Minotaur: Adapting Software Testing Techniques for Hardware Errors

Abdulrahman Mahmoud
amahmou2@illinois.edu
University of Illinois at Urbana-Champaign

Radha Venkatagiri
venktgr2@illinois.edu
University of Illinois at Urbana-Champaign

Khalique Ahmed
kahmed10@illinois.edu
University of Illinois at Urbana-Champaign

Sasa Misailovic
misailo@illinois.edu
University of Illinois at Urbana-Champaign

Darko Marinov
marinov@illinois.edu
University of Illinois at Urbana-Champaign

Christopher W. Fletcher
cwfletch@illinois.edu
University of Illinois at Urbana-Champaign

Sarita V. Adve
sadve@illinois.edu
University of Illinois at Urbana-Champaign

Abstract

With the end of conventional CMOS scaling, efficient resiliency solutions are needed to address the increased likelihood of hardware errors. Silent data corruptions (SDCs) are especially harmful because they can create unacceptable output without the user's knowledge. Several resiliency analysis techniques have been proposed to identify SDC-causing instructions, but they remain too slow for practical use and/or sacrifice accuracy to improve analysis speed.

We develop Minotaur, a novel toolkit to improve the speed and accuracy of resiliency analysis. The key insight behind Minotaur is that modern resiliency analysis has many conceptual similarities to software testing; therefore, adapting techniques from the rich software testing literature can lead to principled and significant improvements in resiliency analysis. Minotaur identifies and adapts four concepts from software testing: 1) it introduces the concept of *input quality criteria* for resiliency analysis and identifies PC coverage as a simple but effective criterion; 2) it creates (fast) *minimized inputs* from (slow) standard benchmark inputs, using the input quality criteria to assess the goodness of the created input; 3) it adapts the concept of test case prioritization to *prioritize error injections* and invoke *early termination* for a given instruction to speed up error-injection campaigns; and 4) it further adapts test case or *input prioritization* to accelerate SDC discovery across multiple inputs.

We evaluate Minotaur by applying it to Approxilyzer, a state-of-the-art resiliency analysis tool. Minotaur's first three techniques speed up Approxilyzer's resiliency analysis by 10.3X (on average) for the workloads studied. Moreover, they identify 96% (on average) of all SDC-causing instructions explored, compared to 64% identified by Approxilyzer alone. Minotaur's fourth technique (input prioritization) enables identifying all SDC-causing instructions explored across multiple inputs at a speed 2.3X faster (on average) than analyzing each input independently for our workloads.

Keywords Hardware reliability; Resiliency analysis; Silent data corruption (SDC); Fault tolerance; Software testing; Coverage metrics; Input minimization and prioritization

ACM Reference Format:
Abdulrahman Mahmoud, Radha Venkatagiri, Khalique Ahmed, Sasa Misailovic, Darko Marinov, Christopher W. Fletcher, and Sarita V. Adve. 2019. Minotaur: Adapting Software Testing Techniques for Hardware Errors. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3297858.3304050

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304050

1 Introduction

As we approach the end of CMOS scaling, hardware is becoming increasingly susceptible to errors in the field [14, 19, 27, 96, 111, 134]. Commodity hardware is used in systems with a range of reliability requirements, from entertainment devices to stringently safety-critical systems such as self-driving cars. This hardware is also used at multiple scales, from small embedded systems to large-scale high-performance computing systems where the sheer scale demands extremely low failure rate for individual components. Traditional reliability solutions, relying on indiscriminate redundancy in space or time, are too expensive for such systems. Therefore, there has been significant research in cross-layer solutions [22, 26] 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 [31, 32, 95, 134].

Early work recognized that a large majority of hardware errors were either masked at the software level (i.e., they did not change the output of the executing program) or resulted

in easily detectable anomalous software behavior (e.g., exceptions due to unaligned or out-of-bounds addresses) [29, 38, 50, 67, 89, 98, 120]. The former errors require no action and the latter can be detected using zero to very low-cost detection mechanisms. While such software-centric resiliency techniques show immense promise, unfortunately, some hardware errors escape detection and result in undetected and potentially unacceptable silent data corruptions (*SDCs*) of the program output.

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) [9, 22, 30, 35, 48, 60, 66, 70, 73, 88, 92, 98, 113].

Underlying all of these solutions is the need for techniques that find SDCs in the applications of interest. We use *software resiliency* or just *resiliency* to mean the ability of a given piece of software to avoid an SDC for a given hardware error. We use *resiliency analysis* to mean the process of characterizing the resiliency of a given piece of software for a given set of hardware errors. We use *resiliency hardening* to mean software modifications (with or without accompanying hardware modifications) to make the software more resilient. This paper concerns techniques for fast and accurate resiliency analysis.

Prior work in resiliency analysis imposes a significant trade-off between speed and accuracy – statistical analyses based on dynamic error-free execution traces or static code [34, 68, 78, 80, 113] are unable to precisely model error propagation paths; randomized error injection campaigns [18, 54, 64, 109, 122] provide only statistical information and are unable to predict resilience for code portions where errors were not injected; and more systematic and comprehensive error-injection techniques [49, 107, 117] precisely identify SDC-causing instructions but are much slower than the previous techniques. Section 7 describes prior work in more detail.

This paper presents Minotaur, a toolkit that improves the speed of resiliency analyses while also precisely identifying more SDC-causing instructions (program counters) or *SDC-PCs*. The novel insight behind Minotaur is that analyzing software for resiliency to hardware errors is similar to testing software for software bugs; therefore, adapting techniques from the rich software testing literature can lead to principled and significant improvements in resiliency analysis. Minotaur can benefit many resiliency analysis techniques; here we evaluate it by applying it to the state-of-the-art Approxilyzer tool [116, 117].

We identify, adapt, and evaluate four bridges between software testing and resiliency 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 2.1.1). Resiliency analysis typically uses generic inputs often developed for performance evaluation; e.g., the reference inputs in benchmark suites. These generic inputs could be sub-optimal for discovering code vulnerable to SDCs, but there is no accepted input-quality criterion for resiliency analysis.

This work introduces the notion of input-quality criteria for resiliency analysis, adapts several widely used software testing criteria to the object-code level, and evaluates these criteria for resiliency analysis. We find that *program counter (PC) coverage*, an analog of the widely used statement coverage, is an effective input-quality criterion for resiliency 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 resiliency 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*).

We apply minimization to seven benchmarks and show that using Min instead of Ref speeds up resiliency analysis by 4.1X on average. Min also finds 96% of all SDC-PCs identified by either Ref or Min. However, Ref only finds 64% of these SDC-PCs. This surprising result that Min is more accurate[1] than Ref parallels recent work from the software testing literature on bug detection [40]. Intuitively, Min can improve accuracy because it can be analyzed more comprehensively due to the improved analysis speed, whereas Ref can be prohibitively expensive to analyze in its entirety [49, 117].

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). In both cases, once a PC is found to generate an SDC, no further error injections are performed on that PC because it needs to be hardened anyway.

We explore several priority orderings for error injections for a given PC. Surprisingly, we find that random ordering reveals SDCs almost as quickly as an oracular best case. Further investigation shows that an SDC-PC often produces SDCs

[1]The accuracy of a binary classifier is typically measured as the ratio of the identified true positives (SDC-PCs for our case) and true negatives relative to the total population with known outcomes. We focus only on the identification of SDC-PCs and assume Approxilyzer's error-injection based methodology results in no false positives. Therefore, we measure accuracy of Min (or Ref) as the ratio of SDC-PCs identified by Min (or Ref) relative to all the known SDC-PCs; i.e., the union of the SDC-PCs identified by Min and Ref. With this interpretation, accuracy is equivalent to *recall* [45].

for a very large number of its injections; therefore, a random ordering quickly finds one such injection. The combination of random ordering and termination of injections on a PC after an SDC discovery, combined with input minimization, provides an average 10.3X speedup (up to 38.9X) in resiliency analysis time by employing Minotaur.

Concept 4: Test-Case Prioritization → Input Prioritization. We also adapt test-case prioritization across multiple inputs. To find SDC-PCs as fast as possible, we prioritize resiliency analysis on the faster Min input over the slower Ref input. Then, for higher accuracy, we can additionally perform resiliency analysis on the larger Ref input, but only for the 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 resiliency analysis enables principled and significant benefits in speed and accuracy. While our evaluation uses Approxlyzer as the underlying resiliency analysis, Minotaur and its concepts apply more generally (Section 7). For example, Concepts 1 and 2 can be applied to speed up any dynamic resiliency analyses that typically study large inputs, by producing a smaller representative input for analysis. Error-injection analyses can greatly benefit from Concept 3, by prioritizing error-injections and employing early termination for SDC-PCs. Concept 4 can propel resiliency analyses to explore multiple inputs, a new direction which previously was daunting due to speed and accuracy concerns of existing techniques. Minotaur provides a foundation for a systematic methodology for efficient resiliency analysis based on software testing, and opens up many avenues for further research.

2 Background

This section provides an overview of the relevant software testing techniques adapted by Minotaur and of Approxilyzer [117].

2.1 Relevant Software Testing Techniques

Software testing is the process of executing a program or system with the intent of finding failures [81]. The objective of testing can be quality assurance, verification, validation, or reliability estimation. We discuss some techniques and best practices adopted by the software testing community.

2.1.1 Test-Case Quality

In software testing, a *test case* is an input and an expected output used to determine whether the system under test satisfies some software testing objective. A *test set* is a collection of 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.

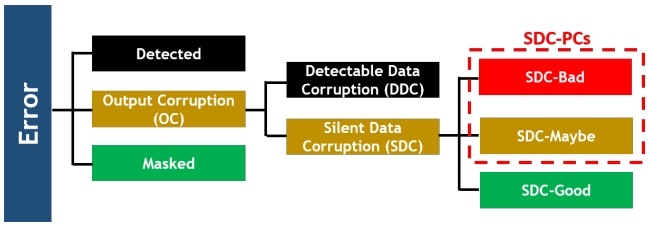

Figure 1. A classification of error outcomes [117]. Only outcomes of SDC-Bad and SDC-Maybe constitute SDC-PCs.

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 [93]. The choice of criterion depends on several factors, including the size of the program, cost requirements, and criticality of failure. Some popular criteria [93], ordered from weaker to stronger, are: (1) statement coverage [7], which measures the fraction of program statements executed by tests; (2) branch coverage [81], which measures the fraction of branch edges executed; and (3) def-use coverage [37, 93], 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 [36]. The software testing literature provides an extensive analysis of various testing criteria [7].

2.1.2 Test-Case Minimization

While running larger (or more) test cases is desirable for thorough testing, time and resources limit the size (or number) of test cases that can be executed. *Test-case minimization* is used to minimize the testing cost in terms of execution time [6, 39, 40, 94, 130–132]. 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.

2.1.3 Test-Case Prioritization

Resource constraints can sometimes make it infeasible to execute all planned test cases. It thus becomes necessary to prioritize and select test cases so that critical failures can surface sooner rather than later [130]. 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 [130].

2.2 Approxilyzer

We evaluate Minotaur using Approxilyzer [116, 117], a state-of-the-art instruction-level 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

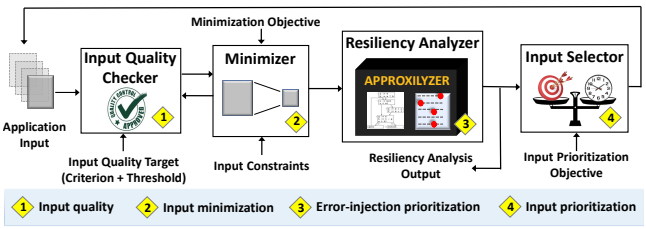

Figure 2. Overview of Minotaur. Approxilyzer may be replaced with another resiliency analyzer.

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. We use *error site* 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 [49]) to reduce the number 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 [117]. 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 [49, 117].

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 1:

- **Detected**: An error that raises observable symptoms and can hence be caught using various low-cost detectors [98] before the end of execution.
- **DDC**: An OC that is detectable via low-cost mechanisms such as range detectors applied on the output [48].
- **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).

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

Figure 3. Statement coverage vs. PC coverage.

- **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.

3 Minotaur

This section describes Minotaur, a novel toolkit for principled and efficient resiliency analysis for hardware errors. Figure 2 illustrates the complete system.

3.1 Input Quality

Ensuring that "good" quality inputs are used for resiliency analysis increases the effectiveness of the analysis. We adapt the concept of test-case quality (Section 2.1.1) to build an *Input Quality Checker* (Figure 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 some work on test coverage at the object-code level [13, 17], but it is not widely studied. Our resiliency analysis examines error models at the object code level and aims to find assembly instructions that are vulnerable to SDCs (SDC-PCs). Hence, it is desirable to measure the quality of the input used for resiliency analysis with quality criteria expressed at the object code.

Figure 3 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 2.1) of the source code measures that this single input will cover (execute) 100% of the code. However, for the same code compiled to assembly, only 75% of the instructions are covered (executed). Analyzing resiliency with just this input does not provide full (100%) assembly instruction coverage, and an error in assembly instruction PC-4 would not be captured.

For resiliency analysis, we adapt three test (input) quality criteria to the object code level—statement, branch, and def-use coverage. The analog of statement coverage at the object code level measures the fraction of static assembly instructions (or *PCs*) executed by the input; we call it simply *PC coverage*. Branch and def-use coverage are analogously 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 2) evaluates whether a given input meets the desired *quality threshold* (e.g., 90%) for a specified *quality criterion* (e.g., PC coverage). We refer to the combination of the input quality criterion and the threshold as the *input quality target*.

3.2 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 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 [7]. Our Minimizer algorithm is specialized for our workloads. Given a Ref, the goal of the Minimizer is to find a reduced input (Min) that minimizes a stated *minimization objective (MinObj)* (e.g., execution time) while satisfying an *input quality target* (e.g., 90% PC coverage relative to Ref). We chose a simple, greedy algorithm based on binary search for the Minimizer and found it effective. More sophisticated optimizers may find better Min inputs; we leave such an exploration to future work.

In addition to the minimization objective and input quality target, the Minimizer is provided with the list of input parameters (e.g., command line and other program-specific parameters) and a set of parameter constraints (e.g, range or boundary conditions) to ensure that the Min generated is both legal and realistic. A realistic Min enables the resiliency analysis to uncover SDC-PCs that are vulnerable for realistic conditions, avoiding over- or under-protection. Domain knowledge enables understanding the realistic range of input values and how to change them (e.g., choosing image shrinking instead of sub-sampling pixels or subsetting image inputs [58]) to achieve realistic inputs.

Algorithm 1 shows the pseudo-code of Minotaur's Minimizer. It first performs a pre-processing pass over the reference input's parameter list and orders the parameters according to their estimated impact on the minimization objective. Our current implementation determines this order by running the program with a few different values for each input parameter and measuring the impact on the minimization objective. This step can be accelerated with additional domain knowledge or automated using more sophisticated optimizers.

Given the ordered parameter list, the Minimizer uses binary search to progressively change each input parameter (one with highest impact on the minimization objective first) while ensuring that the new input value meets the input quality target. Lines 6–10 of Algorithm 1 show this search for applications with (1) numeric inputs and (2) where reducing the value of input parameters reduces (or does not affect) the minimization objective. All applications we study (except Sobel, which takes as input an image) satisfy both characteristics, with binary search sufficing for the value exploration. We reduce images for Sobel using the *resize* utility in the ImageMagick suite [69], 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.

Algorithm 1: Input Minimization Pseudocode

```
1  PList: Parameter List, C: Constraints,
2  IQT: Input Quality Target, MinObj: Minimization Objective,
3  PList_Ref: Reference input's PList
4  Function Minimizer(PList_Ref, C, IQT, MinObj):
5      PList ← OrderParams(PList_Ref, MinObj)
6      for param ∈ PList do
7          lower ← Minimum value of param provided C
8          upper ← Reference value of param
9          PList[param] ←
             BinarySearch(lower, upper, C, IQT)
10     end
11     return PList
12 Function OrderParams(PList, MinObj):
13     return Ordered parameters of PList with respect to
         MinObj
14 Function BinarySearch(lower, upper, C, IQT):
15     Search values between lower and upper provided C,
16     checking if the candidate value satisfies IQT
17     return minimum value that satisfies IQT
```

3.3 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 2 shows the application of error-injection prioritization in Minotaur's workflow.

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

- **Bit position of registers (BitPos)**: Injecting into specific bits first (such as the MSB or LSB).

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

3.4 Input Prioritization

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

In our scheme, an *Input Selector* (Box 4 in Figure 2) chooses inputs for resiliency analysis according to an order specified by an *input prioritization objective*. We choose to analyze the input with the shortest execution time, prioritizing faster analyses first (e.g., we choose Min before Ref). Input prioritization can lead to faster resiliency analysis speed for each subsequent input because the PCs already identified as SDC-PCs by prior inputs need not be (re)analyzed. Thus, we can leverage input-prioritization to find many of the SDC-PCs from one (faster) input, and carry this information onto another (slower but larger) input to avoid unnecessary error injections. Minotaur's Input Selector can successively select inputs for resiliency analysis until it meets an analysis target (e.g., a coverage or resource target).

4 Methodology

4.1 Evaluation Infrastructure and Workloads

Our error-injection infrastructure builds on Approxilyzer [117], based on simulation using Wind River Simics [119] and GEMS [72] running OpenSolaris. Our workloads are compiled to the SPARC V9 ISA with all optimizations enabled.

Approxilyzer's error model uses single-bit architecture-level errors (Section 2.2), which are a limited but effective [104] and realistic subset of hardware errors [24]. With resiliency becoming a first-class software design objective [10], techniques with different speed, precision, and error models are needed at different stages of software development. Evaluating Minotaur with tools that use different error models (lower-level, multi-bit, etc.) is part of our future work.

To evaluate Minotaur, we use seven workloads from three benchmark suites spanning multiple application domains, summarized in Table 1. Column 4 lists the reference (Ref) input parameters used in our study. For five of the benchmarks—Blackscholes, Swaptions, LU, Water, and FFT—we use the same inputs as Approxilyzer [117] for the reference inputs. For Streamcluster, prior evaluations [75, 110] showed that the benchmark benefits from realistic datasets (as opposed to data points generated internally by the application); hence, we use a dataset from the UCI Machine-Learning Repository [28, 52, 99] as its Ref input. For Sobel, we use the bird image from the iACT [76] repository as input. We chose relatively small Ref inputs for almost all applications to be conservative and not over-estimate the benefits of input minimization. To evaluate the quality of the outputs, we use the same metrics as Approxilyzer [117] for Blackscholes, Swaptions, LU, Water, and FFT; for Streamcluster and Sobel, we use maximum relative error (*max-rel-err* from Approxilyzer [117]).

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.

4.2 Input-Quality Criteria

Since no available tool can easily measure test coverage at the object-code level, we developed our own tools using dynamic traces from Simics [119] for PC, branch, and def-use coverage for the object code. For PC coverage, we simply track the PCs executed by the input. For branch coverage, we store the unique branch-target PC pairs that represent control edges exercised by the input. For def-use coverage, we analyze the definition and use of operand registers exercised by the input, and store unique PC pairs that represent a def-use edge. For all criteria, we measure Min's coverage relative to Ref.

4.3 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. We measure PC, branch, and def-use coverage for each Min *relative* to its corresponding Ref; e.g., if Min executes all PCs executed by its Ref, we consider it to have 100% PC coverage. Similarly, if Min exercises all branch-target and def-use pairs exercised by Ref, we consider it to have 100% branch and def-use coverage, respectively.

We choose PC coverage as our quality criterion because it is simple and fast to compute and it is the analog of the widely used statement coverage criterion for software testing (Section 2.1.1). We find that the Min inputs generated using PC coverage are surprisingly effective (Section 5.1.3), and also exhibit high (but not perfect) branch and def-use coverage.

Suite	Application	Domain	Ref Input	Min Input	PC (%)	Branch (%)	Def-Use (%)
Parsec 3.0 [12]	Blackscholes [12]	Financial Modeling	64K options	21 options	100	100	99.38
	Swaptions [12]		16 options 5000 simulations	1 option 1 simulation	99.91	99.23	98.42
	Streamcluster [12]	Data Mining	centers = [10,20] num iterations = 3	centers = [4,5] num iterations = 1	99.97	99.77	98.67
SPLASH-2 [124]	LU [124]	Scientific Computing	512x512 matrix 16x16 block size	16x16 matrix 8x8 block size	100	100	95.56
	Water [124]		512 molecules	216 molecules	99.89	99.36	99.85
	FFT [124]	Signal Processing	2^{20} data points	2^{8} data points	100	100	99.59
ACCEPT [101]	Sobel [101]	Image Processing	100% image size (321x481 pixels)	25.25% image size (81x121 pixels)	100	100	100

Table 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.

4.4 Accuracy Analysis

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

To understand the sources of inaccuracy, we analyze the SDC-PCs identified by Min and Ref by grouping them into categories based on whether they were found by Ref, Min, or both. We further distinguish the cases where certain PCs are explored (i.e., analyzed for resiliency) by one input but not both inputs. The difference occurs when the targeted error-site coverage (Section 2.2) 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.

We group the SDC-PCs into five categories:

1. **Common**: Both Min and Ref classify the PCs as SDC, which are considered accurately classified by both.
2. **MinSDC**: Min classifies these as SDC-PCs and Ref explores them but does not classify them as SDC-PCs. Although Ref did not find these SDC-PCs, they are still candidates for hardening because they were found by a realistic Min input. Hence, these PCs are considered accurately classified by Min, but not by Ref.
3. **MinSDC+**: Min classifies these as SDC-PCs and Ref does not explore them. For similar reasons as MinSDC, this category is also considered accurately classified by Min, but not by Ref.
4. **RefSDC**: Ref classifies these as SDC-PCs and Min explores them but does not classify them as SDC-PCs. These PCs are inaccurately classified by Min because relying only on Min's analysis would leave these PCs unprotected.
5. **RefSDC+**: Ref classifies these as SDC-PCs and Min does not explore them. This category is also considered inaccurately classified by Min.

The error-injection prioritization scheme (Section 4.5) 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 4.6) will result in 100% accuracy since input-prioritization obtains the union of SDC-PCs found by analyzing all inputs (while optimizing resiliency analysis speed).

4.5 Error-Injection Prioritization

We explore 38 different error-injection prioritizations using combinations of the schemes from Section 3.3. For BitPos, DI, and ECS schemes, we test both ascending (A) and descending (D) ordering. We also explore compositional schemes. For example, BitPos_A + ECS_D first orders error injections by bit positions in ascending order (i.e., starting with the LSB), followed by ordering in descending equivalence class size. For RT and OP schemes, we simply pick the type/kind of register (e.g., OP_{Src} or OP_{Dest}) to prioritize.

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

4.6 Input Prioritization

Our Input Selector prioritizes (faster) Min over Ref. Section 5 shows that while Min exhibits high accuracy (Section 4.4), 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 3.4).

4.7 Runtime Analysis of Minotaur

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

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

The time for resiliency analysis for a given application and input (Ref or Min) depends on: (1) equivalence class generation time ($t_{equiv\_class\_gen}$) [49, 117], (2) total injections of each outcome category ($I_{masked}, I_{det}, I_{OC}$) for a target error site coverage, and (3) the average error-injection runtime of each outcome category ($t_{masked}, t_{det}, t_{OC}$). We measure the runtime for each category separately because it can be quite different; e.g., an OC error requires additional post-processing (compared to Masked) to quantify the error quality into Good/Maybe/Bad categories, while Detected outcomes involve simulator and OS overhead to report outcomes such as SegFaults.

We measure the runtime by sampling 1,000 error-injection experiments for each of masked, detected, and OC outcomes per application and input, excluding outliers in the top and bottom 2.5% of runs. The total samples correspond to a 99.8% confidence level with 5% error margin in timing measurements [59]. The time for resiliency analysis is calculated as:

$$TotalRuntime = t_{equiv\_class\_gen} + \sum_n \left(I_n \times t_n\right) \qquad (1)$$

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

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

5 Results

We evaluate Minotaur's impact on a resiliency analysis tool, Approxilyzer [117], by analyzing (1) the speedup and accuracy from a minimized input (Min) for resiliency analysis (Section 5.1); (2) the speedup from error-injection prioritization with early termination (Section 5.2); (3) the combined speedup from minimization and error-injection prioritization (Section 5.3); and (4) the speedup from applying input prioritization across multiple inputs (Section 5.4).

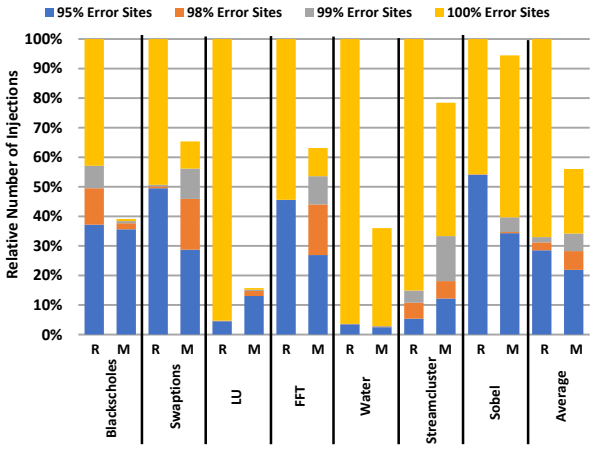

Figure 4. Number of error injections for different error-site coverage targets for each benchmark, relative to 100% error-site coverage for Ref (Ref100). R=Ref, M=Min.

5.1 Input Minimization

5.1.1 Min Quality

Table 1 shows the Min generated by applying Algorithm 1 to each Ref, using PC coverage as the input quality criterion. Most applications show a large reduction of input parameter values in Min (column 5), which translates to faster application runtimes relative to Ref (Section 5.1.2).[2] Additionally, Min maintains very high PC coverage relative to Ref (column 6), which translates to high accuracy in finding SDC-PCs (Section 5.1.3).

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 5.1.3), while allowing Minotaur to benefit from running the faster Min (Section 5.1.2).

The last two columns of Table 1 show the branch and def-use coverage of the generated Min (relative to Ref) and are discussed further in Section 5.1.4.

5.1.2 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 4 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 2.2. By using input minimization, achieving 100%

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

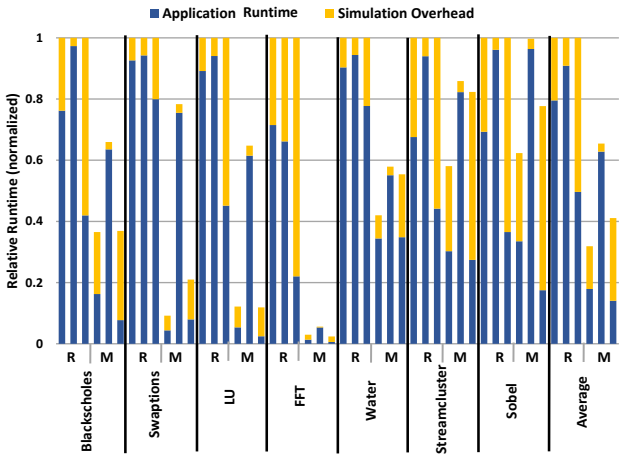

Figure 5. Average runtime per injection, normalized to Ref. Each set of three bars represents (from left to right) Masked, Detected, OC runtime (Section 4.7), divided into application runtime and simulation overhead. R = Ref and M = Min.

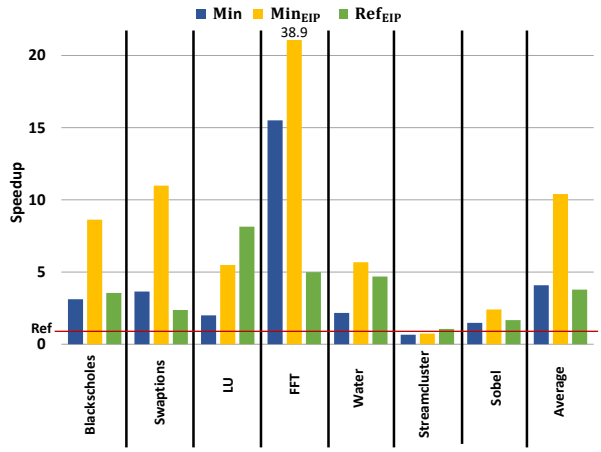

Figure 6. Min, Min$_{EIP}$, and Ref$_{EIP}$ speedup relative to Ref.

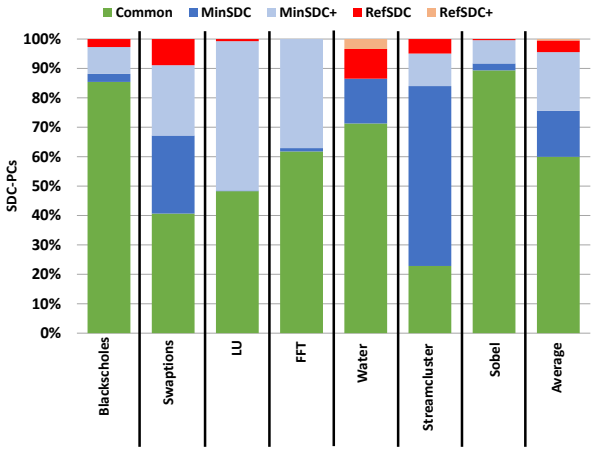

Figure 7. Min and Ref accuracy. The Y-axis represents all SDC-PCs found by Min or Ref in an application.

error-site coverage is no longer elusive for many applications. Figure 4 shows that for the Min inputs of Blackscholes, Swaptions, LU, and FFT, the number of error injections required for 100% error site coverage (Min100) is comparable to the number of error injections for Ref99 Thus, for these applications, it becomes tractable to run resiliency analysis with Min100. The other applications (Water, Streamcluster, and Sobel) also reduce the number of error injections from Ref100 to Min100, but the total number is still very large, presenting a trade-off between resiliency-analysis runtime and error-site coverage. We choose to favor runtime and use 99% error-site coverage for these applications. Henceforth, we use the umbrella term Min (unless otherwise stated) to encompass Min100 for Blackscholes, Swaptions, LU, and FFT, and Min99 for Water, Streamcluster, and Sobel. We use Ref to refer to Ref99 for all applications.

Not only does Min require fewer error injections for most of our workloads, each individual injection runs faster compared to Ref. Figure 5 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 4.7).

Min injections run 2.1X faster on average[3] 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 1). Min's smaller output matrices allow for faster post-processing, further speeding up the resiliency analysis relative to Ref for these applications.

Figure 6 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.

5.1.3 Minimization Accuracy

Figure 7 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 4.4). 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

[3] All averages in this paper refer to the arithmetic mean.

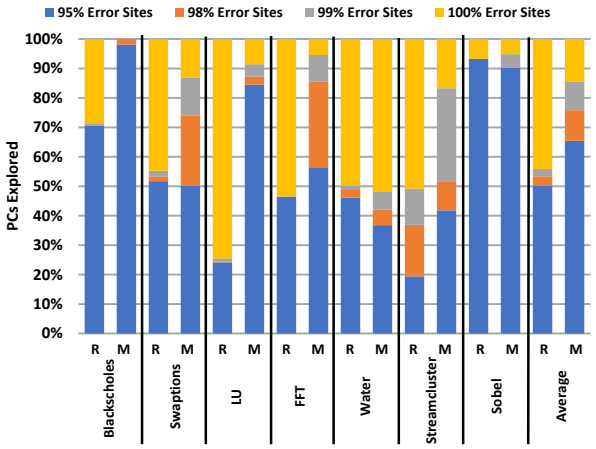

Figure 8. Percentage of PCs explored for different error-site coverage targets. R = Ref, M = Min.

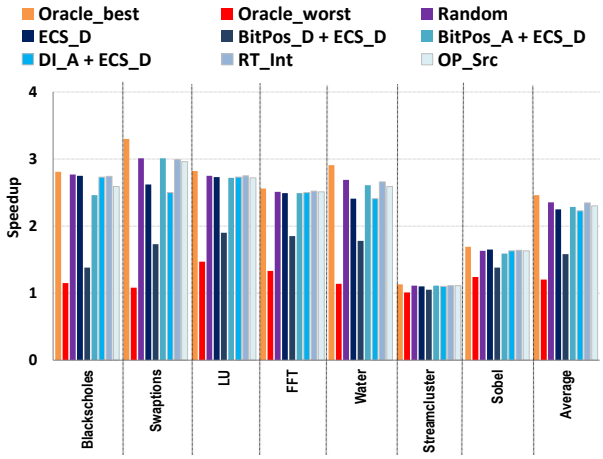

Figure 9. Min speedup with error-injection prioritization.

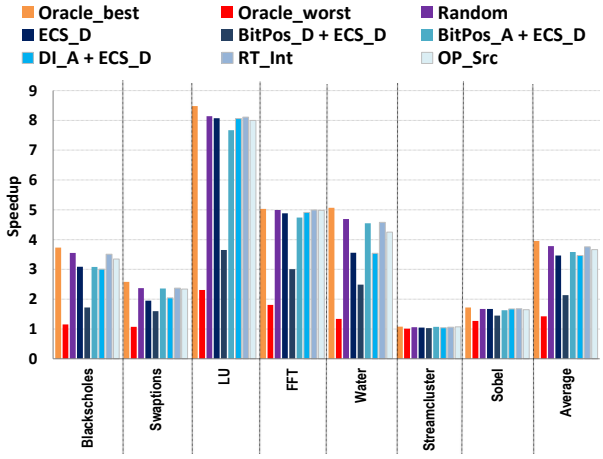

Figure 10. Ref speedup with error-injection prioritization.

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 8 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 5.1.2), generally explore fewer distinct PCs than Min at lower error site coverage targets. Figure 8 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 workloads such as Streamcluster and Sobel, even though the total analysis time is similar to that of running with Ref.

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

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

5.1.4 Improving Min Selection Criteria

We studied branch and def-use coverage of Min (relative to Ref) to understand if these stronger criteria could have been used to generate an alternate Min that provides higher accuracy than PC coverage. Table 1 shows that the Min inputs generated using PC coverage already have very high branch and def-use coverage of 99.76% and 98.78%, respectively, relative to Ref. Further, as discussed, Min already finds 96% of the SDC-PCs discovered by the union of Ref and Min. Thus, the potential improvement from using the more complex criteria is limited.

Nevertheless, we isolated the branch-target and def-use pairs that were in Ref but not in Min to determine if they were responsible for the RefSDCs in Figure 7. 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

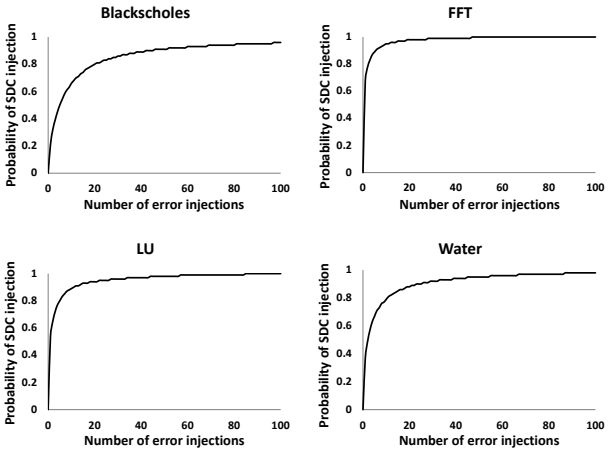

Figure 11. Cumulative probability (Y-axis) of picking an SDC-causing error injection within the first n injections (X-axis) for an SDC-causing PC.

comprehensive analysis would explore the entire control and data flow paths rooted at the isolated branch-target and def-use PCs in Ref to conclusively confirm whether the stronger criteria would add further accuracy. We leave such an analysis and exploration of even more complex input quality criteria (e.g., path coverage) to future work, given that our results already show that PC coverage provides an excellent sweet spot for simplicity, performance, and accuracy.

5.2 Error-Injection Prioritization

We study 38 different error injection prioritization schemes (Section 4.5). For brevity, we show results only for the 7 most effective schemes, in addition to the oracle best-case and oracle worst-case schemes.

Figures 9 and 10 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.

To understand the surprising result that Random performs the best, Figure 11 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 11 shows only four applications using Ref input, but the trends are representative across the workloads and inputs. The figure shows that the probability of finding an SDC injection shoots up within the first few injections. Upon investigation, we uncover an interesting insight – when a PC is SDC-causing, a large fraction of the injections in that PC result in an SDC outcome. Randomly choosing an injection therefore tends to quickly find an SDC for that instruction. Thus, we choose the Random error injection prioritization scheme for the remainder of the evaluations in this paper.

5.3 Minimization Plus Injection Prioritization

This section discusses the benefits of combining input minimization with error injection prioritization. Figure 6 shows the speedup in resiliency analysis, relative to Ref, from (1) using Min (discussed in Section 5.1.2), (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 5.1.2, 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 6). 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 6 and also discussed in Section 5.2).

Recall that the accuracy of Min_{EIP} is the same as that of Min (Section 5.1.3). 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%.

5.4 Input Prioritization

For safety-critical systems which may require even higher accuracy, Minotaur provides the additional optimization of *input prioritization*. This optimization can speed up the analysis of multiple inputs in an attempt to further improve SDC-PC identification without taking the performance hit of running resiliency analysis for each individual input in its entirety. Figure 12 shows the runtime of analyzing both Min_{EIP} and Ref_{EIP}, without and with input prioritization, normalized to the runtime of Min_{EIP} (Section 5.3).

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 5.3), 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 our study) to run for only a fraction of the original resiliency analysis time.

Figure 12 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.

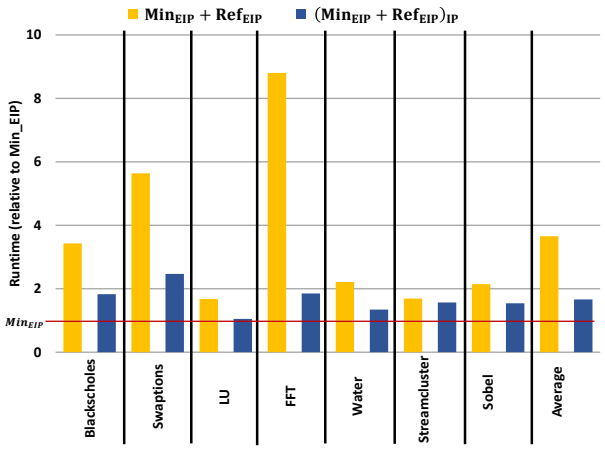

Figure 12. Resiliency analysis time for analyzing both Min_{EIP} and Ref_{EIP}, without and with input prioritization, normalized to analysis time for only Min_{EIP}.

6 Minotaur Extensions

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

6.1 Extension to Approximate Computing

The resiliency analyzer we chose (Approxilyzer [117]) can also be used for approximate computing. Approxilyzer can identify approximable instructions by grouping error sites differently. Whereas for resiliency we focus on SDC-Maybe and SDC-Bad outcomes (Section 2), Approxilyzer classifies an instruction as approximable if no egregious errors – Detected, DDC, or OC above a user-defined threshold – are observed for any dynamic instance of that instruction. We use the following user-defined thresholds: 1) for financial applications, errors in individual outputs that are smaller than a cent are tolerable and 2) for other applications, relative errors up to 5% in individual outputs are tolerable. We use the same Min and Ref inputs as in Table 1, and apply random error injection prioritization with early termination (we observe the same trend that randomized error injection ordering performs close to oracle best). For approximate computing, early termination is triggered when an error-injection reveals a PC as non-approximable, indicating that no further injections are required for that instruction.

For approximate computing, Minotaur's analysis time without error injection prioritization is the same as that for resiliency since we use the same Min and Ref inputs. That is, Min observes an average 4.1X speedup compared to Ref, due to Min's smaller size (Section 5.1.2). 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.

We use an accuracy metric similar to that in Section 4.4, 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%.

6.2 Selective Instruction Analysis

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

7 Related Work

Minotaur is the first work to systematically adapt and apply software testing techniques for fast and effective resiliency analysis. Section 2 describes the key background related work from software testing. We discuss other related work here.

Concepts similar to Minotaur: We discuss the most directly related works from other domains with similarities to different concepts in Minotaur. IRA [58] 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 [90] 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 [47, 51, 108, 129]. A recent example is SnaPEA [129] where convolution operations are terminated early if their output is predicted to be zero.

MinneSPEC [57] aims to provide reduced input workloads to improve performance (usually runtime of applications), which differs from our objective of uncovering SDC-PCs.

Hardware resiliency analysis: Many successful analysis techniques have been proposed to address soft errors in both hardware and software. They can be split into two groups:

1) Techniques that rely purely on static/dynamic program analysis of error-free execution to model error propagation. The widely used ACE [78] analysis is often used to measure the Architectural Vulnerability Factors (AVF) [43, 68,

78, 79, 82] of hardware structures. PVF [113] isolates purely (program or software dependent) architecture-level vulnerabilities in the AVF; ePVF [34] 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 [1, 123]. Shoestring [35] uses a compiler analysis to identify vulnerable program locations. While fast, these techniques do not precisely model an error's impact on the execution because they use information from only an error-free execution.

2) Techniques that employ error injections. While typically slower than the previous group, these techniques employ error injections at different hardware and software abstractions [18, 24, 31, 44, 54, 64, 65, 91, 109, 122]. Some rely predominantly on statistical error injections for vulnerability analysis [25, 41, 60, 62, 118]. Others combine program analysis with selected error-injection campaigns [2, 49, 54, 61, 108, 117]. For example, MeRLiN [54] 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 [61] uses error injections in static instructions to build an input-dependent model on top of Trident's [63] error propagation analysis to predict the instruction's SDC vulnerability. Approxilyzer analysis, used in this paper, 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 will impact the final output. Approxilyzer builds on Relyzer [49], so Minotaur can trivially apply to Relyzer.

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 [55, 62, 83], for example, can potentially benefit from the concepts of Minotaur to improve runtime and/or accuracy.

Approximate computing: Many techniques have been proposed that leverage approximate computing at the level of software [8, 11, 23, 71, 85, 100, 106, 110, 114, 121, 125, 127], programming languages [15, 20, 74, 86, 87, 101, 102] and hardware [5, 11, 16, 33, 42, 46, 53, 56, 77, 103, 105, 112, 126, 128, 133] for improved performance, energy, or reliability. Criticality-testing [3, 4, 21, 75, 84, 97, 115] of approximate computations is important for many domains. Minotaur is an orthogonal set of techniques that can be used to improve many of these analyses that use application input(s).

8 Conclusion and Future Work

We present Minotaur, a toolkit to improve the analysis of software vulnerability to hardware errors by leveraging concepts from software testing. Minotaur adapts several concepts from software testing for software bug detection to resiliency analysis for hardware error detection: 1) identifying test-case quality criteria, 2) test-case minimization, and 3) two adaptations of test-case prioritization. We evaluate Minotaur on a resiliency analysis tool, Approxilyzer. Minotaur's single-input techniques speed up Approxilyze's resiliency analysis by 10.3X on average while significantly improving SDC-PC detection accuracy (96% vs. 64% on average) for the workloads studied. Further, Minotaur presents a technique, *input prioritization*, which enables finding SDC-PCs across multiple inputs at a speed 2.3X 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, we plan to explore more input quality criteria (such as path coverage, loop coverage, or state coverage [7]) as well as develop new quality criteria geared specifically towards resiliency (e.g., criteria derived from ACE bits [78] or PVF [113]) or towards approximate computing (e.g., using parameter range coverage). We also plan to employ 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. We can also improve the Input Selector by tuning analysis speed vs. accuracy for multiple Refs and Mins with variable input quality thresholds.

To widen the applicability of Minotaur, we plan to apply it to other resiliency and approximation analysis techniques proposed in the literature, using a broader range of error models abstracted at lower and higher layers of the system stack than studied here.

Our end goal is a seamless integration of resiliency analysis (and hardening) into the standard software development and testing workflow. We believe Minotaur opens up many avenues for further research towards this ambitious end goal. Modern software development practices such as continuous integration encourage developers to continuously commit their code, which would be ideally checked for resiliency, making fast and accurate resiliency analysis techniques such as Minotaur even more important.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grants Nos. CCF-1320941, CCF-1421503, CCF-1703637, and CCF-1725734, by the 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 by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] Abhinav Agrawal, Bagus Wibowo, and James Tuck. 2017. Software Marking for Cross-Layer Architectural Vulnerability Estimation Model. In *Proc. of IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*.

[2] Khalique Ahmed. 2018. Relyzer+: an Open Source Tool for Application-Level Soft Error Resiliency Analysis. (July 2018).

[3] Riad Akram. 2017. *Performance and Accuracy Analysis of Programs Using Approximation Techniques*. Ph.D. Dissertation.

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

[5] Ismail Akturk, Nam Sung Kim, and Ulya R. Karpuzcu. 2015. Decoupled Control and Data Processing for Approximate Near-threshold Voltage Computing. *IEEE Micro Special Issue on Heterogeneous Computing* (2015), 70–78.

[6] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce. 2016. Evaluating Non-Adequate Test-Case Reduction. In *Proc. of International Conference on Automated Software Engineering (ASE)*.

[7] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press.

[8] Woongki Baek and Trishul M. Chilimbi. 2010. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *Proc. of International Conference on Programming Language Design and Implementation (PLDI)*. 198–209.

[9] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, Paolo Prinetto, and Luca Tagliaferri. 2003. Data Criticality Estimation in Software Applications. In *Proc. of International Test Conference (ITC)*.

[10] David E. Bernholdt, Al Geist, and Barney Maccabe. 2014. Resilience is a Software Engineering Issue. *Software Productivity for Extreme-Scale Science (SWP4XS) Workshop, Oak Ridge National Laboratory* (2014).

[11] Filipe Betzel, Karen Khatamifard, Harini Suresh, David J. Lilja, John Sartori, and Ulya Karpuzcu. 2018. Approximate Communication: Techniques for Reducing Communication Bottlenecks in Large-Scale Parallel Systems. *ACM Comput. Surv.* 51, 1, Article 1 (Jan. 2018).

[12] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[13] Matteo Bordin, Cyrille Comar, Tristan Gingold, Jérôme Guitton, Olivier Hainque, and Thomas Quinot. 2010. Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework. In *Embedded Real Time Software and Systems (ERTSS)*.

[14] Shekhar Borkar. 2005. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro* 25, 6 (2005).

[15] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability Type Inference for Flexible Approximate Programming. In *Proc. of International Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. 470–487.

[16] Rahul Boyapati, Jiayi Huang, Pritam Majumder, Ki Hwan Yum, and Eun Jung Kim. 2017. APPROX-NoC: A Data Approximation Framework for Network-On-Chip Architectures. In *Proc. of International Symposium on Computer Architecture (ISCA)*. 666–677.

[17] Jörg Brauer, Markus Dahlweid, Tobias Pankrath, and Jan Peleska. 2015. 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 (SAFECOMP 2015)*.

[18] Jon Calhoun, Luke Olson, and Marc Snir. 2014. FlipIt: An LLVM based fault injector for HPC. In *European Conference on Parallel Processing*. Springer, 547–558.

[19] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. 2014. Toward Exascale Resilience: 2014 Update. *Supercomput. Front. Innov.: Int. J.* (2014).

[20] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *Proc. of International Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. 33–52.

[21] Michael Carbin and Martin C. Rinard. 2010. Automatically Identifying Critical Input Regions and Code in Applications. In *Proc. of International Symposium on Software Testing and Analysis (ISSTA)*. 37–48.

[22] Eric Cheng, Shahrzad Mirkhani, Lukasz G. Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R. Stan, Klas Lilja, Jacob A. Abraham, Pradip Bose, and Subhasish Mitra. 2016. CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*.

[23] Ting-Wu Chin, Chia-Lin Yu, Matthew Halpern, Hasan Genc, Shiao-Li Tsao, and Vijay Janapa Reddi. 2018. Domain-Specific Approximation for Object Detection. *IEEE Micro* 38, 1 (January 2018), 31–40.

[24] Hyungmin Cho, S. Mirkhani, Chen-Yong Cher, J.A. Abraham, and S. Mitra. 2013. Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design. In *Proc. of International Design Automation Conference (DAC)*. 1–10.

[25] Jeffrey J. Cook and Craig B. Zilles. 2008. A Characterization of Instruction-level Error Derating and its Implications for Error Detection. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*.

[26] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *Proc. of International Symposium on Computer Architecture (ISCA)*.

[27] Nathan DeBardeleben, James Laros, John T Daly, Stephen L Scott, Christian Engelmann, and Bill Harrod. 2009. High-end Computing Resilience: Analysis of Issues Facing the HEC Community and Path-forward for Research and Development. *Whitepaper* (2009).

[28] Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. (2017). http://archive.ics.uci.edu/ml

[29] Martin Dimitrov and Huiyang Zhou. 2007. Unified Architectural Support for Soft-Error Protection or Software Bug Detection. In *Proc. of International Conference on Parallel Archtectures and Compilation Techniques (PACT)*.

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

[31] Waleed Dweik, Murali Annavaram, and Michel Dubois. 2014. Reliability-Aware Exceptions: Tolerating Intermittent Faults in Microprocessor Array Structures. In *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6.

[32] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. 2003. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proc. of International Symposium on Microarchitecture (MICRO)*.

[33] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Proc. of International Symposium on Microarchitecture (MICRO)*. 449–460.

[34] Bo Fang, Qining Lu, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. 2016. 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)*. 168–179.

[35] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[36] International Organization for Standardization. 2011. Road vehicles – Functional safety. Website. (2011). https://www.iso.org/standard/43464.html.

[37] Phyllis G. Frankl and Elaine J. Weyuker. 1988. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering* 14, 10 (Oct 1988), 1483–1498.

[38] Olga Goloubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. 2003. Soft-Error Detection Using Control Flow Assertions. In *Proc. of International Symposium on Defect and Fault Tolerance in VLSI Systems*.

[39] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause Reduction for Quick Testing. In *ICST*. 243–252.

[40] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2015. Cause reduction: Delta debugging, even without bugs. *STVR* 26, 1 (2015), 40–68.

[41] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang. 2003. Characterization of Linux Kernel Behavior Under Errors. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*.

[42] Peizhen Guo and Wenjun Hu. 2018. 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)*. 271–284.

[43] Manish Gupta, Vilas Sridharan, David Roberts, Andreas Prodromou, Ashish Venkat, Dean Tullsen, and Rajesh Gupta. 2018. Reliability-Aware Data Placement for Heterogeneous Memory Architecture. In *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*. 583–595.

[44] Meeta S Gupta, Jude A Rivers, Liang Wang, and Pradip Bose. 2014. Cross-layer System Resilience at Affordable Power. In *2014 IEEE International Reliability Physics Symposium*. 2B.1.1–2B.1.8.

[45] Jiawei Han, Micheline Kamber, and Jian Pei. 2011. *Data Mining: Concepts and Techniques* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[46] Jie Han and Michael Orshansky. 2013. Approximate computing: An Emerging Paradigm for Energy-efficient Design. In *ETS*. IEEE Computer Society, 1–6.

[47] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* (2015).

[48] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. 2012. Low-cost Program-level Detectors for Reducing Silent Data Corruptions. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*.

[49] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[50] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. 2009. mSWAT: Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *Proc. of International Symposium on Microarchitecture (MICRO)*.

[51] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. *CoRR* (2017).

[52] Muhammed Isenkul, Betul Sakar, and O Kursun. 2014. Improved Spiral Test Using Digitized Graphics Tablet for Monitoring Parkinson's Disease. In *The 2nd International Conference on e-Health and Telemedicine (ICEHTM-2014)*.

[53] Djordje Jevdjic, Karin Strauss, Luis Ceze, and Henrique S. Malvar. 2017. Approximate Storage of Compressed and Encrypted Videos. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 361–373.

[54] Manolis Kaliorakis, Dimitris Gizopoulos, Ramon Canal, and Antonio Gonzalez. 2017. MeRLiN: Exploiting Dynamic Instruction Behavior for Fast and Accurate Microarchitecture Level Reliability Assessment. In *Proc. of International Symposium on Computer Architecture (ISCA)*.

[55] Charu Kalra, Fritz Previlon, Xiangyu Li, Norman Rubin, and David Kaeli. 2018. PRISM: Predicting Resilience of GPU Applications Using Statistical Methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Article 69.

[56] S. Karen Khatamifard, Ismail Akturk, and Ulya R. Karpuzcu. 2017. On Approximate Speculative Lock Elision. *IEEE Transactions on Multiscale Computing Systems, Special Issue on Emerging Technologies and Architectures for Manycore Computing* (2017).

[57] A J KleinOsowski and David J. Lilja. 2002. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Comput. Archit. Lett.* 1, 1 (Jan. 2002), 7.

[58] Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. 2016. Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation. In *Proc. of International Conference on Programming Language Design and Implementation (PLDI)*. 161–176.

[59] Régis Leveugle, A Calvez, Paolo Maistri, and Pierre Vanhauwaert. 2009. Statistical Fault Injection: Quantified Error and Confidence. In *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*.

[60] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. 2017. 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)*.

[61] Guanpeng Li and Karthik Pattabiraman. 2018. Modeling Input-Dependent Error Propagation in Programs. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*. 279–290.

[62] Guanpeng Li, Karthik Pattabiraman, Chen-Yang Cher, and Pradip Bose. 2016. Understanding Error Propagation in GPGPU Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 240–251.

[63] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling Soft-Error Propagation in Programs. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*.

[64] Jianli Li and Qingping Tan. 2013. 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)*. 236–242.

[65] Man-Lap Li, Pradeep Ramachandran, Rahmet Ulya Karpuzcu, Siva Kumar Sastry Hari, and Sarita V. Adve. 2009. Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults. In *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*.

[66] Man-Lap Li, Pradeep Ramachandran, Swarup Sahoo, Sarita Adve, Vikram Adve, and Yuanyuan Zhou. 2008. Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*.

[67] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. 2008. Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[68] Xiaodong Li, Sarita Adve, Pradip Bose, and Jude Rivers. 2007. Online Estimation of Architectural Vulnerability Factor for Soft Errors. In *Proc. of International Symposium on Computer Architecture (ISCA)*. 341–352.

[69] ImageMagick Studio LLC. 2018. Image Magick. Website. (2018). https://www.imagemagick.org/.

[70] Galen Lyle, Shelley Chen, Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2009. An End-to-end Approach for the Automatic Derivation of Application-Aware Error Detectors. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*.

[71] Divya Mahajan, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, and Hadi Esmaeilzadeh. 2016. Towards Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration. In *Proc. of International Symposium on Computer Architecture (ISCA)*.

[72] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News* 33, 4 (2005).

[73] Albert Meixner, Michael E. Bauer, and Daniel J. Sorin. 2007. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proc. of International Symposium on Microarchitecture (MICRO)*.

[74] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. *SIGPLAN Not.* 49, 10 (Oct. 2014), 309–328.

[75] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of Service Profiling. In *Proc. of International Conference on Software Engineering (ICSE)*. 25–34.

[76] Asit K Mishra, Rajkishore Barik, and Somnath Paul. 2014. iACT: A Software-hardware Framework for Understanding the Scope of Approximate Computing. In *WACAS*.

[77] T. Moreau, J. San Miguel, M. Wyse, J. Bornholt, A. Alaghi, L. Ceze, N. Enright Jerger, and A. Sampson. 2018. A Taxonomy of General Purpose Approximate Computing Techniques. *IEEE Embedded Systems Letters* 10, 1 (March 2018), 2–5.

[78] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proc. of International Symposium on Microarchitecture (MICRO)*.

[79] Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. 2003. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-performance Microprocessor. In *Proc. of International Symposium on Microarchitecture (MICRO)*. 29–40.

[80] Shubhendu S. Mukherjee, Christopher T. Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. Measuring Architectural Vulnerability Factors. *IEEE Micro* 23, 6 (Nov. 2003), 70–75.

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

[82] Ajeya Naithani, Stijn Eyerman, and Lieven Eeckhout. 2017. Reliability-Aware Scheduling on Heterogeneous Multicore Processors. In *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*. 397–408.

[83] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. 2018. Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 749–761.

[84] Bernard Nongpoh, Rajarshi Ray, Saikat Dutta, and Ansuman Banerjee. 2017. AutoSense: A Framework for Automated Sensitivity Analysis of Program Data. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1.

[85] Jongse Park, Emmanuel Amaro, Divya Mahajan, Bradley Thwaites, and Hadi Esmaeilzadeh. 2016. AxGames: Towards Crowdsourcing Quality Target Determination in Approximate Computing. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 623–636.

[86] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. 2015. FlexJava: Language Support for Safe and Modular Approximate Programming. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. 745–757.

[87] Jongse Park, Xin Zhang, Kangqi Ni, Hadi Esmaeilzadeh, and Mayur Naik. 2014. ExpAX: A Framework for Automating Approximate Programming. In *Technical Report, Georgia Institute of Technology*.

[88] Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2008. SymPLFIED: Symbolic Program-level Fault Injection and Error Detection Framework. In *International Conference on Dependable Systems and Networks*.

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

[90] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proc. of Symposium on Operating Systems Principles (SOP)*. 1–18.

[91] Andrea Pellegrini, Robert Smolinski, Lei Chen, Xin Fu, Siva Kumar Sastry Hari, Junhao Jiang, Sarita V Adve, Todd Austin, and Valeria Bertacco. 2012. CrashTesting SWAT: Accurate, Gate-Level Evaluation of Symptom-Based Resiliency Solutions. In *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*.

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

[93] Sandra Rapps and Elaine J. Weyuker. 1985. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering* SE-11, 4 (1985), 367–375.

[94] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proc. of International Conference on Programming Language Design and Implementation (PLDI)*.

[95] Kevin Reick, Pia N Sanda, Scott Swaney, Jeffrey W Kellington, Michael Mack, Michael Floyd, and Daniel Henderson. 2008. Fault-Tolerant Design of the IBM Power6 Microprocessor. *IEEE Micro* (2008).

[96] Philippe Ricoux. 2013. European Exascale Software Initiative EESI2-Towards Exascale Roadmap Implementation. *2nd IS-ENES workshop on high-performance computing for climate models* (2013).

[97] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. ASAC: Automatic Sensitivity Analysis for Approximate Computing. In *Proc. of Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*. 95–104.

[98] Swarup Sahoo, Man-Lap Li, Pradeep Ramchandran, Sarita V. Adve, Vikram Adve, and Yuanyuan Zhou. 2008. Using Likely Program Invariants to Detect Hardware Errors. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*.

[99] Betul Erdogdu Sakar, M. Erdem Isenkul, Cemal Okan Sakar, Ahmet Sertbas, Fikret S. Gürgen, Sakir Delil, Hulya Apaydin, and Olcay Kursun. 2013. Collection and Analysis of a Parkinson Speech Dataset With Multiple Types of Sound Recordings. *IEEE Journal of Biomedical and Health Informatics* 17 (2013), 828–834.

[100] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning Approximation for Graphics Engines. In *Proc. of International Symposium on Microarchitecture (MICRO)*. 13–24.

[101] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. ACCEPT: A Programmer-guided Compiler Framework for Practical Approximate Computing. In *Technical Report UW-CSE-15-01-01, University of Washington*.

[102] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *Proc. of International Conference on Programming Language Design and Implementation (PLDI)*. 164–174.

[103] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. Doppelganger: A Cache for Approximate

Computing. In *Proc. of International Symposium on Microarchitecture (MICRO)*.

[104] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. 2017. 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)*.

[105] John Sartori and Rakesh Kumar. 2011. Architecting Processors to Allow Voltage/Reliability Tradeoffs. In *Proc. of International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 115–124.

[106] John Sartori and Rakesh Kumar. 2013. Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications. *Multimedia, IEEE Transactions on* 15, 2 (Feb 2013), 279–290.

[107] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V. Adve, and Helia Naeimi. 2014. GangES: Gang Error Simulation for Hardware Resiliency Evaluation. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 61–72.

[108] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V. Adve, and Helia Naeimi. 2014. GangES: Gang Error Simulation for Hardware Resiliency Evaluation. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, 61–72.

[109] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. 2015. FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance. In *European Dependable Computing Conference (EDCC)*. 245–255.

[110] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. 2011. Managing Performance vs. Accuracy Tradeoffs with Loop Perforation. In *SIGSOFT FSE*. 124–134.

[111] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A Debardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. 2013. Addressing Failures in Exascale Computing*. *International Journal of High Performance Computing* (2013).

[112] Haiyue Song, Xiang Song, Tianjian Li, Hao Dong, Naifeng Jing, Xiaoyao Liang, and Li Jiang. 2018. A FPGA Friendly Approximate Computing Framework with Hybrid Neural Networks. In *Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA)*. 286–286.

[113] Vilas Sridharan and David R. Kaeli. 2009. Eliminating Microarchitectural Dependency from Architectural Vulnerability. In *Proc. of International Symposium on High Performance Computer Architecture (HPCA)*.

[114] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. 2016. Proactive Control of Approximate Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 607–621.

[115] Anna Thomas and Karthik Pattabiraman. 2013. Error Detector Placement for Soft Computation. In *International Conference on Dependable Systems and Networks*. 1–12.

[116] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V. Adve. 2016. Approxilyzer Code Repository. Website. (2016). https://cs.illinois.edu/approxilyzer

[117] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V. Adve. 2016. Approxilyzer: Towards a Systematic Framework for Instruction-level Approximate Computing and its Application to Hardware Resiliency. In *Proc. of International Symposium on Microarchitecture (MICRO)*. 1–14.

[118] Radha Venkatagiri, Karthik Swaminathan, Chung-Ching Lin, Liang Wang, Alper Buyuktosunoglu, Pradip Bose, and Sarita Adve. 2018. Impact of Software Approximations on the Resiliency of a Video Summarization System. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*.

[119] Virtutech. 2006. Simics Full System Simulator. Website. (2006). http://www.simics.net.

[120] Nicholas J Wang and Sanjay J Patel. 2006. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (July-Sept 2006).

[121] Ting Wang, Qian Zhang, and Qiang Xu. 2017. ApproxQA: A Unified Quality Assurance Framework for Approximate Computing. In *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*. 254–257.

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

[123] Bagus Wibowo, Abhinav Agrawal, Thomas Stanton, and James Tuck. 2016. An Accurate Cross-Layer Approach for Online Architectural Vulnerability Estimation. *ACM Trans. Archit. Code Optim.* (2016), 30:1–30:27.

[124] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*.

[125] Hans-Joachim Wunderlich, Claus Braun, and Alexander Schöll. 2016. 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)*. 133–136.

[126] Chengwen Xu, Xiangyu Wu, Wenqi Yin, Qiang Xu, Naifeng Jing, Xiaoyao Liang, and Li Jiang. 2017. On Quality Trade-off Control for Approximate Computing using Iterative Training. In *Proc. of International Design Automation Conference (DAC)*. 1–6.

[127] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. 2018. Videochef: efficient approximation for streaming video processing pipelines. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 43–56.

[128] Siyuan Xu and Benjamin Carrion Schafer. 2017. Exposing Approximate Computing Optimizations at Different Levels: From Behavioral to Gate-Level. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 11 (2017), 3077–3088.

[129] Amir Yazdanbakhsh, Kambiz Samadi, and H Esmaeilzadeh. 2018. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*.

[130] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR* 22, 2 (2012), 67–120.

[131] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *TSE* 28, 2 (2002), 183–200.

[132] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. 2014. Using Test Case Reduction and Prioritization to Improve Symbolic Execution. In *ISSTA*. 160–170.

[133] Hengyu Zhao, Linuo Xue, Ping Chi, and Jishen Zhao. 2017. Approximate Image Storage with Multi-level Cell STT-MRAM Main Memory. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 268–275.

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