Harnessing Software Testing Techniques for Hardware Resiliency Analysis*

*

Abstract

As hardware technology scales, 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 in applications without the user's knowledge. Recently proposed techniques have used error injections to identify SDC-causing instructions, but these techniques remain too expensive for practical use or need to sacrifice accuracy to improve performance.

This paper shows how leveraging software testing methodology can advance the state-of-the-art hardware resiliency analysis, improving both accuracy and performance of SDC discovery. We illustrate the potential of software testing when applied to hardware resiliency analysis with the following contributions: 1) we introduce a new test coverage criterion for resiliency analysis, called PC coverage; 2) we use PC coverage to perform test-case minimization of standard benchmark inputs, creating smaller inputs that are easier to analyze; 3) we employ test-case prioritization, both for single inputs and across multiple inputs, to identify SDCs faster. We show that employing all these techniques can assist in SDC discovery by running, on average, 7X faster (and up to 22x) than the state-of-the-art resiliency techniques while uncovering 67% more SDC-causing instructions on average for the workloads studied.

1. Introduction

As we approach the end of CMOS scaling, efficient resiliency solutions are needed to address the increased likelihood of hardware failures in the field. Traditional solutions relying on redundancy in space and time are generally considered too expensive. There has therefore been significant research in low-cost solutions that rely on observations of anomalous software behavior as symptoms of hardware faults [11, 15, 21, 26, 37, 41, 56]. They are attractive due to their extremely low cost, but they can occasionally result in undetected silent data corruptions or *SDCs*.

The possibility of undetected SDCs has been an obstacle in the widespread adoption of software-anomaly-based resiliency techniques; therefore, there has been significant recent research focused on reducing these SDCs. For example, once instructions potentially resulting in such SDCs are identified, they can be protected using software checks and assertions in the application code [19, 41]. Most such work is evaluated using error injection campaigns, where an error is injected at different points of an application's execution (typically one error per execution) and the output is evaluated to determine if the error caused an unacceptable corruption.

These error injection campaigns are expensive in analysis time; e.g., an error that is masked or results in an SDC requires running the application from the point of error injection to completion. Naïvely testing all potential error sites[2] in an application is impractical [20]. An alternative is to perform a limited number of injections in randomly selected error sites, but this provides no insight whatsoever for error sites where errors are not injected, making those vulnerable to SDCs. The Relyzer tool [20] addresses this problem by showing that many error sites give equivalent error outcomes (for the studied error model); therefore, only one error injection is required for a given equivalence class of error sites. This dramatically reduced the number of required error injections and, for the first time, made it possible to predict the outcome of errors injected in virtually every instruction of a program for a given input (for the given error model). A more recent tool, Approxilyzer [54], builds on Relyzer to make finer distinctions between error outcomes, distinguishing between output corruptions that are acceptable approximations for the programmer from those that are truly unacceptable SDCs and must be protected.

While Approxilyzer[3] dramatically reduces the number of required error injections from a naïve scheme and provides vastly more information than limited randomly selected injections, it is still prohibitively slow. For example, the work on these tools so far only considers a single input for an application and analyzes only 99% of the error sites for each such application-input combination. To become commercially acceptable, resiliency solutions must be tested on inputs with maximal coverage of error sites on a range of workloads.

This paper observes that we can make testing for hardware errors more effective by leveraging techniques for testing for software bugs. Comprehensive testing for software bugs continues to remain an active area of research. While the holy grail of perfect test coverage (and bug elimination) is still out of reach, the field of software testing has made significant progress with high commercial impact. This paper takes an ini-

*This work was supported in part by the National Science Foundation under Grant CCF-1320941 and by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

[2]An error site in this work is a bit in a source or destination register in a dynamic instruction, analogous to previous work [20].

[3]We henceforth use Approxilyzer to refer to the techniques originally proposed in Relyzer as well.

tial step towards the direction of systematic testing of hardware resiliency solutions analogous to software bug testing, with the eventual goal of incorporating hardware resiliency analysis seamlessly within the software testing workflow. Specifically, we make the following contributions.

1) *Quality of Test Cases:* At the heart of testing a program is a set of test cases (inputs) for the program, characterized by a certain goodness or quality criterion. A commonly used software test quality criterion is *statement coverage;* i.e., the fraction of statements in the program exercised by the test. We adapt this criterion to quantify the goodness of an input at the machine level in the form of *PC coverage;* i.e., the fraction of static instructions (program counters or PCs) in the program exercised by the analyzed input. We find that this simple coverage criterion is very effective – as discussed below, it enables us to judiciously select minimized inputs that improve both analysis overheads and reveal more SDCs, compared to reference inputs (called *Ref* below).

2) *Test-Case Minimization:* This software testing technique starts from a high quality, potentially expensive (e.g., in terms of execution time) test and creates a smaller, cheaper (e.g., with faster execution time) test that has similar quality. We adapt this technique to our purpose to create inputs (referred to as *Min*) that are smaller than but have similar PC coverage as the Ref inputs for our benchmarks. We find that on average Approxilyzer requires 3X fewer error injections for Min than for Ref while also taking much less time per injection. Since previous work was able to analyze injections into only 99% of the error sites (referred to as 99% error site coverage), we compared Min with 100% error site coverage (*Min100*) against Ref with 99% error site coverage (*Ref99*). Since minimization is still a creative and mostly manual task, in this paper we present a methodology for systematically identifying Min inputs and are the first to do so in the context of hardware reliability. We found that compared to Ref99, Min100 discovered 67% *more* PCs that led to SDCs and does so on average 1.7X *faster* than Ref99.

3) *Test-Case Prioritization (Single Input):* The software testing technique of test-case prioritization systematically prioritizes test cases to find critical software failures as early as possible [60]. We apply this technique to prioritize error injections, and terminate injections for instructions as soon as they are classified as SDCs. Applying this technique to Min100, we see an average 7x speedup over the prior techniques at Ref99 without these optimizations.

4) *Test-Case Prioritization (Multiple Inputs):* Although Min is a great improvement on Ref, there are a few SDCs that Ref finds that Min does not. This final technique aims to curb this issue, by prioritizing multiple inputs to combine their results in a hierarchical fashion. To do this, we first prioritize running error injections in Min (with early termination). Then we run error injections in Ref, but only for PCs that are not already exposed as generating SDCs by Min. This combination always identifies the maximum number of PCs that are vulnerable to SDCs, while also running 2.47X faster on average than running both Min and Ref analysis in their entirety. The multiple-input prioritization adaptation is beneficial for applications that require very high resilience (minimal undetected SDCs) and can tolerate the increase in runtime from testing multiple inputs.

This work does not claim to find the optimal testing strategy or eliminate all possible SDCs exposed by all possible inputs. Our goal is to take a step towards a more disciplined methodology of testing hardware resiliency solutions based on software anomaly monitoring. We believe this work shows that such disciplined methodologies can be used both to identify more vulnerabilities and to increase analysis speed, often both together. It opens up many avenues of future work inspired by decades of software testing literature.

This work is orthogonal to other potential improvements in SDC analysis tools [27, 32, 31, 13] and are discussed in detail in the Section 6.

2. Background

2.1. Relevant Software Testing Techniques

Software testing is the process of executing a program or system with the intent of finding failures [33]. It broadly involves any activity aimed at evaluating an attribute or capability of a program or system to determine whether it meets its requirements. The purpose of testing can be quality assurance, verification and validation, or reliability estimation.

Due to the complexity of modern software, it is infeasible to completely test a program to reveal all its failure modes. Instead, software testing methodologies provide a trade-off between cost, time, and quality to help the user understand the quality of the program under test and determine when the software is "good enough" to deploy. In this section, we discuss some of the techniques and best practices adopted by the software testing community and associated definitions.

2.1.1. Quality of Test Cases In software testing, a *test case* is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly, and a *test set (TS)* is a collection of one or more test cases. Since there may be an intractably large number of test cases for a given system under test, selecting appropriate test cases has a large impact on testing time and the thoroughness of the testing. Evidence of the effectiveness of a set of test cases is often sought by measuring if it satisfies some quality criterion deemed important to thorough testing.

Statement coverage, which measures the number of statements executed in a program, is typically regarded as a baseline criterion that is enough for use in most commercial code. Another criterion, branch coverage, which checks that each edge of branch in a program is taken, is the required quality measure for automobile system safety, as outlined by the ISO 26262 standard for automotive safety [1]. While other forms of coverage may result in higher quality, they may be too expensive and are not used much in practice (e.g., def-use or path coverage). The quality criterion differs based on the intent of the test, and will take into account the application domain, complexity of analysis, and evaluation goals. A thorough analysis of various testing criteria has been explored extensively in the software testing literature [3].

2.1.2. Test-Case Minimization While running larger (or more) test cases is likely to increase the thoroughness requirement for "good enough" testing, time and cost limit the size (or number) of test cases that can be practically executed. *Test-case minimization* is a technique used to minimize the testing cost in terms of execution time [60, 62, 39, 17, 16, 2]. The purpose of test-case minimization is to generate a smaller test case that satisfies similar or (ideally) the same requirements as the original test case (e.g., the quality criteria described by branch coverage). Minimizing test cases not only enables faster testing but also helps in debugging [61].

2.1.3. Test-Case Prioritization Even in a well-planned testing workflow, it is possible that time and cost are not sufficient to execute all the planned test cases. It thus becomes necessary to prioritize and select test cases in a suitable way such that critical failures can surface sooner rather than later [60]. Test-case prioritization techniques schedule test cases for execution in an order that attempts to increase their effectiveness in meeting some performance goal. Prioritizing and executing the most important test cases first also ensures important problems can be found and dealt with early. A large number of test-case prioritization techniques have been proposed, as surveyed by Yoo and Harman [60].

2.2. Approxilyzer

Approxilyzer is an open-source error analysis framework used to understand the impact of errors in a program's execution on its final output quality [54]. Approxilyzer uses a single-bit error model in registers of dynamic instructions to identify all possible error sites in a program. Using program analysis and some heuristics, Approxilyzer groups error sites that behave similarly into *equivalence classes*. It then identifies just one error injection experiment to perform on a representative error site (called a *pilot*) per equivalence class. It uses the outcomes of the error injection experiments to reason about virtually all the error sites in the application. Thus, Approxilyzer is able to construct a comprehensive error impact profile for a given program, with relatively few error injection experiments. Approxilyzer uses the generated error impact profile to identify which instructions in the application need resiliency protection or which instructions are first-order candidates for approximation.

This work uses Approxilyzer's resiliency analysis as a proof-of-concept to demonstrate the employment of the various software testing techniques.[4] In particular, we use Approxilyzer to identify the set of static instructions that produce a Silent Data Corruption (SDC) in the presence of errors, and hence, require resiliency protection.

[4]The same techniques can also be applied to study approximation opportunities in a program, but we do not focus on that in this work.

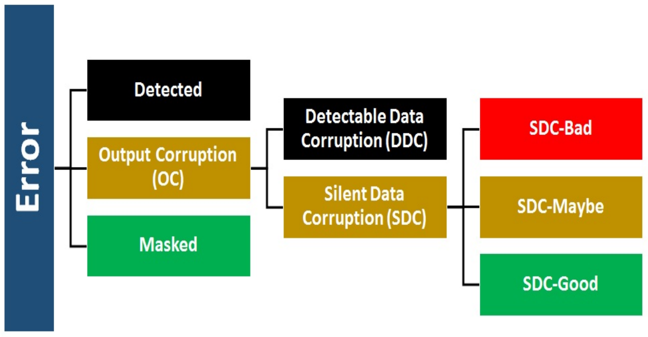

Figure 1: A classification of errors, from [54].

We adopt the quality-aware error outcome classification used in the Approxilyzer work [54], which classifies each injection as one of six possible outcomes, shown in Figure 1:

- **Detected**: An error which raises software symptoms and can hence be caught using various low-cost detectors [41] before the end of execution.
- **DDC**: An output corruption (OC) which is detectable via low-cost mechanisms such as range detectors [19].
- **SDC-Bad**: SDCs with very high quality degradations.
- **SDC-Maybe**: SDCs that may be tolerable provided an acceptable quality threshold.
- **SDC-Good**: SDCs that produce negligibly small quality degradations.
- **Masked**: Errors which produce no output corruption.

A complete discussion of the various error categories and their implications appears in [54]. We assume conservatively that no quality threshold has been provided by the user and hence the entire spectrum of SDC-Maybe errors along with the SDC-Bad error sites and constituent instructions need to be protected. In the rest of the paper we will refer to SDC-Bad and SDC-Maybe outcomes collectively as SDC.

Approxilyzer conservatively sets the category of a PC based on the worst-case outcome for all error injections based on that PC. The order of outcomes, from best to worst is: Masked, Detected, DDC, SDC-Good, SDC-Maybe, then SDC-Bad. Only the true SDCs (SDC-Maybe and SDC-Bad) need protection – other error outcomes can rely on low-cost detectors.

3. Approach

Approxilyzer has advanced the state-of-the-art in resiliency analysis by making it feasible to study virtually all program error sites (dynamic instruction instance + register + bit). However, analyzing 100% of the error sites in programs is still extremely costly, so previous work has reported results for 99% of the error sites. While this reduction in *error-site coverage* reduces the analysis time considerably, the analysis time is still large, restricting analysis to a few applications. For example, previous work analyzed only five benchmarks with a single (reference) input each [54] and reported that the experiments took multiple days on a cluster with 200 cores. Furthermore, the lower error-site coverage potentially misses an unknown number of SDCs in the error sites that are not explored. Finally, in choosing the pilots to cover the 99% error sites, previous work prioritized the pilots (and hence, the equivalence classes) that covered the most error sites. Section 5.2.1 shows that this prioritization leaves a large fraction of PCs unexplored, potentially compromising reliability.

Our work can significantly reduce Approxliyzer's analysis cost by showing how to judiciously choose an input and injection prioritization strategy, inspired by the test-case minimization and test-case prioritization techniques from the software testing literature (Section 2.1). We start with one given reference input for each application, referred to as *Ref*, assuming it to be our gold standard for that application. Section 3.1 proposes a criterion, called *PC coverage* for evaluating smaller inputs that could be analyzed faster but without compromising accuracy of the analysis, relative to Ref. Although the actual input generation is usually application-specific and requires domain knowledge (analogous to test-case generation in software testing), Section 3.2 uses the PC coverage criterion to develop a systematic methodology to generate an appropriate minimal input, referred to as *Min*. Section 3.3 proposes a variety of prioritizations for error injections within a single input, Ref or Min. Our experimental results show that, in general, Min provides better accuracy with lower analysis time than Ref. Nevertheless, there are a few SDCs that Ref uncovers but Min does not and vice versa. Section 3.4 uses these results to motivate a hierarchical approach that prioritizes error injections across a combination of different inputs. Here we focus on combining injections in Min and Ref, but the same concept can be used to study combinations of other inputs (e.g., multiple reference or multiple minimized inputs).

The concepts proposed in this section (PC coverage, input minimization, injection prioritization for single inputs, and injection prioritization for multiple inputs) are mostly independent of error-site coverage. For example, given a desirable and practical error-site coverage for Ref, our Min can be run at the same or higher error-site coverage. The former will potentially see a higher improvement in analysis time while the latter in accuracy. Further, injection prioritization is mostly orthogonal to input minimization. Thus, injection prioritization can be applied directly to Ref or Min (for additive analysis time improvements over those already provided by Min).

3.1. Input Quality Criterion: PC Coverage

Many types of coverage criteria can be used to measure a test case's quality (Section 2.1.1). Statement coverage, which requires covering each source-code program statement, is simple yet effective in finding bugs in software, so it is a commonly used criterion in software testing [3]. However, statement coverage may not be enough for measuring resiliency of compiled code. The objective of our resiliency analysis is to find SDC-causing static instructions, corresponding to unique program counter (PC) values in the compiled code. Obtaining 100% statement coverage at the source-code level may not obtain 100% PC coverage at the compiled-code level. For example, consider a statement with a ternary operator, "v = c ? e1 : e2", or an "if" statement with many conditions that the compiler may short-circuit. Executing this statement at the source-level may not execute all compiled instructions, resulting in a coverage hole. Thus, we use a new coverage criterion, called **PC coverage**, which simply measures how many PCs are executed by a test case.

While more complex coverage criteria can be used for resiliency analysis – including branch coverage or def-use coverage [3, 33, 38, 14] – we chose to start with PC coverage because of its simplicity and similarity to the popular statement coverage criterion in software testing. Our evaluation in Section 5 shows that PC coverage is surprisingly effective – it results in Min inputs that provide improved analysis time and accuracy relative to Ref. We discuss possible refinements for even better results; however, exploring the tradeoff of potentially increased analysis times for small improvements in accuracy with more complex test quality criteria is beyond the scope of this first paper on applying software testing inspired techniques.

3.2. Input Minimization

Judiciously reducing the input size for an application can greatly improve resiliency analysis by speeding up error-injection simulations and reducing the number of error sites needing injections. We designed a systematic technique for choosing a minimal input, Min, provided a reference input, Ref. The technique uses PC coverage as the test quality criterion, although it can be applied to any other criterion.

We assume a provided reference input (Ref) as our "gold standard," and we measure PC coverage of Min relative to Ref. More precisely, we define PC coverage of Min (relative to Ref) as the intersection of PCs executed by Min and Ref divided by the total PCs executed by Ref. Any PCs that Ref covers but Min does not cover can become a cause of inaccuracy (missed SDCs) for Min. It is, however, also possible that Min covers some PCs that are not part of Ref (e.g., if there is a small cutoff value for some input below which different instructions are executed); In this case, Min might expose potential SDCs that Ref will miss.

Algorithm 1 describes our selection process. Generating ideal test cases remains a "holy grail" even in software testing [3]. Not surprisingly, our algorithm consists of several heuristics and steps that need to be refined with domain-specific knowledge. Nevertheless, we found straightforward adaptations to work well with our applications. The key steps for generating a minimal input are as follows.

1. Run the application with its Ref input and measure the base PC coverage provided by Ref.
2. Extract the input parameters for the application. These may already be provided by the benchmark suite.

Algorithm 1 Minimize a set of application parameters while achieving target coverage. The PROFILE subroutine uses architectural simulation to collect PC coverage and timing information.

```
1:  procedure MINIMIZE($InputParamList, TargetThreshold$)
2:      $RefCoverage$ = PROFILE($InputParamList$)
3:      $TargetCoverage = RefCoverage * TargetThreshold$
4:      Reorder $InputParamList$ for minimization
5:      for $Param$ in $InputParamList$ do
6:          Minimize $Param$ to smallest legal value
7:          $MinCoverage$ = PROFILE($InputParamList$)
8:          while $MinCoverage < TargetCoverage$ do
9:              Incrementally adjust $Param$
10:             $MinCoverage$ = PROFILE($InputParamList$)
11:     return $InputParamList$
```

3. Order the input parameters for minimization. Although any order would suffice, prioritizing parameters that have a larger impact on application runtime can produce a better overall Min selection.
4. Starting with the first input parameter, change the value of the parameter to minimize the application execution time.
5. Measure the PC coverage of the execution with the new input parameter(s).
6. If the PC coverage matches or exceeds the desired PC coverage (measured relative to Ref's coverage), move on to the next parameter to minimize.
7. Else, adjust the value of the parameter incrementally up to the original value of Ref. Using binary search or similar techniques can reduce the number of iterations for updating the parameter.
8. Once all parameters are minimized to the target PC coverage, the set of input parameters represents the Min input.

Section 4.2 describes in detail how this algorithm can be applied to systematically minimize the applications we study.

3.3. Single-Input Error-Injection Prioritization

Approxilyzer chooses pilots for error injections in the order of the size of the equivalence class they represent and ceases error injection when it reaches the desired error-site coverage (99% in [54]). We observe, however, that the primary goal of error injections in many studies is to determine whether a given (static) PC must be protected against an SDC. In this case, once an injection reveals an SDC outcome, there is no need to perform further injections for any error sites corresponding to that PC. We refer to this form of injection prioritization as *early termination*. Given early termination, it is advantageous to order injections such that SDC-causing error sites are run first. We explore several strategies for such ordering. These ideas are inspired by the concept of test prioritization in the testing literature (Section 2.1.3). As previously mentioned, an error site in Approxilyzer is a bit in a source or destination register of a dynamic instruction instance. We explored the following specific prioritization strategies and their various combinations.

- **Equivalence class size**: Ordering injections based on the equivalence class size represented by the error site.
- **Bit position of registers**: Injecting into specific bits first (such as the MSB or LSB).
- **Dynamic instance of error site**: Error sites from an earlier dynamic instance may be more prone to SDCs than later dynamic instances.
- **Register type (integer vs. floating point)**: Certain register types could be more susceptible to SDCs than others.
- **Operand type (source vs. destination)**: Prioritizing source vs. destination register may also show a pattern for SDC-causing instructions. Ordering injections based on operand type can expose this trend.
- **Random ordering**: Included as as simple scheme for prioritization.

3.4. Multiple-Input Prioritization

Section 5 shows that, in general, Min identifies almost all the SDCs identified by Ref as well as several that are not identified by Ref, but Min may still miss a few SDCs. For mission-critical applications with high resiliency requirements, this motivates analyzing multiple inputs. A naive scheme would be to analyze each input in its entirety, a prohibitively expensive proposition. Instead, we propose a hierarchical scheme called *multiple-input injection prioritization*. Applied to Min and Ref, the scheme proceeds as follows: We first run Approxilyzer on Min to relatively quickly identify many SDC-causing instructions. Subsequently, we run Approxilyzer with Ref *only* for the PCs that Min failed to identify as SDCs. This technique has lower analysis time than the naive method of analyzing both Min and Ref in their entirety. It can also have lower analysis time than running Ref alone, especially if Ref were to be run at the same error-site coverage as Min. The technique does have a higher analysis time than running only Min, but the technique finds SDCs that are a combination of those found by both Min and Ref, resulting in higher accuracy than running only Min or only Ref.

4. Methodology

4.1. Evaluation Infrastructure

Approxilyzer's error injector is based on Wind River Simics [55] and GEMS [29]. We use Simics to gather PC coverage information. All applications are run on OpenSolaris compiled with full optimizations enabled to the SPARC V9 ISA. Workloads studied are summarized in Table 1. Error-injection simulations are run on a 150-node cluster of 2.4GHz Intel Xeon E7-8870 processors with 252GB of total memory. Most error injection studies typically run only thousands of random error injections per application. Using Approxilyzer, we systematically choose injections for a more comprehensive study per application, resulting in many more injections.

For this study, we ran 14.5 *million* error injections across five applications.

Application	Ref Input	Min Input
Blackscholes [5]	64K options	21 options
Swaptions [5]	16 options 5000 simulations	1 option 1 simulation
LU [59]	512x512 matrix 16x16 block size	16x16 matrix 8x8 block size
FFT [59]	2^{20} data points	2^8 data points
Water [59]	512 molecules	216 molecules

Table 1: Key input parameters provided by the benchmark suite, and the minimized inputs chosen by test minimization

4.2. Input Minimization

We evaluate five applications from the Parsec 3.0 [5] and the SPLASH-2 [59] benchmark suites. The rest of this section discusses in detail the procedure used in choosing the Min input for each application, following the Algorithm 1. Table 1 summarizes the input parameters used for Ref and Min.

Blackscholes - Blackscholes calculates European options prices using the Black-Scholes partial differential equation. The options are stored in a single input file, with the reference input consisting of 1,000 unique options repeated many times to obtain a total of 65,536 options. The only input parameter for Blackscholes is the number of options. Starting from 1 option, we iterate through the available options to find an option that matches our target coverage. Since 1 option does not produce high coverage, we increment the number of options to two, and iterate again, checking the PC coverage. At 21 options, we obtain 100% PC coverage relative to Ref for a certain set of options, and use that as our Min for Blackscholes.

Swaptions - Swaptions calculates the swaptions price within a portfolio and uses the Heath-Jarrow-Morton (HJM) framework, which is a Monte-Carlo (MC) simulation. The application accepts the number of swaptions and the number of MC simulations as its input parameters. The reference input used for comparison contains 16 swaptions and 5,000 simulations. Of these two inputs parameters, we minimize first for the number of simulations, followed by the number of swaptions, because the number of simulations influences the runtime more than the number of swaptions. By reducing the number of simulations from 5,000 to 1 and the number of swaptions from 16 to 1, we obtain 99.91% PC coverage. Although not exactly 100%, this minimization is extremely good for a loss of coverage of just 2 out of 2,270 PCs. Going above these values results in diminishing returns for coverage increase with increased analysis time.

LU - LU is a matrix-based application which factorizes a dense square matrix. The input matrix is divided into smaller blocks to improve spatial locality. The two input parameters are the matrix size and the block size, which are 512x512 and 16, respectively, in Ref. This application has a dependency between the two inputs, since the block size cannot be larger than the matrix size. Thus, we minimize the matrix size first, and then increase the block size to the maximal matrix size, while checking the PC coverage. Using this procedure to adjust the input parameters, we attain 100% PC coverage with an input matrix size of 16x16 and a block size of 8.

FFT - FFT is a matrix-based application that computes the FFT of a randomly generated square matrix. The matrix size is the key parameter for this specific implementation of FFT. The Ref matrix is of size 2^{20}, or 1024x1024. Minimizing the matrix to 2x2 gives the PC coverage of 61.83%, which is not near our target of 100%. We incrementally increase the size of the matrix by powers of 2, until we obtain 100% PC coverage with 2^8, or 16x16.

Water - Water-Spatial computes forces between water particles. The application has a uniform 3D grid of cells in which the water molecules interact. The input parameter that scales best proportionally to the runtime is the number of molecules being simulated. We incrementally increase the number of molecules until we find that at 216 molecules we obtain 99.89% PC coverage relative to Ref. Above this value, the PC coverage asymptotically gets closer to 100%, but the slower runtime associated with having a higher molecule count provides less benefit. At 216 molecules, we also find a few PCs not covered by Ref. Reducing any of the other parameters such as the number of timesteps below their Ref values results in lower PC coverage than acceptable. Consequently, we leave other parameters identical to Ref.

4.3. Error-Injection Prioritizations

To implement single-input injection prioritization, we order the injections that Approxilyzer performs based on the various prioritization schemes covered in Section 3.3. Before a new injection is performed, Approxilyzer first checks to see if the PC of the associated dynamic instance has been injected into previously, by hashing a list of PCs explored and the result of the injections. If a PC has been identified as an SDC-causing instruction, the new injection is not performed. This process is automated and incorporated into the Approxilyzer workflow.

Since this optimization is applicable to any input, it is enabled for both Ref and Min analysis. For each prioritization scheme, we test both ascending and descending ordering. For example, we inject into bit positions by prioritizing the LSB (ascending) for injection, as well as the reverse, by prioritizing MSB (descending). Similarly, for register type, we explore injecting first into all integer registers followed by all floating point registers, and the reverse. In total, we explore 38 combinations of these various orderings.

To evaluate these prioritization techniques and understand the bounds on the gains provided, we also report an Oracle best and worst case. The best case assumes that for a given PC, the Oracle always picks an SDC-causing injection first, ensuring that only one injection is needed to identify an SDC-causing PC, if a PC indeed has any such injection. For the worst case, the Oracle performs all the non-SDC causing injections in the PC before picking the first SDC-causing injection, effectively eliminating the benefit of early termination.

For multiple-input prioritization, we first run Approxilyzer with Min (with single-input injection prioritization enabled), and save a list of SDC- causing PCs. Then, before running Approxilyzer with Ref, we remove all injections associated with SDC-causing PCs, as identified by Min. Subsequently, we run Approxilyzer with Ref, with single-input injection prioritization enabled.

4.4. Accuracy Analysis

This paper uses the term *accuracy* to refer to the ability to find SDCs. To measure the accuracy of our test quality criterion of PC coverage as applied to a minimized input, we compare the SDC-causing PCs found by Min with those found by Ref. Although we use Ref as the baseline comparison, there is no known ground truth for the total number of SDC-causing PCs in these applications. Similarly, in software testing, a test suite may not have all test cases that find bugs. Thus, while we focus on evaluating whether Min finds all SDCs that Ref find, we also show that Min finds some SDCs that Ref does not.

For each PC outcome in Min, we compare with the PC outcome in Ref. If the results match, with both producing an SDC or not an SDC, we consider the PC as a correctly classified PC. In the case that Min and Ref do not match, we distinguish whether Ref found the SDC or Min found the SDC for further analysis. For PCs explored by one but not both inputs (e.g., because of the error-site coverage chosen), we classify SDC-causing PCs as “Exclusive" to either Min or Ref. For example, if Min found an SDC-causing PC that was not injected into by Ref, we label that SDC as Exclusive to Min. Due to the prohibitively high analysis time for Ref100 (Ref with 100% error-site coverage) for some applications, and because prior work used Ref99 (Ref with 99% error site coverage), we focus accuracy comparisons between Ref99 and Min100 (Min with 100% error-site coverage).

The single-input injection prioritization schemes do not affect accuracy because they find the same SDCs as without the optimization, albeit faster. The multiple-input prioritization scheme has the same accuracy as the combination of the two inputs because it finds all SDCs found by the individual inputs.

4.5. Performance Analysis

For this paper, *performance* refers to the runtime performance of the resiliency-analysis tool (Approxilyzer) and not to the performance of the program. Thus, it measures the overhead of resiliency analysis. Approxilyzer's overhead includes the following.

1. Static and dynamic analysis time for equivalence class generation ($t_{analysis}$)
2. Total injections required for a desired error site coverage per outcome category ($I_{masked\_total}, I_{det\_total}, I_{OC\_total}$)
3. The average runtime of error injection experiments per outcome category ($t_{masked\_ave}, t_{det\_ave}, t_{OC\_ave}$)

The input to an application affects all three aspects of the total runtime, as the size of the input directly influences the number of dynamic instructions and the total runtime of an application. Since different error injection outcomes results in different simulation times (for e.g., error injection simulations resulting Masked and OC run to completion, but Detected outcomes result in aborted executions) , the average runtime is calculated for each outcome category.

To avoid measurement inaccuracies with nearly 15 million injections, average runtimes per injection are statistically calculated for different injection outcomes. We sampled 1,000 error-injection experiments for each of masked, detected, and OC outcomes per application. The 3,000 samples correspond to a 99.8% confidence level with 5% error margin in timing measurements [23].

The total runtime is calculated as:

$$TotalRuntime = t_{analysis} + \sum_n I_{n\_total} \times t_{n\_ave} \tag{1}$$

where the error-injection totals are weighted for each of the types of outcomes, $n \in \{masked, detected, OC)\}$.

In practice, error injections dominate the total runtime of analyzing an application, taking hundreds of wall clock hours in comparison to minutes for analysis and equivalence class generation. Thus, even though the analysis time is much shorter for Min (< 30 minutes) compared to Ref (< 2 hours), it is negligible compared to the total time for injection experiments.

5. Results

5.1. PC Coverage

Applying the algorithm described in Section 3.2 for generating Min inputs produces PC coverage results in Table 2. The table shows the PC coverage for the Min input relative to the Ref input for our applications. The first several columns show the absolute number of PCs covered by Ref, Min, both Ref and Min, and only by one of Ref or Min. The last column shows the percentage of Ref PCs covered by Min. BlackScholes, FFT, LU show 100%, while Swaptions and Water are over 99.8%. The results validate that our algorithm generates Min inputs with very high quality, as measured by PC coverage.

5.2. Input Minimization

5.2.1. Error-Site Coverage Running Approxilyzer's static and dynamic analysis results in a list of error sites for injections. Previous work found it intractable to perform error injections that covered *all* error sites [20, 54]. Instead, that

Application	Ref PCs Covered	Min PCs Covered	PC Intersection	Disjoint PCs	PC Coverage (%)
Blackscholes	538	538	538	0	100
Swaptions	2270	2268	2268	2	99.91
LU	1124	1124	1124	0	100
FFT	1483	1483	1483	0	100
Water	3740	3740	3736	8	99.89

Table 2: PC coverage. The last column shows the PC coverage of Min relative to Ref (column 4/column 2) as a percentage.

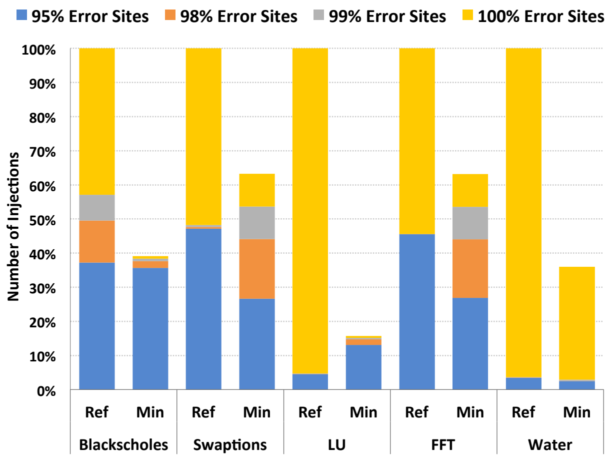

Figure 2: Number of injections for different error-site coverage

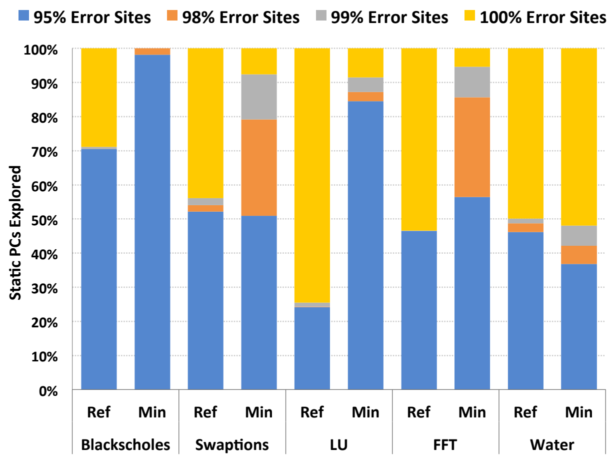

Figure 3: Static PCs explored for different error-site coverage.

work chose to inject errors in 99% of all error sites, referred to as 99% *error-site coverage*. The equivalence classes are ordered by the number of error sites they contain, and errors are injected into a random pilot of the equivalence class until the first 99% of the error sites are explored.

Figure 2 shows the total number of error injections needed for different values of error-site coverage, ranging from 95% to 100%, for Ref and Min. The figure shows that for all applications, Ref sees a sharp increase in the total number of injections going from 99% to 100% error-site coverage, while Min always have fewer injections overall to run compared to Ref100. By using the Min input for analysis, we can explore all 100% error sites in an application. In this work, we target 100% error-site coverage for Min (*Min100*) while comparing with the reference input at 99% error-site coverage (*Ref99*).

A potential implication of using Ref99 instead of Ref100 is that the error injections may miss SDC-causing error sites in some PCs altogether. Figure 3 shows the number of static PCs explored for an error-site coverage ranging from 95% to 100% (normalized to that for 100%) for both Ref and Min. The figure clearly shows that dropping error-site coverage even to 99% may drop the exploration of a large number of PCs (potentially hiding SDCs). The reduction of error-site coverage can result in less than ideal exploration of a program's resiliency.

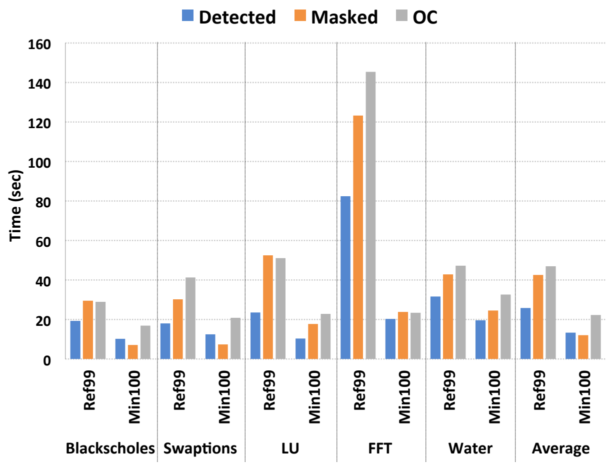

Figure 4: Average runtime for a single injection.

5.2.2. Performance As discussed in Section 4.5, the total runtime for a tool such as Approxilyzer is primarily dictated by the number of error injections and the injection's runtime.

Figure 4 shows the runtime of a single error injection for Min and Ref for different error outcomes on our simulation infrastructure as described in Section 4. Min clearly takes less time to complete an injection for all types of outcomes compared to Ref. This is a direct result of input minimization, since the inputs are now smaller and thus the application can

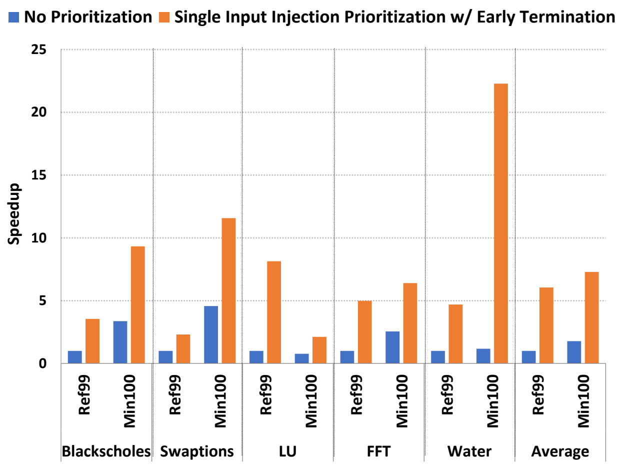

Figure 5: Speedup from Min (blue bars), the early termination optimization (orange bars), and both, relative to Ref100 with no optimization.

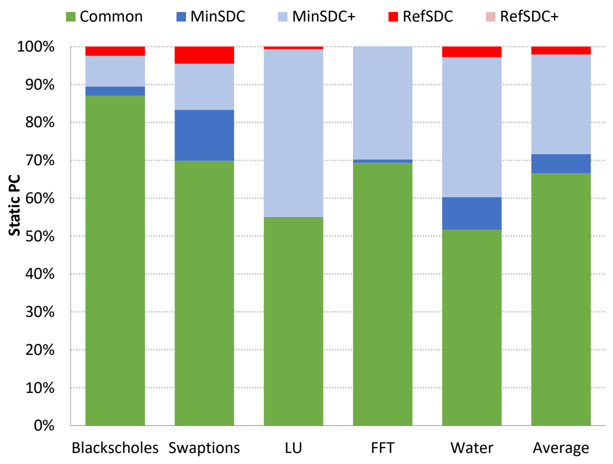

Figure 6: Accuracy of Min100 vs. Ref99

run to completion much faster.

Using Equation 1 on the data in Figure 2 provides the total runtime for error injections with Ref99 and Min100. Figure 5 shows the speedup relative to Ref99 with the blue bars. By using Min100 (without optimizations), the analysis time of Approxilyzer is, on average, 1.7X faster than for Ref99 (up to 4.6X faster), while additionally covering more error sites.

For one application (LU), Min100 is slightly slower than Ref99. This is because of the larger number of injections needed by Min100 over Ref99, and the time per injection does not compensate for the difference. Even though Min100 for LU has about 3.3X the number of injections needed by Ref99 (amounting to 418,000 more injections), it is not that much slower overall, while also obtaining higher error-site coverage. The next section shows that this higher error site coverage reveals many more SDCs for Min100 than Ref99 for LU.

5.2.3. Accuracy To assess the accuracy of Min, we compare the error outcome classifications of Min100 with that of Ref99. PCs are classified into SDC-causing instructions (ones that generate outcomes of SDC-Bad or SDC-Maybe) and PCs that do not need protection (due to masking or low-cost detectors). We measure accuracy of Min100 relative to that of Ref99 using the following five categories.

1. **Common**: Min100 and Ref99 classify these PCs in the same way.
2. **MinSDC**: Min100 classifies these PCs as needing protection. Ref99 classifies them as not needing protection.
3. **MinSDC+**: Min100 classifies these PCs as needing protection. Ref99 does not explore these PCs at all.
4. **RefSDC**: Ref99 classifies these PCs as needing protection. Min100 classifies them as not needing protection.
5. **RefSDC+**: Ref99 classifies the PCs as needing protection. Min100 does not explore these PCs at all.

Figure 6 shows, for each application, all PCs explored by Ref99 or Min100 divided into the above five categories. The majority of all PCs explored by both Ref99 and Min100 (67% of all PCs on average, and 80% of intersecting PCs) result in a common classification, with two applications having above 97% of their PCs common among intersecting PCs. Surprisingly, a large number of PCs fall in the MinSDC and MinSDC+ categories (31% on average). That is, Min100 is able to find many SDCs that Ref99 misses. Most of these MinSDC+ (25% on average and up to 44%) – Ref99 misses these PCs altogether as a consequence of trading off error-site coverage for analysis time. Min100 can explore all PCs because it runs significantly faster due to fewer and/or shorter injection runs.

The last two categories, RefSDC and RefSDC+, reflect a loss of accuracy from Min. Fortunately, we do not see any RefSDC+ because of our judicious minimization based on PC coverage that ensures that Min100 explores no fewer PCs than Ref. The RefSDC category is small but not insignificant (2% on average). We analyzed several cases in this category and found two predominant trends.

1) 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 (sometimes significantly so) 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.

2) A more fundamental cause of mismatches is that the equivalence class heuristic used by Approxilyzer results in different error sites grouped together in order to prune the large error site space. Specifically, a cap is set as to how many equivalence classes can be associated with a PC – in the scenario when too many equivalence classes are generated, Approxilyzer combines similar equivalence classes together based on certain heuristics. This indicates that a stronger criterion which can mimics Ref's equivalence classes in a Min input might give even better accuracy than simple PC coverage. We show however that, for its simplicity, PC coverage does exceptionally well.

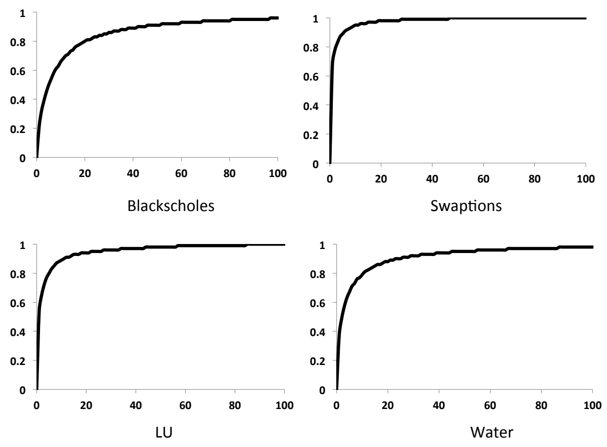

Figure 7: Cumulative probability (on the y-axis) of picking an SDC-causing injection within the first n trials (x-axis).

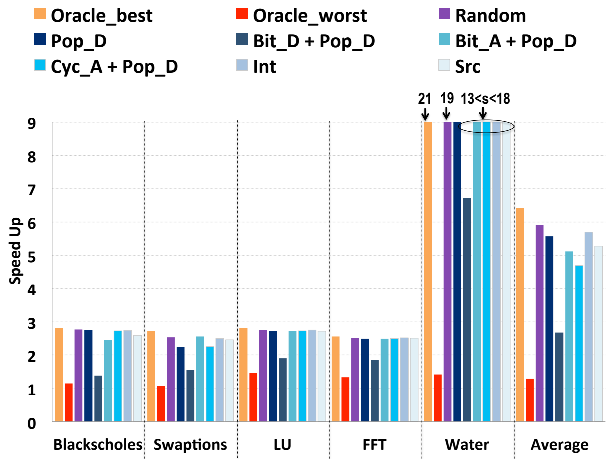

Figure 8: Min100 speedup with Single-Input Prioritization.

5.3. Single-Input Error Injection Prioritization

The sooner a PC can be classified as causing an SDC, the fewer error injections are necessary to probe the PC. We study 38 different prioritization combinations in order to best enable early termination. We include for brevity the 7 most interesting orderings, in addition to an oracle best and worst case ordering.

Figures 8 and 9 show the results of different prioritizations as presented in Section 3.3. The figures show the injection prioritizations used in lexicographic ordering. For example, "Bit_D + Pop_D" means that error injections are first ordered by descending bit position (started from the MSB), then ordered by their population size (number of error sites in the equivalence class, in descending order).

One interesting observation is that random generally performs as well as or better than most injection prioritization schemes, and is very close to the Oracle best case. Upon further inspection, this is related to the fact that if a PC is SDC-causing, a large fraction of injections in that PC result in an SDC outcome. Based on the observed ratios of SDC-causing injections in a given PC, Figure 7 shows the expected cumulative probability (on the y-axis) of finding an SDC-causing instruction after n (x-axis) random trials. As can be seen from the curves, the probability of finding an SDC injection shoots up within the first few trials. Thus, randomly choosing an injection tends to quickly find the SDC for that instruction, and we default to this case for later analysis in the paper. Overall, we see that prioritizing injections and using early termination can result in a 4.7X and 5.9X speedup for Min and Ref respectively, as compared to not using early termination.

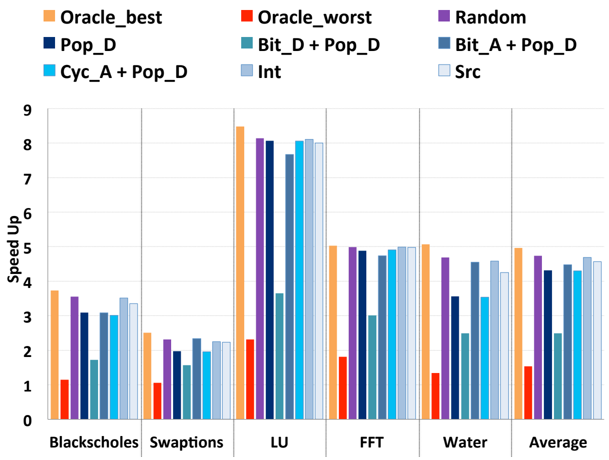

Figure 9: Ref99 speedup with Single-Input Prioritization.

In terms of total analysis runtime, we measure early termination of Ref99 compared to Min100. The orange bars in Figure 5 shows the speedups obtained for Min100 and Ref99 when using random injection prioritization for single input. By both choosing a minimal input *and* employing the early termination optimization, we observe a 7.2X analysis speedup over baseline Ref99 on average, and a 1.3X average speedup compared to Ref99 applying early termination.

5.4. Multiple-Inputs Prioritization

Prioritization across multiple inputs also shows significant performance improvements. Figure 10 illustrates the relative runtime of using multiple inputs sequentially versus prioritizing inputs to find SDCs and use that information during the analysis of another, potentially longer running, input. While Min100 has a high accuracy and finds many SDCs, running Min and Ref in this fashion can find all SDCs from both Min and Ref, while being, on average, only slower than Min100 itself and faster than Ref99 itself. We find that running multiple input prioritization results in 2.47X faster analysis time, than simply running Min100 and Ref99 in their entirety.

6. Related Work

This work is the first to apply a more systematic testing methodology to hardware resiliency analysis. The most directly related work from software testing literature is discussed in Section 2. We discuss other related work in three broad categories below.

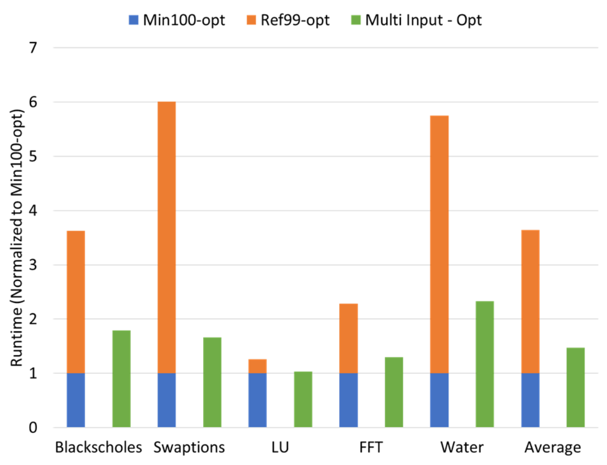

Figure 10: Multiple-Input Prioritization to find all SDC-causing PCs from both Ref and Min

High-level analysis techniques for hardware resiliency: High-level methodologies for analyzing hardware resiliency can generally be split into two categories: 1) analytically estimating critical hardware needed for architecturally correct execution (ACE) [31] and 2) injecting errors/faults and measuring their impact. ACE analysis is typically used to measure the Architectural Vulnerability Factors (AVF) [27, 32, 31] of hardware structures. PVF [51] uses ACE analysis to isolate purely architecture-level (program vs. microarchitecture) vulnerabilities in the AVF; ePVF [13] further isolates bits that may lead to crashes and achieves tighter SDC-bounds. The primary goal of AVF/PVF/ePVF is statistical characterization of hardware/software vulnerabilities, and they all consider the dynamic trace of a fault-free program execution on a given input. While powerful, these techniques are limited by their inability to precisely model an error's impact on the execution since they use information from only an error-free execution.

Approxilyzer analysis used in this paper differs from these techniques: the primary goal is not a statistical average – it is to determine precisely if/how an error in any given instruction will impact the final output, and whether that specific instruction needs protection against an SDC. For this purpose, it uses a combination of program analysis (including ACE-like data-flow analysis) and error injection. The second category of techniques use injection frameworks to introduce errors in execution at different program abstractions [48, 57, 49, 7, 24]. All these techniques rely of simulating high-level hardware errors in execution and observing their impacts on the final output to estimate the application's resiliency.

The techniques described in this paper is orthogonal to all the techniques described above. For e.g., minimization could be employed to broaden the range of inputs used for tighter ACE/AVF evaluation. Test-case minimization and injection prioritization can be applied to improve the analysis speed and accuracy (enabling analysis with wider set of inputs for given overheads) of the error-injection techniques described above. Enabling more accuracy in high-level resiliency analysis can potentially further improve AVF estimates via cross-layer resilience [58, **?**].

Approximate Computing

Many techniques have been proposed that leverage approximate computing at the software [50, 4, 42, 46, 28, 52], programming language [8, 30, 35, 43, 44, 6, 36] and hardware [47, 18, 12, 45] level for improved performance, energy or reliability. This work can be extended to study error tolerance of instructions in an approximate computing environment. These analyses can directly benefit criticality-testing [40, 9, 53, 34] for approximate computing.

Perhaps the most relevant related work is by Laurenzano et al. [22] who present approximate program optimization with canary inputs. These are subsampled versions of the full input units that perform on-line determination of appropriate approximations. Our approach uses test-case minimization and prioritization to speed up the analysis. Second, we present an off-line systematic methodology to select minimized inputs using software-testing inspired coverage criterions.

Low-level hardware resiliency techniques and error models: Hardware resiliency techniques that use lower-level error models (microarchitectural, gate-level) [10, 25] target accuracy in analysis at the cost of significantly increased overheads. The error model in this work assumes single-bit architectural errors. Extending this work to multi-bit errors is a future direction we would like to explore. Single bit architectural errors, albeit limited [10], are a realistic subset of errorswidely in the literature. Analyzing all possible single bit errors sites in a program (amounting to billions) is a steep and unique challenge that is met by the Approxilyzer tool.To our knowledge no other tool/technique can identify all SDCs stemming from a given error model. We believe that the techniques described in this work can enable future researchers to take the next step of systematically exploring other error models (multi-bit, RTL etc.) to further understand SDC behavior. For example, accurate dictionaries in the future could use faster architectural error models to comprehensively reason about lower-level errors. Even in the absence of such dictionaries, simpler error models can be used to perform targeted analysis using more expensive error models (for e.g., if portions of code produce SDC using simpler models, then they do not need to be analysed further). Thus, continuing to significantly improve the speed, accuracy and comprehensiveness of simpler error models is a worthy goal to pursue.

7. Conclusion

As hardware technology scales, efficient resiliency solutions are needed to address the increased likelihood of hardware errors. SDCs are especially harmful, and several recently proposed techniques use error injections to identify SDC-causing instructions. However, these techniques are rather expensive as they need to analyze a huge number of error sites.

We show some initial, highly promising applications of software testing methodology to advance the state-of-the-art resiliency analysis. In particular, we (1) introduce a new test coverage criterion, PC coverage, and (2) use it to perform test-case minimization of standard benchmark inputs, creating smaller inputs that are easier to analyze. To identify SDCs even faster, we also (3) employ test-case prioritization for single inputs and (4) employ test-case prioritization across multiple inputs. The experiments show that our approach can improve both accuracy and performance of SDC discovery. These highly promising results open an avenue for future work on applying many other software testing techniques [3] to further improve resiliency analysis.

References

[1] Road vehicles — Functional safety. Website. https://www.iso.org/standard/43464.html.

[2] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce. Evaluating non-adequate test-case reduction. In *Proc. of the 31st IEEE/ACM Conference on Automated Software Engineering (ASE)*, pages 16–26, Singapore, Singapore, September 2016.

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

[4] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[5] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 470–487, New York, NY, USA, 2015. ACM.

[7] Jon Calhoun, Luke Olson, and Marc Snir. Flipit: An llvm based fault injector for hpc. In *European Conference on Parallel Processing*, pages 547–558. Springer, 2014.

[8] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 33–52, New York, NY, USA, 2013. ACM.

[9] Michael Carbin and Martin C. Rinard. Automatically identifying critical input regions and code in applications. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 37–48, New York, NY, USA, 2010. ACM.

[10] Hyungmin Cho, S. Mirkhani, Chen-Yong Cher, J.A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Proc. of International Design Automation Conference*, pages 1–10, 2013.

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

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

[13] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 168–179, June 2016.

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

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

[16] A. Groce, M.A. Alipour, Chaoqiang Zhang, Yang Chen, and J. Regehr. Cause reduction for quick testing. In *ICST*, pages 243–252, 2014.

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

[18] Jie Han and Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *ETS*, pages 1–6. IEEE Computer Society, 2013.

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

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

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

[22] Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input responsiveness: Using canary inputs to dynamically steer approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 161–176, New York, NY, USA, 2016. ACM.

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

[24] J. Li and Q. Tan. Smartinjector: Exploiting intelligent fault injection for sdc rate analysis. In *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 236–242, Oct 2013.

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

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

[27] Xiaodong Li, Sarita Adve, Pradip Bose, and Jude Rivers. Online Estimation of Architectural Vulnerability Factor for Soft Errors. In *Submitted for publication*, 2007.

[28] Divya Mahajan, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, and Hadi Esmaeilzadeh. Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, ISCA, 2016.

[29] 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. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4), 2005.

[30] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. *SIGPLAN Not.*, 49(10):309–328, October 2014.

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

[32] Shubhendu S. Mukherjee, Christopher T. Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. Measuring architectural vulnerability factors. *IEEE Micro*, 23(6):70–75, November 2003.

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

[34] B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee. Autosense: A framework for automated sensitivity analysis of program data. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[35] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. Flexjava: Language support for safe and modular approximate programming. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 745–757, 2015.

[36] Jongse Park, Xin Zhang, Kangqi Ni, Hadi Esmaeilzadeh, and Mayur Naik. Expax: A framework for automating approximate programming. In *Technical Report, Georgia Institute of Technology*, 2014.

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

[38] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[39] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *PLDI*, pages 335–346, 2012.

[40] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. Asac: Automatic sensitivity analysis for approximate computing. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 95–104, New York, NY, USA, 2014. ACM.

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

[42] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.

[43] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. In *Technical Report UW-CSE-15-01-01, University of Washington*, 2015.

[44] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[45] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelganger: A cache for approximate computing. In *International Symposium on Microarchitecture*, 2015.

[46] J. Sartori and R. Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications. *Multimedia, IEEE Transactions on*, 15(2):279–290, Feb 2013.

[47] John Sartori and Rakesh Kumar. Architecting processors to allow voltage/reliability tradeoffs. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '11, pages 115–124, New York, NY, USA, 2011. ACM.

[48] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V. Adve, and Helia Naeimi. Ganges: Gang error simulation for hardware resiliency evaluation. *SIGARCH Comput. Archit. News*, 42(3):61–72, June 2014.

[49] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk. Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *Dependable Computing Conference (EDCC), 2015 Eleventh European*, pages 245–255, Sept 2015.

[50] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[51] Vilas Sridharan and David R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *Proc. of International Symposium on High Performance Computer Architecture*, 2009.

[52] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. Proactive control of approximate programs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 607–621, New York, NY, USA, 2016. ACM.

[53] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. In *International Conference on Dependable Systems and Networks*, pages 1–12, 2013.

[54] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, Oct 2016.

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

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

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

[58] Bagus Wibowo, Abhinav Agrawal, Thomas Stanton, and James Tuck. An accurate cross-layer approach for online architectural vulnerability estimation. *ACM Trans. Archit. Code Optim.*, 13(3):30:1–30:27, September 2016.

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

[60] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2):67–120, 2012.

[61] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *TSE*, 28(2):183–200, 2002.

[62] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *ISSTA*, pages 160–170, 2014.