sample of execution cycles. While such a technique can provide statistical summaries of the program's behaviour under errors (for example, silent data corruption rates), it does not accurately determine the impact of errors that are unsampled.

(2) Program analysis: Some evaluation techniques examine certain (static or dynamic) program properties to identify program locations that are vulnerable to errors [15, 75, 76, 77, 78, 80, 82, 98, 107]. These techniques are much faster than error injection based techniques, but their accuracy (in precisely modeling the impact of errors on program output) has not been previously validated

(3) Hybrid injection+analysis: A recent work, called Relyzer [33], demonstrates the use of a hybrid technique – which uses a combination of program analysis and error injection – to comprehensively analyze virtually all single-bit errors in integer registers of dynamic instructions within a program. Relyzer applies static and dynamic program analyses (with some heuristics) to determine when different errors in the program's instruction result in the same outcome (i.e., impact the output similarly). For a set of errors that are shown to produce equivalent outcomes, Relyzer performs error injection for only one of these errors (referred to as the *pilot*) to determine that outcome. Relyzer uses program analysis to prune the number of error injections needed for comprehensive error analysis by 99.78% and demonstrates an accuracy of >96% (for error outcome prediction). Relyzer's analysis is used to accurately determine which instruction errors will lead to Silent Data Corruptions (SDCs), where an incorrect output is produced.

Although Relyzer is 2 to 6 orders of magnitude faster than comprehensive error injection, it unfortunately still spends a significant amount of time in error injection for the pilot errors. Thus, while Relyzer is certainly more practical than comprehensive pure error injection, compared to program analysis based techniques, its accuracy comes at a significant cost in speed – the program analysis techniques studied average running times of approximately 5 CPU hours compared to 2.5 days of wall clock time on a 188 cluster node for Relyzer (90% of this time is spent on error injections).

A legitimate question therefore is whether current program analysis based techniques obviate the need for hybrid tools such as Relyzer altogether. As mentioned above, it has not been previously possible to test the accuracy of program analysis based techniques. The availability of Relyzer allows us to perform such a test.

In this work, we evaluate the accuracy of several program analysis based techniques using Relyzer [30]. We use program based metrics proposed by [98] and some derivatives as examples of pure program analysis based techniques that have been suggested to measure the vulnerability of instructions to SDC. By comparison with Relyzer, *we show that, in general, these program analyses based metrics and their various derivatives and combinations are unable to adequately predict an instruction's vulnerability to SDCs [30].* Although it is possible that other analysis based techniques (and more sophisticated machine learning techniques) are more accurate, a comprehensive evaluation of all such techniques is outside the scope of this work. Nevertheless, *the negative results provided here do indicate that there is much work needed to develop accurate pure program analysis based techniques, and hybrid techniques (like Relyzer) provide a valuable solution to the error analysis problem. Thus, a hybrid error analysis technique is chosen for developing the automated error analysis tools described in this work.*

3.2.1 Methodology to evaluate program analyses based techniques

To evaluate the accuracy of pure program analyses based techniques, we study metrics proposed by [98] and some derivatives. Particularly, we explore the following two metrics from [98] for a given static instruction, as an indicator of its vulnerability to producing SDCs. (1) *Fanout* is defined for a static instruction that writes to a register as the cumulative fanout of all the dynamic instances of the instruction. Fanout for a dynamic instruction that writes to a register R is defined as the number of dynamic uses of R before the next dynamic write to R. (2) *Av.lifetime* is defined for a static instruction that writes to a register as the average of the lifetimes of dynamic instances of the static instruction. Lifetime for a dynamic instruction I_d that writes to a register R is defined as the number of cycles from the execution of I_d to the last use of R before the next dynamic write to R.

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

We evaluate the five metrics using five single-threaded applications – two (randomly selected) from PARSEC [108] and three (randomly selected) from the SPLASH-2 [109] benchmark suites. Table 3.4 provides a brief description of these applications and inputs used. We collect the values of these metrics at the instruction level using Wind River Simics [110]. All the metric values are normalized to one. Since lifetime and fanout for an instruction are derived from its destination register, the metrics evaluation is restricted to errors in destination registers. For a given static instruction, we also obtain the number of SDCs it produces by employing Relyzer and use it as the golden metric (*sdc*), also normalized to one.

Although Relyzer analysis is orders of magnitude faster than naiv̈e error injections campaigns, it still requires a significant number of error injections (in the pilots chosen through

	Application	**Input**
PARSEC 2.1	Blackscholes	sim-large
	Swaptions	sim-small
SPLASH 2	FFT	64K points
	LU	512 × 512 matrix, 16 × 16 blocks
	Water	512 molecules

Table 3.1: Applications studied.

error equalization) to cover all the error-sites (program locations where an error can occur; which for Relyzer's error model is integer register bits in dynamic instructions). In the interest of simulation time, prior work has instead chosen to analyze or *cover* a large subset (>95% but <100%) of error-sites, which is sufficient for most applications [33]. In this work, an aggregated Relyzer coverage of 97% is used. To ensure that the missing error information did not skew the metrics evaluation, we additionally ensured that all error-sites belonging to static instructions that cover at least the top 75% of the metric values were analyzed. Any missing information, therefore, is from static instructions that have among the lowest metric values and the lowest dynamic instruction counts.

Metric evaluation:

Three different methods are used to evaluate the metrics described above. The first two methods quantify how accurately the individual metrics predict SDCs in isolation while the third evaluates a combinations of the metrics.

Correlation coefficient: For each application, correlation coefficients[1] between individual metrics and *sdc* (golden metric) is measured to study the linear relationship between them.

Cost vs. SDC reduction: We note that the objective of estimating SDCs with metrics is to identify optimal set of SDC-targeted error detectors. We therefore employ a 0/1 knapsack algorithm to find an optimal set of detectors that will provide the largest SDC reduction at a given cost – we assume duplication for detectors and charge one instruction as the cost of duplicating and comparing results for one instruction on average (similar to [13]). Thus, SDC reduction vs. cost graph for each application is obtained using the known SDC count for each instruction from Relyzer. We call this Relyzer curve (*RC*).

We then apply the same knapsack algorithm using the metric of interest, instead of the

---

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

SDC count, and plot a similar tradeoff curve which we call Prediction curve (*PC*). This curve is the predicted SDC reduction vs. cost curve if the metric were accurate. We also plot an Actual curve (*AC*) as follows: for each point on the *PC* curve, we calculate and plot the actual number of SDCs (from Relyzer) covered by the instructions actually identified by the metric in the PC curve. This gives us the actual SDC reduction vs. cost curve of the metric. For a given cost, the gap between *AC* and *RC* tells us how well the metric estimates SDCs (the smaller the gap, the better).

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

$$\begin{aligned} sdc_i = \beta_0 lifetime_i + \beta_1 fanout_i + \beta_2 av.lifetime_i + \\ \beta_3 av.fanout_i + \beta_4 dyn.inst_i + \epsilon_i \end{aligned} \tag{3.1}$$

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

$$\begin{aligned} sdc_i = \beta_0 lifetime_i + \beta_1 (lifetime_i)^2 + \beta_3 (lifetime_i)^3 + \\ \beta_4 fanout_i + \beta_5 (fanout_i)^2 + \beta_6 (fanout_i)^3 + \\ \beta_7 av.lifetime_i + \beta_8 (av.lifetime_i)^2 + \beta_9 (av.lifetime_i)^3 + \\ \beta_{10} av.fanout_i + \beta_{11} (av.fanout_i)^2 + \beta_{12} (av.fanout_i)^3 + \\ \beta_{13} dyn.inst_i + \beta_{14} (dyn.inst_i)^2 + \beta_{15} (dyn.inst_i)^3 + \epsilon_i \end{aligned} \tag{3.2}$$

3.2.2 Accuracy of pure program analyses based metrics

This section presents the results for the accuracy of program analyses based techniques, evaluated using the methodology described in Section 3.2.1

Correlation coefficients

Table 3.2 shows the correlation coefficients between *sdc* and individual metrics for all the metrics and applications studied. It shows that *av.lifetime* and *av.fanout* have virtually no correlation with *sdc* for our workloads. *Lifetime* displays weak to virtually no correlation

---

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

and *fanout* exhibits moderate correlation for Blackscholes, FFT and LU. Although *dyn.inst* is the only metric that shows high correlation with *sdc* for a few applications (FFT and LU), there is no single metric that uniformly demonstrates a strong linear relationship with *sdc* for all our workloads. Furthermore, when correlation is calculated on the combined data points from all the benchmarks (represented by *All*), none of the metrics display a strong association with *sdc*.

Applications	vs	*lifetime*	*fanout*	*av. lifetime*	*av. fanout*	*dyn.inst*
Blackscholes	*sdc*	0.25	0.56	-0.05	-0.04	0.68
Swaptions		-0.04	0.21	-0.03	-0.02	0.27
FFT		0.08	0.52	-0.03	-0.01	0.82
LU		0.19	0.56	-0.02	-0.01	0.80
Water		0.08	0.40	-0.02	-0.01	0.52
All		0.13	0.49	-0.02	-0.01	0.62

Table 3.2: Correlation coefficients between *metrics* and *sdc* for different workloads.

Cost vs. SDC reduction

Here we compare optimal cost vs. SDC reduction curves for the metrics vs. Relyzer by plotting the *RC*, *PC*, and *AC* curves as described in Section 3.2.1. For brevity, the cost vs. SDC reduction curves are presented for a representative subset from our workload and metric combinations in Figure 4.7.[3]

In the remainder of this section the term *gap* is used to signify the difference in the Y axis (SDC) for a given value on the X axis between the different curves.

Graphs for LU:*dyn.inst* and FFT:*dyn.inst* show a high correlation between PC and AC, which was expected based on Table 3.2. (The gap between the PC and AC curve – which shows the inaccuracy in the SDC coverage claimed by the corresponding metric – is lower for these graphs).

However, even for these best cases there is a significant gap in SDC reduction between the AC and RC curves, showing that these metrics do not pick the optimum set of detectors. For example, at dynamic instruction overhead of 20%, the loss in SDC reduction for LU:*dyn.inst* and FFT:*dyn.inst* is 37% and 17%, respectively (compared to RC). This is primarily because

---

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

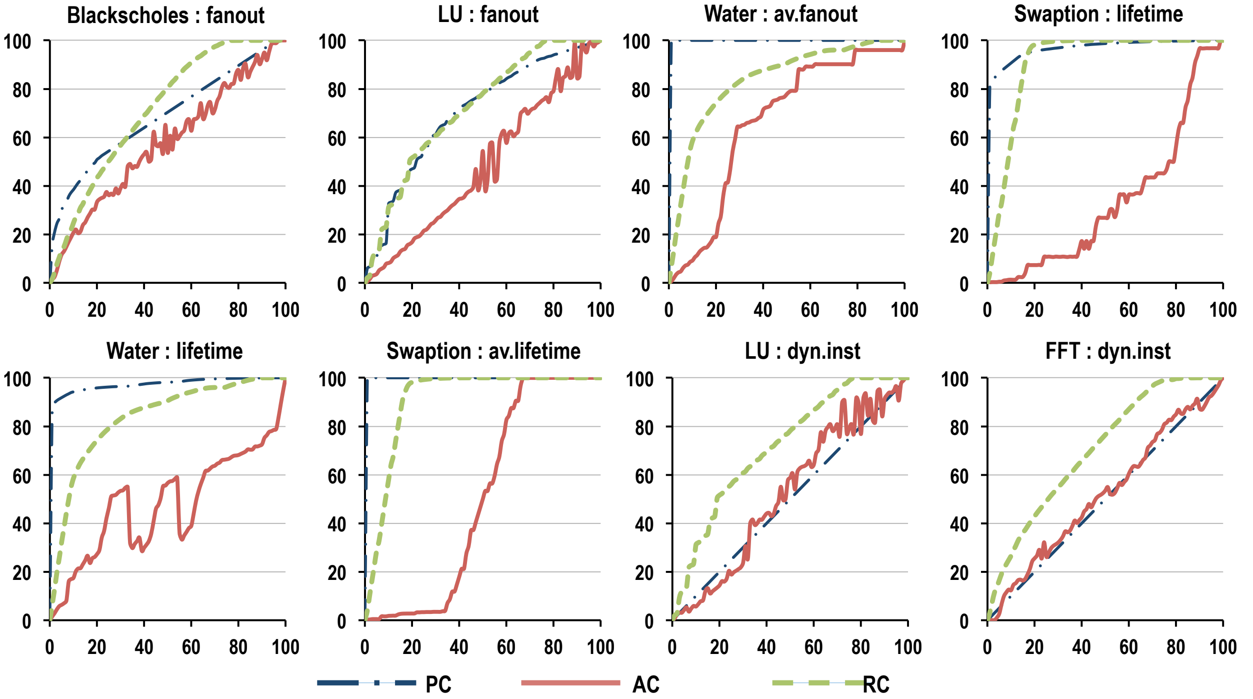


Figure 3.1: SDC reduction vs. execution overhead. The X axis plots *% execution overhead* (in terms of increase in dynamic instructions) and the Y axis represents *% reduction in SDCs*.

the instructions that do not produce SDCs were also selected for protection (false positives) by the metrics.

Overall, the significant gaps we observed in the SDC reduction between AC and RC for a given overhead reveals that the individual metrics are poor predictors of SDC causing instructions. This also indicates that correlation coefficient alone is not a determining factor in predicting SDCs.

Combining multiple metrics

Table 3.3 shows the result of the linear regression (Equation 3.1) for the workloads studied. It shows the metrics that are significant[4] to the model and the model's adjusted R^2. The adjusted R^2 value estimates the percentage of variance in *sdc* that is explained by the metrics. If the adjusted R^2 is high then the derived model is considered robust. For example, 0.66 adjusted R^2 for LU implies that only 66% of the variance in *sdc* can be explained by the metrics, which leaves 34% as unexplained or caused by randomness. However, a low adjusted R^2 value can be interpreted either as *(a)* the model is missing key additional explanatory

---

[4] A standard t-test is used to calculate the significance of the individual linear regression coefficients.

Applications	Significant metrics	Adjusted R^2	CV_{10}	CV_4	CV_2
			RMSE/Mean		
Blackscholes	*fanout*, *av.fanout*, *dyn.inst*, *lifetime*	0.61 [0.67]	1.46 [$> 10^4$]	264 [$> 10^3$]	285 [$> 10^4$]
Swaptions	*dyn.inst*, *lifetime*	0.07 [0.26]	4.48 [4.16]	4.55 [4.25]	4.64 [4.44]
FFT	*dyn.inst*, *lifetime*	0.68 [0.69]	6.91 [$> 10^7$]	10.1 [$> 10^7$]	5.94 [$> 10^5$]
LU	*dyn.inst*, *fanout*	0.66 [0.77]	4.96 [152]	4.73 [85.1]	4.95 [201]
Water	*lifetime*, *fanout*, *av.lifetime*, *av.fanout*, *dyn.inst*	0.28 [0.49]	5.82 [$> 10^3$]	6.03 [$> 10^3$]	10.3 [$> 10^4$]
All	*dyn.inst*, *lifetime*, *fanout*, *av.lifetime*	0.39 [0.50]	4.55 [9.97]	4.40 [14.3]	4.45 [79.2]

Table 3.3: Linear regression summary. The significant metrics and the main number in each cell are the result of using linear regression based on Equation 3.1. The numbers in the square brackets are results using linear regression on polynomials based on Equation 3.2.

variables (other metrics), or *(b)* that a linear model is not sufficient to explain the relationship between the metrics and *sdc*. Overall, we make the following observations:

- No common model (formed by a linear combination of our metrics) that offers a best fit for all our workloads was identified. For different applications, different metrics were identified as being significant contributors. For metrics that prove to be significant for multiple applications, the respective regression coefficients (β_i) were different. For example, even though *fanout* is identified as a significant metric for Blackscholes, LU, and Water, the regression coefficients (β_1) were 0.60, -0.21, and -0.33 respectively.

- The adjusted R^2 values varied between 0.07 (for Swaptions) to 0.68 (for FFT) and were mostly lower than desired.

- The last three columns of Table 3.3 show the ratio of the Root Mean Square Error (RMSE) to the Mean for K-fold cross validations (CV_K)[5] with K = 10, 4 and 2. Even for models that have relatively high adjusted R^2 value (e.g., for LU or FFT), the cross validation showed high errors (values >1) in the predicted and observed SDCs. For example, for FFT, the average error for CV_4 is a very high 10.1 times the mean.

The results from nonlinear regression (Equation 3.2 in Section 3.2.1) are presented in brackets in Table 3.3. They show a trend similar to that of linear regression – no common model for the studied workloads is identified. The adjusted R^2 has improved for all the workloads, which indicates that a nonlinear combination of the metrics can perform better

---

[5] Cross validation (*CV*) is a model validation technique for accessing how well the results of the analysis generalize to an independent set. A K fold cross validation splits the population randomly into K parts. K-1 parts are used for training the model and the remaining one part is used for testing. This is done K times until all the parts have been used for testing.

than a linear combination in predicting SDCs. However, for several applications the adjusted R^2 value is still poor, indicating that other metrics and/or different nonlinear regression models are required. The last three columns show that the error from cross validation is also high. In the data sets for some of the applications, there are outliers that have significantly larger metric values than others. During CV when these outlier metric values are fed into the model, they produce large deviations in the output which result in higher RMSE. Regression on polynomials further exacerbates this problem as the model exponentially increases the error. For example, in FFT just one instruction accounts for approximately 96% of *av.fanout* of the entire application and removing it brings the CV error for polynomial regression down from approximately 10^7 to approximately 5. Although removing these outliers may improve the error rate, it also means that we are removing from our analysis instructions that the metrics identify as the most vulnerable. Since we are evaluating the predictive capacity of the metrics, we choose to not remove these instructions.

In summary, simple linear and nonlinear models using the pure program analyses based metrics we study do not uniformly explain or predict SDCs in the workloads studied. While other program analysis based metrics may be more effective, these results provide evidence that developing such metrics is not straightforward and establish the value of hybrid error analysis techniques (such as Relyzer) in accurately determining the impact of errors on output.

3.3 COMMON CONCEPTS AND METHODOLOGY FOR AUTOMATED ERROR ANALYSIS TOOL SUITE

The following sections provide a brief description of common concepts, terminology and methodology employed in the development and evaluation of all three automated error analysis tools described in this work – Approxilyzer, gem5-Approxilyzer and Minnow (collectively referred to as *AEA tools*) .

3.3.1 High-Level Objective of Error Analysis

The goal of the error analysis undertaken in this work is to characterize the impact of any given error (per the error model) in a program's execution with high accuracy and precision. The term *error site(s)* is used to refer to specific points in the application's execution where an error could be encountered. Thus, if the error model used is single-bit transient errors in architectural registers (as in the case of Approxilyzer [31] and gem5-Approxilyzer [25]), then an error site will refer to a specific bit in a specific operand register of a specific dynamic instruction.

The error analysis must accurately determine the outcome (impact on program output) of an error - no output, correct output or corrupted output with quality degradation Q (measured compared to the output generated by an error-free execution) – for virtually *all* error-sites in the program. We will use the term *output quality* hereafter to refer to the degradation in output quality caused by an error in the execution.

Since error-efficient techniques trade-off output quality for resource savings, the error analysis must determine output quality with high precision – for example, the tools must be able to distinguish if an error resulted in an output quality degradation of 20% vs 23% etc. Finally, it must do this using a general methodology (applicable to different applications and error models) that is (relatively) inexpensive and imposes minimal burden on the programmer.

3.3.2 Error Models

The AEA tools are designed to analyze errors in both instructions (compute) and data (storage) in a given application. In this work we focus on transient bit-flip errors [98, 111, 112, 113, 114, 115]. Targeted bits (in instruction and data) are flipped (value changed from 0 to 1 or 1 to 0) and remain in corrupted state till they are overwritten. The error models for instructions (*used in Approxilyzer and gem5-Approxilyzer*) and data (*used in DataApproxilyzer*) are described below.

Error Model for Instructions

The error model used is a single bit transient error in instruction registers. This error model has been widely used in literature [13, 33, 98, 111, 115] and shown to be effective [115]. Specifically, our error model is a single bit transient error in an operand register of a dynamic instruction in the program. Hence, an error site refers to a specific bit in a specific operand register in a specific dynamic instruction. Errors in both integer and floating point architectural registers are studied. The error can occur in the source operand register (the error occurs just *before* the register value is read by the instruction) or the destination operand register (the error occurs just *after* the register value has been written by the instruction) of a given dynamic instruction. To illustrate with an example, consider a dynamic instance of an multiply instruction with register operands r1, r2 and r3. We consider single-bit flips in registers r1, r2 and r3 that are accessed by this instruction (one at a time, in different bits).

Error Model for Data

The error model used is multi-bit transient errors in (data stored in) system memory. We focus on multi-bit errors in memory words as they are are an increasing concern [116, 117, 118, 119] in modern memory systems. Furthermore, unlike in the case of single-bit memory errors, widely used ECC schemes are incapable of protecting the memory against multi-bit errors in the same word (two-bit errors can be detected but not corrected; errors in three or more bits can neither be detected or corrected). We study errors in data memory regions (both heap and stack); errors in memory regions containing code are not studied.

Error sites for data corruption (during a program's execution) in memory have both a spatial (*where*, i.e., which memory address) and a temporal (*when*, i.e, what cycle in the program execution) component. For example, consider a data byte D that is accessed (read-/written) multiple times (by different dynamic instructions) during a program's execution. A given error in D could impact the output quality differently depending on whether it occurs while being accessed by dynamic instruction X vs. dynamic instruction Y. Thus, it is essential to distinguish and identify data errors based on both when and where the error occurred. In order to capture the temporal component of errors we record the first dynamic instruction in the program execution that reads (error occurs just *before* the read) or writes (error occurs just *after* the write) the corrupted data bit(s). Thus, an error site is defined as individual (random) bits in data bytes that are accessed (read/written) by a given dynamic memory instruction. To illustrate using the example described above, for a given dynamic instance X of a memory instruction (load/store) that accesses (reads/writes) data byte(s) D, we consider n-bit errors (one at a time, in different combinations of n bits) in D.

Specifically, we study the following four error models:

1. 1-bit: A single bit error in a given data bit accessed by a given dynamic instruction
2. 2-bit: Error in any two data bits of the n (n >= 1) bytes of data accessed by a given dynamic instruction.
3. 4-bit: Error in any four data bits of the n (n >= 1) bytes of data accessed by a given dynamic instruction.
4. 8-bit: Error in any eight data bits of the n (n >= 1) bytes of data accessed by a given dynamic instruction.

In this work, we study errors in an abstract memory device from which data is read or written. This allows us to study the impact of errors on data while being agnostic to different memory configuration implementations. We choose this design since our goal is show that

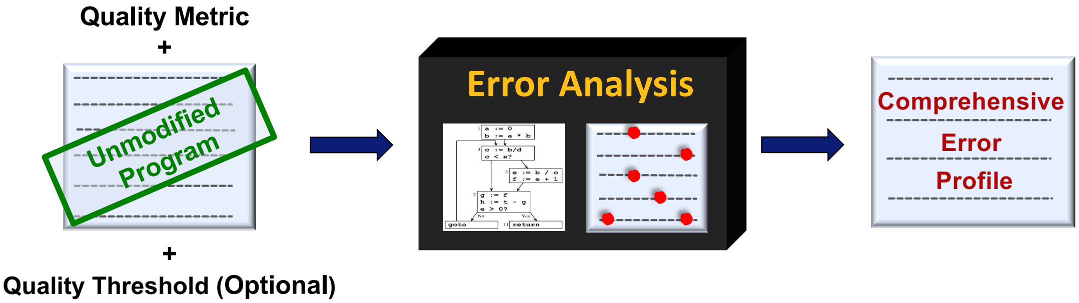


Figure 3.2: Overview of the input and output interface of the Automated Error Analysis tools.

it is possible to accurately determine the impact of any and all data errors (for a given error model). Thus, we chose an error model where data errors are always "activated" or consumed by the execution. This need not be true in systems that have caches. For instance, an error in DRAM will not be activated if a copy is present in SRAM (the execution will access the error-free data from the SRAM). Similarly, an error in a clean (not dirty) line in SRAM will not be activated if it is evicted before it can be read. The techniques presented in this work are generally applicable to errors in both SRAM and DRAM. Depending on particular system memory configurations, a subset of data errors will simply not be activated (and hence need not be analyzed).

3.3.3 Inputs for Error Analysis

As shown in figure 3.2, the AEA tools require the following inputs from the user:

1. Application: An (unmodified) application that the user wishes to analyze along with its relevant program input parameters.

2. Quality Metric: Underlying any error-efficient computing solution is the need to quantify output quality through an end-to-end quality metric. To quantify the quality (degradation) of a corrupted output, it is essential to find a measure of its difference from the golden (error-free) output. This "difference measure" is referred to as the quality metric – technically, this is a quality degradation metric since the higher the value of the difference, the lower the quality. The quality metric is domain-specific [87, 120, 121] and the AEA tools assume that the programmer or user will supply it.

3. Quality Threshold (Optional): Another parameter pertinent to many use cases for error-efficient computing is the quality threshold that sets a bound on the maximum

quality degradation that is acceptable to the user. The quality threshold is an optional parameter that the AEA tools can take as input from the user. Since programmers may want to use the AEA tools for analysis or tuning, these tools enables them to specify quality threshold ranges if they so desire. In the limit, no threshold range may be specified, in which case the tools will perform their analyses for the full range of quality degradation.

Along with the application, the quality metric is the absolute minimal input required by any error analysis tool. It is a tall order for any tool to predict how the user intends to use the program output and, hence, how they wish to calculate output quality. For example, in a program with numerical outputs, the user might want to measure the average deviation (from the golden output) across all components or might want to measure the maximum deviation for each individual output component. Error analysis tools (including the AEA tools developed in this work) assume that the user will provide the quality metrics. Although an orthogonal concern, in order to assist the user, we envision incorporating simple domain-specific libraries in the error analysis 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. Section 3.3.5 describes specific metrics and thresholds used in evaluating the AEA tools. The error analysis methodology developed in this work are independent of the quality metric and quality thresholds chosen.

In summary, the AEA tools developed in this work place the absolute minimal burden on the user, by only only requiring an (unmodified) application, an end-to-end quality metric and, optionally, acceptable quality thresholds.

3.3.4 Quality Aware Error Outcome Categorization

The impact of an error of the output of an execution is referred to as *error outcome*. Traditional error analysis schemes broadly classify error outcomes as either **Masked** (the effects of the error are masked by the execution and correct output was generated), **Detected** (no output was generated, as in the case of program crash or hang), or **Silent Data Corruption** (an incorrect or corrupted output was generated). Prior tools like Relyzer do not consider output quality, marking all corruptions in an output (no matter how trivial) as

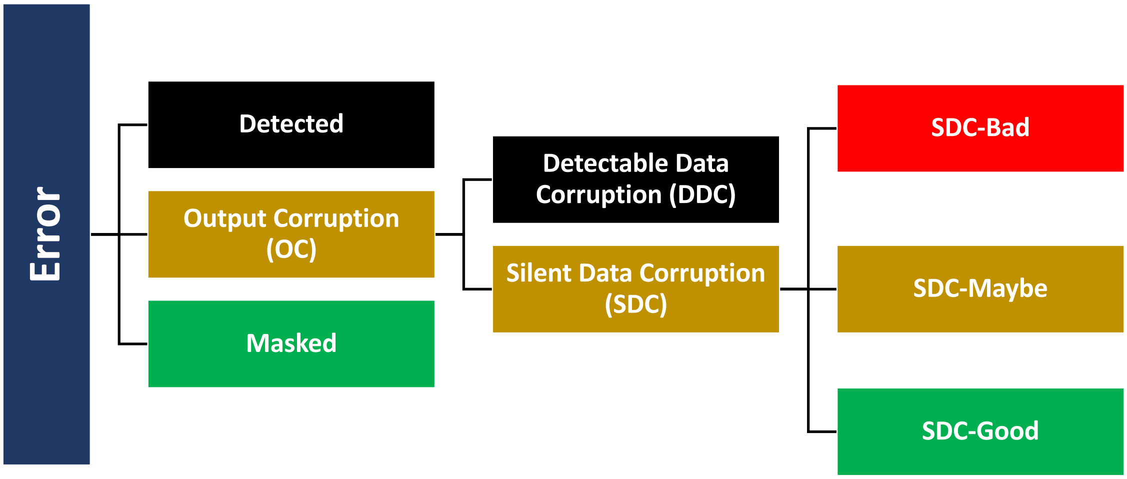


Figure 3.3: A quality aware error outcome classification and their implication for error-efficient approximation and resiliency.

a Silent Data Corruption (SDC) outcome.

The AEA tools refine the notion of an error outcome by including the quality of the erroneous output as part of this outcome. They assess the quality of the corrupted/erroneous output by measuring its deviation from the error-free/precise output using the quality metrics provided by the user. Thus, an error that produces a quality degradation of (say) 20% is said to have a different outcome from one with a quality degradation of (say) 25%.

In this work, we introduce a new categorization of error outcomes [31] to incorporate the notion of output quality into the category of errors traditionally known as Silent Data Corruptions (SDCs), as illustrated in Fig 3.3.

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 (error-free) 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 [12, 13]. Additional detectors are introduced in the AEA tools 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 (SDC).

The SDCs are further categorized as follows:

SDC-Good: These SDCs are *highly tolerable* SDCs 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).

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 threshold.

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), the the SDC error sites are classified into SDC-Good, SDC-Bad, and SDC-Maybe.

To assist the user, the AEA tools incorporate simple domain- specific and common sense quality metrics and thresholds that a user can choose. For example, negative values and infinities are examples of obvious DDC outputs for many financial applications, and quality differences of less than one-hundredth of a cent could be assumed to be SDC-Good. 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 3.3.5); Otherwise all the SDC error sites are classified as SDC-Maybe.

For each error site belonging to the SDC-Maybe error class, the AEA tools also records its associated output quality degradation. Hence, the output quality for a given error site is characterized by its error outcome class (also called error outcome category or simply error category) and additionally, in the case of SDC-Maybe, by the amount of quality degradation introduced in the output.

3.3.5 Workloads and Quality Measures

To evaluate the AEA tools, we use workloads across three different benchmark suites (Parsec [122][6], Splash-2 [109] and Accept [4]) and spanning multiple application domains.

---

[6] Approxilyzer uses Parsec 2.1, whereas gem5-Approxilyzer and DataApproxilyzer use Parsec 3.0. The two applications (Blackscholes and Swaptions) studied from the Parsec suite do not significantly vary between

	Application	Domain	Description
PARSEC [122]	Blackscholes	Financial Modeling	Calculates prices of options with Black-Scholes partial differential equation
	Swaptions	Financial Modeling	Computes prices of a portfolio of swaptions using Monte Carlo simulations
SPLASH-2 [109]	FFT	Signal Processing	1D Fast Fourier Transform
	LU	Scientific Computing	Factors a matrix into the product of a lower & upper triangular matrix
	Water	Scientific Computing	Evaluates forces and potentials that occur over time in a system of water molecules
ACCEPT [4]	Sobel	Image Processing	Image convolution kernel implementing the Sobel filter

Table 3.4: Applications studied.

Application	DDC	SDC-Good	SDC-Bad	SDC-Maybe
Blackscholes	$F_i > \$500$ $F_i < \$0$	max-abs-diff $< \$10^{-4}$	max-rel-err $> 100\%$	max-rel-err
Swaptions	$F_i > \$500$ $F_i < \$0$	max-abs-diff $< \$10^{-4}$	max-abs-diff $> \$1$	max-abs-diff
LU	No App-Specific Detectors	max-rel-err $< 10^{-4}\%$	max-rel-err $> 100\%$	max-rel-err
Water	No App-Specific Detectors	max-rel-err $< 10^{-4}\%$	max-rel-err $> 100\%$	max-rel-err
FFT	No App-Specific Detectors	rel-l2-norm $< 10^{-4}\%$	rel-l2-norm $> 100\%$	rel-l2-norm
Sobel	No App-Specific Detectors	mean-pixel-diff $< 10^{-4}\%$	mean-pixel-diff $> 100\%$	mean-pixel-diff
Common to all apps	F_i = NaN F_i = Inf #F != #G	Errors in non-significant portions of the output		

Table 3.5: Quality metrics and quality thresholds used for different applications.

Table 3.4 provides a brief description of the applications studied.

Table 3.5 details the quality metrics and quality threshold ranges used (per application) in evaluating the AEA tools[7]. In the absence of specific domain studies and standardization [87, 120, 121], 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, one must first determine a difference function for each component and then a method to aggregate across the compo-

---

the two versions.

[7] Determining the quality metrics and quality thresholds to use involved a multi-month effort where we consulted different domain experts to understand how they use use the application and how much quality loss they can tolerate

nents. 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 (average can hide very large errors in individual components when the number of components is large). 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 output quality[8]. In the case of Sobel (which produces an image output), we measure the pixel difference between images [4]. More precisely, given a golden output G and a faulty (erroneous) output F, both having n components, where $n \geq 1$, Table 3.5 uses the following quality metrics.

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

$$\textit{max-abs-diff} = max(|G_1 - F_1|, |G_2 - F_2|, \ldots, |G_n - F_n|) \tag{3.3}$$

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

$$rel\_err_i = \frac{|G_i - F_i|}{G_i} * 100 \tag{3.4}$$

$$\textit{max-rel-err} = max(rel\_err_1, rel\_err_2, \ldots, rel\_err_n) \tag{3.5}$$

(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,

$$\|A\|_{BL2N} = \frac{\|A\|_{L2}}{n} \tag{3.6}$$

where,

$$\|A\|_{L2} = \sqrt{\sum_{i=1}^{n} a_i^2} \tag{3.7}$$

The rel-l2-norm is thus defined as:

$$\textit{rel-l2-norm} = \frac{\|G - F\|_{BL2N}}{\|G\|_{BL2N}} * 100 \tag{3.8}$$

---

[8] 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.

(4) *mean-pixel-diff*: This metric is used for image outputs and calculates the mean of the differences between individual pixels within the images (normalized as percentage).

$$\textit{mean-pixel-diff} = \frac{|G_i - F_i|}{255 * n} * 100 \tag{3.9}$$

Table 3.5 also lists the quality threshold ranges for identifying SDC-Good and SDC-Bad (Section 3.3.4). We use fairly conservative values that we believe will be reasonable for most users and applications. As mentioned in Section 3.3.4, the AEA tools uses detectors to catch DDCs. These detectors include range violations, 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 (components) as the golden output, irrespective of deviation. The DDC detectors used for different applications are detailed in Table3.5.

3.3.6 Output of Error Analysis

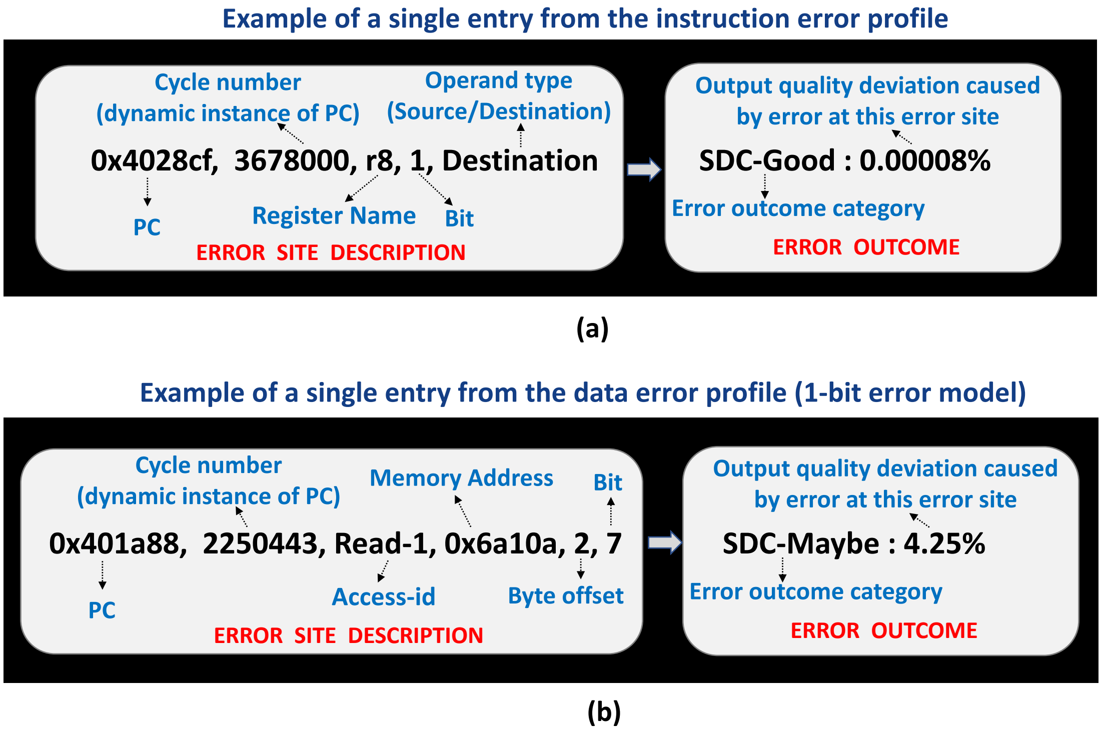

Example of a single entry from the instruction error profile
Cycle number
(dynamic instance of PC)
Operand type
(Source/Destination)
0x4028cf, 3678000, r8, 1, Destination
PC
Register Name
Bit
ERROR SITE DESCRIPTION
Output quality deviation caused
by error at this error site
SDC-Good : 0.00008%
Error outcome category
ERROR OUTCOME
(a)
Example of a single entry from the data error profile (1-bit error model)
Cycle number
(dynamic instance of PC)
Memory Address
Bit
0x401a88, 2250443, Read-1, 0x6a10a, 2, 7
PC
Access-id
Byte offset
ERROR SITE DESCRIPTION
Output quality deviation caused
by error at this error site
SDC-Maybe : 4.25%
Error outcome category
ERROR OUTCOME
(b)

Figure 3.4: A single example entry, describing a single error site and its corresponding error outcome, is shown for (a) instruction error profile (error model - one bit transient error in dynamic instruction register) and (b) data error profile (error model - one bit transient error in data bit accessed by a dynamic instruction).

The output of the error analysis (for a given application) is a comprehensive *error profile* of the application. The error profile contains a detailed list of the error outcomes for virtually all errors (for a given error model) in the program's execution. Each entry in the error profile contains two pieces of information, (1) the description of an error site in the program and (2) the error outcome (Section 3.3.4) of an error at the described error site.

Approxilyzer and gem5-Approxilyzer use an error model of transient errors in instructions (Section 3.3.2) and hence the error profile they output characterizes the behavior of errors in program instructions, and is called the *instruction error profile*. DataApproxilyzer, on the other hand, analyzes transient errors in program data (Section 3.3.2) and accordingly its output error profile is called the *data error profile*.

Figure 3.4 shows a single example entry, each from the instruction and data error profiles. The instruction error profile entry, for example, shows that an error, in bit 1 of register 8 in the dynamic instance of instruction 0x4028cf (this is the static PC) at cycle number 3678000, will lead to an output with a quality degradation of 0.00008%, which, using the common-sense thresholds supplied by the tool, is an error outcome categorized as SDC-Good. The entry for the data error profile in Figure 3.4 shows an example error model of a 1-bit error in data that is stored in system memory (DRAM in this case). Hence, the error-site describes the particular data bit at a specified location (address) in memory that was corrupted when it was read/written by the given dynamic instruction. Note that the examples shown in Figure 3.4 show just a single entry each from the error profiles. In reality, the error profiles can have millions (or even billions – depending on the size of the application) of such entries corresponding to different individual error sites in the application.

3.3.7 Use-case Aware Error Outcome Categorization

Error-efficient computing environments often trade accuracy in the program output for gains in other system parameters such as energy or performance. Two error-efficient techniques are discussed in this work – (1) *approximate computing* which deliberately introduces errors in computation for improved performance or energy, and (2) *ultra-low cost hardware resiliency* which allows some unintentional hardware errors to escape as user-tolerable output corruptions (rather than incurring high overheads to prevent all errors).

The application error profile generated by the AEA tools, which quantify the output quality of each error site in the program, can be used to understand the trade-offs between loss in output accuracy with respect to other system benefits for different error-efficiency schemes. In this section, we explain how, based on the error outcomes reported in the error profiles, the AEA tools can identify which error sites in the application need protection from

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	✗	✗

Table 3.6: Error outcomes and their potential for approximation and resiliency overhead savings.

transient errors (low cost resiliency), or alternatively, which error sites could be approximable (approximate computing). we will demonstrate how this information can be used to enable different error-efficiency techniques in Chapter 4.

Section 3.3.4 describes the various categories of error outcomes. The knowledge of each error site's output quality can be used to further categorize and equalize error sites according to the error-efficiency context in which the error analysis (and hence the application's error profile) is used. The error outcome equalization for two error-efficiency use-cases are described below (summarized in Table3.6).

(1) Low-Cost Resiliency: Each error site's output quality is used to decide whether that error site needs protection from transient errors (Table 3.6). Error sites that result in Masked outcomes do not need to be protected since they produce the golden (correct) output even in the presence of transient errors. Low cost detectors (as discussed earlier in Section 3.3.4) 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 AEA tools, we would have to protect all OC error sites. With the quality information provided by the AEA tools, 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.

(2) Approximate Computing: Table 3.6 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 are considered as not approximable. SDC-Good and Masked error sites are considered approximable. SDC-Maybe error sites are potential candidates for approximation depending on whether their quality meets the

acceptable quality threshold set by the user.

3.3.8 Error Pruning

The comprehensive error profile generated by the AEA tools contain the error outcomes for millions (or even billions) of error sites present in typical programs. In order to accurately analyze such a large number of error sites, the AEA tools employ error site pruning (or simply *error pruning*) techniques (pioneered by Relyzer [33]) to reduce the number of error sites needing detailed study (by error injection), either by predicting their outcomes or showing them equivalent to other errors.

Compared to a naïve campaign that performs an error injection for every error site, the AEA tools dramatically reduce (by many orders of magnitude) the number of error injections required to predict the error outcome for all error sites. They use a hybrid technique of program analysis and error injections to perform this comprehensive analysis with high accuracy while performing relatively few error injections.

The AEA tools build on key insights gained by prior work [33] that errors propagating through "similar" control and data flow paths in the program result in similar outcomes. Program analysis (both static and dynamic) and some heuristics are used to determine this similarity and group resulting error sites predicted to have similar outcomes in an *equivalence class*. Using an error injection experiment on a single representative from an equivalence class (called the *pilot*), they predict that all members of the class will have the same error outcome (as the pilot). Hence, these tools are able to predict the error outcomes of virtually all the error sites in the application, with high accuracy, using relatively few error injection experiments (in the pilots). Dynamic instances (or data accesses) of (from) the same static instruction are searched to find equivalent-outcome instructions (or data).

We do not claim that the error pruning techniques and heuristics used in this work are the fastest or the most accurate. However, as we will show later in this chapter, the techniques used hit the sweet spot of very high accuracy ($> 95\%$) with very high error pruning (up to 5 orders of magnitude reduction in errors). Exploring improvements to our error pruning technique is part of our future work.

The rest of this section describes the error pruning techniques developed by the prior work Relyzer. Specific error pruning techniques employed by the individual AEA tools will be described later in this Chapter.

Background: Relyzer's Error Pruning Techniques

Prior work called Relyzer [33] was the first to employ the novel error-pruning techniques described in this section. Relyzer used the error model of transient errors in (integer) registers of dynamic instructions. Relyzer systematically analyzes all application error sites (according to its error model) and carefully selects a small subset for thorough error injection experiments such that it can still estimate the outcomes of all the errors in the application. To achieve this goal, Relyzer applies a set of pruning techniques that are classified as *known-outcome* and *equivalence-based* pruning techniques. The known-outcome techniques largely use static (and some dynamic) program analyses to predict the outcome of an error. The equivalence-based techniques prune errors by showing them equivalent to others using static and dynamic analyses and/or heuristics. This section briefly describes these techniques; detailed explanations and examples can be found in [33].

The first step of error analysis (for Relyzer as well as the AEA tools from this work) is to enumerate all the errors that can impact the application. This requires an error-free execution trace for a given application (and input). Each dynamic instruction instance in the trace forms a potential application error site. Relyzer studies errors in architectural integer registers and in output latches of address generation units (this is equivalent to errors in operand registers of load and store instructions that hold the memory address).

While enumerating the list of all application error sites, Relyzer stores all the error site related information such as static instruction (program counter), error sites within the instruction (e.g., names of registers), number of dynamic instances of the instruction etc. Error pruning techniques are then applied on this initial set of errors. The rest of the section enumerates the various error pruning techniques applied.

Known-outcome pruning technique

Bounding addresses: Transient errors can make applications access memory locations that fall out of the range of the allocated address space. Such accesses are likely to result in detectable error outcomes (e.g., fatal traps, segmentation faults, application aborts, and kernel panic). Hence, there is no need for injection experiments to identify the outcome of most such errors and these can be directly pruned as follows. The range of valid addresses are determined, for both the stack and the heap, by studying the dynamic memory profile of the application.

Once the range of the valid addresses are identified, the errors that would allow a memory instruction to access an invalid address are pruned (e.g., errors in high order bits of the address when the error-free trace shows valid addresses are within lower order bits). The pruned errors (error sites) are apriori determined to produce detected error outcomes. This technique is applicable to memory instructions (both loads and stores).

Equivalence-based pruning techniques

The equivalence-based class of pruning techniques eliminates errors that are equivalent to each other from the initial set of errors and retains only the representative errors (pilots) for thorough error injection experiments. The pruning techniques are further categorized as as *precise* and *heuristics-based*, based on whether they use accurate analyses or heuristics to form the equivalence classes.

Precise equivalence technique

Def-use analysis: A register definition is created whenever a register is used as a destination operand in an instruction. Errors in the definition of a register have similar behavior to that of errors in the first use of this definition. Therefore, errors in the definition are pruned out and errors in the first use are retained. Note that this technique prunes errors only in the definition and not in the uses. There can be multiple uses of a definition, and errors in different uses may have different error propagation. Whenever a definition is pruned, the information of the first use is recorded. This allows relating the error outcomes of the first use to the definition's at a later stage. Ideally, the destination register operands of all the instructions can be pruned by this technique. However Relyzer only prunes errors in those destination registers that have a first use within the same basic-block. This was done to keep the implementation simple yet precise; Relyzer uses a static program pass for to identify def-use pairs and the presence of conditional moves makes the precise association of a definition with its first use non-trivial.

Heuristics-based equivalence techniques

Control-equivalence: This heuristic pruning technique uses the observation that errors propagating through similar code sequences are likely to behave similarly. It also uses the observation that a majority of the errors appear in code sequences that are executed many times. Consider a static instruction I with many dynamic instances in the error-free execution under consideration. The pruning technique attempts to partition all these dynamic instances of I into equivalence classes, based on the control flow path followed after the dynamic instance.

It is convenient to describe and implement the algorithm at the basic block level. The technique uses the error-free application execution to enumerate all possible control flow paths up to a depth n starting at the basic block that contains the instruction of interest. Depth is defined as the number of branch or jump instructions encountered. For the paths that were exercised multiple times in the execution, it randomly selects one dynamic occurrence, a pilot. It prunes all other unselected executions of such paths (population) and assumes that errors in those dynamic executions are represented by the selected ones (pilots). More precisely, a dynamic instruction instance on a pilot path serves as a pilot for other instances with the same PC on the other paths in its population.

Figure 3.5 (from [33]) explains through an example how this pruning technique selects pilots. The figure presents a control flow graph of a small program, with the basic blocks represented by the black and grey circles with numbers on their sides. Assume the grey basic block is not exercised by the dynamic execution of interest. Assume $n = 5$ (depth until which control flow is tracked). Suppose that the objective is to find representative pilots for an instruction in basic block 1. All control flow paths starting at basic block 1 up to a depth of 5 that are executed in the dynamic error-free execution of interest are enumerated. Basic block 4 is never executed and hence it does not appear in the list of dynamically exercised paths. Each path is identified as forming a new equivalence class. There will be potentially many instances of such paths in the dynamic execution trace. One dynamic execution sequence is randomly identified for each equivalence class and named as the pilot for that class. As mentioned before, a dynamic instruction instance on a pilot path serves as a pilot for other instances with the same PC on the other paths in its population.

Relyzer applies this technique to prune errors in all instructions other than stores and those that affect stores within a basic block. This is because the propagation of an error in a store also depends on the addresses of the loads in the control flow path taken (only loads to the same address as the store will propagate the error). The next technique described deals with this distinction. Exceptions to the above are a few SPARC specific instructions; namely, *save*, *restore*, *call*, *return*, and *read state register*. Relyzer does not not inject errors in these instructions and therefore does not consider them any further. Relyzer also does not inject errors in dead instructions and does not consider those any further either.

Overall, control-equivalence has the potential of pruning a large fraction of the errors by softening the constraint on evaluating all dynamic occurrences from a specific code section.

Store-equivalence: An error in a store instruction propagates through the loads that read the erroneous values. Load addresses are not entirely captured by the control flow path taken after the store. Therefore, an alternate heuristic, called store-equivalence, is used for errors in store instructions or in instructions that a store depends on within the same basic block. This heuristic captures the error propagation behavior by observing the addresses

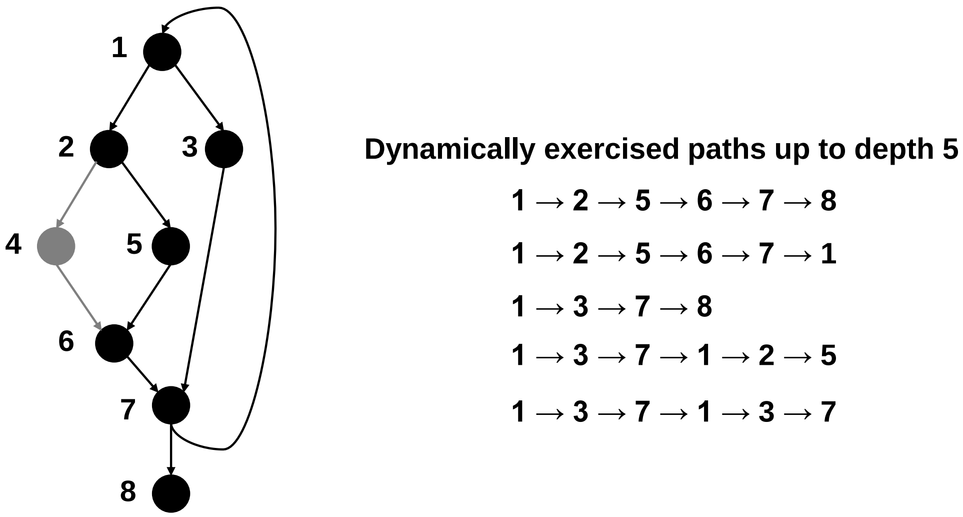


Figure 3.5: Control-equivalence example from [33]. The figure shows a CFG for a small program starting at basic block 1 and ending at basic block 8. All dynamically exercised control paths up to a depth, say 5, are enumerated. Here basic block 4 (showed in grey) never gets exercised. Therefore control flow paths through this node do not appear on the list of dynamically exercised paths. The executions along each of these paths form the equivalence classes for similar error outcomes.

that a store writes in an error-free execution and recording all read accesses to this address. It treats the errors in stores differently whenever a different permutation of load instructions read the stored value.

Figure 3.6 (from [33]) illustrates this heuristic with an example. Consider Store 1 and Store 2 as two dynamic store instruction instances from the same static instruction. To determine if the errors in these two store instructions will have the same outcomes, Relyzer examines all the loads that return the values written by these stores in the error-free execution, i.e., Load 1a and Load 1b for Store 1 and Load 2a and Load 2b for Store 2 from the figure. It first checks whether the number of such loads is the same (two for each store in the figure). If this is the case, it then checks whether the static instructions (program counters) of the corresponding loads are the same; e.g., if the program counters of Load L1a and Load L2a are the same and if those of Load L1b and Load L2b are the same in the figure. If these match, then Relyzer concludes that the two dynamic store instructions are very likely to have similar error outcomes and places them both in the same equivalence class.

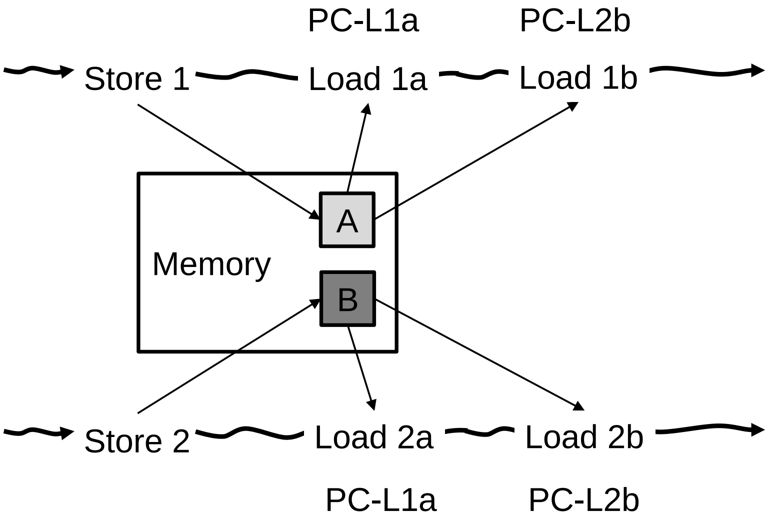


Figure 3.6: Store-equivalence example from [33]. Store 1 and Store 2 are two store instructions from the same static instruction writing to addresses A and B respectively. Load 1a with program counter PC-L1a and Load 1b with program counter PC-L1b are two load instructions reading the value from address A. Similarly, Load 2a and Load 2b are two loads from address B with program counters PC-L2a and PC-L2b respectively. The store-equivalence heuristic requires that PC-L1a equal PC-L2a and PC-L1b equal PC-L2b.

3.3.9 Validation of Error Analysis

The AEA tools use similar error pruning techniques as those described in Section 3.3.8. Since heuristics-based equivalence techniques rely on heuristics to predict the outcome of error sites, validation experiments are needed to measure the accuracy of these techniques, and hence, the accuracy of the tool in predicting error outcomes. Note that known-outcome and precise equivalence techniques use precise a priori knowledge, to determine error outcome and show equivalence respectively, and hence do not need validation (and will not be discussed further). In this section, the validation experiments undertaken to show the accuracy and precision of the AEA tools are described.

1. **Baseline Validation**:

   The heuristic-based equivalence techniques rely on heuristics to group error sites that produce similar quality outcomes into an equivalence class. They predict the quality of each member of an equivalence class based on the outcome of an error injection experiment on its pilot (Section 3.3.8). Similar to the methodology used in prior work [33], the validity of the equivalence techniques is measured by the extent to which the they correctly group error sites (with the same outcome) into equivalence classes.

Specifically, the validation attempts to answer the following question: how accurately does the error outcome of the pilot predict the error outcome of the other error sites in its equivalence class? For validating a single pilot, error injections are performed in a sample of error sites (not including the pilot) – called the *population* – chosen randomly from the pilot's equivalence class. The error outcome of the population is then compared 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.

To validate a pilot of an SDC-Maybe class, it is further required that the quality degradation (refered to a QD) 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 quality degradation of 12% (written as QD-12 for brevity). Suppose 86% of its population is SDC-Maybe with QD-12, 6% is SDC-Maybe with QD-13, 5% is SDC-Maybe with QD-10, and 3% is SDC-Bad. Then the prediction accuracy of pilot X is 86%.

The baseline validation described above is general and measures the accuracy of the techniques independent of any error-efficiency use-case or context (discussed further below). Hence, this type of evaluation is referred to as *CF* (context-free) validation.

2. **Flexible Quality Window:**

   Quality is a continuous parameter and requiring the pilot's QD to exactly match the QD of the associated population is unnecessarily conservative and a tall order for any tool. We therefore introduce a flexibility parameter, δ, that allows a fine-grained margin of error at QD boundaries. For the validation of pilot X described above, setting $\delta = x$ means that an error site in its population with QD 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%.

   Note that Masked, SDC-Good and SDC-Bad error-sites also have quality information associated with them. Masked outcomes have a $QD = 0$ (no quality degradation), SDC-Good outcomes have $QD < Th_1$ and SDC-Bad outcomes have $QD > Th_2$, where Th_1 and Th_2 are common-sense application specific thresholds set by the AEA tools (as described in Section 3.3.5 and Table 3.5). Thus, the δ parameter can also allow for fine-grained error margins in quality at the boundaries of these categories. For example, given a pilot SDC-Maybe with QD-1, setting $\delta = 2$ will result in all Masked outcomes in the associated population to be counted as correct prediction.

3. **Equalizing Error Outcomes for Error-Efficiency Use-Case:**

   Since the output of the AEA tools can be used to enable different error-efficiency techniques, it is interesting to extend the baseline validation to include the context in which the error analysis is used. As mentioned in Section 3.3.7, we study low-cost resiliency and approximate computing as two representative error-efficiency use-cases in this work and we evaluate the AEA tools for both these contexts.

   Validation for Resiliency: In this first case, the AEA tools are used to determine which error sites in the application need to be protected for resiliency (Section 3.3.7). There is no need to distinguish between Masked, SDC-Good, DDC, and Detected outcomes since all of them do not require protection. Therefore these error outcomes can be grouped together.

   Validation for Approximation: In the second case, the AEA tools are used to determine which error sites are approximable (Section 3.3.7). Therefore, there is no need to distinguish between Masked and SDC-Good outcomes since they are approximable, and they can be grouped together. Similarly, SDC-Bad, DDC, and Detected outcomes can be grouped 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 henceforth use *Res* and *Approx* when considering use-specific validations for resiliency and approximation respectively.

   To illustrate the above with an example, consider an equivalence class whose pilot Y generates a DDC. Suppose 86% of its population is DDC, 6% is Masked, 5% is SDC-Maybe with QD-2, and 3% is Detected. Then the prediction accuracy of pilot Y for *CF* is 86%, for *Res* is 95% and for *Approx* is 89%.

   In the presence of user specified output quality thresholds (referred to as QT), SDC-Maybe error sites can be equalized based on if their QD falls above or below the QT. For instance, if the QT provided is 5% (i.e., the user is willing to tolerate a QD of up to 5%), then the SDC-Maybe error sites with QD-2 in the example above are classified as being approximable and not needing resiliency protection (for the approximation and resiliency use-case respectively). Thus the prediction accuracy for pilot Y, from the example above, is refined to 100% for *Res* (*Approx* validation accuracy remains at 89%).

To illustrate all of the above concepts with a combined example, consider a pilot Z that generates an SDC-Maybe with QD-6. Suppose the error outcomes of its population are as follows: (a) 84% of its population is SDC-Maybe with QD-6, (b) 6% is SDC-Maybe with QD-3, (c) 5% is SDC-Maybe with QD-8, (d) 3% is Masked, and (e) 2% is DDC. Then the validation accuracy of Z for the different validation strategies is as follows:(1) $CF_{\delta=0}$ (Context free and no flexible quality) is 84% (a is correct), (2) $CF_{\delta=2}$ is 89%=84%+5% (a and c are correct), (3) $Res_{\delta=2}$ is 89%=84%+5% (a and c are correct), (4) $Approx_{\delta=2}$ is 89%=84%+5% (a and c are correct), (5) $Res_{\delta=2,QT=7}$ is 100% (all are correct, even c, which lies above the QT, because its QD falls within the $\delta = 2$ margin of the pilot), and (6) $Approx_{\delta=2,QT=7}$ is 98% (e is incorrect).

The overall validation 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. Validation accuracy for individual equivalence-based pruning techniques (for example, control-equivalence, store-equivalence etc.) can be obtained by studying the accuracy of pilots from equivalence classes built using the individual techniques.

To gain confidence in the AEA tools, it is imperative that the accuracy of a sufficient number of equivalence classes (and their pilots) are evaluated using the validation methodology described in this section. To evaluate each AEA tool, we perform the validation experiments for ~750 pilots from each application. This corresponds to a 99% confidence interval with a 5% error margin. Each pilot is validated against a sample population size of 750 (drawn randomly from the equivalence class), which also corresponds to a statistical confidence of 99% with a 5% error margin. In all, I perform approximately 7.6 million error injection experiments – 2.6 million for Approxilyzer, 1.6 million for gem5-Approxilyzer and 3.4 million for DataApproxilyzer – to validate the AEA tools.

The AEA tool-suite described in this work has been developed over many years and hence, there are minor methodological differences in the evaluation of individual tools (inspired by collaborator and reviewer feedback). These include variations in benchmarks, inputs, (minor variations in) validation methodology etc. For example, Approxilyzer discretizes output quality into multiple Quality Bins (QB), fine-grained enough to capture small quality variations. For example, with the max-rel-err metric, the bins are 1% wide, and quality degradation values of 12.1%, 12.6% and 13.3% are assigned a QB of 13, 13, and 14 respectively. And thus the QD value stored with the SDC-Maybe outcomes correspond to the QB values. The various quality bins associated with different applications and metrics are detailed in [24]

(Table II).

DataApproxilyzer's validation is slightly modified to evaluate "purity" of the equivalence classes (analogous to validation in clustering techniques) which identifies the largest set of similar error outcomes within the population [26]. Hence, the accuracy of a given equivalence class is based on the extent to which it contains error outcomes of a single category (instead of comparing it to the pilot). This eliminates effects (over- or under- estimation) introduced by the randomness of picking pilots (which may or may not belong to the largest error outcome set within the equivalence class). For example, consider an equivalence class E whose population distribution is 97% Masked and 3% DDC. If its randomly chosen pilot belongs to the DDC set, then baseline validation described in this section will assign E and accuracy score of 3%. On the other hand if the pilot is Masked, then the accuracy of E is 97%. Using the purity metric, the accuracy of E is 97% (largest common outcome class) and is independent of the pilot. All the concepts of flexible quality window and use-case equalization still apply (each error-site is compared to the most common error outcome).

The methodological differences within the AEA tools are relatively minor and are not anticipated to affect the accuracy measurements significantly.

3.4 APPROXILYZER

In this section, we present Approxilyzer [24], an automated error analysis framework that determines the quality impact of errors in program instructions. The error model that Approxilyzer uses is single bit transient errors in operand registers of dynamic instructions (Section 3.3.2). To our knowledge, Approxilyzer is the first tool that quantifies the quality impact of a single-bit error in (operand registers of) virtually *all* dynamic instructions of an execution with high accuracy and low-cost. The output of Approxilyzer's error analysis is the comprehensive instruction error profile for a given application (Section 3.3.6).

The evaluation results presented in this section show that Approxilyzer can predict output quality, at very fine granularities, with high accuracy (95% on average) and at low-cost (up to 5 orders of magnitude fewer error injections than naïve techniques).

3.4.1 Error Pruning

To build the application's comprehensive error profile, Approxilyzer employs a hybrid error analysis technique which uses a combination of program analysis and few error injections (Section 3.2). Approxilyzer builds upon Relyzer [33], a tool that predict the outcomes of errors in integer registers of dynamic instructions. Relyzer's predictions, however, only consider *whether* an error will affect the output but does not quantify *how* the output will be affected. Relyzer determines if an error results in being Masked, Detected, or a Silent Data Corruption (SDC). Relyzer does not consider output quality, marking all corruptions in an output (no matter how trivial) as an SDC outcome.

Approxilyzer refines the notion of an error outcome by including the quality of the erroneous output as part of this outcome (Section 3.3.4). 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 (Section 3.3.3).

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 error pruning techniques and heuristics – bounding addresses, def-use, control-equivalence (control depth is set to N=50, as in prior work [30]) and store-equivalence (described in Section 3.3.8) – to predict similarity and to divide error sites into equivalence classes. Validation experiments (as described in Section 3.3.9) are performed to test this hypothesis and we show that this is indeed the case (Section 3.4.4). Additionally, while Relyzer only studied errors in a integer registers (which in turn only allowed it to study errors in a small subset of integer instructions), Approxilyzer's analysis extends to errors in both integer and floating point registers (thereby enabling analysis of errors in virtually all instructions in the program).

Thus, Approxilyzer is able to enumerate, with high confidence, the output quality generated for a given error in the program. And it is able to do this for virtually all errors in the program instructions with relatively few error injections – 3 to 5 orders of magnitude fewer error injections when compared to naïve techniques.

3.4.2 Workloads and Inputs

To evaluate Approxilyzer, we select five benchmarks from two different benchmark suites spanning multiple application domains (application descriptions can be found in Section 3.3.5). Table 3.7 lists the workloads and inputs used to evaluate Approxilyzer. The quality metrics and quality thresholds (common-sense thresholds provided by the tool) used for each of the applications is detailed in Section 3.3.5.

Application	Input
Blackscholes	sim-large
Swaptions	sim-small
LU	512x512 matrix 16x16 blocks
FFT	64K points
Water	512 Molecules

Table 3.7: Workloads and inputs.

3.4.3 Error Injection Framework and Speed

The error injection simulation infrastructure is similar to that used for [33], based on Wind River Simics [110] and GEMS [123] running the 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 any of the error-efficiency techniques demonstrated in this work.

Although error injections are performed 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, thereby trading off simulation time for a modest loss in coverage [33]. The 1% of error sites not included in the study do not detract from the observations and gains reported.

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 (and hence, Approxilyzer) is able to prune error injections by 3 to 5 orders of magnitude [33]. The remaining error injections complete on a cluster of 200 machines in a few days. Approxilyzer adds a few hours to this process to perform quality calculations and error outcome categorizations.

3.4.4 Approxilyzer Validation

We carry out validation experiments as described in Section 3.3.9. We discuss the validation accuracy of Approxilyzer for the approximation and resiliency contexts separately below.

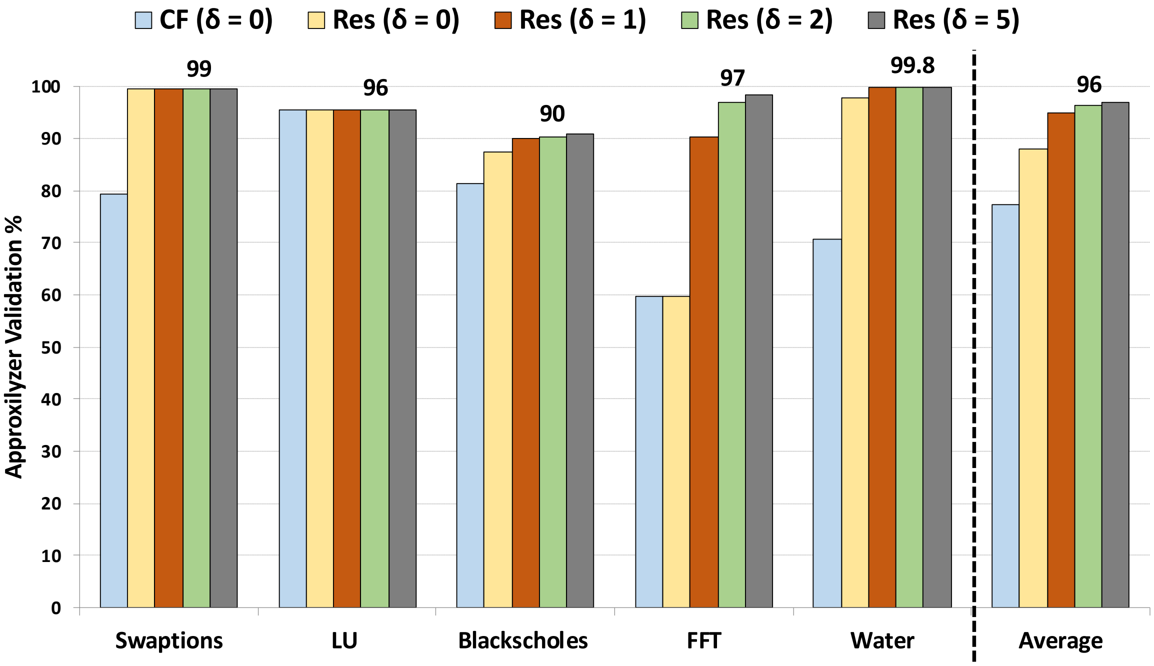


Figure 3.7: Approxilyzer validation geared towards resiliency.

Validation for Resiliency: The results for validation geared towards resiliency are shown in Fig. 3.7. 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 (bar corresponding to $Res(\delta = 2)$).

Swaptions and Water see big gains in validation just by applying the Res optimization (Note that $Res(\delta = 0)$ equals just applying Res without any flexible quality window). Both of these applications have high SDC-Good rates 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 $CF(\delta = 0)$.

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 Res optimization causes the validation rate of Water to jump from 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 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.

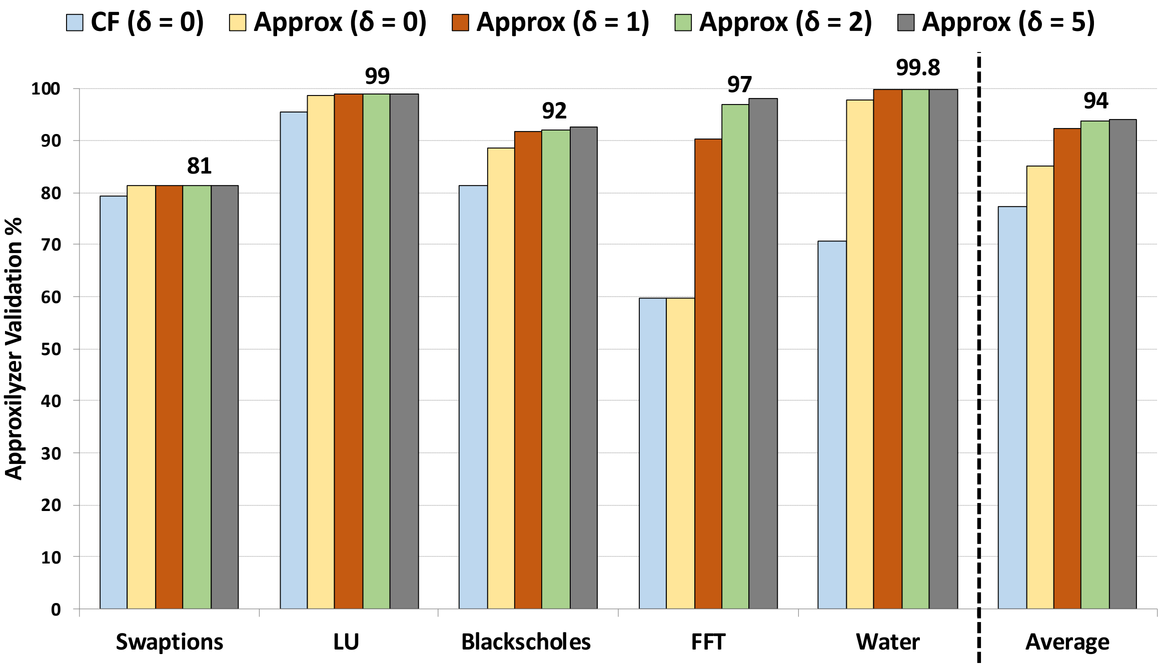


Figure 3.8: Approxilyzer validation geared towards approximation.

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 $CF(\delta = 0)$. Applying the Res optimization, causes the validation rate of Swaptions to go up from 79% to 99%.

While still high at 90% (for $Res(\delta = 2)$), 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 instructions produce OCs of different quality based on the input being processed at any given execution cycle. While range detectors to capture certain SDC-Bad outcomes and specialized heuristics to better capture variations in data patterns can be applied to handle some of these cases, their implementation is left to future work. Applying user-specified quality thresholds can also equalize some of the SDC-Maybe and SDC-Bad outcomes and improve the accuracy. In spite of these special cases, Blackscholes shows high prediction rate.

Validation for Approximation: Fig. 3.8 shows the graph for Validation considering Approximation. On average, the validation percentage for $Approx(\delta = 2)$ 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 $Approx(\delta = 2)$ is 97%) operating on random numbers. While the *Res* optimization equalized these outcomes, the *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 $Approx(\delta = 2)$ and $Res(\delta = 2)$, for the applications studied is 95%. *Thus, we conclude that Approxilyzer can capture the output quality – at very fine granularities – with high accuracy for the purposes of both Resiliency and Approximate Computing.*

3.5 GEM5-APPROXILYZER

Approxilyzer has significantly furthered the state-of-the-art in automated error analysis, by providing the precise impact (execution anomalies and output quality) of single-bit transient errors on *every* operand register bit in *virtually every* dynamic instruction in a program execution, with high accuracy and low-cost. Furthermore, Approxilyzer can analyze general-purpose applications while placing minimal burden on the programmer.

Approxilyzer's unique features enable new avenues of research, but limitations in its original implementation hinder its usability. Approxilyzer relies on Wind River Simics [110], a proprietary full-system simulator and is designed to handle only applications compiled for the SPARC ISA. The restrictions imposed from both the simulator and ISA make a wide adoption of the tool challenging.

To mitigate these restrictions, we develop gem5-Approxilyzer [25], a fully open-source [28] implementation of Approxilyzer that enables support for more ISAs, beginning with x86 in this work. gem5-Approxilyzer is built using the open-source gem5 simulator [83] 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. For example, Approxilyzer's original implementation for SPARC assumes constant register size and instruction encoding length, which is not the case for x86.

This work is the first to show Approxilyzer's effectiveness/accuracy with a different ISA, namely x86. Since gem5-Approxilyzer analyses errors in applications compiled to x86 (and Approxilyzer techniques were evaluated for SPARC), validation experiments are performed to show the accuracy and effectiveness of Approxilyzer's error pruning techniques (Section 3.4.1) for x86. We show that gem5-Approxilyzer is effective in error pruning and reduces the number of error injections required by up to two orders of magnitude over a naïve campaign[9]. We also show that gem5-Approxilyzer predicts the impact of errors on the program's output quality with high accuracy ($> 92\%$ on average and up to 99.9% for some applications). Hence, gem5-Approxilyzer produces accurate and precise instruction error profiles at relatively low-cost.

The remainder of this section details the implementation challenges of building gem5-Approxilyzer and demonstrate the effectiveness and accuracy of gem5-Approxilyzer.

3.5.1 Error Model

gem5-Approxilyzer uses the same error-model as Approxilyzer – single-bit transient errors in operand registers of dynamic instructions in the program (Section 3.3.2). However, because gem5-Approxilyzer analyzes applications compiled to CISC x86 (vs. Approxilyzer that analyzes RISC SPARC) there are additional considerations which are detailed below.

In this work, 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 the analysis to macro-instructions.

For this study, we only consider general-purpose registers and SSE[10] registers in x86. We do not inject errors in special-purpose, status, and control registers (e.g., %rsp, %rbp, rflags) to simplify the 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[11] registers in this work. Extending gem5-Approxilyzer to support these registers is relatively straightforward and we leave it to future work.

---

[9] The gains from error-pruning are lower in this work (compared to Approxilyzer) since we use minimized program inputs that have been shown to considerably speedup error analysis [29]. Since these smaller inputs lead to programs with less dynamic instructions (per static instruction), the potential for error pruning is considerably reduced. Yet, even for these smaller programs, considerable error pruning is achieved. The gains from larger inputs are expected to be commensurate with Approxilyzer.

[10] 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.

[11] 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.

3.5.2 Implementation Details

The inputs required by gem5-Approxilyzer are detailed in Section 3.3.3. 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 instruction error profile (Section 3.3.6).

(1) 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. gem5-Approxilyzer further simplifies the trace to only contain the execution within the ROI (if an ROI is provided).

(2) Phase 2 prunes error sites using the same techniques as Approxilyzer – 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.

(3) Phase 3 performs the error-injection experiments using the *error injector* module built for gem5. The error injector takes as input 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. If there is an OC, gem5-Approxilyzer uses the user-provided quality metric to evaluate the output quality.

(4) 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 instruction error profile containing all the error sites studied 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.

3.5.3 x86 Implementation Challenges

While phases 3 and 4 (described in the previous section) are largely ISA independent, phases 1 and 2 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.

3.5.4 Extensions to gem5-Approxilyzer

gem5-Approxilyzer has been designed to be reasonably modular (e.g., each phase in Section 3.5.2 is a separate module) to enable future extensions to support different ISAs, error models, and pruning techniques. This section 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. 2) ISA-specific behaviors that affect control flow (e.g., branch delay slots for SPARC) should be incorporated into the control-equivalence algorithm accordingly. 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 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.

3.5.5 Workloads and Inputs

To evaluate gem5-Approxilyzer, five benchmarks are selected from three different benchmark suites spanning multiple application domains (application descriptions can be found in Section 3.3.5). Table 3.8 lists the workloads and inputs used in the evaluation. These inputs are chosen because we have shown (in Chapter 5) that performing error analysis on these inputs is much faster and at least as accurate as for larger reference inputs provided by benchmark suites [29]. We use gem5 to simulate an Ubuntu-16.04 system, and use GCC 7.3 with -O3 optimization to compile the applications. The quality metrics and quality thresholds (common-sense thresholds provided by the tool) used for each of the applications is detailed in Section 3.3.5.

3.5.6 Error Pruning Effectiveness

gem5-Approxilyzer employs the same error pruning techniques as Approxilyzer – control-equivalence (depth set to N=50) and store-equivalence as well as address-bound and def-use techniques. The address-bound and def-use techniques together are referred to as *non-heuristic (NH)* pruning techniques, since they do not employ any heuristics (Section 3.3.8).

The effectiveness of gem5-Approxilyzer is measured by observing how many error sites were pruned using the various pruning techniques. For each application, we measure first the number of error sites in the application's region of interest (column 3 in Table 3.8) and then the number of error sites remaining after the pruning (column 4 in Table 3.8)

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

Table 3.8: Benchmarks, inputs, and error-site pruning by technique (C: Control-Equivalence, S: Store-Equivalence, C+S+NH: total pruning using control, store, and non-heuristic techniques)

to calculate the number of error sites that have been pruned. 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.

The last column of Table 3.8 shows the percentage of error sites pruned by gem5-Approxilyzer using the control-equivalence (C), store-equivalence (S), and non-heuristic (NH) pruning techniques. At 56.77%, Blackscholes has the smallest total (C+S+NH) 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.

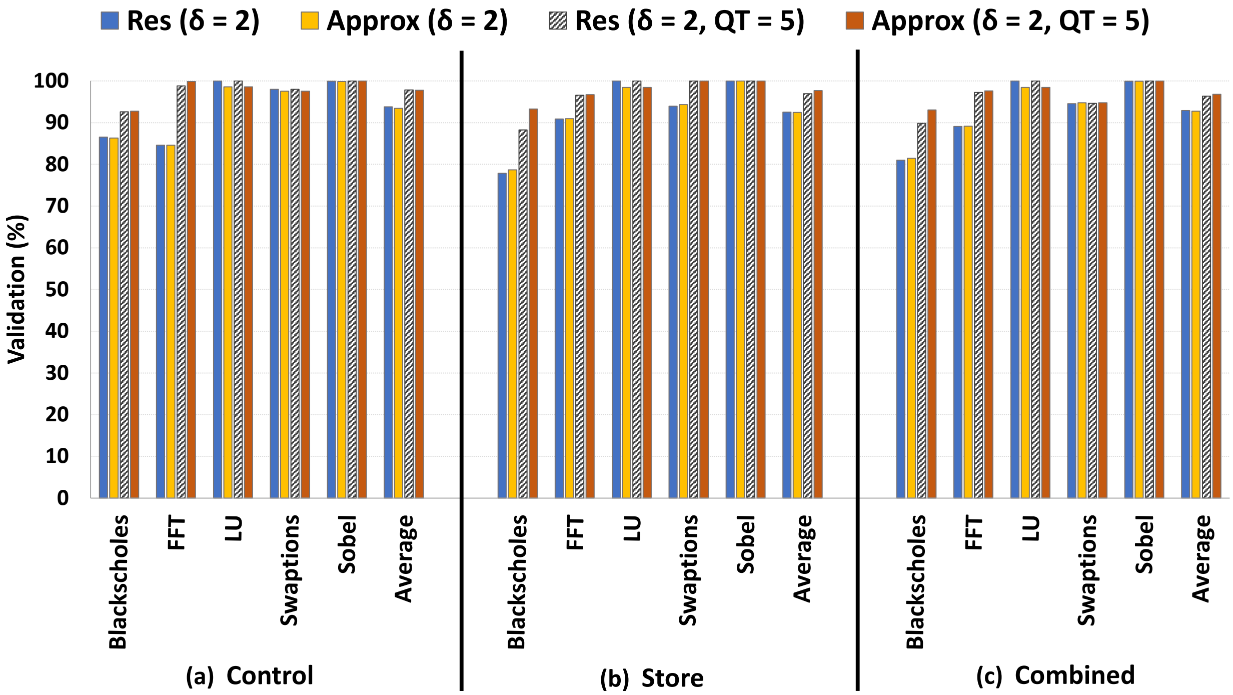


Figure 3.9: gem5-Approxilyzer validation for (a) control equivalence, (b) store equivalence, and (c) combined (control + store) equivalence.

3.5.7 gem5-Approxilyzer Validation

We evaluate the accuracy of gem5-Approxilyzer using validation experiments as described in Section 3.3.9. Figures 3.9(a), 3.9(b), and 3.9(c) show the validation accuracy for the 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 $\delta = 2$. Note that the flexible quality parameter $\delta = 2$ used here corresponds to 2% error margin for Blackscholes, FFT, LU, and Sobel, while for the financial applications, Blackscholes[12] and Swaptions, it corresponds to the absolute difference in the dollar value set to \$0.01 (difference of 1 cent or less).

In summary, *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

---

[12] Two different quality metrics are employed for Blackscholes (Table 3.5) and hence the flexibility parameter δ is applied to both metrics.

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.

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 observe that when a quality threshold is supplied, the validation accuracy is $> 97\%$ for both store and control heuristics (and hence their combination).

In summary, the results show that the techniques used by gem5-Approxilyzer are very accurate in precisely quantifying the impact of errors on output quality for x86 applications.

3.6 DATAAPPROXILYZER

In this section, we describe *DataApproxilyzer*, the first automated application-level error analysis tool to perform extensive analysis of errors in program data. DataApproxilyzer extensively analyzes many, and in some cases all, errors (for a given data error model) and determines the output quality produced (for each error) with high accuracy (98% on average) and at fine granularity (within a 2% error margin). The error models used by DataApproxilyzer are multi-bit (1-, 2-, 4- and 8-bit) transient errors in (data stored in) system memory (details of the error model can be found in Section 3.3.2). The output of DataApproxilyzer is the given application's *data error profile* that extensively lists the data errors (for a given data error model) that can impact program execution along with the corresponding output quality produced in the presence of each error (Section 3.3.6).

DataApproxilyzer builds on the hybrid error analysis techniques (Section 3.2) developed by Approxilyzer (and by extension gem5-Approxilyzer) and employs a combination of program analysis and (relatively) few error injections to generate the application's data error profile. DataApproxilyzer introduces a novel equivalence-based error pruning technique (Section 3.3.8) called *Load-Order-Equivalence* that prunes the number of data error sites that need detailed analysis through error injections by up to 99.9% (98% on average) for the workloads studied. The main insight offered by the Load-Order-Equivalence technique is that errors in data bits that are read (and therefore consumed and propagated) by the same sequence of load instructions in the program, produce outputs of similar quality. We undertake validation experiments (as described in Section 3.3.9) to show the high accuracy (98% on average) of the Load-Order equivalence technique (Section 3.6.5). DataApproxilyzer is the first to show that equivalence-based error pruning techniques can be applied to multi-bit data errors.

In this section, we elaborate on the the Load-Order-Equivalence technique used by DataApproxilyzer and show that DataApproxilyzer can predict error outcomes for data errors in the program with high accuracy and at low cost.

3.6.1 Error Pruning Technique

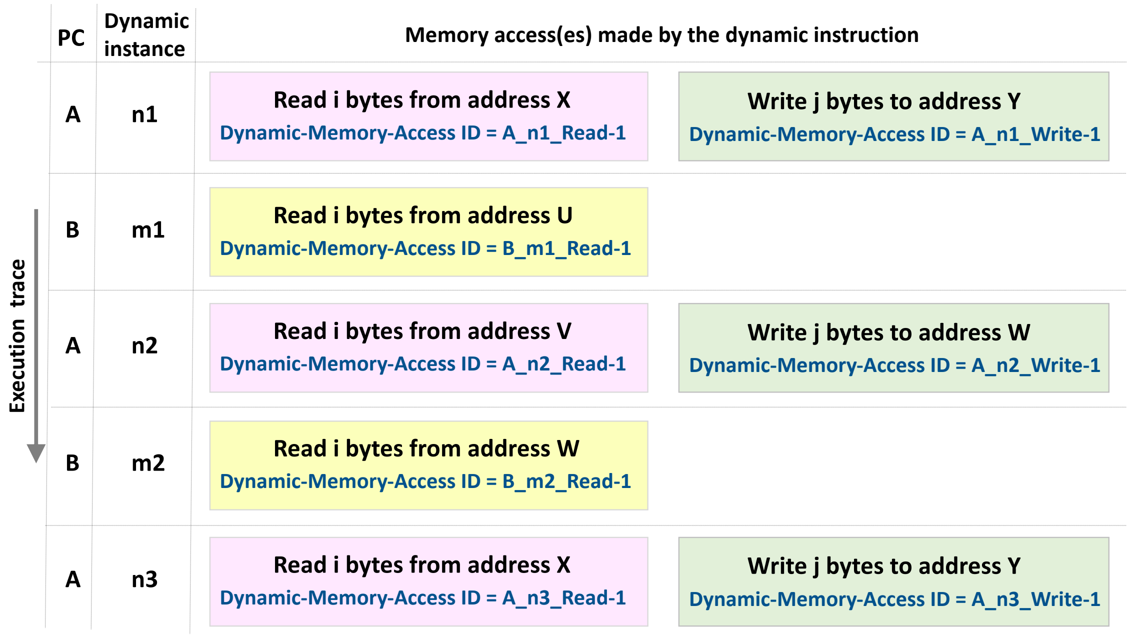
Like the other AEA tools (Approxilyzer and gem5-Approxilyzer) described in this work, DataApproxilyzer also uses error pruning techniques to identify just a small subset of pilot error sites which require error injections (Section 3.3.8). DataApproxilyzer uses a new equivalence based pruning technique called Load-Order equivalence, which uses a data-flow based heuristic (simply referred to as the *Load-Order* heuristic) to equalize "similar" data errors into equivalence classes. This heuristic captures how errors in program data are propagated and consumed by subsequent computation(s) in the execution. Specifically, the *Load-Order* heuristic captures the sequence of dynamic instructions in the program that read (and thus propagate) the corrupted data.

The Load-order heuristic improves upon the store-equivalence heuristic (used by Approxilyzer and gem5-Approxilyzer) that equalizes single-bit errors in operand registers of store instructions (Section 3.3.8). Single-bit errors in operand registers of store instructions constitute a small subset of memory (data) errors, when the store instruction writes the corrupted values in the register to system memory; The Load-Order technique used by DataApproxilyzer improves upon store-equivalence to make analysis of these more efficient. Additionally, DataApproxilyzer analyzes many more data errors in memory (both single- and multi-bit) that lie outside this subset of errors. Note that while some registers directly access data (such as destination operands of load instructions), errors in such registers is fundamentally different than errors affecting (the same data in) memory since the errors in memory will persist across multiple instructions that access this data. And hence, the analysis of such memory errors need a modified approach.

The remainder of this section details the methodology of the Load-Order equivalence technique.

Pre-processing for Error Equalization:

The first step DataApproxilyzer performs is to identify and label all the error sites in the application. To do this, DataApproxilyzer extracts the application's (error-free) dynamic memory trace and labels each memory access made by a dynamic instruction (referred to as a *Dynamic-Memory-Access*) with a unique identifier (called a *Dynamic-Memory-Access ID*). Each Dynamic-Memory-Access ID consists of (1) the instruction's static PC (Program Counter), (2) the instruction's dynamic instance, usually identified by some abstract sequence number in program order (such as the cycle number at fetch) and (3) a tuple, called the *micro-access-ID*, consisting of the type of memory access (read or write) and a number to differentiate between multiple Dynamic-Memory-Accesses of the same type within a given dynamic instance (common in CISC instructions). An example toy memory trace is shown in Figure 3.10 to illustrate this process of assigning Dynamic-Memory-Access IDs.



PC	Dynamic instance	Memory access(es) made by the dynamic instruction	
A	n1	Read i bytes from address X Dynamic-Memory-Access ID = A_n1_Read-1	Write j bytes to address Y Dynamic-Memory-Access ID = A_n1_Write-1
B	m1	Read i bytes from address U Dynamic-Memory-Access ID = B_m1_Read-1	
A	n2	Read i bytes from address V Dynamic-Memory-Access ID = A_n2_Read-1	Write j bytes to address W Dynamic-Memory-Access ID = A_n2_Write-1
B	m2	Read i bytes from address W Dynamic-Memory-Access ID = B_m2_Read-1	
A	n3	Read i bytes from address X Dynamic-Memory-Access ID = A_n3_Read-1	Write j bytes to address Y Dynamic-Memory-Access ID = A_n3_Write-1

Figure 3.10: Toy trace showing the terminology of Dynamic Memory Accesses and when they may be equalizable.

Each data bit accessed by a given Dynamic-Memory-Access is an individual error site. Hence, if Dynamic-Memory-Access D accesses data of size n (where n>=1) bytes starting at memory address X, then each of the n$*$8 data bits starting at X is a potential error site.

Each error site (individual data bit) is described/labeled by the Dynamic-Memory-Access ID, the memory address of the access and a byte offset and bit number.

Load-Order Equalization Technique:

In this section we describe the Load-Order equalization heuristic used in DataApproxilyzer. We evaluated a few different equalization heuristics for DataApproxilyzer (described in the next section) before picking the heuristic with the best performance and accuracy (reported here).

Error equalization is attempted only within Dynamic-Memory-Accesses that have the same PC and micro-access-ID. The goal of equalization is to identify error sites that are consumed (used) and propagated similarly through the program; data bits accessed from different PCs or micro-access-IDs are considered as being dissimilar for this purpose. Thus, error sites belonging to Dynamic-Memory-Accesses from different PCs or from the same PC but having different micro-access-IDs will never be in the same equivalence class. The Dynamic-Memory-Accesses in Figure 3.10 are colour coded to show Dynamic-Memory-Accesses that can potentially be equalized (those with the same colours) with each other.

Given Dynamic-Memory-Accesses with the same PC and micro-access-ID (called *equalizable Dynamic-Memory-Accesses*), we use an equivalence heuristic to identify Dynamic-Memory-Accesses that have similar error propagation in the program execution and hence can be equalized.

The equivalence heuristic used is called the *Load-Order* heuristic. For a given Dynamic-Memory-Access D, the Load-Order heuristic records the PCs of all the loads (Dynamic-Memory-Accesses of type Read) executed after D that read from the same memory location (same memory address) as D, in a list called the *Load-Order-Chain.* The Load-Order-Chain concatenates consecutive reads from the same PCs into a single entry so as to observe longer load patterns. The Load-Order-Chain stops being updated when its size reaches a pre-determined limit l or when a write to the same address as D is encountered. Such Load-Order-Chains are built for each Dynamic-Memory-Access in the program. Equalizable Dynamic-Memory-Accesses that have the same Load-Order-Chains are grouped together in the same meta-equivalence class. Figure 3.11 illustrates the Load-Order equalization heuristic for two equalizable Dynamic-Memory-Accesses.

The size l of the Load-Order-Chain is set to 50 in this work to restrict the number of equivalence classes (and hence the number of pilots for error injection) generated. We start with l=50 since prior work[30] has shown that a control depth of 50 can adequately capture different program paths for error propagation. The high pruning percentage and accuracy of this technique validates this choice and we stop there. Exploring smaller Load-Order chains for more efficient analysis is left for future work.

Individual data bits within an data byte or word(s) accessed by a Dynamic-Memory-Access can display different error-characteristics. For example, an error in the most significant bit (MSB) of a data word can lead to a different error outcome compared to an error in the least significant bit (LSB). Hence, it is crucial that error analysis distinguishes between individual data bits within Dynamic-Memory-Accesses and only group together data bits that occupy the same relative positions within an equivalence class (for example, all MSBs across different Dynamic-Memory-Accesses in the same meta-equivalence class will be grouped together and all LSBs will be grouped together). Thus, given an meta-equivalence class of Dynamic-Memory-Accesses E, all data bits in the same position (byte offset and bit number) across all the Dynamic-Memory-Accesses in E will be grouped together into equivalence classes. To illustrate with an example, say that a single representative Dynamic-Memory-Access D is chosen from E. Since D accesses data of size n (where n >= 1) bytes, it contains $n * 8$ data bits. Each of these data bits is an individual error site (for the 1-bit error model) and are each chosen individually as pilots for error injection. The error outcome O from an error injection at the data bit located at byte m (m <= n), bit b of *D's* access is assigned to the corresponding data bits at byte m and bit b for all the Dynamic-Memory-Accesses in E. For the multi-bit error models the appropriate number of bits are chosen and equalized to the same bit positions across the members of E.

Other Heuristics Considered

The "goodness" of an equivalence heuristic is measured by (1) how effective it is at pruning the number of error sites that need analysis by error injections (referred to simply as *heuristic effectiveness*) and (2) how accurate it is at grouping "similar" errors into equivalence classes, i.e, it did not group "dissimilar" errors together (referred to as *heuristic accuracy*).

A "good" equivalence heuristic will equalize more error sites into fewer equivalence classes leading to fewer pilots and larger pruning. But this alone is not sufficient since accuracy is a necessary requirement for an equivalence heuristic. Accuracy of equivalence heuristic determines if the pilot chosen for analyses truly represents the other members of the equivalence class and hence determines the accuracy of the final memory error profile generated. Thus, the best heuristic is one that shows high pruning effectiveness and high accuracy.

We also considered and evaluated other equivalence heuristics for DataApproxilyzer based on data and control flow, before picking the Load-Order heuristic. For example, we considered two variation of the Load-Order heuristic: (1) which tracks not only the PC order

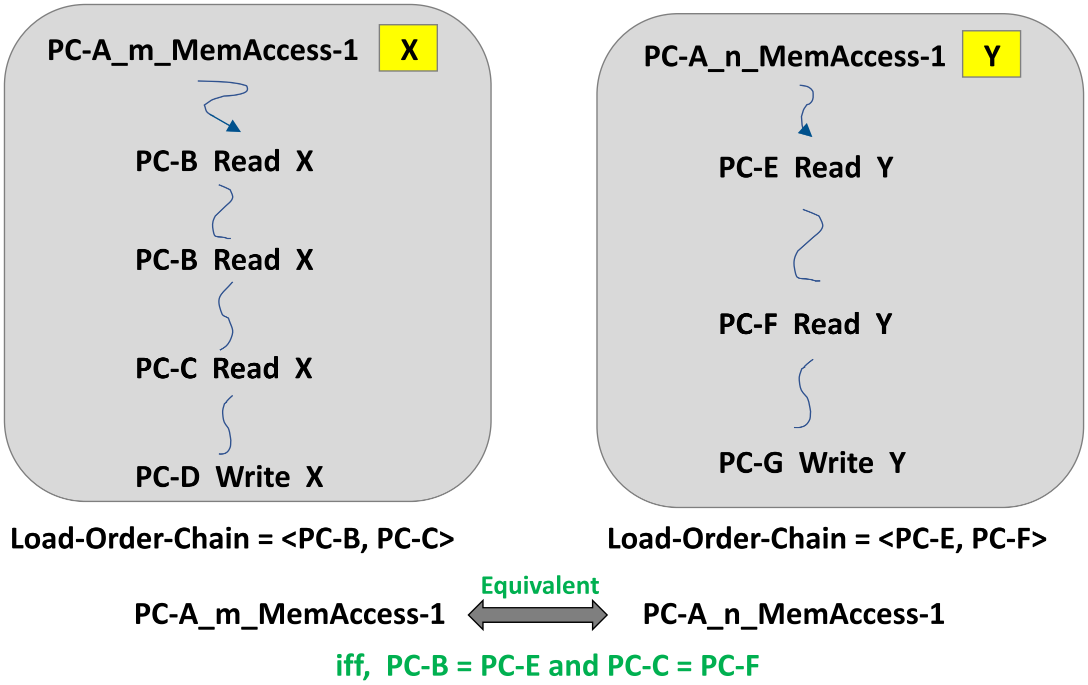


Figure 3.11: Equalization of two Dynamic-Memory-Accesses using Load-Order heuristic. Both Dynamic-Memory-Accesses have the same micro-access-ID and PC and hence are equalizable. The micro-access-ID is shown simply as MemAccess-1 to illustrate that the methodology is the same irrespective of whether the memory access is a read or a write.

but also the number of times each load PC reads a corrupted value and (2) full load chains (this is similar to the store equivalence used by Approxilyzer [31]) with different limits on the size of the load chains. We also considered simply using the control-equivalence heuristic from Approxilyzer. We found the Load-Order heuristic to provide the largest pruning percentage; the number of pilots to be analyzed were reduced by up to 80% compared to the next best heuristic, which was a variation of the Load-Order heuristic but with no cap on size of the load chain. This is because, in hot code with large loops, dynamic instructions (belonging to the same static instructions) in different iterations may never be equalized if we strictly track every subsequent load or branch within the loop (they will monotonically lead to decreased chain lengths with each iteration); leading to more equivalence classes and hence smaller pruning. We will show in Section 3.6.5 that the Load-Order heuristic is also extremely accurate at predicting the error outcomes at very fine output quality granularity.

Application	Input
Blackscholes	64k options
Swaptions	1 option 1 simulation
LU	16x16 matrix 8x8 blocks
FFT	2^8 data points
Sobel	321x481 pixels

Table 3.9: Workloads and inputs.

3.6.2 Workloads, Input and Quality Thresholds

To evaluate DataApproxilyzer, five benchmarks from three different benchmark suites spanning multiple application domains (application descriptions can be found in Section 3.3.5) are used. Table 3.9 lists the workloads and inputs used to evaluate DataApproxilyzer. To facilitate extensive analysis of data errors for large applications like Swaptions, FFT and LU, we chose minimized inputs (Chapter 5) that can enable fast and comprehensive error analysis. For relatively smaller benchmarks like Blackscholes and Sobel we use reference inputs provided in the Parsec [122] benchmark suite and Intel's Approximate Computing Toolkit (iACT) [124] respectively. The quality metrics and common-sense domain-specific quality thresholds provided by the tool (to identify SDC-Good, SDC-Bad and DDC outcomes) used for each of the applications is detailed in Section 3.3.5.

Beyond this, a user can optionally specify quality thresholds that make SDC-Maybe outputs with quality degradation within this threshold acceptable. For validating DataApproxilyzer (Section 3.3.9), we use the following 3 different quality thresholds (QT) values (with varying degrees of error-tolerance):

1. **Threshold_1:** 2% quality degradation is acceptable for Blackscholes, LU, FFT and Sobel. Absolute loss in dollar value of less that one-tenth of a cent (<\$0.001) is acceptable for Swaptions.

2. **Threshold_2:** 5% quality degradation is acceptable for Blackscholes, LU, FFT and Sobel. Absolute loss in dollar value of less that one cent (<\$0.01) is acceptable for Swaptions.

3. **Threshold_3:** 10% quality degradation is acceptable for Blackscholes, LU, FFT and Sobel. Absolute loss in dollar value of less that ten cents (<\$0.1) is acceptable for Swaptions.

3.6.3 Evaluation Infrastructure

We use the gem5 [83] simulator for extracting memory execution trace and for simulating error injections. We use the full system (Ubuntu-16.04) functional simulator in gem5 with atomic CPU and atomic memory accesses (fast-mem option). We use GCC 7.3 with -O3 to compile the studied applications.

For carrying out the error injection simulations, we developed an error injector module in gem5 that is capable of stopping at specific target cycle numbers (corresponding to specific dynamic instruction instances) and injecting n-bit errors at the targeted address bits.

3.6.4 Error Pruning Effectiveness

To evaluate the error pruning effectiveness of the Load-Order equivalence heuristic, we first calculate the total number of error sites in the application. Next we calculate the error sites corresponding to the pilots chosen after equalization using the Load-Order heuristic. The relative difference in the number of error sites between the pilots and rest of the application is the application's pruning percentage.

Application	**Number of Data Bytes Analyzed**	**Total Error Sites in Application**	**Remaining Error Sites after Pruning (% Pruned)**
Blackscholes	1835.3K	251,313,760	205,248 (99.9%)
Swaptions	30.75K	2,820,064	44,864 (98.4%)
LU	2.8K	410,432	35,616 (91.3%)
FFT	13.3K	1,373,592	25,368 (98.15%)
Sobel	309K	371,105,472	7,960 (>99.9%)

Table 3.10: Error site pruning statistics for DataApproxilyzer.

Table 3.10 lists the number of error sites before and after pruning as well as the percentage of error sites pruned (in the last column) using the Load-Order equalization heuristic. On average 98% of error sites are pruned across all the workloads with Sobel showing the highest pruning at >99.9%. Most of the applications see orders of magnitude reduction in the number of error sites to be analyzed, with the largest being a 5 orders of magnitude reduction for Sobel. Even LU, which has the lowest pruning percentage at 91.3% sees an order of magnitude reduction in error sites that need to be analyzed with error injections. LU is a small application (with minimized inputs) with only 6K dynamic memory instructions and an average of 12 dynamic instances per static memory instruction. Thus the size of the equivalence classes formed are small which equates to less pruning. The amount of error site pruning is directly proportional to the size of the equivalence classes; larger equivalence classes equates to a single pilot representing a bigger number of error sites. Thus, larger applications have more potential for error pruning.

Hence, we conclude that the Load-Order equivalence technique is very effective at error pruning.

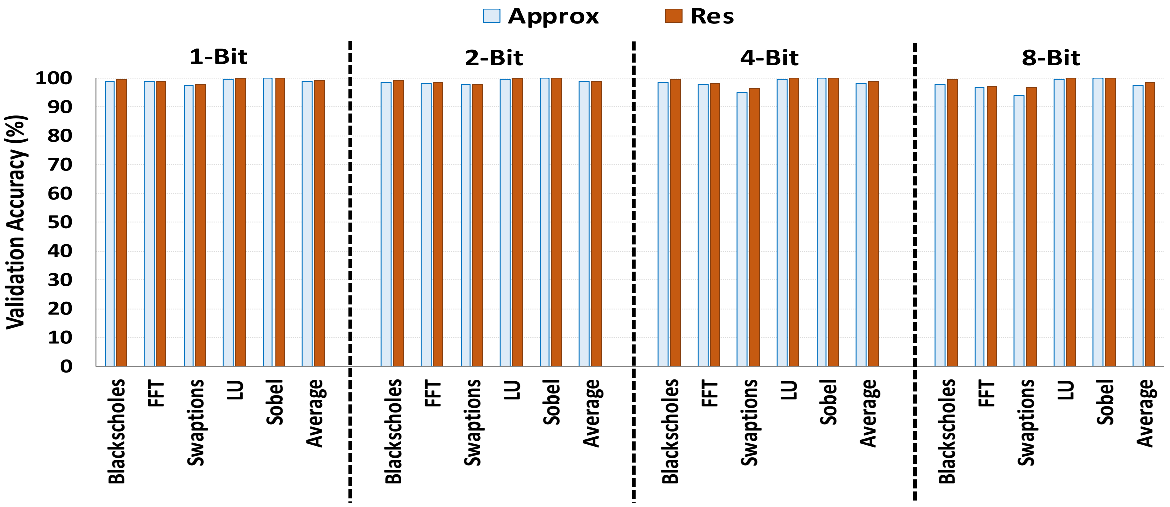
3.6.5 DataApproxilyzer Validation

We undertake validation experiments as described in Section 3.3.9 to evaluate the accuracy of the Load-Order equivalence technique and hence, DataApproxilyzer. Fig 3.12 shows the validation accuracy of DataApproxilyzer (on the Y axis), targeted to approximate computing (Approx) and resiliency (Res), for the workloads studied, across the four error models and three quality thresholds (Threshold_1 in Fig 3.12(a), Threshold_2 in Fig 3.12(b) and Threshold_3 in Fig 3.12(c)). We can see from the figure that the validation accuracy is very high across the board with average validation accuracy (across workloads and quality thresholds) for Approx of 1-bit and 2-bit error models at 99% and the validation accuracy of 4-bit and 8-bit error models at 97%; For Res, the validation accuracy of 1-bit, 2-bit and 4-bit error models is 99% and the validation accuracy for 8-bit error models is at 98%. LU and Sobel show almost perfect validation for all error models and across all quality thresholds.

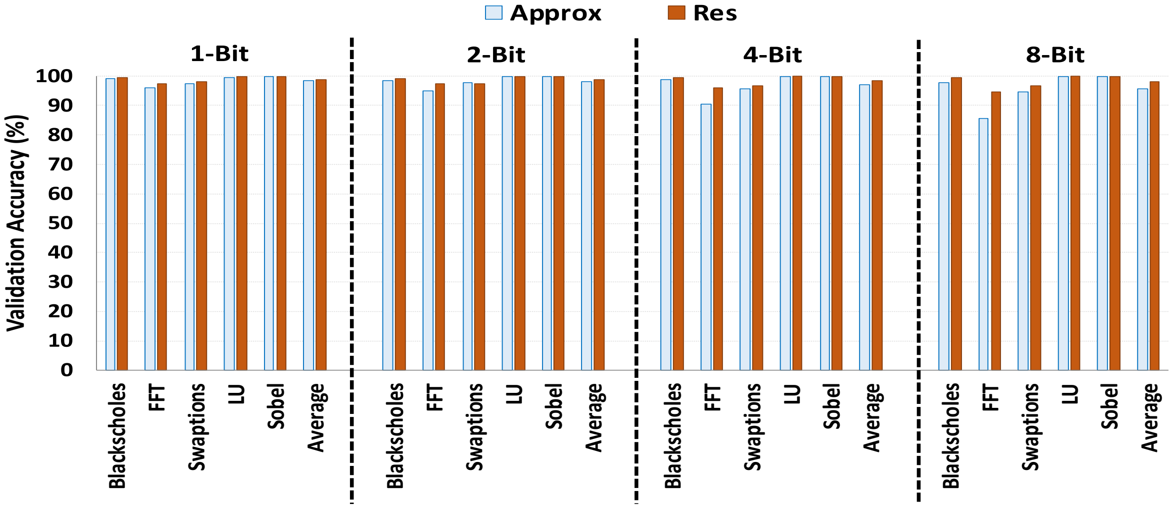
FFT shows the worse validation, especially for 4-bit and 8-bit error models, with the worst case validation accuracy of 86% for 8-bit errors with Threshold_3 for Approx. Upon closer examination, we find that the bad validation of FFT at 4-bit and 8-bit error models comes from a few equivalence classes that contain SDC-Maybe outcomes with a wider range of output quality degradation than that allowed by the validation methodology. Adjusting the flexibility quality window δ to 5% increases FFT's validation accuracy across the board to 96% (and up to 99%). Further increasing δ to 10% increases FFT's validation accuracy to >98% across all the error-models and quality thresholds.

Thus, these results show that error equalization performed by DataApproxilyzer using the Load-Order equalization technique is accurate – for both Approx and Res – for all the studied workloads across a range of realistic quality thresholds. Not only is the equalization accurate for 1-bit data errors, it also holds for 2-bit, 4-bit and 8-bit errors. Furthermore, we have shown that it can accurately predict the impact of 1-bit, 2-bit, 4-bit and 8-bit data errors on the quality of the program output up to a very fine granularity (within an error margin δ of 2%).

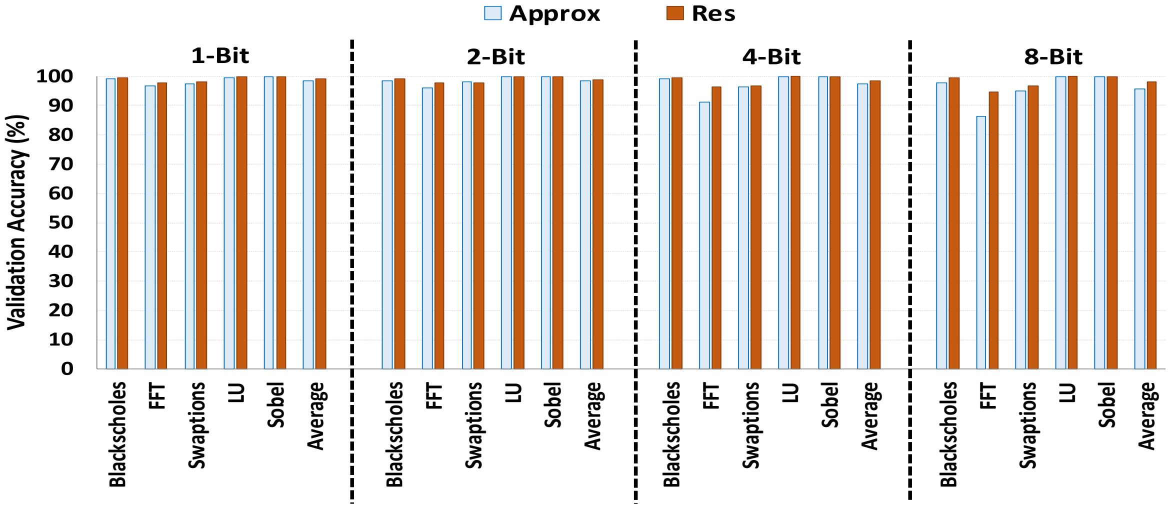
In summary, we show that DataApproxilyzer is not only effective at reducing the number of error injections required to analyze program data but is also highly accurate in precisely



(a) Threshold_1 : 2% , $0.001



(b) Threshold_2 : 5% , $0.01



(c) Threshold_3 : 10% , $0.1

Figure 3.12: Validation accuracy for DataApproxilyzer

predicting the program's output quality in the presence of 1-bit, 2-bit, 4-bit and 8-bit data errors.

3.6.6 Building the Application's Data Error Profile

The application data error profile is a list of all the data error sites analyzed in the program along with the output quality expected when they encounter an error. This is straightforward for error models where the number of pilots generated are feasible to comprehensively analyze by error injection experiments. However, for complex error models, comprehensive error injections is all pilots can be prohibitively expensive. To illustrate with an example, suppose that DataApproxilyzer equalizes data errors in a program P, prunes 99% of data and identifies just 64 bits of data (as pilot) for detailed analysis by error injection. We can now comprehensively analyze P for all 1-bit and 2-bit errors by performing just 64 and 2016 error injections respectively. However, to analyze P for all 8-bit errors would require approximately 4 billion error injections. Nevertheless, even in cases of n-bit errors where combinatorial explosion makes 100% comprehensive analysis infeasible, statistical methodologies can be employed to generate an extensive data error profile that can provide a first-order estimate of the program data's error characteristic.

In this section we describe how we build the data error profile for the different error models.

1-bit: We build a comprehensive data error profile by (1) doing error injections in all pilot data bits and (2) assigning their error outcomes to the corresponding data bits in their equivalence classes. Hence, the final Data Error Profile comprehensively lists *all* the error sites (for this error model) in the application along with their error outcomes.

2-bit, 4-bit, 8-bit: Ideally for an n-bit error model, a comprehensive data error profile will list the error outcome for all n-bit error sites in the program. But this can quickly become prohibitive as n increases. Hence, we use a statistical methodology to build a data error profile that provides a first order estimate of the error characteristics of the program data that can be used for targeted error-efficiency technique. In Chapter 4, we will demonstrate how an application's data profile can enable a customized approximate computing technique. And hence, here we show an example of how an extensive data error profile can be built for a use-case of approximate computing. Note that this methodology can be tuned based on analysis budget and use-case. While we leave the exploration of other methodologies to future work, we note that the methodology described here is sufficient for the error-efficiency use case described in Section 4.4.

For a given n-bit error model ($n > 1$), 16 different n-bit error injections are performed in each pilot's data (note that there are multiple pilots in the program). We chose 16 different combinations of multi-bit errors since we observed that most of our pilots spanned 16 or 32 data bits and using 16 different combinations increased the probability that each of the different bits were included in at least one n-bit error-site composition. We observe the outcomes from 16 error injections per pilot and calculate the percentage of approximable error outcomes (Section 3.3.7). We then assign this percentage (as an approximation probability) to each of the data bits in the pilot and therefore to each of the data bits in its corresponding equivalence class. Thus the data error profile contains each single data bit in the application along with the probability that an n-bit injection spanning the given data bit will result in an approximate outcome.

3.7 CONCLUSION

This work introduces a suite of automated application-level error analysis tools – Approxilyzer, gem5-Approxilyzer and DataApproxilyzer – that can determine the output that the program will produce for each of the billions of error that the program might encounter in its computation and data. To the best of our knowledge, the error analysis tools and techniques developed as part of this work are the first-of-their-kind that satisfy all six requirements of automation, accuracy, precision, comprehensiveness, (relatively) low-cost and generality.

The automated error analysis tools accurately (>95% on average, up to 99.9%) quantify the impact of virtually all errors (for a given error model) in a program's computation and data on its final output quality at very fine granularities (within a 2% error margin). Using a hybrid approach (built upon prior work [33]) of program analysis and relatively few (up to five orders of magnitude less) error injections, these tools can comprehensively analyze billions of possible errors that can impact a program's execution at low cost. They impose the absolute minimal programmer burden by only requiring the user to provide an unmodified program (along with relevant program inputs), a (domain-specific) quality metric and optionally a quality threshold. These tools are general and can be used to analyze any general-purpose application and have been evaluated for five different error models. The output of automated error analyses are *comprehensive application error profiles* that list the errors that can affect the program's execution (instructions and data) along with the corresponding output quality expected for each of the errors. The application error profiles can be used by programmers or systems to derive a comprehensive view of the application's error characteristics without the need for programmer expertise.

Chapter 4: AUTOMATED ERROR ANALYSIS TO CUSTOMIZED ERROR-EFFICIENCY

The paradigm of error-efficient computing exploits new opportunities for compute efficiency by allowing the system to make controlled errors. One of the fundamental requirements for achieving safe and effective error-efficiency is to understand how errors encountered during a program's execution impact the quality of the program output. In the absence of systematic methodologies to guarantee output quality in the presence of errors, error-efficient techniques today rely on program or domain experts to understand the error tolerance characteristics of their program and articulate it via custom annotations, datatypes, error-bounds on function parameters etc. [3, 4, 5, 17, 18, 19]. This is very limiting since such expertise is sparse and can take years to develop for specific domains [20, 21, 22]. To alleviate this dependence on programmer expertise, this work developed automated error analysis tools (Chapter 3) that can automatically determine the impact of billions of errors in the program's computation and data on its output quality.

The application error profiles generated by the automated error analysis tools allow us to derive a comprehensive view of the error characteristics of the application without the need for programmer expertise. To demonstrate the versatility of this approach, this work demonstrates how these automatically generated error profiles can be used to devise different error-efficiency solutions – from low-cost resiliency to approximate computing – that are tailored to the application and user requirements.

In the rest of this chapter, we first show how the application instruction and data error profiles can enable an understanding of the application's error characteristic, which can inform their potential for error-efficiency. For example, the application's error profile is used to perform a first-order analysis to identify promising subsets of instructions and/or data that can be potentially targeted by approximate computing techniques. The error profiles can also be used to gain insights that can motivate further research. For example, we show that error profiles for the same application, compiled to different ISAs (SPARC vs. x86), can vary widely [25]; motivating the need for customized error-efficiency for different architectures, and validating the choice of error analysis at the machine-code level (as opposed to at the intermediate representation or source-code level).

Next, we demonstrate how the first-order understanding built from the application's error profile can be used to build error-efficiency solutions that are customized to the application and user targets. For example, we demonstrate how an application's instruction error profile is used for error-efficiency solutions targeted towards ultra-low cost resiliency to transient hardware errors [31]. We show that large resiliency overhead reductions (up to 55%) are possible if the user is willing to tolerate a very small loss in output accuracy (1%) while still providing high (99%) resiliency coverage [31]. In another example, we demonstrate how the data error profile can be used to enable an approximate computing technique. Specifically, the data error profile is used to (1) automatically identify non-critical or approximable data (for a given user-specified quality threshold) and (2) map them to be stored in approximate DRAM with low refresh rates (which saves power – 23% on average for our workloads – but incurs modest errors), without programmer intervention [26]; prior techniques required the programmer to identify non-critical data with annotations. We show that this automatic mapping to approximate memory meets the quality target set by the user 98% of the time, on average, over (tens of) thousands of executions.

In summary, this chapter demonstrates the versatility of the automated error analysis methodology in enabling a diverse range of customized hardware and software error-efficiency solutions targeting both instructions and data.

4.1 METHODOLOGY TO IDENTIFY ERROR-EFFICIENCY OPPORTUNITIES

Error-efficient computing environments often trade accuracy in the program output for gains in other system parameters such as energy, performance or resiliency overheads. The application error profiles, which quantify 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. Two error-efficient techniques are discussed in this work – (1) *approximate computing* which deliberately introduces errors in computation for improved performance or energy, and (2) *ultra-low cost hardware resiliency* which allows some unintentional hardware errors to escape as user-tolerable output corruptions (rather than incurring high overheads to prevent all errors).

In Section 3.3.7 (Table 3.6), we detail how, based on the error outcomes reported in the error profiles, the AEA tools can identify which error sites in the application need protection from transient errors (low cost resiliency), or alternatively, which error sites could be approximable (approximate computing). The reasoning about the error-efficiency potential of individual error sites can be extended to larger granularities, such as instructions (both dynamic and static) and data bytes, based on the error outcome of their constituent error sites. The methodology is described below.

4.1.1 Low-Cost Resiliency

The application error profiles can be used to tune the loss in output accuracy with respect to reduction in the overhead costs related to resiliency. Section 3.3.7 (Table 3.6) describes how each error site's output quality is used to decide whether that error site needs protection from transient errors. Error-sites with Masked, SDC-Good, DDC and Detected outcomes do not need resiliency protection. SDC-Bad error sites need to be protected. SDC-Maybes error sites 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 larger granularities such as instructions (both dynamic and static) based on the quality of their constituent error sites. For example, the instruction error profile contains error sites that describe individual register bits in operand registers of dynamic instructions. The description for each error site in the instruction error profile (Section 3.3.6) identifies the register, the dynamic instance and the static instruction that contains the given error site. Thus, this information can be used to identify dynamic and static instructions in the application that need resiliency protection. For a given dynamic instruction, if any of its constituent error sites (spanning all bits of all operand registers accessed by the dynamic instruction) needs resiliency protection, then the dynamic instruction is marked as needing resiliency protection. Similarly, if any of the dynamic instructions belonging to a single static instruction (identified by program counter or PC), is marked as needing resiliency protection, then that static instruction is deemed as needing resiliency protection.

4.1.2 Approximate Computing

The application error profiles can be used to identify which error sites in the program are potential first order candidates for approximations. Section 3.3.7 (Table 3.6) describes how each error site's output quality is used to decide whether that error site is a candidate for approximation (referred to as approximable) or not. Error sites with Detected, DDC and SDC-Bad outcomes are considered to be not approximable. Error sited with Masked and SDC-Good outcomes are considered approximable. SDC-Maybe error sites can be approximable (or not) depending on whether their output quality meets (or does not meet) the output quality threshold set by the user.

Approximation Potential of Instructions

The knowledge of individual error sites can be extended to decide the approximation potential of instructions (static and dynamic) based on the quality of their constituent error sites. Using the instruction error profile, if any error site in a dynamic instruction (spanning all bits of all operand registers accessed by the dynamic instruction) is deemed not approximable, then the dynamic instruction is marked as non approximable. Otherwise, the dynamic instruction is marked as a potential candidate for approximation. The knowledge of the approximation potential of dynamic instructions can be similarly composed to decide if a static instruction is approximable (all constituent dynamic instructions are approximable) or not (at least one constituent dynamic instruction is deemed not approximable). Note that individual error sites can be composed to different granularities (using the same methodology described here) depending on the approximation techniques. For example, individual operand registers within dynamic instructions may be considered for approximation depending on the approximation potential of their constituent error sites (all bits within the given operand register).

Approximation Potential of Data

The approximable potential of error sites within the application data error profile (Section 3.3.6) can similarly be used to identify approximable data at different granularity within the program. For example, we can understand the approximation potential of a single byte of program data by looking at the error outcome for all of its constituent bits accessed at different dynamic instructions in the entire program. To illustrate with an example, consider data byte D stored at memory starting at address A. Whether data byte D is approximable is decided by looking at error sites corresponding to all data bits located at memory locations A to $A+7$, across all dynamic instructions that access these data bits. If all of the constituent error sites are approximable then data byte D is considered approximable.

For n-bit error models ($n > 1$), for each data error site, the data error profile generated by DataApproxilyzer lists the probability (as a percentage) that an error at the given error site will result in an approximable outcome (Section 3.6.6). Thus, when using the data error profile across these error-models, approximable data bytes are identified for different approximation targets (for example, data bytes that produce approximable error outcomes 100% of the time they encounter an error vs. those that are approximate only 50% of the time they encounter an error).

4.1.3 Discussion

Since this work uses transient single bit errors as the intruction error model, instructions marked as approximable 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. Similarly, for data since the error analysis only considers the impact of one (n-bit) transient error in isolation, data bytes marked as approximable may be false positives; false negatives are similarly expected to be rare.

While this 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 and data on which it is feasible to do further rigorous and targeted analysis. Another benefit of this approach is that the identification of approximable instructions and data 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.

The AEA tools significantly reduce analysis time by performing error injections only in the pilots of the generated equivalence classes (Section 3.3.8). 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 [24] analysis), thereby trading off simulation time for a modest loss in coverage [33]. The 1% of error sites not included in the study do not detract from the observations and gains reported in this work. While identifying approximable instructions, these remaining error sites might introduce some false positives (in the event that they produce unacceptable errors). This is, however, consistent with the overall goal to tolerate some false positives, while minimizing false negatives, in the quest to uncover approximation opportunities in the application. For resiliency overhead tuning, these unexplored error sites might represent missed opportunity (in the event that they produce SDCs) for further overhead reduction than that reported in this work.

4.2 EXPLORING FIRST-ORDER ERROR-EFFICIENCY OPPORTUNITIES

In this section, we will detail how a given application's instruction and data error profile can be used to derive a first-order understanding of its potential for error-efficiency.

4.2.1 Output Corruption Distribution

In this study, we examine the instruction error profiles generated by Approxilyzer (Section 3.4) to gain insights into application error characteristics. Fig. 4.1 shows the distribution of error outcomes for error sites in the studied applications[1] (Section 3.4.2). 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.

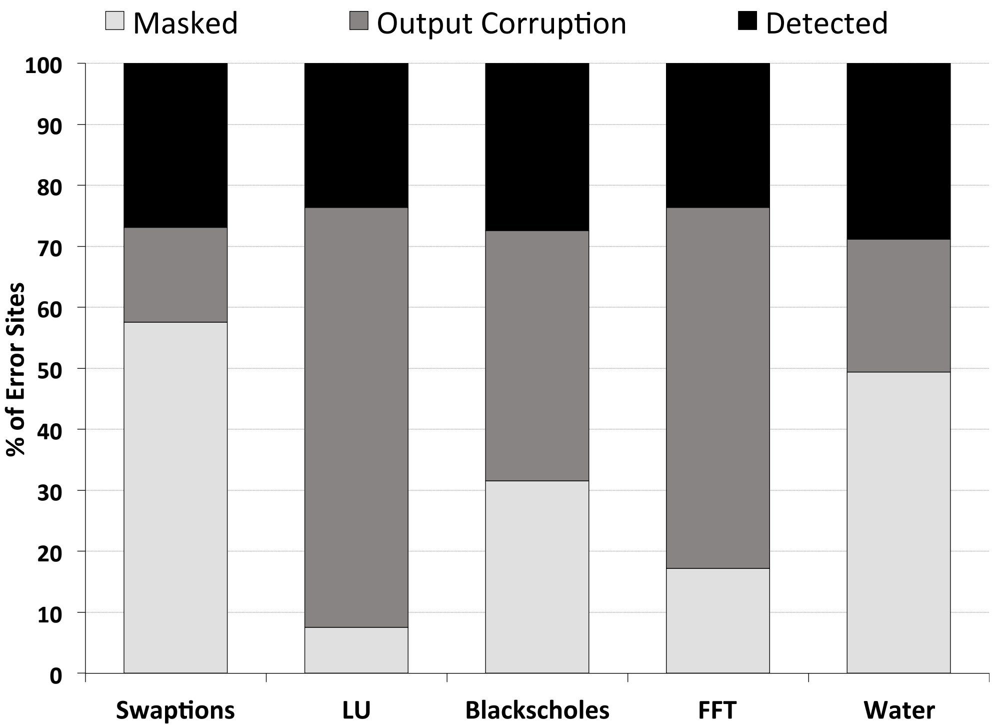


Figure 4.1: Distribution of error outcomes for the applications studied.

Fig. 4.2 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 vs. 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

---

[1] The OC (originally SDC) rates reported in this work are different from the rates reported in previous work [13, 33] as our error model is different. We study errors in both integer and floating point architectural registers, while prior work only considered integer registers.

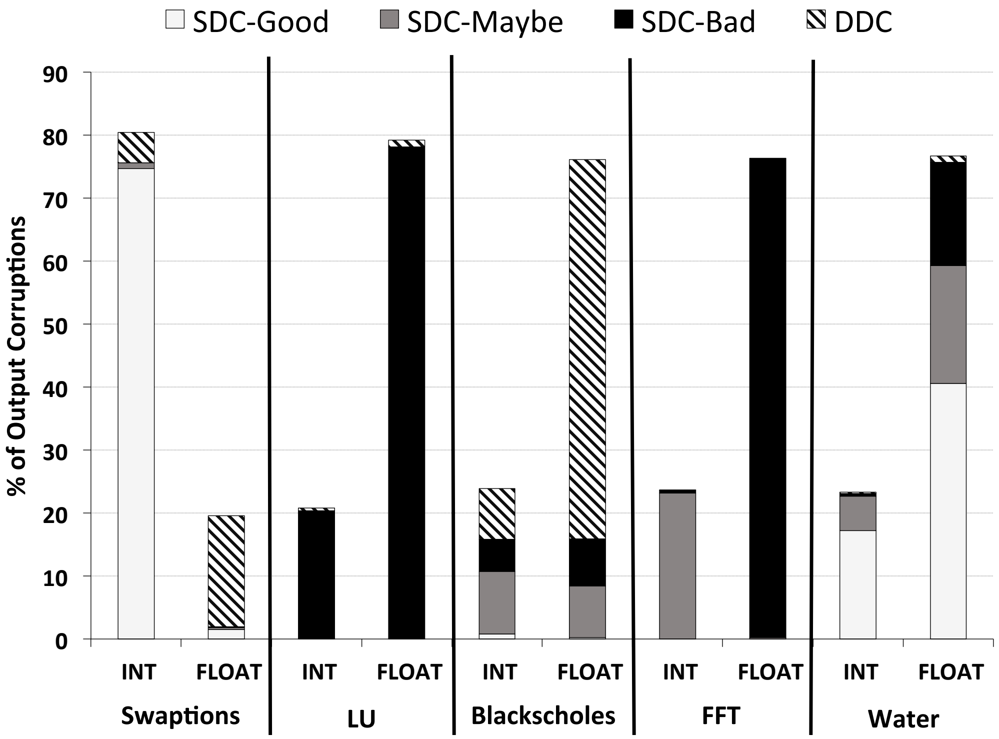


Figure 4.2: Distribution of output corruptions (OC) in integer (INT) and floating point (FLOAT) registers.

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 the instruction error profile can be employed to automatically analyze an application to gain insights into its behavior in the presence of errors in its instructions.

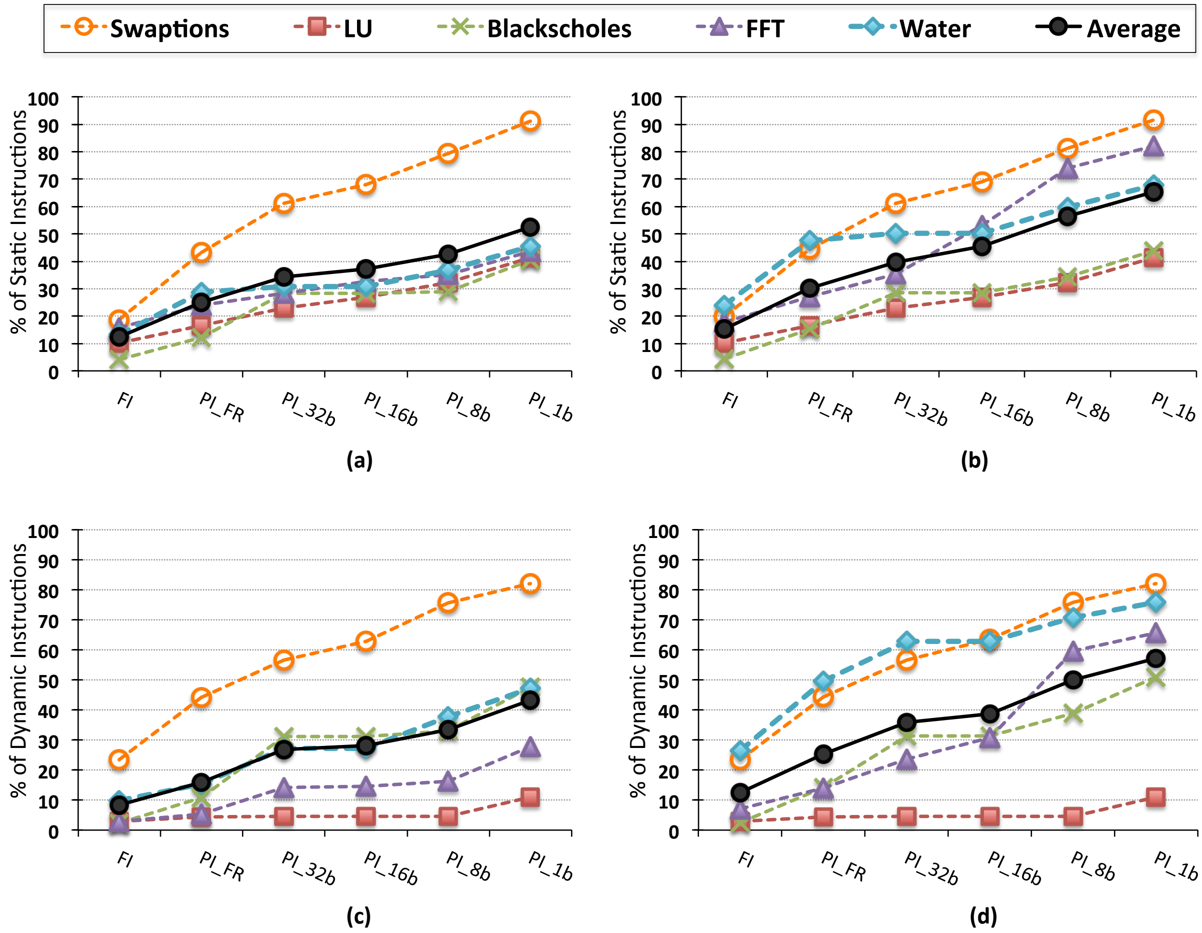
4.2.2 Exploring Approximation Opportunities in Program Instructions

In this section, we use the instruction error profile (generated by Approxilyzer) to analyze the approximation potential of static instructions for five workloads (Section 3.4.2). We use the same technique used to identify which static instructions are potentially approximate (Section 4.1.2), 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:

(1) *Full Instruction (FI)*: The entire static instruction (i.e., all register bits) is approximable.

(2) ***Partial Instruction, Full Register (PI_FR)*****:** At least one full register in the static instruction is approximable. **(3)** ***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.



(a) (b) (c) (d)

Figure 4.3: 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.

Figure 4.3(a) and 4.3(b) show, for each application the worst and best case bound, respectively, on the number of static instructions, marked as candidates for approximation (as described in Section 4.1.2). In order to understand the potential impact of approximating these static instructions, Figure 4.3(c) and 4.3(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 that Approxilyzer does not study (Section 3.4.3).

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, as shown in Section 4.2.1.

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[2].

In summary, the instruction error profile 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 much hidden approximation potential that can then be targeted by specialized techniques.

4.2.3 Exploring Approximation Opportunities in Program Data

In this case study, we use the data error profile generated by DataApproxilyzer (Section 3.6) to perform a first order analysis that estimates the approximation potential of

---

[2] This can be useful to determine if the application will benefit from approximation techniques at the dynamic instruction granularity (e.g., task skipping [125] and loop perforation [39]).

different data bytes in each workload studied (Section 3.6.2), across different approximation targets and different data error models (Section 3.3.2). To calculate the approximation potential of a single data byte for the 1-bit error model, we look at all the error sites listed in the data error profile corresponding to all bits (across all dynamic accesses) of the given data byte and calculate the percentage of error outcomes that are approximable. For the multi-bit error models, we similarly identify constituent error sites in the data error profile and average the corresponding approximation probability listed (Section 3.6.6). This gives an estimate of the approximation potential of the data byte under consideration. We do this for all application data bytes, thereby building a list of data bytes that are approximable for given approximation targets. The building of this list of approximable data bytes takes a few seconds per application.

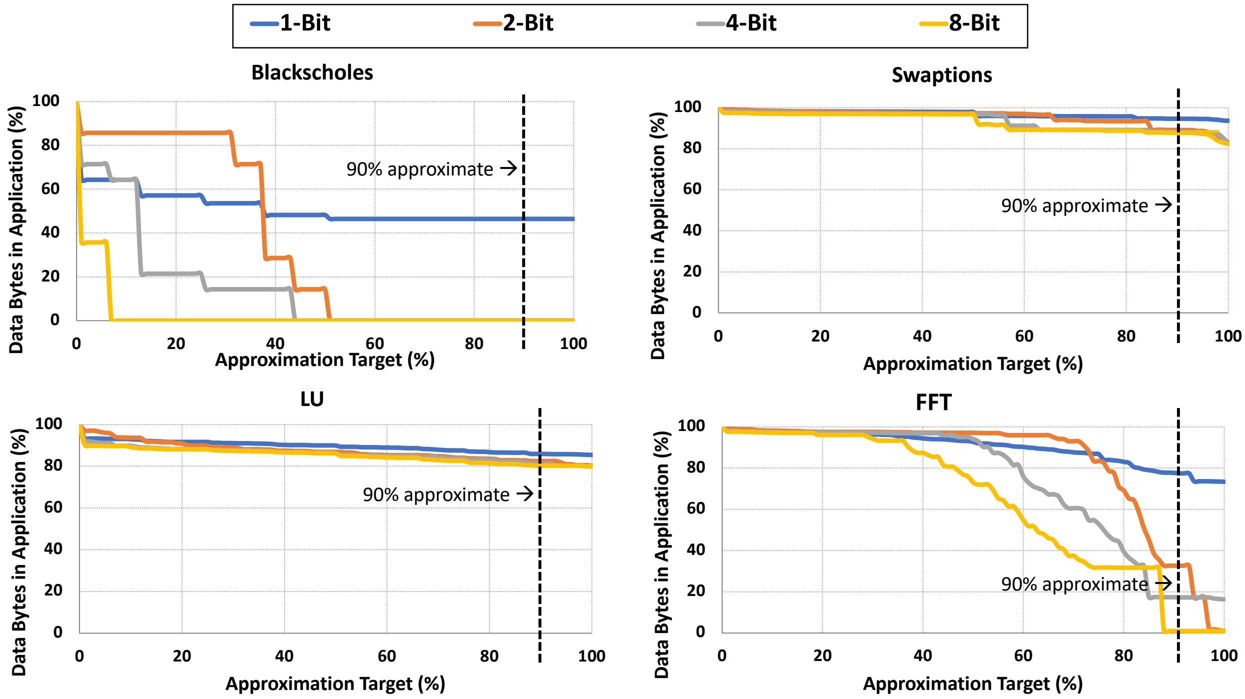


Figure 4.4: Percentage of approximable data bytes across different error models and approximation targets.

Fig. 4.4 shows the percentage of application data bytes (on the Y axis) that are approximable at different approximation targets (shown on the X axis). The line at 90% on the X axis, for example, represents data bytes that are approximable 90% of the times they encounter an n-bit error. We only show four workloads here for brevity. We see that 46% of the data bytes in Blackscholes are 100% approximable when perturbed by single-bit errors. However, they quickly become non-approximable (yielding egregious outcomes), when perturbed by multi-bit errors. FFT shows a similar trend of being less tolerant to multi-bit errors. However, we note that most of the data bytes in FFT are tolerant to a significant number of errors. For example, >80% of the data bytes in FFT are approximate 40% of the time (40 on the X axis) they encounter errors of any bit length. Further analyzing FFT's data error profile to understand patterns in error sites that are approximable is an interesting future direction.

The trend-lines for LU, Swaptions and Sobel (not shown for brevity) show that a high percentage of program data in these workloads are approximable – perturbing the data using the 1-bit, 2-bit, 4-bit and 8-bit error models leads to acceptable loss in output quality a large fraction of the time. For example, >80% of data bytes in Swaptions are approximable 99% of times they encounter an error. This number is even higher for Sobel where 99% of data bytes are approximable 100% of the time for all the error models studied.

In summary, the data error profile (generated by DataApproxilyzer) makes it possible to automatically perform such analysis that enable programmers and approximate computing systems to make informed decisions about the approximation potential of the program data and possibly inform data targeted approximations.

4.2.4 Application Error Profiles under different ISAs

In this case study, we compare the error profiles of the same applications compiled to two different ISAs – SPARC and x86. First we use gem5-Approxilyzer to generate the instruction error profile for workloads (Section 3.5.5) compiled to x86 binary. Next, we generate instruction error profiles for the same workloads compiled to a SPARC binary using Approxilyzer. This allows a first-order comparison comparison of the resiliency and approximation characteristics of the same workloads for two different ISAs.

Figure 4.5 compares the distribution of error outcomes (for all the error sites) in each application for the x86 and SPARC ISAs. The instruction error 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 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.

Next, we use the instruction error profiles (for x86 and SPARC) to identify the sets of

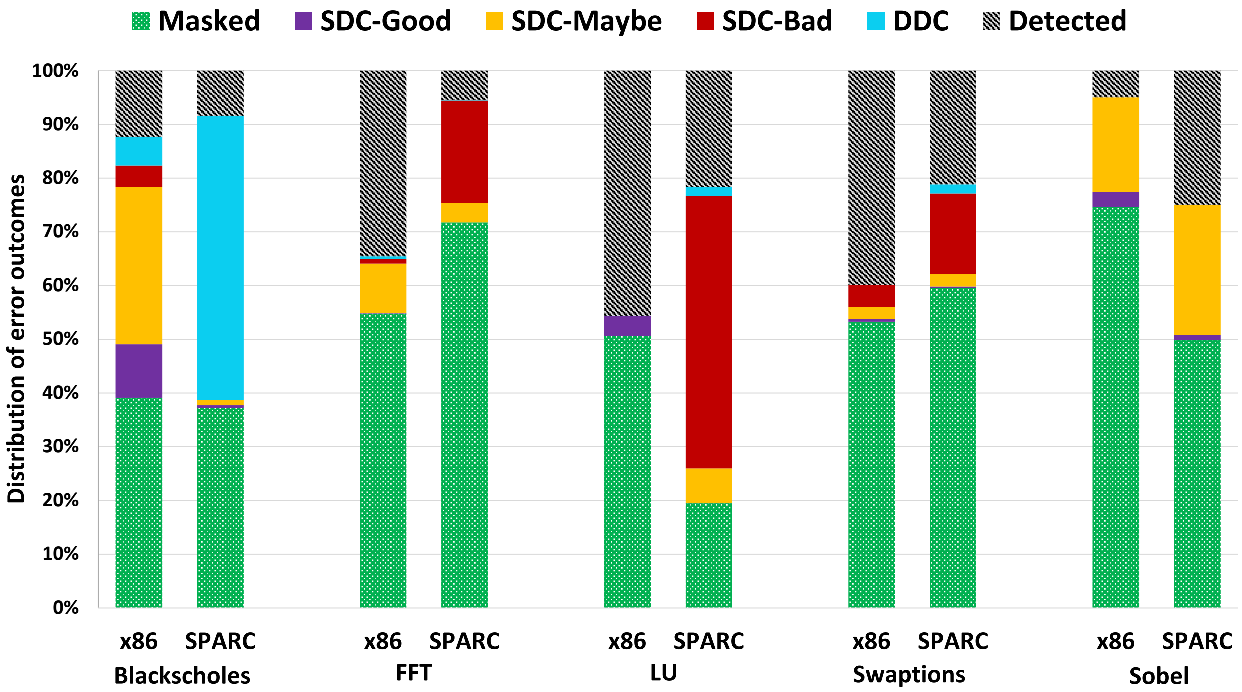


Figure 4.5: Distribution of error outcome categories for the applications studied using the x86 and SPARC ISAs.

static instructions that need (1) resiliency protection and (2) are candidates for approximate computing (using the methodology described in Section 4.1) under given output quality thresholds. Figure 4.6 shows the percentage of static instructions that need resiliency protection and those that are approximable for the same quality threshold (QT) across the two ISAs.

The wide differences across the two ISAs and the lack of a clear trend further underscore the importance of error analysis tools (such as those developed in this work) 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.

4.3 CUSTOMIZED LOW-COST RESILIENCY USING THE APPLICATION ERROR PROFILE

In the previous section, we demonstrated how the application error profile can be used to build a first-order understanding of the application's error-efficiency potential. In this section we will show an end-to-end workflow of how this understanding of the application's error characteristic can be used to enable an error-efficiency solution. Specifically, we will

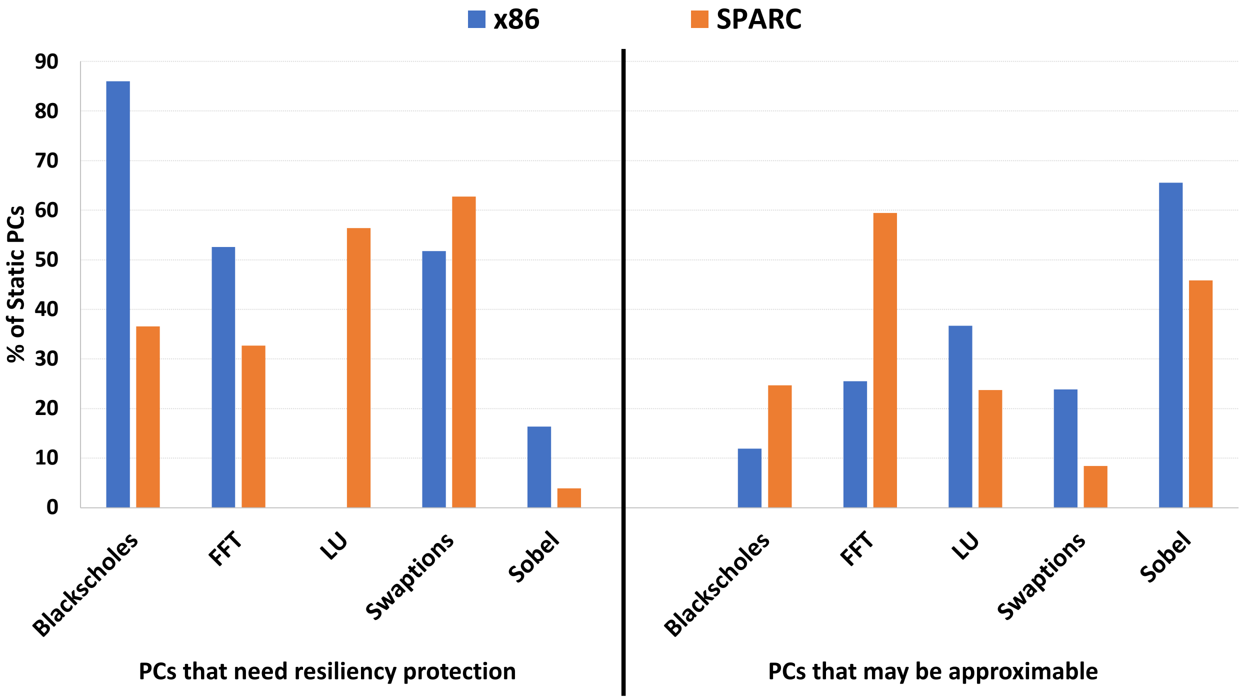


Figure 4.6: Percentage of static PCs in the application that need resiliency protection and percentage of PCs that are approximable across x86 and SPARC. The same quality threshold is used across both ISAs: 5% for Blackscholes, Sobel, FFT, and LU; $0.001 for Blackscholes and Swaptions.

show how the application's automatically generated *instruction error profile* is used to enable an *ultra-low cost resiliency* solution to transient hardware errors that can be tuned to user or system requirements.

4.3.1 Background for Resiliency Analysis

As we have discussed in Chapter 1 and Chapter 3, increasing susceptibility of hardware to errors pose a challenge to the reliability of modern systems. In this section, we will briefly recap some background on resiliency analysis as well as establish the goal of resiliency analysis performed in this study.

Traditional reliability solutions, relying on indiscriminate redundancy in space or time, are too expensive for a range of systems – from small embedded systems to large-scale high-performance computing systems. Therefore, there has been significant research in cross-layer solutions [126, 127] that rely on the software layers of the system stack to provide acceptable end-to-end system resiliency for hardware errors at lower cost than hardware-only solutions [128, 129, 130, 131].

Early work recognized that a large majority of hardware errors were either masked at the software level or resulted in easily detectable anomalous software behavior [12, 57, 132, 133, 134, 135, 136]. Since the former errors require no action and the latter can be detected using zero to very low-cost detection mechanisms, such software-centric resiliency techniques show immense promise. Unfortunately, these techniques allow some hardware errors to escape detection and result in undetected and potentially unacceptable silent data corruptions (SDCs)

Such 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 ECC in hardware memory structures) or software solutions (e.g., insertion of software checks in application code regions determined to be too vulnerable to SDCs) [12, 13, 14, 15, 16, 56, 68, 82, 107, 126, 137, 138, 139].

Underlying all of these solutions is the need for techniques that find SDCs in the applications of interest. In this study, we use the term *resiliency* to mean the ability of a given piece of software to avoid an SDC for a given hardware error. This study is concerned with identifying SDC-causing instructions in a given program.

4.3.2 Methodology to tune Output Quality vs. Resiliency vs. Overhead

As explained in Section 4.1.2, using the application's instruction error profile, specific static instructions can be targeted 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, hereby 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 (range of quality degradations). Then a 0/1 knapsack algorithm is used to pick the instructions for resiliency protection that offer the specified coverage for the least overhead. A similar methodology is used in [33] to tune resiliency vs. overhead, but without relaxation of output quality.

The (software-based) resiliency protection scheme we assume in this work is instruction redundancy ([140]) 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.

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 that the maximum quality degradation (QD) 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%.

For this study, we use the instruction error profile generated by Approxilyzer and hence use the same workloads as Approxilyzer (Section 3.4.2). The analysis to generate quality vs. resiliency vs. overhead curves for an application takes several minutes of CPU time for the workloads studied.

4.3.3 Customizable Low-Cost Resiliency

Fig. 4.7 shows the resiliency overhead cost vs. coverage for different levels of acceptable output quality degradation (QD) using the methodology in Section 4.3.2. Graphs for four of our five benchmarks is shown (the fifth, LU, is discussed later). Each graph shows the following curves corresponding to different levels of acceptable quality degradation.

(1) *All Output Corruptions*: This curve represents the optimal overhead vs. coverage when all output corruption (OC) causing instructions are protected. It represents the state-of-the-art in the absence of the output quality impact information (provided by the automatically generated instruction error profiles) to distinguish instructions that produce acceptable quality output.

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

(3) *All SDC-Bad + SDC-Maybe with QD$>x$*: These are the optimal overhead vs. coverage curves with user-specified quality threshold x. For these curves, Approxilyzer does not protect instructions that produce SDC-Maybe with QD$\leq x$ from the list of instructions protected. This essentially means that if a user says that they are willing to tolerate x%

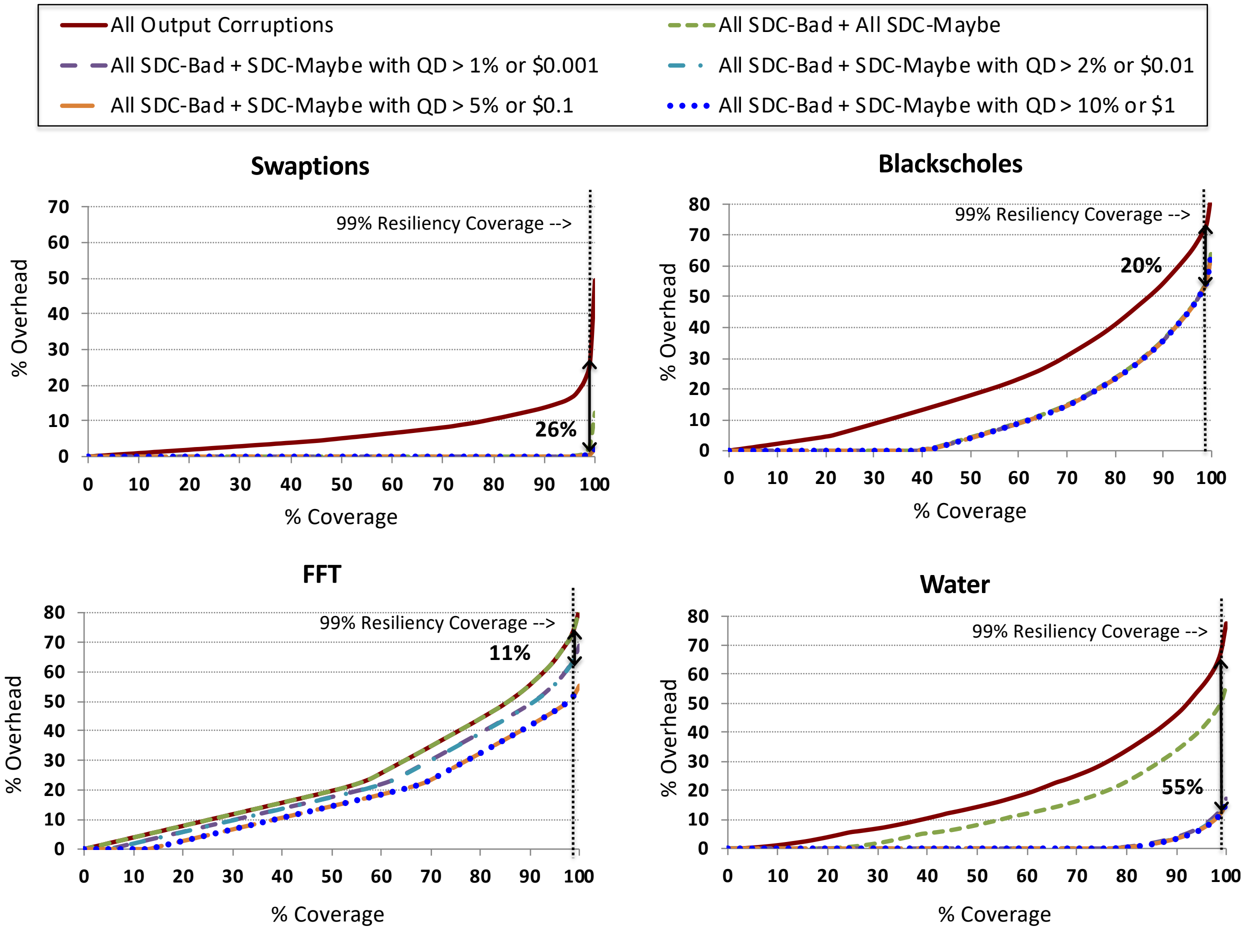


Figure 4.7: Tuning resilience vs. execution overhead vs. output quality for different applications.

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.

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 Fig. 4.7 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 QD>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 QD>$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, but for a different reason. As mentioned in Section 3.4.4, many (static) instructions in Blackscholes produce a mix of SDC-Bad and SDC-Maybe outcomes (with wide QD 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 *QD>1%* to *QD>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 (not shown in the figure for brevity) shows no gains from either quality tuning or from not protecting SDC-Good and DDCs. This is because, as seen from Fig 4.2, LU produces only SDC-Bad corruptions and hence all of the instructions need protection.

In summary, most of the applications studied show significant resiliency overhead reductions while suffering very small accuracy losses. Thus, the application's instruction error profile can be used to target ultra-low cost resiliency solutions in an error-efficient environment.

4.4 CUSTOMIZED APPROXIMATE COMPUTING USING THE APPLICATION ERROR PROFILE

To demonstrate the versatility of the (automatically generated) application error profiles in enabling error-efficient techniques, this section shows a second end-to-end workflow targeting a different error-efficient technique – Approximate Computing. Specifically, the application's data error profile (generated by DataApproxilyzer) is used to automatically identify approximable data bytes in the program (for a given user-specified quality threshold) and simulate their storage in approximate memory with sub-optimal (but low energy) refresh rates (Flikker [18]). The evaluation shows that, on average, this automatic approximate data partitioning and mapping to approximate memory meets the quality target set by the user 98% of the time over (tens of) thousands of simulations.

4.4.1 Approximate Memory Technique

There have been different techniques suggested for approximate memory solutions that trade-off errors in memory for performance or energy benefits. One such technique suggested for approximate memory is Flikker [18]. Flikker allows developers to specify critical and non-critical (approximate) data in programs and the runtime system then allocates this data in separate parts of memory. The portion of memory containing critical data is refreshed at the regular refresh-rate, while the portion containing non-critical data (referred to as *approximate memory*) is refreshed at substantially lower rates. This partitioning saves energy at the cost of a modest increase in data corruption in the non-critical data.

While Flikker needed programmer annotations to identify approximate data, here the application's data error profile is used to automatically identify critical data to store in approximate memory for a given quality target (thus removing the burden on the programmer). Note that while the data error analysis performed by DataApproxilyzer analyses (multi-bit) errors in one data block at a time, a model like Flikker can induce errors in multiple data blocks in an execution. We show (in Section 4.4.4) that the partitioning of approximate and critical data (to store in approximate and precise memory regions respectively) is accurate; i.e., it leads to acceptable (within user specified thresholds) output quality over tens of thousands of simulated runs.

4.4.2 Flikker Design Overview

This section provides a brief overview of the Flikker hardware design described in detail in [18]. Flikker enhances existing DRAM architectures (in modern mobile devices) that allow for a partial refresh of DRAM memory by allowing different refresh rates for different sections in memory.

Memory systems in smartphones devices consume power both when the device is active (active mode) and when it is suspended (standby mode). In standby mode, the refresh operation is the dominant consumer of power, and hence the focus of Flikker [18] is on reducing refresh power. In modern mobile systems, several low-power DRAM states are used in different system scenarios [18]. Self-refresh is a feature of low power mobile DRAMs in which the DRAM array is periodically refreshed when the processor is in standby or sleep mode. Partial Array Self Refresh (PASR) is a low power state in which only a portion of the DRAM array is refreshed. DRAM cells that are not refreshed will lose their data in PASR. The main difference between Flikker DRAM and PASR is that instead of discarding the data in a part of the memory array (by not refreshing the data), Flikker lowers the reliability of the data (by refreshing the data at a lower rate). As a result, Flikker is able to achieve power savings without compromising on the amount of memory available to applications [18].

In the Flikker DRAM architecture, each bank is partitioned into two different parts, the high refresh (error-free) part and the low refresh (erroneous) part. DRAM rows in the high refresh part are refreshed at a regular refresh cycle time (64 milliseconds or 32 milliseconds in most systems). The error rate of data in the high refresh parts is negligible (similar to data in state-of-the-art DRAM chips). On the other hand, the low refresh part is refreshed at a much lower rate (longer refresh cycle time) and its error rate is a function of the refresh cycle time.

Figure 4.8 [18] shows the Flikker Bank Architecture. The DRAM bank is partitioned into two parts, the high refresh part, which contains critical data, and the low refresh part, which contains non-critical data. The high refresh/low refresh partition can be assigned at discrete locations as shown in Figure 4.8. Henceforth, we refer to the the low-refresh part of memory as simply *approximate memory*. The non-critical data that is stored in approximate memory is referred to as *approximate data*.

4.4.3 Methodology to Map Data to Approximate Memory

Identifying Approximate Data

The data error profile (generated by DataApproxilyzer) is used to perform a first order analysis that estimates the approximation potential of different data bytes in each workload studied across different approximation targets and different error models. Using the methodology described in Section 4.2.3, for each of the four error models studied (Section 3.3.2), we first build a list of data bytes that are approximable for the given error model and given approximation targets. The approximation target that we use for this study is 100%, i.e., for each error model, we list data bytes that are approximable 100% of the times they encounter an error. Next, we pick the data bytes that are approximable (for a given quality threshold) across *all* 4 error models (1-bit, 2-bit, 4-bit and 8-bits). We denote these data bytes as

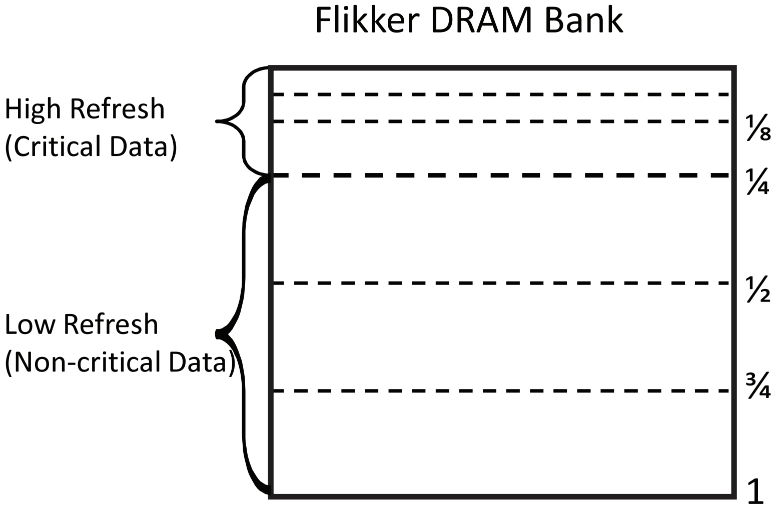


Figure 4.8: Flikker DRAM bank architecture [18]. The bank is partitioned into high- and low-refresh parts at discreet locations. The curly brackets on the left show a partition with 1/4 high refresh rows and 3/4 low refresh rows.

approximable or non-critical and mark them to be mapped to approximate memory.

Non-critical data bits are selected using the above methodology for different user-acceptable quality thresholds. The quality thresholds used in this study are described in Section 3.6.2. For the convenience of the reader, the quality thresholds used are recapped below:

1. **Threshold_1:** 2% quality degradation is acceptable for Blackscholes, LU, FFT and Sobel. Absolute loss in dollar value of less that one-tenth of a cent (<$0.001) is acceptable for Swaptions.

2. **Threshold_2:** 5% quality degradation is acceptable for Blackscholes, LU, FFT and Sobel. Absolute loss in dollar value of less that one cent (<$0.01) is acceptable for Swaptions.

3. **Threshold_3:** 10% quality degradation is acceptable for Blackscholes, LU, FFT and Sobel. Absolute loss in dollar value of less that ten cents (<$0.1) is acceptable for Swaptions.

Simulating Errors in Approximate Memory

The approximate memory technique of lowering refresh rate saves power at the cost of incurring modest errors in the non-critical (approximate) data stored in the approximate memory. To simulate these errors in the non-critical data (stored in approximate memory), we develop an error injector framework in the gem5 [83] simulator.

The injector starts the application and executes it for an initial period. No errors are injected during this initialization period. Then a self-refresh period is inserted, after which errors are injected in non-critical memory to emulate the effect of lowering their refresh rate. In order to keep track of the errors injected during the self-refresh period, the injector maintains a *modified* bit for each non-critical data bit (in low refresh memory) which tracks whether this bit has been accessed after the self-refresh period. Before a non-critical bit is read, the corresponding modified bit is checked. If it is a 0, then the bit is flipped with a pre-computed probability (error rate) and the modified bit is set to 1 (the modified bit is set to 1 to prevent future error injections into these bits). This methodology is the same as that employed by Flikker [18]).

In the original Flikker implementation, since a self-refresh cycle is only inserted once in the beginning of the application (as described above), a write to a data bit automatically sets the corresponding modified bit from 0 to 1 (no error is injected into this data). In our evaluation, however, we also simulate errors in data values that are written during the execution (this is similar to a self-refresh cycle occurring after the the data value is written but before it is read); hence, the modified bit is reset to 0 on a write to the data bit. Note that this results in our evaluation being stricter (more errors are injected in our study) than in the original Flikker work. We will show later (Section 4.4.4) that even with this more conservative methodology, we achieve high accuracy in identifying and mapping non-critical data to approximate memory. Figure 4.9 shows the state transition diagram of the modified bit.

The error rate that is used in this work is the same as Flikker [18] – 4×10^{-8} – which corresponds to a refresh cycle of 1 second at an operating temperature of 48°C. A refresh cycle of 1 second is chosen as it is shown to achieve a desirable trade-off between power savings and reliability [18].

Power Savings Estimate

An analytical power model (based on real power measurements in mobile DDR DRAMs with PASR) is used in Flikker to estimate the power consumption of different Flikker DRAM configurations [18]. Figure 4.10 (originally appearing in [18]) provides the self-refresh current of different PASR and Flikker DRAM configurations (with different refresh cycle times for the low-refresh part). We will use the values reported in Figure 4.10 to estimate power savings. Note that for the results reported in this work, we assume a refresh time of 1s and hence only use the values from the corresponding column. The self-refresh power is calculated as self-refresh current times power supply voltage (1.8V in this work). The power

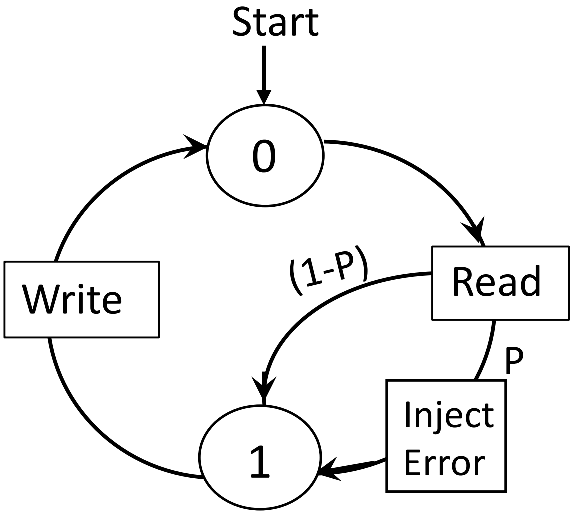


Figure 4.9: State transition diagram of "modified" bit in fault-injector. Error is injected in non-critical bits with probability P.

saved by Flikker is estimated as the difference between the power consumed in refreshing all arrays at a high refresh rate (corresponding to the PASR with a high refresh size of 1 from Figure 4.10) and power consumed by Flikker configuration that uses low refresh rate for part of the arrays. Hence, for a flikker configuration with 1/4 arrays at high refresh rate, results in a power savings of 22.5%. .

High Refresh Size	**Self-Refresh Current [mA]**			
	PASR	**Flikker**		
		1s	10s	100s
1	0.5	0.5	0.5	0.5
3/4	0.47*	0.4719	0.4702	0.4700
1/2	0.44	0.4438	0.4404	0.4400
1/4	0.38	0.3877	0.3807	0.3801
1/8	0.35	0.3596	0.3509	0.3501
1/16	0.33	0.3409	0.3310	0.3301

Figure 4.10: Self-refresh current in different PASR and Flikker configurations [18].

Note that since self-refresh (in PASR and therefore Flikker) is employed in standby mode, the power savings estimate reported here are only for the power saved in standby mode. This is still significant since prior work [141] shows cell phone usage profiles with 5% being busy and 95% in standby mode (self-refresh state).

To estimate power savings for the workloads studied, we first compute the proportion of critical and non-critical data for each of the applications studied. As discussed in Section 4.4.2 (and shown in Figure 4.10), the partitioning of approximate memory is done at discrete boundaries. Therefore, only non-critical (or approximate) data that can fit into a discrete partition is considered for power calculations. Furthermore, for power savings estimation, we allocate approximate data in 4KB pages, so we do not count the data that is fragmented, i.e., resides in the same page with non-approximate (critical) data. To illustrate with an example, suppose 80% of data bytes in an application are identified to be approximate (using the application's error profile). Discarding the fragmented approximate data-bytes, 78% of the data bytes are identified and marked into approximate pages. However, only 75% of data bytes are considered as being stored in approximate memory for power savings (corresponding to a configuration where 1/4 of arrays are refreshed at normal rate and 3/4 of the arrays are refreshed at lower rate). Note however that this is done only for the power savings estimate using the methodology described above. To generalize the accuracy of the techniques described in this work – which automatically identify approximate data and map them to approximate memory – errors are simulated in all non-critical data (80% of data bytes from the above example) that can potentially be stored in approximate memory. We describe the methodology to determine the accuracy of mapping approximate data to low-refresh approximate memory next.

Accuracy of Mapping Data to Approximate Memory

In this study, for a given user-specified quality target, we use the data error profile to automatically identify non-critical data to be mapped to approximate memory. The application is executed while simulating errors in data stored in approximate memory. The accuracy of our approximate memory mapping is validated by verifying that the quality of the final output produced by the execution is within the user specified quality target (used to generate the mapping). Since each execution can produce different errors in the non-critical data (errors are randomly injected according to a given bit error rate), it is imperative that the accuracy of a given mapping (for a given quality target) is verified over many executions.

We perform 10,000 simulations for each mapping (for a given quality threshold) of non-critical data to approximate memory. The output produced at the end of the simulations is observed. If the output quality degradation is below the quality threshold (that was used to generate the mapping), the mapping is considered to be accurate. Thus, if all 10,000 simulations lead to acceptable outputs, the mapping accuracy is considered to be 100%.

Memory System

Flikker [18] simulates approximate DRAM memory by reducing DRAM refresh rate. In this work, we simulate an abstract approximate memory device that mimics the DRAM with lower refresh rate. Caches are not simulated in this system. This is a deliberate design choice to ensure that all the errors introduced in the non-critical data (due to the lowered refresh rate) are "activated" or consumed by the execution. This need not be true in systems that have caches. For instance, an error in DRAM will not be activated if a copy of the data is present in SRAM. The design chosen is to present a strict (conservative) evaluation of the accuracy of the approximate data mapping. Hence in real systems with caches, the accuracy of the automatic approximate data mapping is expected to be higher than that reported here.

4.4.4 Accuracy of Mapping Data to Approximate Memory

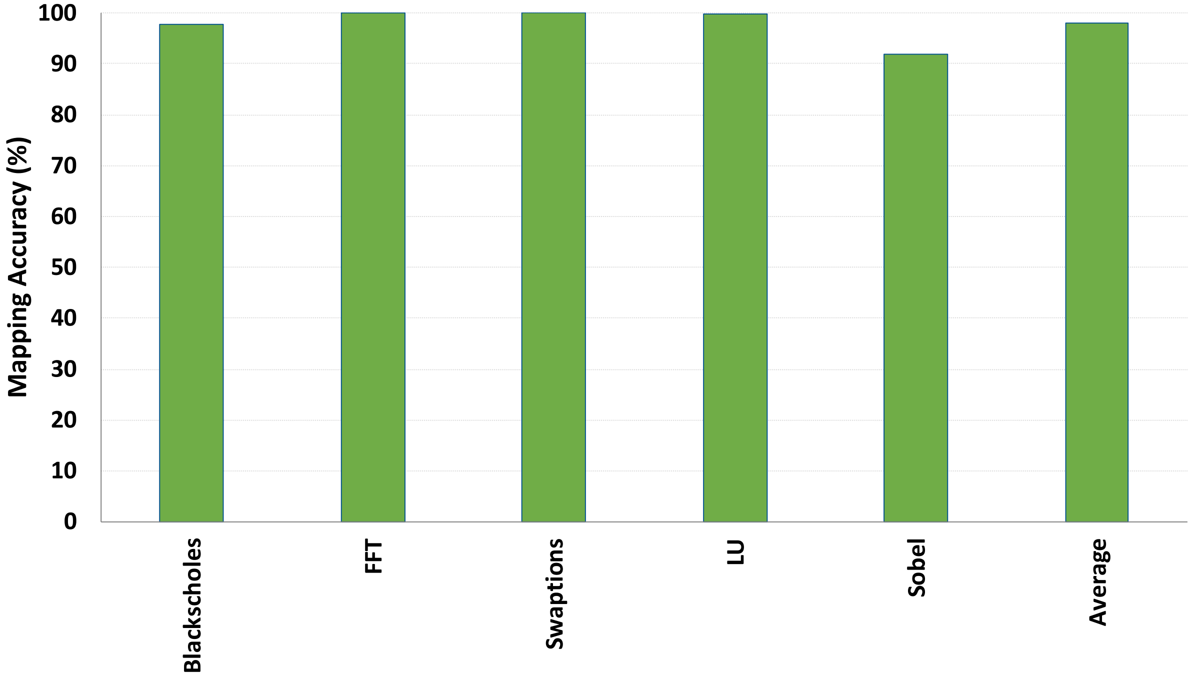


Figure 4.11: Accuracy of Mapping non-critical (Approximate) data to Approximate Memory

We evaluate the accuracy of mapping non-critical data to approximate memory using the methodology described in Section 4.4.3. We observe empirically that each simulated execution run (across workloads) incurs errors that number in the thousands. Figure 4.11 shows the mapping accuracy (on Y axis) for the workloads studied using Threshold_1. The figure shows that, on average, the workloads generate acceptable outputs (with quality degradation less than Threshold_1), 98% of the time over tens of thousands of simulation runs. Sobel shows the least accuracy at 92%. Upon closer inspection, a significant number of incorrect outcomes in Sobel are caused by Segmentation Faults. As mentioned in Section 4.2.3, 99% of data bytes are designated as approximable for Sobel. This includes data that is stored in stack and heap including pointers, which can lead to egregious errors if corrupted with multiple errors over time in the execution (the data error profiles are generated by analyzing errors in a single data block – single data value read or written by a given instruction – at a time). Extending the automated error analysis to analyze errors in multiple data blocks within an execution is part of our future work.

We also study the accuracy of the approximate memory mapping generated using Threshold_2 and Threshold_3. We observe similarly high mapping accuracy (>97% and 98% on average across all thresholds) across these quality thresholds as well.

Hence, we conclude that the data error profile of applications can enable automatic extraction of non-critical (approximable) data for mapping to approximate memory.

4.4.5 Power Savings Estimate Using Flikker Power Model

We calculate the estimated power savings of mapping non-critical data to approximate memory using the methodology described in Section 4.4.3. Figure 4.12 shows the power savings observed across the studied workloads. Swaptions and LU show power savings of 22.5% and Sobel shows a power savings of 32%. Blackscholes and FFT show no power savings as the original analytical model assumes a discrete DRAM partitioning which requires at least 25% of approximable data (corresponding to 3/4 of high refresh rows) to achieve any power savings. Choosing a different approximate memory architecture to exploit fine grained approximate data or using a different criteria for selecting approximate data (such as requiring non-critical data to be approximate 90% of the time, as opposed to our strict criteria of 100%) could offer better power savings, but we leave their exploration to future work.

4.5 CONCLUSION

The application error profiles generated by the automated error analysis tools enable a fundamental understanding of the application's error characteristic. The first-order un-

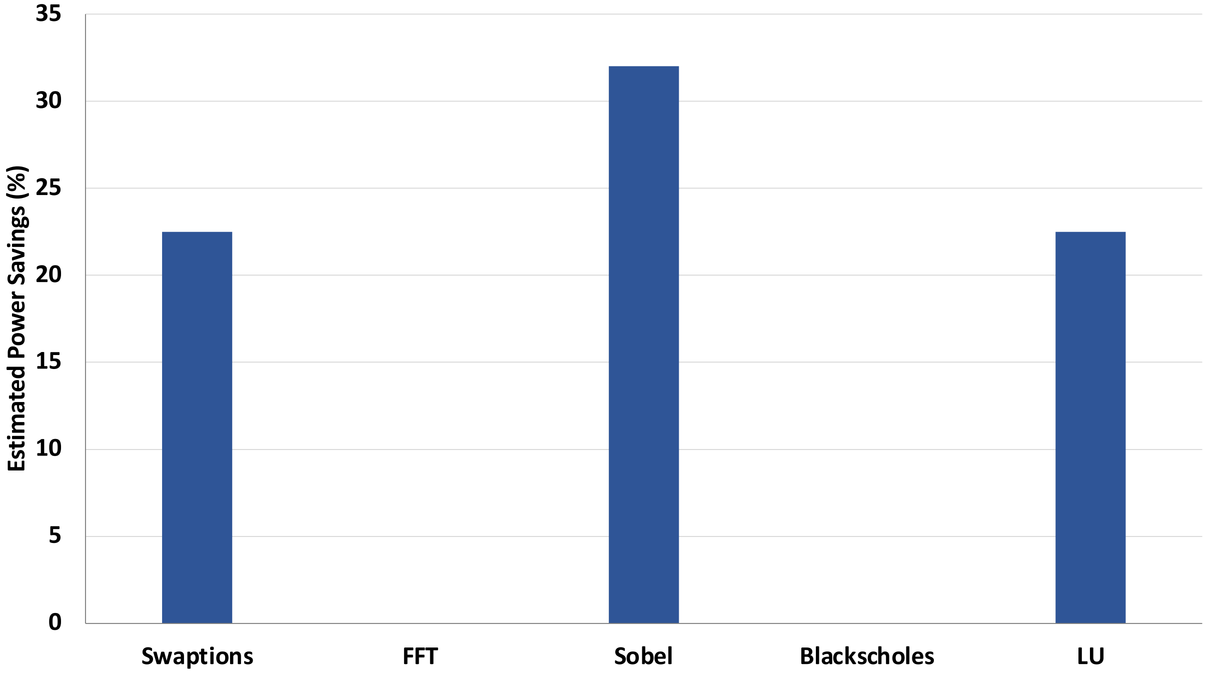


Figure 4.12: Estimated power savings using the analytical power model from Flikker.

derstanding built from the application's error profile can be used to build error-efficiency solutions that are customized to the application and user quality targets. This work demonstrates the versatility of this approach by showing two end-to-end workflows that use the application error profiles to devise customized error-efficiency solutions.

First, the application's instruction error profile is used for error-efficiency solutions targeted towards ultra-low cost resiliency to transient hardware errors. It is shown that large resiliency overhead reductions (up to 55%) are possible if the user is willing to tolerate a very small loss in output accuracy (1%) while still providing high (99%) resiliency coverage. Next, we demonstrate how the data error profile can be used to enable an approximate computing technique. Specifically, the data error profile is used to automatically (without programmer intervention) identify non-critical or approximable data (for a given user-specified quality threshold) and map them to be stored in approximate DRAM with low refresh rates (resulting in up to 32% power savings). The evaluations show that this automatic mapping to approximate memory meets the quality target set by the user 98% of the time, on average, over (tens of) thousands of executions.

Chapter 5: IMPROVING SPEED AND SCALABILITY OF ERROR ANALYSIS

The AEA tools discussed in this work significantly further the state-of-the-art in automated error analysis. But, their dependence on error injections (albeit orders of magnitude fewer than naive techniques) can still make them too slow for some practical use cases. In order to improve speed, some error analysis techniques may sacrifice comprehensiveness or coverage by only analyzing a fraction, say 99%, of error-sites in the program. While this is practical for most applications, it may be unacceptable for safety-critical applications which need 100% coverage to enable highly accurate application of error-efficiency. For example, resiliency analysis (Section 4.3.1) informed by an incomplete error profile (coverage $lt100\%$) may risk not protecting against some SDCs produced by the remaining 1% of error-sites not covered. Since the hybrid error analysis described in this work use dynamic program analysis and error injections, they are also input dependant. For robust error analysis, these techniques must be scalable across different inputs and workloads. For all of the above reasons, it thus is essential for error analyses to be be efficient and scalable.

This work presents *Minotaur*, a toolkit that shows a systematic adaptation of software testing concepts to (hardware) error analysis, thereby significantly improving the speed and scalability of error analyses. The novel insight behind Minotaur is that analyzing software for (hardware) errors is similar to testing software for software bugs; therefore, adapting techniques from the rich software testing literature can lead to principled and significant improvements in error analysis. Minotaur techniques can benefit many error analysis techniques; it is evaluated in this work by applying it to Approxilyzer [24].

Minotaur identifies, adapts, and evaluates four bridges between software testing and error analysis:

Concept 1: Test-Case Quality → Input Quality. A key concept in software testing is test-case (input) quality; i.e., an input's effectiveness in finding bugs in the target software. Several input quality criteria have been proposed in the literature, typically at the source-code level, with statement coverage as a simple and widely used criterion (Section 5.1.1). Typical error analysis techniques, for example those used for resiliency analysis [33] typically use generic inputs often developed for performance evaluation; e.g., the reference inputs in benchmark suites. Error analysis performed using these generic inputs could be sub-optimal for specific error efficiency techniques, that, say identify SDC instructions (for resiliency analysis) or for discovering approximable instructions (for approximate computing). Yet there is no accepted input-quality criterion for error analysis (or for traditional resiliency analysis or approximate computing).

This work introduces the notion of input-quality criteria for error analysis, adapts several widely used software testing criteria to the object-code level, and evaluates these criteria for error analysis targeting both resiliency and approximate computing. *Program counter (PC) coverage*, an analog of the widely used statement coverage, is found to be an effective input-quality criterion for error analysis. Intuitively, PC coverage measures the fraction of assembly instructions executed for a given input.

Concept 2: Test-Case Minimization → Input Minimization. Test-case minimization for software takes a high quality, expensive/slow test and creates a cheaper/faster test with similar high quality. Minotaur adapts minimization to error analysis by creating minimized inputs (referred to as *Min*) that are smaller and execute faster than, but have similar quality as, the reference inputs (*Ref*).

Minotaur uses a Minimizer module to systematically (using a greedy binary search) reduce the input till the Min input generated meets the specified minimization objective (speed in this case) and input quality target (100% PC coverage). Applying minimization to seven benchmarks (across multiple application domains) shows that using Min instead of Ref speeds up error analysis by 4.1X (up to 15.5X for some benchmarks) on average.

The improved analysis speed afforded by Min enables it to be analyzed more comprehensively, whereas Ref can be prohibitively expensive to analyze in its entirety [31, 33]. This more comprehensive error analysis can, in turn, lead to more accurate resiliency and approximate computing solutions. For example, resiliency analysis using Min can correctly find 96% of SDC causing static instructions (SDC-PCs) in the program whereas an incomplete (albeit high coverage at 99%) analysis using Ref only finds 64% of the SDC PCs. In the context of approximate computing, Min identifies 96% of first-order approximable instructions whereas Ref only identifies 81%. This result that Min is more accurate (by enabling more comprehensive analysis) than Ref parallels recent work from the software testing literature on bug detection [142].

Concept 3: Test-Case Prioritization → Error-Injection Prioritization.

Test-case prioritization for software systematically prioritizes test cases to find critical software failures as early as possible. Minotaur adapts test-case prioritization in two ways—prioritization of error injections for a given PC with a given input (Concept 3) and prioritization of inputs (Concept 4). Error injection prioritization can be effectively applied if the context for which the analysis is performed is known. For example, if the context is to find SDC PCs for resiliency protection then once a PC is found to generate an SDC, no further error injections are performed on that PC because it needs to be hardened (protected against SDCs) anyway. Similarly, if the context is to find approximable instuctions, then once a PC is found to generate an output of unacceptable quality (egregious outcome), then it can be classified as non-approximable and no further injections are needed on it.

Several priority orderings are explored for error injections for a given PC. Surprisingly, we find that random ordering reveals SDCs/egregious outcomes almost as quickly as an oracular best case. Further investigation shows that an SDC-PC often produces SDCs for a very large number of its injections; therefore, a random ordering quickly finds one such injection (same trend for approximation). The combination of random ordering and termination of injections on a PC after an SDC (or egregious outcome) discovery, combined with input minimization, provides an average 10.3X speedup (up to 38.9X) in error analysis targeting resiliency by employing Minotaur. Similarly, error analysis for first-order approximate instructions sees a speedup 18X (up to 55X) by using Minotaur techniques of minimization and error injection prioritization.

Concept 4: Test-Case Prioritization → Input Prioritization. Minotaur also adapts test-case prioritization across multiple inputs. For example, to find SDC-PCs as fast as possible, error analysis on the faster Min input is prioritized over the slower Ref input. Then, for higher accuracy, we can additionally perform error analysis on the larger Ref input, but only for those PCs not already classified as SDCs by Min. This prioritization of *inputs* for resiliency analyses results in finding the union of SDC-PCs across both inputs, while running on average 2.3X faster than analyzing both inputs independently in their entirety.

To summarize, Minotaur shows, for the first time, that leveraging software testing concepts for error analysis enables principled and significant benefits in speed and accuracy. While the evaluation in this work uses Approxliyzer as the underlying error 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 error analysis based on software testing, and opens up many avenues for further research. By building a bridge between software testing and (hardware) error analysis, Minotaur lays the foundation for systematically integrating (hardware) error analyses into the software development workflow.

As a working example, the rest of this study shows an application of Minotaur to Approxilyzer analysis targeting resiliency, with the goal of identifying SDC-PCs in a given application. Henceforth, for brevity, Approxilyzer analysis targeted to resiliency will be simply referred to as *resiliency analysis*. Section 5.5 discusses evaluation of using Minotaur for

Approxilyzer analysis targeting approximate computing.

5.1 BACKGROUND

This section provides an overview of the relevant software testing techniques adapted by Minotaur and a brief recap of Approxilyzer (Section 3.4).

5.1.1 Relevant Software Testing Techniques

Software testing is the process of executing a program or system with the intent of finding failures [143]. The objective of testing can be quality assurance, verification, validation, or reliability estimation. This section discusses some techniques and best practices adopted by the software testing community.

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 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 using fewer test cases [144]. The choice of criterion depends on several factors, including the size of the program, cost requirements, and criticality of failure. Some popular criteria [144], ordered from weaker to stronger, are: (1) statement coverage [145], which measures the fraction of program statements executed by tests; (2) branch coverage [143], which measures the fraction of branch edges executed; and (3) def-use coverage [144, 146], 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 [147]. The software testing literature provides an extensive analysis of various testing criteria [145].

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 [142, 148, 149, 150, 151, 152, 153]. The goal 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.

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 [148]. 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 [148].

5.1.2 Approxilyzer

Minotaur is evaluated in this work using Approxilyzer (Section 3.4). In this work, the error analysis performed in the context of resiliency with the end goal of identifying static instructions (PCs) in the application that result in a Silent Data Corruption when perturbed by errors (Section 4.3.1). While a detailed explanation of Approxilyzer methodology and its use for application resiliency is provided in Chapter 3 and Chapter 4, some of the key concepts are briefly recapped here for the convenience of the reader.

Approxilyzer is a state-of-the-art instruction-level error (resiliency) analysis tool that is fine-grained (it identifies individual SDC-PCs) and comprehensive (it analyzes nearly all instructions). Approxilyzer uses a combination of program analysis and error injections to determine the outcome of a single-bit transient hardware error occurring during the execution of any dynamic instruction—in any source or destination register bit of the instruction—of the given program and its input. The term *error site* is used to refer to the combination of the dynamic instruction instance and its register bit that incurs the error.

Approxilyzer dramatically reduces the number of required error injections to predict the error outcome for all application error sites for a given input. It systematically analyzes all error sites, and carefully picks a small subset to perform selective error injections. It uses novel error-site pruning techniques (pioneered by Relyzer [33]) to reduce the number

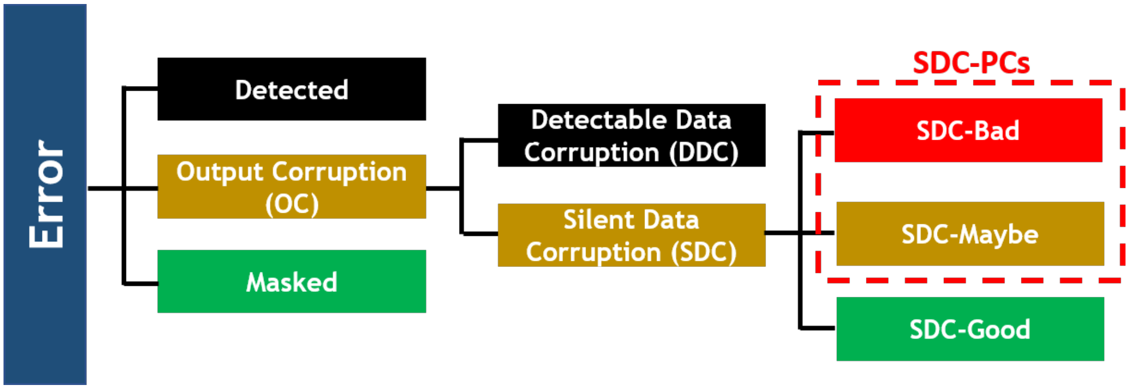


Figure 5.1: A classification of error outcomes. Only outcomes of SDC-Bad and SDC-Maybe constitute SDC-PCs.

of error-sites needing detailed study, either by predicting their outcomes or showing them equivalent to other errors. To prune error sites, Approxilyzer partitions error sites into *equivalence classes* such that the error outcome ofa single representative of each class is needed to predict error outcome for all error sites in the class. However, it is still slow, requiring millions of error injections for standard benchmarks with reference inputs [31]. Past studies, therefore, performed error injections only for the classes that contain 99% of the error sites (sorted by equivalence class size), referred to as 99% *error-site coverage*—analyzing the last 1% was deemed too expensive, because it can involve many more classes and would require many more error injections [31, 33].

Approxilyzer distinguishes error-injection outcomes as masked, detected, or output corruptions (OCs). While most prior work considers all OCs as SDCs, Approxilyzer analyzes the quality (degradation) of the corrupted outputs to further differentiate between output corruptions that are tolerable to the user from those that are not. A comprehensive list of error outcomes follows, also summarized in Figure 5.1:

- **Detected**: An error that raises observable symptoms and can hence be caught using various low-cost detectors [12] before the end of execution.

- **DDC**: An OC that is detectable via low-cost mechanisms such as range detectors applied on the output [13].

- **SDC-Bad**: An OC with very large (unacceptable) output quality degradations.

- **SDC-Maybe**: An OC that may be tolerable if the output-quality degradation is within a user-provided acceptability threshold (if no threshold is provided, all SDC-Maybe's default to SDC-Bad).

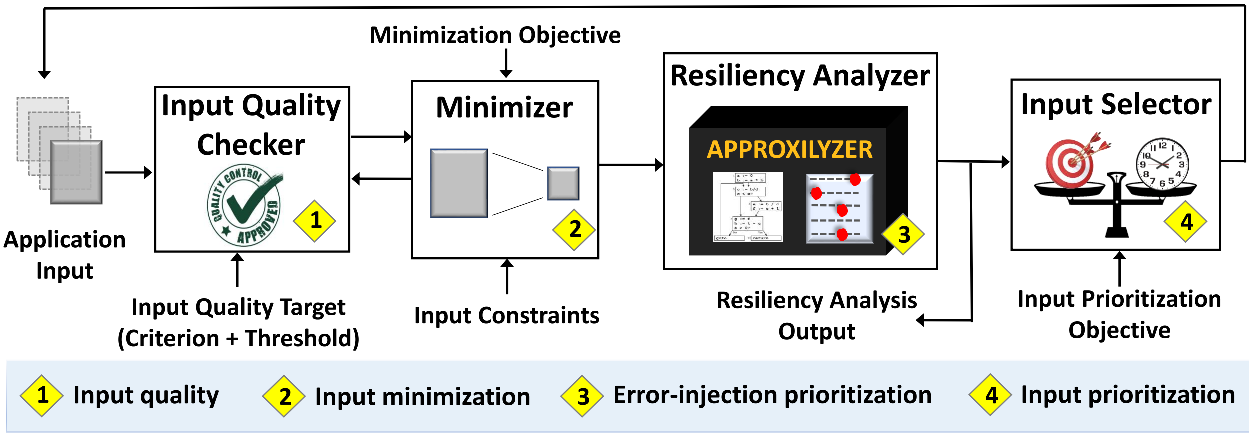


Figure 5.2: Overview of Minotaur. Approxilyzer may be replaced with another resiliency analyzer.

- **SDC-Good**: An OC that produces negligibly small (and acceptable) output quality degradations.
- **Masked**: Errors that produce no output corruption.

To identify an SDC-PC, Approxilyzer examines the error outcomes for all error sites in a given static PC. If even a single error site results in an unacceptable outcome (SDC-Bad or SDC-Maybe for quality degradations outside the acceptability threshold), the PC is classified as an SDC-PC. Because SDC-Good outcomes are tolerable, their error sites do not need hardening and do not contribute to SDC-PCs.

5.2 MINOTAUR

This section describes Minotaur, a novel toolkit for principled and efficient resiliency analysis for hardware errors. Figure 5.2 illustrates the complete system.

5.2.1 Input Quality

Ensuring that ”good” quality inputs are used for resiliency analysis increases the effectiveness of the analysis. The concept of test-case quality (Section 5.1.1) is adapted to build an *Input Quality Checker* (Figure 5.2) 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

```
// 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 5.3: Statement coverage vs. PC coverage.

some work on test coverage at the object-code level [154, 155], but it is not widely studied. Approxilyzer's 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 5.3 demonstrates the difference between using input quality criteria at the source vs. 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, statement coverage (Section 5.1.1) 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, three test (input) quality criteria are adapted 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 adapted from the source to the object code level to consider assembly-level branches and def-uses pairs, respectively.

The Input Quality Checker (Box 1 in Figure 5.2) evaluates whether a given input meets the desired *quality threshold* (e.g., 90%) for a specified *quality criterion* (e.g., PC coverage). The combination of the input quality criterion and the threshold is referred to as the *input quality target*.

5.2.2 Input Minimization

Minimizing the input size 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 5.2), 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 [145]. The Minimizer algorithm in this work is specialized for the workloads studied . 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; such an exploration is left 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 [156]) to achieve realistic inputs.

Figure 5.4 shows the algorithm (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. The current implementation in this work 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 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 Figure 5.4 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 studies (except Sobel, which takes as input an image) satisfy both characteristics, with binary search sufficing for the value exploration. The images for Sobel are reduced using the

```
$PList$: 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 $IQT$
```

Figure 5.4: Algorithm for Input Minimization.

resize utility in the ImageMagick suite [157], 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.

5.2.3 Error-Injection Prioritization

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. Box 3 of Figure 5.2 shows the application of error-injection prioritization in Minotaur's workflow.

We study the following ordering schemes for error-injection prioritization are 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.

5.2.4 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 this scheme, an *Input Selector* (Box 4 in Figure 5.2) chooses inputs for resiliency analysis according to an order specified by an *input prioritization objective.* This work chooses 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, input-prioritization can be leveraged 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).

Application	**Domain**	**Ref Input**	**Min Input**	**PC (%)**	**Branch (%)**	**Def-Use (%)**
Blackscholes [122]	Financial Modeling	64K options	21 options	100	100	99.38
Swaptions [122]	Financial Modeling	16 options 5000 simulations	1 option 1 simulation	99.91	99.23	98.42
Streamcluster [122]	Data Mining	centers = [10,20] num iterations = 3	centers = [4,5] num iterations = 1	99.97	99.77	98.67
LU [109]	Scientific Computing	512x512 matrix 16x16 block size	16x16 matrix 8x8 block size	100	100	95.56
Water [109]		512 molecules	216 molecules	99.89	99.36	99.85
FFT [109]	Signal Processing	2^{20} data points	2^{8} data points	100	100	99.59
Sobel [4]	Image Processing	100% image size (321x481 pixels)	25.25% image size (81x121 pixels)	100	100	100

Table 5.1: 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.

5.3 METHODOLOGY

5.3.1 Evaluation Infrastructure and Workloads

The error-injection infrastructure used here builds on Approxilyzer, based on simulation using Wind River Simics [110] and GEMS [123] running OpenSolaris. The workloads used are compiled to the SPARC V9 ISA with all optimizations enabled.

Approxilyzer's error model uses single-bit architecture-level errors (Section 3.3.2), which are a limited but effective [115] and realistic subset of hardware errors [101]. With resiliency becoming a first-class software design objective [158], techniques with different speed, precision, and error models are needed at different stages of software development.For example, early design stages specifically emphasize speed, so techniques using lower-level error models may be inappropriate—either too slow, not available, or eschewed to maintain generality. Thus, working with tools that use high-level models is a key application of Minotaur. Evaluating Minotaur with tools that use different error models (lower-level, multi-bit, etc.) is part of our future work.

To evaluate Minotaur, this work uses seven workloads from three benchmark suites spanning multiple application domains, summarized in Table 5.1. Column 4 lists the reference (Ref) input parameters used in this study. For five of the benchmarks—Blackscholes, Swaptions, LU, Water, and FFT—we use the same inputs as Approxilyzer(Section 3.4.2) for the reference inputs. For Streamcluster, prior evaluations [39, 73] showed that the benchmark benefits from realistic datasets (as opposed to data points generated internally by the application); hence, a dataset from the UCI Machine-Learning Repository [159, 160, 161] is used as its Ref input. For Sobel, the bird image from the iACT [124] repository is used as input.

This work chooses 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, the same quality metrics as described in Section 3.3.5 are used Blackscholes, Swaptions, LU, Water, Sobel and FFT; for Streamcluster, the maximum relative error (*max-rel-err* from Section 3.3.5 is used.

Evaluating Minotaur using the above workloads involved performing over 8.4 million error-injection experiments spanning approximately seven weeks of simulation time on a 200-node cluster of 2.4GHz Intel Xeon processors.

5.3.2 Input-Quality Criteria

Since no available tool can easily measure test coverage at the object-code level, custom tools were developed (as part of this work) using dynamic traces from Simics [110] 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, Min's coverage is measured relative to Ref.

5.3.3 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. PC, branch, and def-use coverage are measured for each Min *relative* to its corresponding Ref; e.g., if Min executes all PCs executed by its Ref, it is considered to have 100% PC coverage. Similarly, if Min exercises all branch-target and def-use pairs exercised by Ref, it is considered to have 100% branch and def-use coverage, respectively.

PC coverage is chosen 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 5.1.1). We find that the Min inputs generated using PC coverage are surprisingly effective (Section 5.4.1), and also exhibit high (but not perfect) branch and def-use coverage.

5.3.4 Accuracy Analysis

Minotaur uses input minimization to generate a Min that is a good representative of a Ref. Minotaur's accuracy for a given input is quantified 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, the SDC-PCs identified by Min and Ref are analyzed 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 (Section 5.1.2) 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.

The SDC-PCs are grouped 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 5.3.5) 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 5.3.6) will result in 100% accuracy since input-prioritization obtains the union of SDC-PCs found by analyzing all inputs (while optimizing resiliency analysis speed).

5.3.5 Error-Injection Prioritization

This work explores 38 different error-injection prioritizations using combinations of the schemes from Section 5.2.3. For BitPos, DI, and ECS schemes, both ascending (A) and descending (D) orderings are tested. Compositional schemes are also explored. 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, an Oracle best and worst case are also evaluated. 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.

5.3.6 Input Prioritization

Minotaur's Input Selector prioritizes (faster) Min over Ref. Section 5.4 shows that while Min exhibits high accuracy (Section 5.3.4), 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 5.2.4).

5.3.7 Runtime Analysis of Minotaur

The time that Minotaur takes to perform resiliency analysis on a single input is evaluated. The Input Quality Checker, Minimizer, and Input Selector (boxes 1, 2, and 4 in Figure 5.2) take negligible time compared to the resiliency analysis (Approxilyzer) time (box 3); therefore, the focus here is 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. The total runtime is estimated 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 ($t_{equiv\_class\_gen}$) [31, 33], (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.

The runtime is measured 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 [162]. The time for resiliency analysis is calculated as:

$$TotalRuntime = t_{equiv\_class\_gen} + \sum_n (I_n \times t_n) \tag{5.1}$$

where each outcome type $n \in \{masked, det, OC\}$ 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 5.1) 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. The measurements are broken down 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.

5.4 RESULTS

Minotaur's impact on a resiliency analysis tool, Approxilyzer [31], is evaluated by analyzing (1) the speedup and accuracy from a minimized input (Min) for resiliency analysis (Section 5.4.1); (2) the speedup from error-injection prioritization with early termination (Section 5.4.2); (3) the combined speedup from minimization and error-injection prioritization (Section 5.4.3); and (4) the speedup from applying input prioritization across multiple inputs (Section 5.4.4).

5.4.1 Input Minimization

Min Quality

Table 5.1 shows the Min generated by applying Algorithm **??** to each Ref, using PC coverage as the input quality criterion. Most applications show a large reduction of input parameter values in Min (column 5), which translates to faster application runtimes relative to Ref (Section 5.4.1).[1] Additionally, Min maintains very high PC coverage relative to Ref (column 6), which translates to high accuracy in finding SDC-PCs (Section 5.4.1).

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 (Section 5.4.1), while allowing Minotaur to benefit from running the faster Min (Section 5.4.1).

The last two columns of Table 5.1 show the branch and def-use coverage of the generated Min (relative to Ref) and are discussed further in Section 5.4.1.

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 5.5 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), as discussed in Section 5.1.2. By using input minimization, achieving 100% error-site coverage is no longer elusive for many applications. Figure 5.5 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,

---

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

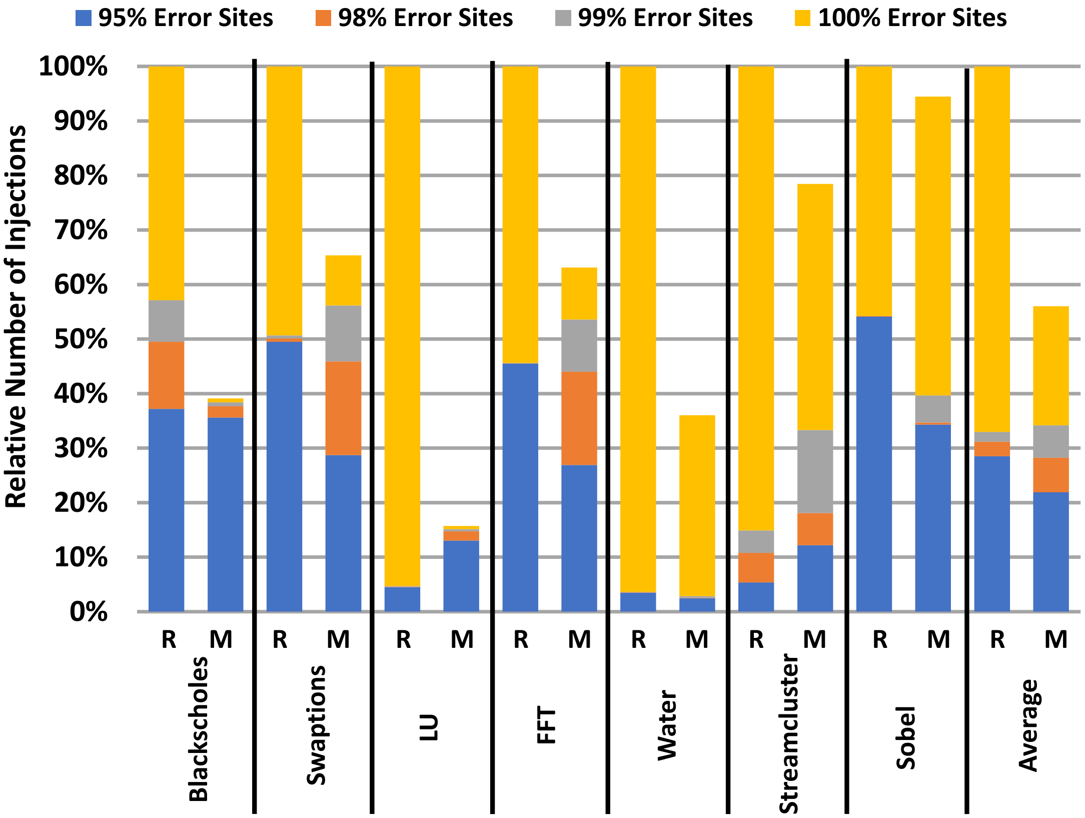


Figure 5.5: Number of error injections for different error-site coverage targets for each benchmark, relative to 100% error-site coverage for Ref (Ref100). R=Ref, M=Min.

Swaptions, LU, and FFT, and Min99 for Water, Streamcluster, and Sobel. We use Ref to refer to Ref99 for all applications.

Not only does Min require fewer error injections for most of the workloads studied, each individual injection runs faster compared to Ref. Figure 5.6 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 5.3.7).

Min injections run 2.1X faster on average[2] than Ref for all outcome types for two primary reasons. First, the application runtime itself is faster (2.3X 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 (1.8X 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 5.1). Min's smaller output matrices allow for faster post-processing, further speeding up the resiliency

---

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

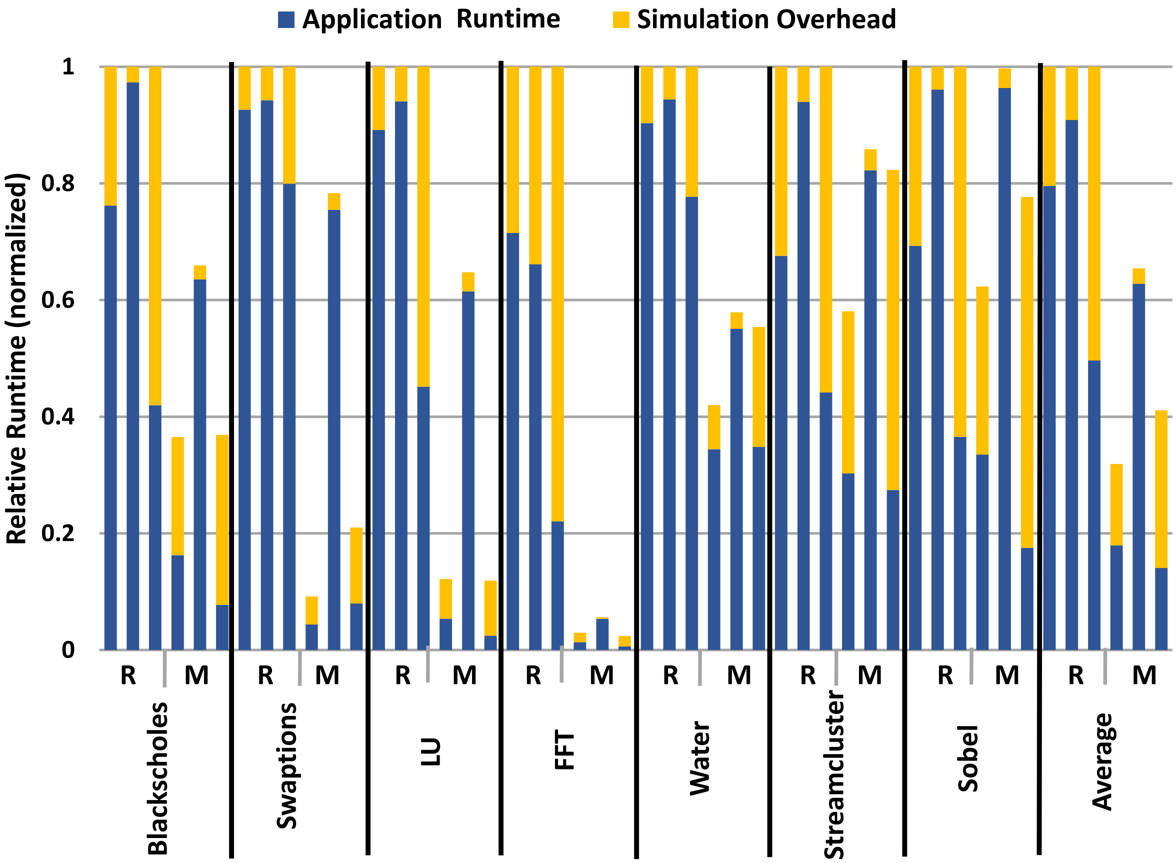


Figure 5.6: Average runtime per injection, normalized to Ref. Each set of three bars represents (from left to right) Masked, Detected, OC runtime (Section 5.3.7), divided into application runtime and simulation overhead. R = Ref and M = Min.

analysis relative to Ref for these applications.

Figure 5.7 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 4.1X speedup for Min over Ref on average (up to 15.5X 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.

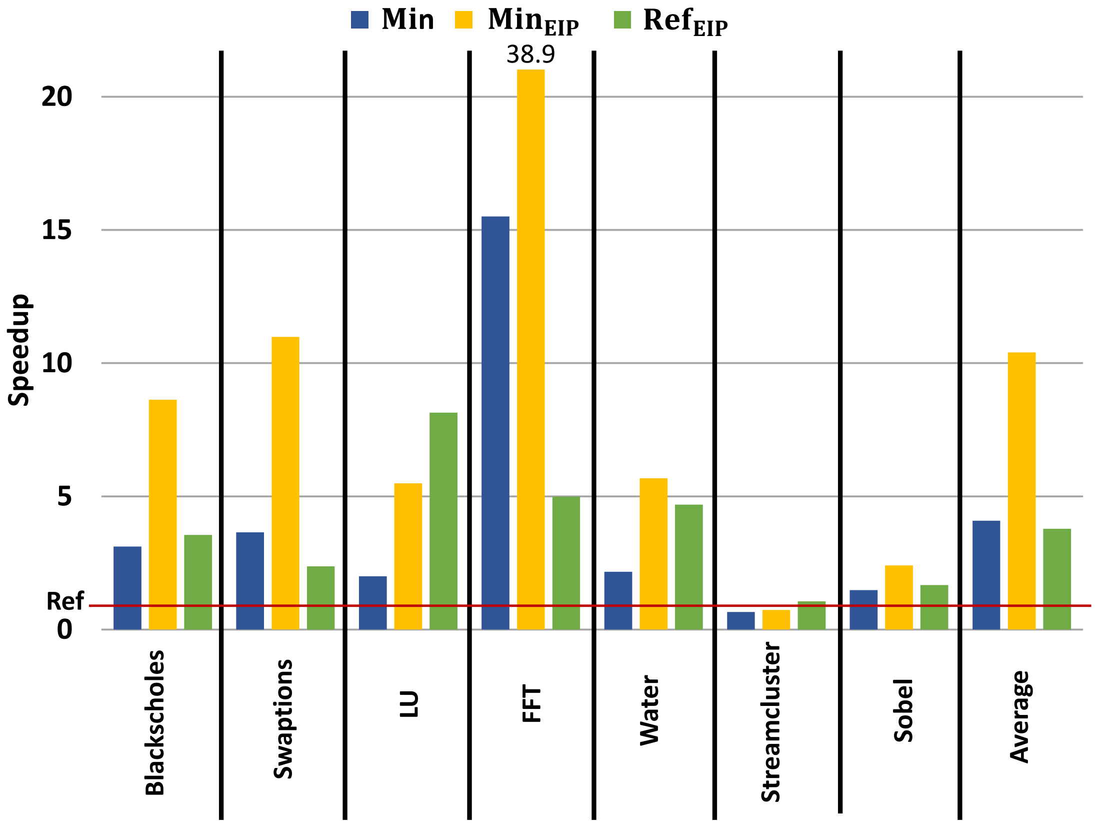


Figure 5.7: Min, Min$_{EIP}$, and Ref$_{EIP}$ speedup relative to Ref.

<u>Minimization Accuracy</u>

Figure 5.8 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 5.3.4). 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 5.9 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 (Section 5.4.1), generally explore fewer distinct PCs than Min at lower error site coverage targets. Figure 5.9 shows that, on average, for 99% error-site coverage (sorted by equivalence class 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

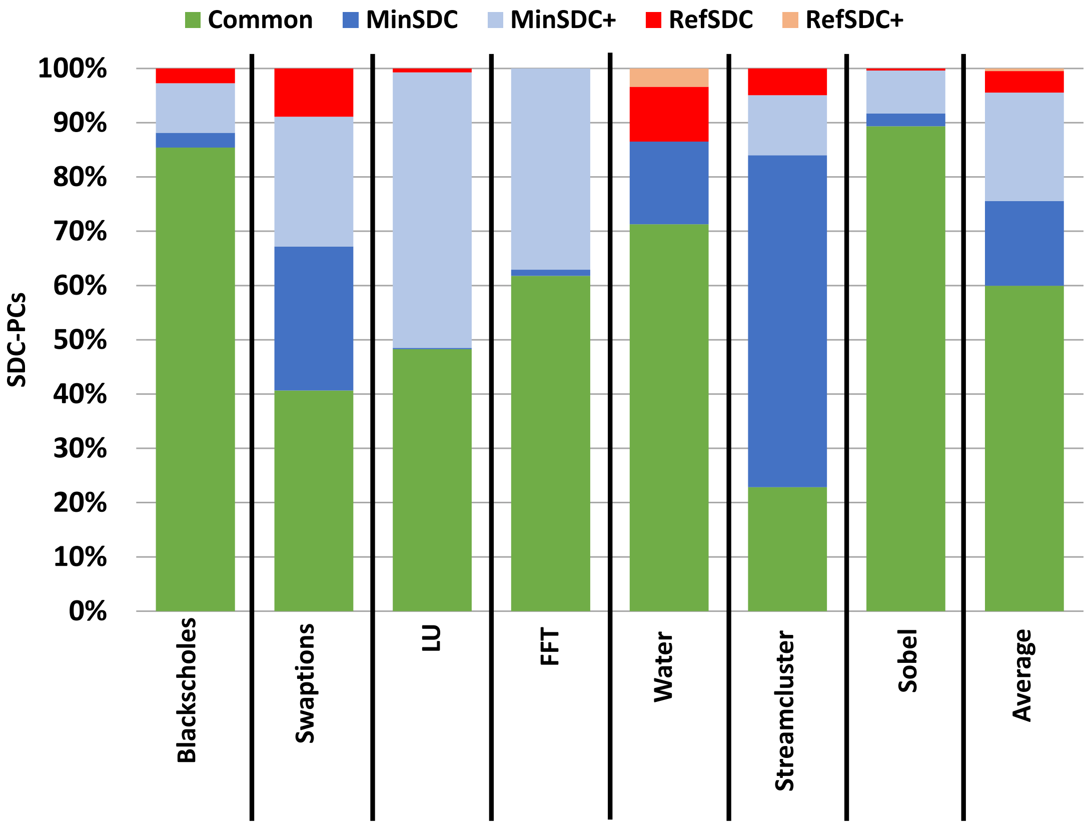


Figure 5.8: Min and Ref accuracy. The Y-axis represents all SDC-PCs found by Min or Ref in an application.

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, a majority of the mismatches are found to 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 (the sum of Common, MinSDC, and MinSDC+ categories) while Ref finds only 64% (the sum of Common, RefSDC, and RefSDC+) of these SDC-PCs.

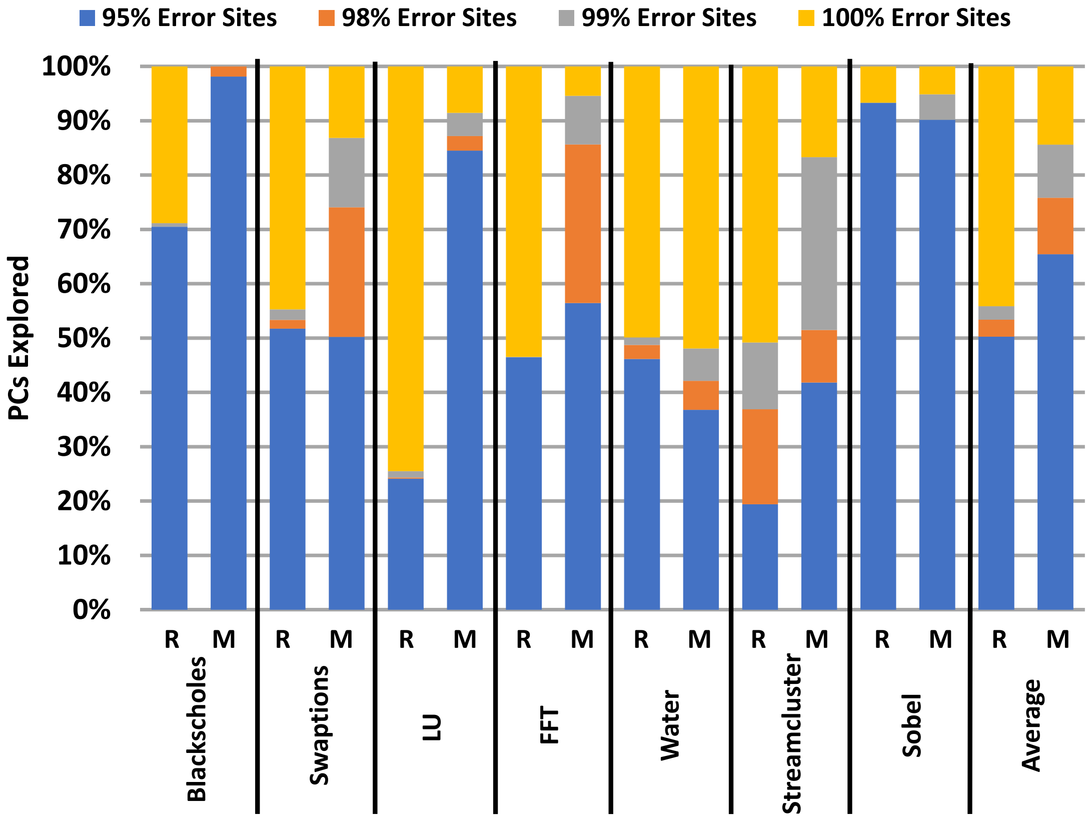


Figure 5.9: Percentage of PCs explored for different error-site coverage targets. R = Ref, M = Min.

Improving Min Selection Criteria

The branch and def-use coverage of Min (relative to Ref) is examined to understand if these stronger criteria could have been used to generate an alternate Min that provides higher accuracy than PC coverage. Table 5.1 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, the branch-target and def-use pairs that were in Ref but not in Min are identified in order to determine if they were responsible for the RefSDCs in Figure 5.8. 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

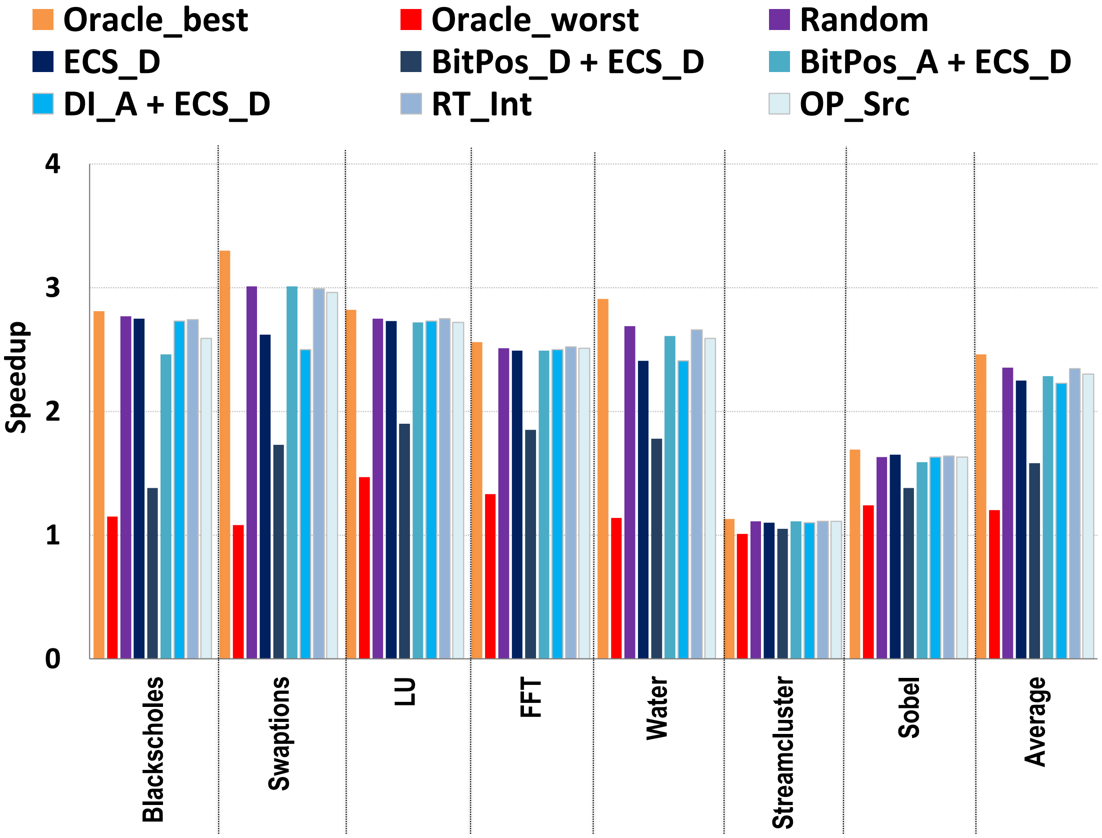


Figure 5.10: Min speedup with error-injection prioritization.

the stronger criteria would add further accuracy. Such an analysis and exploration of even more complex input quality criteria (e.g., path coverage) is left to future work, given that the results presented here already show that PC coverage provides an excellent sweet spot for simplicity, performance, and accuracy.

5.4.2 Error-Injection Prioritization

38 different error injection prioritization schemes are studied in this work (Section 5.3.5). For brevity, results for only the 7 most effective schemes are shown, in addition to the oracle best-case and oracle worst-case schemes.

Figures 5.10 and 5.11 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 2.4X and 3.8X for Min and Ref (upto 3X and 8.1X), respectively, while also being very close to the oracle best-case.

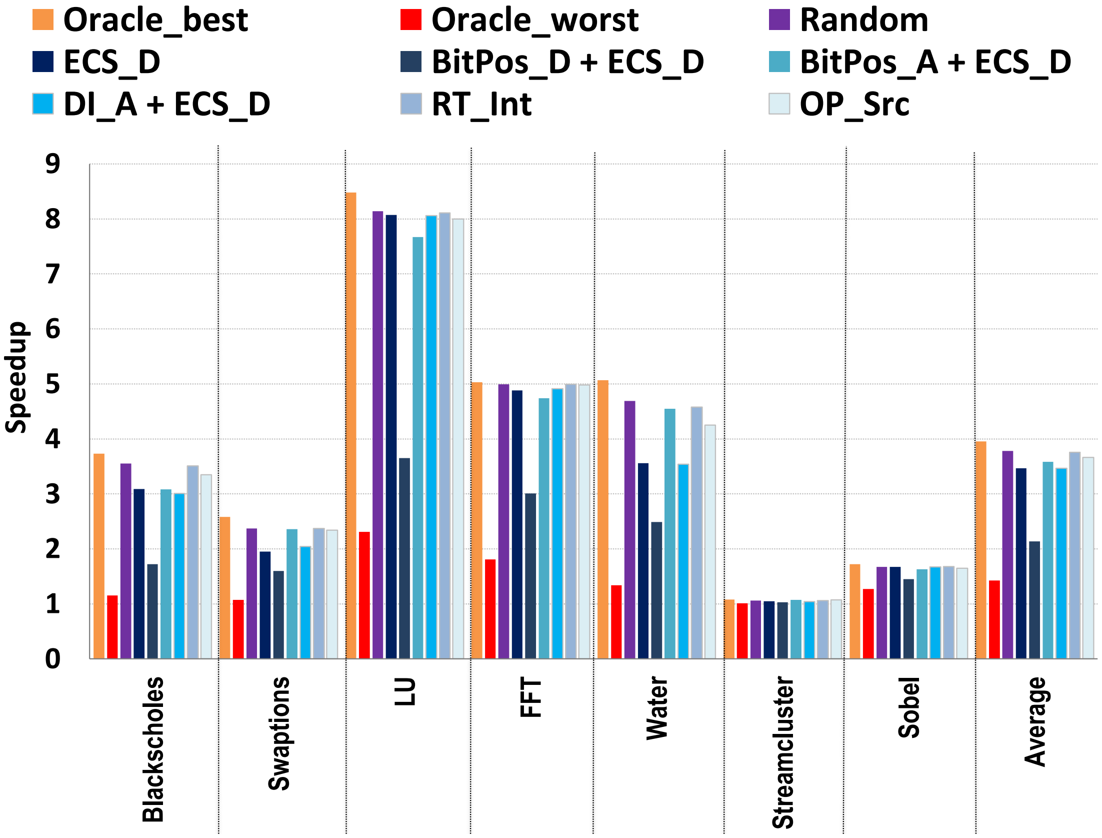


Figure 5.11: Ref speedup with error-injection prioritization.

To understand the surprising result that Random performs the best, Figure 5.12 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 5.12 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, an interesting insight is uncovered – 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, Random error injection prioritization scheme is chosen for the remainder of the evaluations in this chapter.

5.4.3 Minimization Plus Injection Prioritization

This section discusses the benefits of combining input minimization with error injection prioritization. Figure 5.7 shows the speedup in resiliency analysis, relative to Ref, from

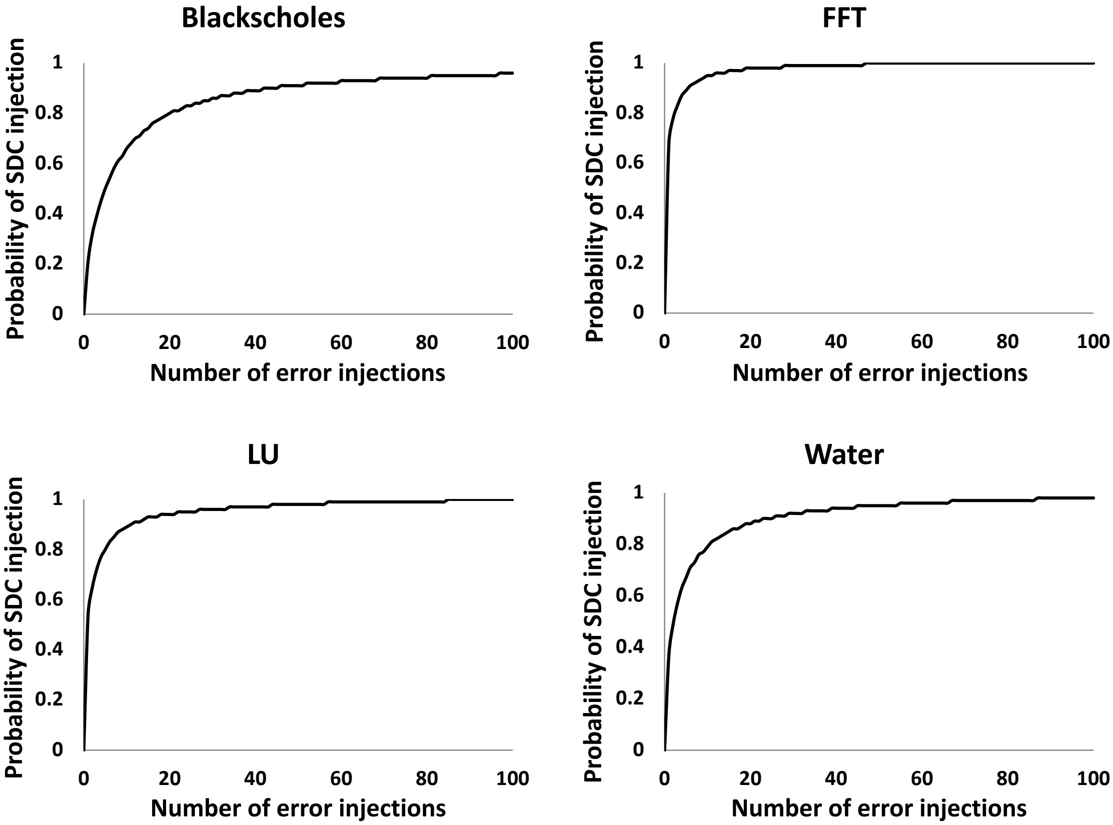


Figure 5.12: Cumulative probability (Y-axis) of picking an SDC-causing error injection within the first n injections (X-axis) for an SDC-causing PC.

(1) using Min (discussed in Section 5.4.1), (2) using Min with error injection prioritization (referred to as Min_{EIP}), and (3) using Ref with error injection prioritization (Ref_{EIP}). As previously discussed in Section 5.4.1, using only Minotaur's input minimization optimization for resiliency analysis provides a 4.1X average speedup (up to 15.5X) compared to Ref (first bar for each application in Figure 5.7). Combining Minotaur's input minimization optimization with error injection prioritization results in an average speedup of 10.3X (up to 38.9X for FFT), relative to Ref. In contrast, Ref_{EIP} observes only a 3.8X average speedup (up to 8.14X for LU) relative to Ref (third bar for each application in Figure 5.7 and also discussed in Section 5.4.2).

Recall that the accuracy of Min_{EIP} is the same as that of Min (Section 5.4.1). 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%.

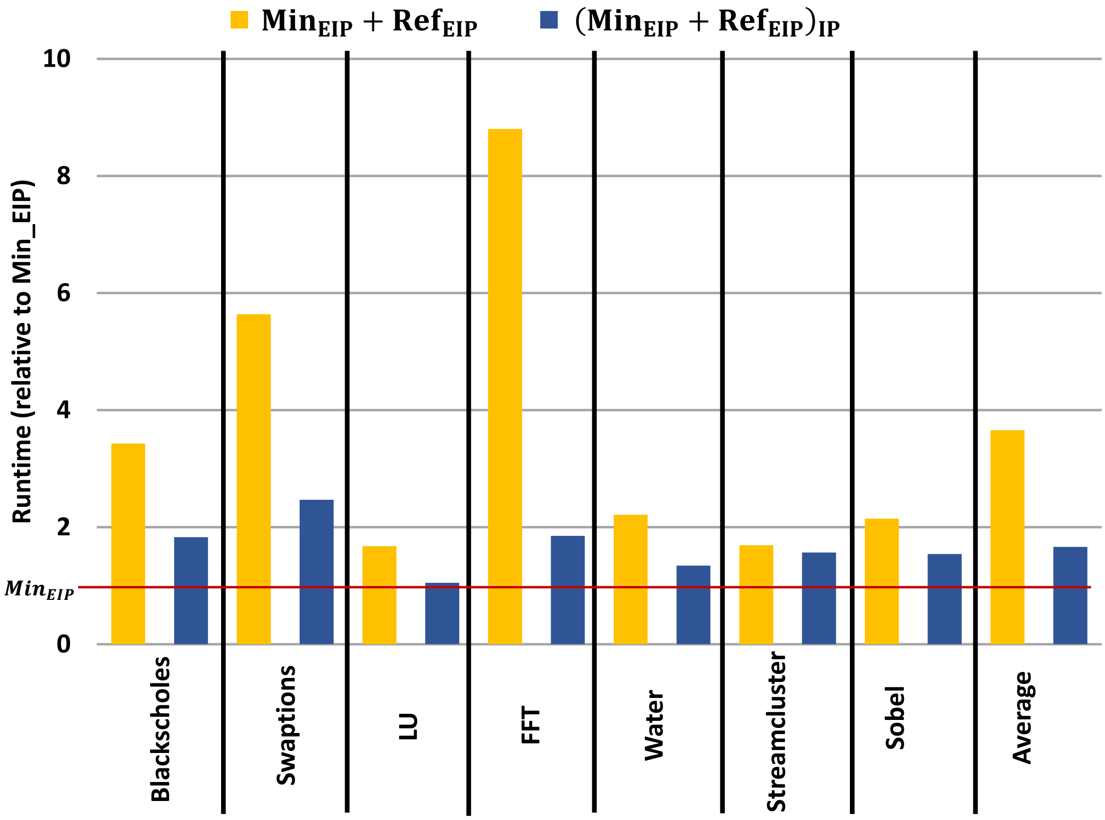


Figure 5.13: Resiliency analysis time for analyzing both Min_{EIP} and Ref_{EIP}, without and with input prioritization, normalized to analysis time for only Min_{EIP}.

5.4.4 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 5.13 shows the runtime of analyzing both Min_{EIP} and Ref_{EIP}, without and with input prioritization, normalized to the runtime of Min_{EIP} (Section 5.4.3).

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 5.4.3), followed by Ref_{EIP} *but only for PCs not identified as SDC-PCs* by Min_{EIP}. Thus, input prioritization requires the second input (Ref_{EIP} in this study) to run for only a fraction of the original resiliency analysis time.

Figure 5.13 shows that without input prioritization, Min_{EIP} + Ref_{EIP} runs 3.7X slower than Min_{EIP}. Using input prioritization $((\text{Min}_{EIP} + \text{Ref}_{EIP})_{IP}$ in the figure) brings the analysis time to only 1.6X slower than Min_{EIP}. Thus, leveraging input prioritization allows Minotaur to analyze both inputs 2.3X 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.

5.5 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.

5.5.1 Extension to Approximate Computing

Approxilyzer analysis can also be used to target approximate computing. 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. The following user-defined thresholds are used here: 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. The same Min and Ref inputs as in Table 5.1 are used, and random error injection prioritization with early termination is applied (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 4.1X speedup compared to Ref, due to Min's smaller size (Section 5.4.1). 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 4.4X on average, while Ref shows an average speedup of 5.53X. Combining the two optimizations, Min_{EIP} shows an average speedup of 18X compared to Ref for approximate computing analysis.

An accuracy metric similar to that in Section 5.3.4, is 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%.

5.5.2 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 6.8X over Ref for the same PCs and with 100% accuracy.

5.6 CONCLUSION AND FUTURE WORK

This work presents Minotaur, a toolkit to improve the error analysis by leveraging concepts from software testing. Minotaur adapts several concepts from software testing for software bug detection to error analysis (for resiliency and approximate computing): 1) identifying test-case quality criteria, 2) test-case minimization, and 3) two adaptations of test-case prioritization. Minotaur is evaluated on the error analysis tool, Approxilyzer. Minotaur's single-input techniques speed up Approxilyzer's resiliency analysis (or approximate computing analysis) by 10.3X (18X) on average while significantly improving SDC-PC detection accuracy (96% vs. 64% on average for resiliency, and 96% vs. 81% on average for approximate computing) for the workloads studied. Further, Minotaur presents a technique, *input prioritization*, which enables finding SDC-PCs across multiple inputs at a speed 2.3X 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 interesting to explore more input quality criteria (such as path coverage, loop coverage, or state coverage [145]) as well as develop new quality criteria geared specifically towards resiliency (e.g., criteria derived from ACE bits [78] or PVF [15]) or towards approximate computing (e.g., using parameter range coverage). Employing 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 is another future direction. The Input Selector can also be improved by tuning analysis speed vs. accuracy for multiple Refs and Mins with variable input quality thresholds.

To widen the applicability of Minotaur, it can be applied 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.

The end goal of this line of research is a seamless integration of (hardware) error analysis 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 error-efficiency, making fast and accurate error analysis techniques such as Minotaur even more important.

Chapter 6: RELATED WORK

In this chapter, we discuss related works that are pertinent to the contributions of this thesis.

6.1 AUTOMATED ERROR ANALYSIS

This work develops a suite of automated application-level error analysis tools – Approxilyzer, gem5-Approxilyzer and DataApproxilyzer (collectively referred to as the AEA tools) [25, 26, 31] – that can automatically extract the error characteristics of applications (Chapter 3). The AEA tools developed in this work significantly advance the state-of-the-art in error analysis to meet all the six requirements of accuracy, precision, comprehensiveness, automation, generality and low-cost.

While error analysis techniques that satisfy all six of the above requirements posed a significant research challenge before this work, researchers have made progress by relaxing some of these requirements. For example, one class of techniques rely on empirically introducing a small (statistically determined) sample of errors during a program's execution (referred to as *error injection*) and observing the resultant behaviour [22, 58, 60, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 99, 101, 104, 105, 106, 131, 163]. These techniques aim to provide statistical averages or probabilistic bounds of program behaviour under errors; they cannot comprehensively and accurately guarantee impact on output quality for the large majority of errors that are left out during sampling.

Another class of techniques leverage *program analysis* of error-free execution to understand how errors in data may propagate to affect program output [15, 75, 76, 77, 78, 79, 80, 81, 82]. The widely used ACE [78] analysis is often used to measure the Architectural Vulnerability Factors (AVF) [75, 76, 77, 78, 79] of hardware structures. PVF [15] isolates purely (program or software dependent) architecture-level vulnerabilities in the AVF; ePVF [80] 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 [81, 164]. Shoestring [82] uses a compiler analysis to identify vulnerable program locations. While fast (low-cost), such techniques cannot accurately and precisely model an error's impact on execution since they use information from an error-free execution.

The AEA tools developed in this work use a hybrid technique of program analysis and (relatively) few error injections. The techniques employed by the AEA tools build upon the error equalization and pruning techniques developed by prior work called Relyzer [33].

Relyzer analyzed single-bit errors in integer instructions of programs to identify which errors lead to silent data corruptions (SDCs). While Relyzer determined *if* an error affected the final output, the AEA tools can precisely determine *how* an error affects output by quantifying the output quality produced. Furthermore, the AEA tools can precisely quantify the impact of errors for a wider range of errors – integer, floating point, instruction, data, single and multi-bit.

There are other works in the literature that combine program analysis with selected error-injection campaigns. MeRLiN [103] 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 [165] uses error injections in static instructions to build an input-dependent model on top of Trident's [166] error propagation analysis to predict the instruction's SDC vulnerability. Other works have attempted fault equalization for GPU [167] and Neural Network resiliency [168]. The analysis technique used by the AEA tools, is also a hybrid technique, but its primary goal is not a statistical average or probability—it is to determine precisely if/how an error in any specific instruction or data will impact the final output.

Relationships between errors in program and effect on output has also been widely studied in the fields of software engineering[169, 170] and formal methods[171, 172]. Error analysis employed in this work differs from these methods since the goal is not to find buggy code or derive error bounds; rather the goal is to precisely determine the output quality produced in the presence of errors caused by underlying hardware faults.

As part of the AEA tool-suite, this work develops gem5-Approxilyzer which is an open-source error analysis tool built using the open-source simulator gem5 [83]. Other error analysis tools have used gem5 as their base simulator. For instance, MeRLiN [103] uses GeFIN [173] (which is also a built on top of gem5) to simulate micro-architectural error injections. GemFI [174] is another error-injection tool that operates at the micro-architectural level, is built on top of gem5 and supports both Alpha and x86 ISA. Other error-injection tools, such as LLFI [104], analyze applications at the compiler intermediate representation (IR) level. IR is ISA-independent by design, so such an analysis is independent of the hardware architecture. However, there may be a loss in error site accuracy because IR still requires additional transformations before producing the assembly [175]. FAIL* [105] performs ISA-level analysis. FAIL* also uses gem5 and supports ARM but is limited to one pruning technique: def-use analysis.

6.2 AUTOMATED ERROR ANALYSIS TO CUSTOMIZED ERROR EFFICIENCY

This work demonstrates the versatility of the automated error analysis approach, by showing how the automatically generated application error profiles can be used to devise different (hardware and software) error-efficiency solutions – from low-cost resiliency to approximate computing – that can be customized to user and system requirements with minimal programmer intervention (Chapter 4). We undertake a discussion of related works below.

6.2.1 Approximate Computing

Many techniques have been proposed that leverage approximate computing at the level of software [10, 39, 40, 41, 42, 43, 44, 84, 85, 86, 87, 88], programming languages [2, 3, 4, 5, 17, 19, 45] and hardware [6, 7, 8, 9, 10, 11, 89, 90, 91, 92, 93, 94, 95, 96, 97] for improved performance, energy, or reliability. The techniques described in this work is orthogonal to these and the application error profile (generated by the AEA tools) can potentially be used to provide approximation hints to all of these techniques.

Programming language support, such as that in [2, 3, 4, 5, 45] helps programmers abstractly express approximations and check program correctness at the cost of increased programmer burden. Recent frameworks [17, 19, 176] build on these languages to automatically identify approximate regions of the code while providing some statistical [17] or probabilistic [176] 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 the AEA tools developed in this work are complementary to these technique and can be used as a front end plugin to these frameworks. A concurrent work [43] 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, the AEA tools provide a general framework to study approximation at the fine granularity of single instructions or single data byte.

SAGE [41] automatically generates approximate kernels for GPUs but like other methods [40, 177] uses an online mechanism to catch unacceptable quality degradation in a reactive fashion. On the other hand, the error profile generated by the AEA tools provides offline output quality information. Techniques such as [44] control output quality constraints by tuning various knobs in an approximate program. The AEA tools solves the problem of identifying approximate code/data and as such is an orthogonal technique.

6.2.2 Resiliency

There exists a large body of work on resiliency analysis that does not assume that output quality degradation is allowed [13, 15, 33, 56, 57, 82, 107, 132, 133, 135, 136, 138]. We will focus here on works that allows quality degradation.

The idea of identifying unacceptable output corruptions for selective reduced-cost resiliency protection has previously been explored. A combination of error injections and static analysis is used in [71] to identify Egregious Data Corruption (EDC) prone code and data segments in computations that can then be protected by detector placements. IPAS [178] uses machine learning to identify and protect only those Silent Output Corruptions (SOC) instructions that alter the output of scientific codes. Khudia et al. [179] 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. The application error profiles generated by the AEA tools classify error outcomes into categories based on approximation potential and predicts the impact of errors in individual instructions (and data) 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. [180] study Electrocardiogram (ECG) monitoring wireless body sensor nodes and trade-off inaccuracies inherent to the domain to achieve resiliency overhead savings. The AEA tools also exploits accuracy loss for resiliency overhead savings but does so in a manner that can be used by any general-purpose program.

6.3 MINOTAUR

Minotaur is the first work to systematically adapt and apply software testing techniques for fast and effective error analysis (Chapter 5). Section 5.1 describes the key background related work from software testing. Other related works are discussed here.

6.3.1 Concepts similar to Minotaur

The most directly related works from other domains with similarities to different concepts in Minotaur are discussed here. IRA [156] 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 [168] 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 [32, 181, 182, 183]. A recent example is SnaPEA [181] where convolution operations are terminated early if their output is predicted to be zero.

MinneSPEC [184] aims to provide reduced input workloads to improve performance (usually runtime of applications), which differs from our objective of uncovering SDC causing instructions.

6.3.2 Error Analysis Techniques:

As discussed in detail in Section 6.1, different error analysis techniques have been proposed in the literature. These include resiliency analysis techniques that employ error-injections [58, 60, 99, 101, 104, 105, 106, 131, 163], program analysis [15, 75, 76, 77, 78, 78, 79, 80, 81, 82, 164] or a hybrid combination [33, 103, 165, 166]. Criticality-testing [69, 70, 71, 72, 73, 74, 185] of approximate computations is another important analysis technique for many domains.

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.

Minotaur can potentially be applicable to other hardware platforms as well. Although this work focuses on CPUs, recent resiliency analyses on GPUs [67, 167, 186], for example, can potentially benefit from the concepts of Minotaur to improve runtime and/or accuracy.

6.4 ERROR-EFFICIENCY FOR VIDEO SUMMARIZATION APPLICATION

We undertake a motivational study to demonstrate the effectiveness of error-efficiency techniques for edge-computing applications used in Unmanned Aerial Vehicles or UAVs (Chapter 2). Specifically, we study a state-of-the-art Video Summarization (VS) application (developed at IBM Research) that constitutes key end-to-end video and image analytics aboard UAVs. In this first-of-a-kind work that studies the effect of approximations on system resiliency, we show that software approximations to the VS application yields significant energy and performance without degrading the overall resiliency of the system. In this section, we discuss other works related to this study.

6.4.1 Error-Efficiency Techniques

Different error-efficient techniques (related to resiliency and approximate computing) have been discussed in Section 6.1 and Section 6.2. Section 2.1.2 discusses many trends in approximate computing. A related area is analysis that performs criticality testing [70, 72] and works that take advantage of soft computations - resilient code regions that result in tolerable output corruptions, when perturbed by errors - to reduce resiliency overheads in approximate environments [24, 69, 71, 187]. To the best of our knowledge, this work is the first that directly measures the resilience of approximate algorithms.

In [188] Thomas *et al.* propose the term *EDC* describing outcomes that deviate significantly from the error-free outcomes of an application. Based on heuristics learned from EDC characterization, they propose a detection mechanism to identify variables and locations to protect against EDC. We propose a novel quantitative metric for EDC evaluation on a complete video stitching algorithm and approximation algorithms to achieve improvement in energy efficiency and performance without significant loss in end-quality.

6.4.2 Computer vision for UAV-based mobile cognition

Extensive research has gone into image-stitching algorithms in the field of computer vision. In [189], Szeliski describes various algorithms for aligning and stitching images into seamless 2D photo-mosaics. Various state-of-the-art algorithms to handle and summarize video content captured on-board a UAV-based processor, have been described in [49]. Rane et al. [190] proposed a method to evaluate mosaic quality using maximum information retrieval. The method uses the similarity between the stripes of the mosaic and the original frames to evaluate performance of mosaicking methods. The videos in the VIRAT dataset contain both translational and rotational movements. However, unlike the algorithm evaluated in our study, this method works when camera has only translational movement. The evaluation method proposed by Camargo *et al.* [191] uses the distances between the corresponding keypoints in all frames after the mosaic is generated. This method is used to compare different optimization methods for parameter estimation, but does not consider the image distortions caused by error injection. Another work [192] empirically evaluates the detectability of objects of interest for human observers when temporally local mosaics are applied on the live aerial video, but cannot provide quantitative evaluation for error injection. Paalanen *et al.* [193] proposed a method to evaluate the mosaic quality using ground truth data. However, ground truth data can only be obtained in synthetic datasets. Since the dataset that we use for the evaluations in our study is a real-word one, determining the ground truth is difficult. El-Saban et al. [194] use human eye to measure precision/recall of the mosaic quality of image pairs. However, this method too cannot provide scientific measurement of the distortion caused by error injection.

Chapter 7: CONCLUSION AND FUTURE WORK

Error-efficient computing has the potential to be a key enabler for many emerging application domains – Edge Computing, AI, Robotics, etc. – with strict power, performance and reliability requirements. However, despite its promise, the lack of general methodologies and excessive programmer burden have limited its widespread adoption. This work aims to democratize error-efficient computing by building systematic methodologies that remove excessive programmer burden and enable the adoption of error-efficient computing.

The automated application-level error analysis tools developed in this work – Approxilyzer, gem5-Approxilyzer and DataApproxilyzer – are the first-of-their-kind that can quantify the impact of billions of individual errors in a program's computation and data on its final output quality. The comprehensive application error profiles (generated using automated error analysis) can not only improve fundamental understanding of how applications/systems behave when perturbed by errors, they can facilitate users to tune the trade-off between output quality with other system benefits in an error-efficient environment while providing output quality guarantees. Two such proof-of-concept workflows are demonstrated that use the application error profiles to devise customized error-efficiency solutions – targeting approximate computing and low-cost resiliency to hardware errors – that meet user/system requirements and quality targets.

In order to facilitate the use of error analyses within the computing stack, a framework called Minotaur is developed to significantly improve the speed (up to 55X) and scalability (across multiple workloads and inputs) of error analyses. By showing a principled adaptation of software testing techniques to (hardware) error analysis, Minotaur takes the first step towards the goal of integrating (hardware) error analysis into the standard software development and testing workflow.

While this work shows the promise of automated error analysis techniques that can potentially be integrated within a software development framework, there is still a long way to go towards the research vision of automated full-stack methodologies that can provide optimal system-wide error-efficiency solutions for emerging applications. The remainder of this section describes some of the future work towards that vision.

7.1 ERROR-EFFICIENCY AT SCALE FOR EMERGING COGNITIVE APPLICATIONS

The work in this thesis has thus far shown fast, comprehensive and automated error analyses for single-threaded CPU workloads from different domains (image processing, data mining, scientific computing, etc.); precisely analyzing these workloads in their entirety was a challenge before this work. Performing scalable and efficient automated error analysis for large multi-threaded applications is a natural next step. It is especially interesting to analyze cognitive applications – such as image processing, autonomous driving, virtual/augmented reality and robotics – that aim to extract context/insight from vast quantities of data. These applications heavily use sensory signals (image, audio, etc.) that can inherently tolerate inaccuracies in data and/or computation and hence, are natural candidates for error-efficient computing. Error-efficiency techniques applied to these domains today are largely empirical and focused on single/few techniques which is sub-optimal and may leave additional sources of efficiency untapped. An optimal solution that considers different combinations of error-efficiency techniques (customized to system/user specifications) is an intractable optimization problem today; automatic, scalable and comprehensive error analysis of these applications can provide insights and be the first step towards a systematic methodology to extract maximum resource efficiency for emerging domains.

7.2 ERROR-EFFICIENCY FOR PERFORMANCE, ENERGY AND BEYOND

The output of automated error analysis (for a given application) is a *comprehensive application error profile* that lists each error (for the error model under consideration) that can perturb the program's execution along with its corresponding impact on output quality. This work shows a few examples (targeted to approximate computing and hardware resiliency) of how these comprehensive error profiles can enable customized error-efficiency, but this is just a beginning. Exploring other use-cases for application error profiles, such as selective ECC deployment, mixed dynamic precision or criticality testing for emerging storage solutions like non-volatile memories (NVMs) is an interesting future direction.

Error-efficiency techniques have largely been focused on trading functional correctness, in the form of output inaccuracies, to improve power/energy or performance. But the promise of error-efficiency and automated error analysis can potentially be extended to other domains that deal with anomalies in computing behavior; for instance, efficient protection from fault attacks, analyzing interference behaviour of secure/private data being computed on erroneous substrates, using application error profiles to assist with debugging during hardware validation, etc. Another interesting direction is to explore if the error profiles can be leveraged to design more efficient accelerators that exploit error-efficiency opportunities [195, 196, 197].

7.3 ERROR-EFFICIENCY ACROSS THE COMPUTING STACK

An ideal error-efficient solution should take any application, along with user inputs (like quality of service guarantees or optional domain-specific information) and map it to optimal combinations of hardware/software error-efficiency techniques. To realize this, we need integrated, full-stack solutions that treat error-efficiency as a first class metric at each layer of the compute stack. This work thus far has been at the application layer of the stack and it has shown how error-efficiency can be integrated into the software development workflow. Other researchers have made progress at different layers of the stack – hardware/software error-efficiency techniques, schedulers, optimizers, compilers, programming languages, etc. However, we are missing the right abstractions to enable integration across the different layers.

We are also seeing a trend towards specialization and heterogeneity at different scales. This has been a boon for error-efficiency techniques that have leveraged hardware/software co-design to extract maximum compute efficiency for specialized domains. However, it has made the task of developing general full-stack error-efficiency methodologies, that still retain the flexibility to specialize/customize at different layers, very challenging. Going forward, it will be interesting to explore the following fundamental questions that need to be answered for error-efficiency solutions across the compute stack: What are the right metrics to quantify an application's error characteristics? What is the right abstraction for programming languages to express error-specifications in inputs, return values, instructions and data? How can compilers encode/translate dynamic error-efficiency opportunities of applications? Can the heterogeneous error characteristics of hardware be abstracted as standardized knobs that can be tuned at the direction of software? Answers to these questions will require a cross-disciplinary solutions.

7.4 SYSTEM WIDE ERROR-EFFICIENCY OVER HETEROGENEOUS SOURCES OF ERRORS

As mentioned in the beginning of this thesis, part of the larger research vision is to develop a *principled and unified methodology for analyzing different sources of errors.* This is an important but extremely challenging problem since modern workloads can encounter a wide variety of heterogeneous components that have varied sources of errors, each with a different error model and no established methodology for translation between them. It is unclear how different error-efficiency techniques across a range of devices will interact, combine and complement or negate each other to provide end-to-end application and/or system benefits. System wide solutions that can holistically optimize for error effects arising from heterogeneous compute, data and networks will require a systematic methodology for incremental and compositional error-efficiency analyses over a range of workloads, devices and error models. Conceptual intuition gained by this work postulates that, albeit hard, this is a problem that can be solved. For example, this work provides an intuition for how the problem of finding the errors in compute that lead to output corruptions can be tackled by systematically performing incremental analysis over increasingly complex error models. Formalizing these concepts will be crucial for establishing a systematic methodology for system-wide error-efficiency.

BIBLIOGRAPHY

[1] P. Stanley-Marbell and M. Rinard, “Error-efficient computing systems,” *Found. Trends Electron. Des. Autom.*, vol. 11, no. 4, p. 362–461, Dec. 2017. [Online]. Available: https://doi.org/10.1561/1000000049

[2] M. Carbin, S. Misailovic, and M. C. Rinard, “Verifying quantitative reliability for programs that execute on unreliable hardware,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, ser. OOPSLA ’13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509546 pp. 33–52.

[3] 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. [Online]. Available: http://doi.acm.org/10.1145/2714064.2660231

[4] 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.

[5] 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 *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993518 pp. 164–174.

[6] 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.

[7] J. Han and M. Orshansky, “Approximate computing: An emerging paradigm for energy-efficient design.” in *ETS*. IEEE Computer Society, 2013. [Online]. Available: http://dblp.uni-trier.de/db/conf/ets/ets2013.html#HanO13 pp. 1–6.

[8] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, 2012, pp. 449–460.

[9] J. San Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, “Doppelganger: A cache for approximate computing,” in *International Symposium on Microarchitecture*, 2015.

[10] 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.

[11] 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.

[12] 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*, 2008.

[13] 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*, 2012.

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

[15] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *Proc. of International Symposium on High Performance Computer Architecture*, 2009.

[16] 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.

[17] 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.

[18] 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

[19] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814301 pp. 470–487.

[20] G. Stazi, L. Adani, A. Mastrandrea, M. Olivieri, and F. Menichelli, "Impact of approximate memory data allocation on a h.264 software video encoder," in *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 545–553.

[21] G. Stazi, F. Menichelli, A. Mastrandrea, and M. Olivieri, "Introducing approximate memory support in linux kernel," in *2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, June 2017, pp. 97–100.

[22] 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.

[23] R. Venkatagiri, K. Swaminathan, C. Lin, L. Want, A. Buyuktosunoglu, P. Bose, and S. Adve, "Resilience characterization of a vision analytics application under varying degrees of approximation," in *International Symposium on Workload Characterization (IISWC)*, 2016.

[24] 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 *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–14.

[25] R. Venkatagiri, K. Ahmed, A. Mahmoud, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve, "gem5-approxilyzer: An open-source tool for application-level soft error analysis," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*

[26] R. Venkatagiri, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve, "Automated application-level error analysis of program data," in *Under review.*

[27] "Approxilyzer," https://ma3mool.github.io/Approxilyzer.

[28] "gem5-approxilyzer," https://github.com/rsimgroup/gem5-approxilyzer.

[29] 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: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304050 pp. 1087–1103.

[30] 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. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665685

[31] 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," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 00, pp. 1–14, 2016.

[32] 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.

[33] 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*, 2012.

[34] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *IEEE Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[35] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, Jan 2017.

[36] R. Viguier et al., "Resilient mobile cognition: Algorithms, innovations, and architectures," in *ICCD*, 2015.

[37] L. Wang et al., "Power-efficient embedded processing with resilience and real-time constraints," in *ISLPED*, 2015.

[38] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," *SIGPLAN Not.*, 2012.

[39] 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.

[40] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806620 pp. 198–209.

[41] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/2540708.2540711 pp. 13–24.

[42] 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.

[43] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, ser. ISCA, 2016.

[44] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive control of approximate programs," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872402 pp. 607–621.

[45] 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.

[46] 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.

[47] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. I. Jordan, S. Madden, B. Mozafari, and I. Stoica, "Knowing when you're wrong: building fast and reliable approximate query processing systems," in *International Conference on Management of Data, SIGMOD*, 2014, pp. 481–492.

[48] K. Swaminathan et al., "A case for approximate computing in real-time mobile cognition," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2015.

[49] C. Lin et al., "Moving camera analytics: Emerging scenarios, challenges, and applications," *IBM JRD*, 2015.

[50] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking." in *ICCV*, 2005.

[51] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *ECCV*, 2006.

[52] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: an efficient alternative to sift or surf," in *ICCV*, 2011.

[53] M. Fischler and R. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, 1981.

[54] S. Oh et al., "A large-scale benchmark dataset for event recognition in surveillance video," in *CVPR*, 2011.

[55] A. Vega et al., "Resilient, UAV-embedded real-time computing," in *ICCD*, 2015.

[56] 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*, 2007.

[57] M. Li et al., "Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[58] 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*, 2009.

[59] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "CrashTest: A Fast High-Fidelity FPGA-based Resiliency Analysis Framework," in *Proc. of International Conference on Computer Design*, 2008.

[60] A. Pellegrini, R. Smolinski, X. Fu, L. Chen, S. K. S. Hari, J. Jiang, S. V. Adve, T. Austin, and V. Bertacco, "CrashTest'ing SWAT: Accurate, Gate-Level Evaluation of Symptom-Based Resiliency Solutions," in *Proc. of the Conference on Design, Automation, and Test in Europe*, 2012.

[61] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam, "Sampling + DMR: Practical and Low-overhead Permanent Fault Detection," in *Proc. of International Symposium on Computer Architecture*, 2011.

[62] S. Hari, S. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults," in *ASPLOS*, 2012.

[63] "perf: Linux profiling with performance counters." [Online]. Available: \url{https://perf.wiki.kernel.org/index.php/Main_Page}

[64] "Open source computer vision library (OpenCV)," 2015. [Online]. Available: \url{https://github.com/itseez/opencv}

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] P. Roy, R. Ray, C. Wang, and W. F. Wong, "Asac: Automatic sensitivity analysis for approximate computing," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2597809.2597812 pp. 95–104.

[70] M. Carbin and M. C. Rinard, "Automatically identifying critical input regions and code in applications," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831713 pp. 37–48.

[71] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *International Conference on Dependable Systems and Networks*, 2013, pp. 1–12.

[72] 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.

[73] 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.

[74] R. Akram and A. Muzahid, "Approximeter: Automatically finding and quantifying code sections for approximation," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, 2017.

[75] 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.

[76] X. Li, S. Adve, P. Bose, and J. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors," in *Submitted for publication*, 2007.

[77] 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.

[78] 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 *International Symposium on Microarchitecture*, 2003.

[79] 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.

[80] 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.

[81] B. Wibowo, A. Agrawal, T. Stanton, and J. Tuck, "An accurate cross-layer approach for online architectural vulnerability estimation," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, pp. 30:1–30:27, Sep. 2016. [Online]. Available: http://doi.acm.org/10.1145/2975588

[82] 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*, 2010.

[83] 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*, 2011.

[84] 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.

[85] 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.

[86] 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.

[87] J. Park, E. Amaro, D. Mahajan, B. Thwaites, and H. Esmaeilzadeh, "Axgames: Towards crowdsourcing quality target determination in approximate computing," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872376 pp. 623–636.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, "Application-based metrics for strategic placement of detectors," in *Proc. of Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2005.

[99] 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.

[100] 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*, ser. CGO '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854059 pp. 227–238.

[101] 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*, 2013, pp. 1–10.

[102] F. Bower, D. Sorin, and S. Ozev, "Online Diagnosis of Hard Faults in Microprocessors," *ACM Transactions on Architecture and Code Optimization*, vol. 4, no. 2, 2007.

[103] 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.

[104] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 375–382.

[105] 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 *Dependable Computing Conference (EDCC), 2015 Eleventh European*, Sept 2015, pp. 245–255.

[106] 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)*, Oct 2013, pp. 236–242.

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

[108] 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.

[109] 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.

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

[111] 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*, 2005.

[112] Weining Gu, Z. Kalbarczyk, and R. K. Iyer, "Error sensitivity of the linux kernel executing on powerpc g4 and pentium 4 processors," in *International Conference on Dependable Systems and Networks, 2004*, 2004.

[113] D. Skarin, R. Barbosa, and J. Karlsson, "Goofi-2: A tool for experimental dependability assessment," in *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, 2010.

[114] L. Palazzi, G. Li, B. Fang, and K. Pattabiraman, “A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection,” in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019.

[115] 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.

[116] S. Cha, O. Seongil, H. Shin, S. Hwang, K. Park, S. J. Jang, J. S. Choi, G. Y. Jin, Y. H. Son, H. Cho, J. H. Ahn, and N. S. Kim, “Defect analysis and cost-effective resilience architecture for future dram devices,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[117] D. Zhang, V. Sridharan, and X. Jian, “Exploring and optimizing chipkill-correct for persistent memory based on high-density nvrams,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[118] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, “Memory errors in modern systems: The good, the bad, and the ugly,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[119] S. Gurumurthi, “Advanced memory device correction (amdc) for servers,” *AMD Whitepaper*, 2020.

[120] 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.

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

[122] C. Bienia, “Benchmarking modern multiprocessors,” Ph.D. dissertation, Princeton University, January 2011.

[123] 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.

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

[125] M. Rinard, “Probabilistic accuracy bounds for fault-tolerant computations that discard tasks,” in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS ’06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1183401.1183447 pp. 324–334.

[126] 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.

[127] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An Architectural Framework for Software Recovery of Hardware Faults," in *International Symposium on Computer Architecture*, 2010.

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

[129] K. Reick, P. N. Sanda, S. Swaney, J. W. Kellington, M. Mack, M. Floyd, and D. Henderson, "Fault-Tolerant Design of the IBM Power6 Microprocessor," *IEEE Micro*, 2008.

[130] D. Ernst et al., "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," in *MICRO*, dec 2003.

[131] 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.

[132] 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*, 2007.

[133] 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.

[134] 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*, 2009.

[135] 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*, 2006.

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

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

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

[139] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, “Perturbation-based Fault Screening,” in *International Symposium on High Performance Computer Architecture*, 2007.

[140] 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*, March 2005, pp. 243–254.

[141] R. K. Venkatesan, S. Herr, and E. Rotenberg, “Retention-aware placement in dram (rapid): software methods for quasi-non-volatile dram,” in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, 2006.

[142] 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.

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

[144] 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, April 1985.

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

[146] 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.

[147] “Road vehicles — Functional safety,” Website, https://www.iso.org/standard/43464.html.

[148] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: A survey,” *STVR*, vol. 22, no. 2, pp. 67–120, 2012.

[149] C. Zhang, A. Groce, and M. A. Alipour, “Using test case reduction and prioritization to improve symbolic execution,” in *ISSTA*, 2014, pp. 160–170.

[150] 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.

[151] A. Groce, M. Alipour, C. Zhang, Y. Chen, and J. Regehr, “Cause reduction for quick testing,” in *ICST*, 2014, pp. 243–252.

[152] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce, "Evaluating non-adequate test-case reduction," in *Proc. of the 31st IEEE/ACM Conference on Automated Software Engineering (ASE)*, Singapore, Singapore, Sep. 2016, pp. 16–26.

[153] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *TSE*, vol. 28, no. 2, pp. 183–200, 2002.

[154] 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.

[155] 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.

[156] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: Using canary inputs to dynamically steer approximation," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908087 pp. 161–176.

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

[158] 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.

[159] 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.

[160] 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.

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

[162] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, April.

[163] 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.

[164] A. Agrawal, B. Wibowo, and J. Tuck, “Software marking for cross-layer architectural vulnerability estimation model,” in *SELSE*, 2017.

[165] 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.

[166] 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.

[167] B. Nie, L. Yang, A. Jog, and E. Smirni, “Fault site pruning for practical reliability analysis of gpgpu applications,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 749–761.

[168] 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.

[169] W.-C. Lee, T. Bao, Y. Zheng, X. Zhang, K. Vora, and R. Gupta, “Raive: Runtime assessment of floating-point instability by vectorization,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2814270.2814299 p. 623–638.

[170] D. Jeffrey, N. Gupta, and R. Gupta, “Fault localization using value replacement,” in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA ’08. New York, NY, USA: Association for Computing Machinery, 2008. [Online]. Available: https://doi.org/10.1145/1390630.1390652 p. 167–178.

[171] M. P. D. Lohar and E. Darulova, “Sound probabilistic numerical error analysis,” in *iFM*, ser. iFM ’19, 2019.

[172] E. D. H. Becker, P. Panchekha and Z. Tatlock, “Combining tools for optimization and analysis of floating-point computations,” in *FM*, ser. FM ’08, 2018.

[173] A. Chatzidimitriou and D. Gizopoulos, “Anatomy of Microarchitecture-level Reliability Assessment: Throughput and Accuracy,” in *ISPASS*, 2016.

[174] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, “GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates,” in *DSN*, 2014.

[175] N. Hasabnis and R. Sekar, “Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers,” in *ASPLOS*, 2016.

[176] 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.

[177] 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.

[178] 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.

[179] 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.

[180] 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.

[181] 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.

[182] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, 2015.

[183] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," *CoRR*, 2017.

[184] 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.

[185] R. Akram, "Performance and accuracy analysis of programs using approximation techniques," Ph.D. dissertation, 2017.

[186] C. Kalra, F. Previlon, X. Li, N. Rubin, and D. Kaeli, "Prism: Predicting resilience of gpu applications using statistical methods," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[187] Q. Shi, H. Hoffmann, and O. Khan, "A cross-layer multicore architecture to tradeoff program accuracy and resilience overheads," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 85–89, 2015.

[188] A. Thomas and K. Pattabiraman, "Error Detector Placement for Soft Computation," in *DSN*, 2013.

[189] R. Szeliski, "Image alignment and stitching: A tutorial," 2004.

[190] K. Rane et al., "Mosaic evaluation: An efficient and robust method based on maximum information retrieval," *Int. J. Computer Applications*, 2013.

[191] A. Camargo, Q. He, and K. Palaniappan, "Performance evaluations for super-resolution mosaicing on UAS surveillance videos," *Int J Adv Robotic Systems*, 2013.

[192] B. Morse et al., "Application and evaluation of spatio-temporal enhancement of live aerial video using temporally local mosaics," in *CVPR*, 2008.

[193] P. Paalanen, J.-K. Kämäräinen, and H. Kälviäinen, "Image based quantitative mosaic evaluation with artificial video," in *Image Analysis*, 2009, pp. 470–479.

[194] M. El-Saban, M. Izz, A. Kaheel, and M. Refaat, "Improved optimal seam selection blending for fast video stitching of videos captured from freely moving devices," in *ICIP*, 2011.

[195] Y. S. Shao, S. L. Xi, V. Srinivasan, G. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[196] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G. Wei, and D. Brooks, "Masr: A modular accelerator for sparse rnns," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[197] M. Minutoli, V. G. Castellana, C. Tan, J. Manzano, V. Amatya, A. Tumeo, D. Brooks, and G. Y. Wei, "Soda: a new synthesis infrastructure for agile hardware design of machine learning accelerators," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020.