gem5-Approxilyzer: An Open-Source Tool for Application-Level Soft Error Analysis

Radha Venkatagiri*, Khalique Ahmed*, Abdulrahman Mahmoud,
Sasa Misailovic, Darko Marinov, Christopher W. Fletcher, Sarita V. Adve
{venktgr2, kahmed10, amahmou2, misailo, marinov, cwfletch, sadve}@illinois.edu
University of Illinois at Urbana-Champaign, USA

***Abstract*—Modern systems are increasingly susceptible to soft errors in the field and traditional redundancy-based mitigation techniques are too expensive to protect against all errors. Recent techniques, such as approximate computing and various low-cost resilience mechanisms, intelligently trade off inaccuracy in program output for better energy, performance, and resiliency overhead. A fundamental requirement for realizing the full potential of these techniques is a thorough understanding of how applications react to errors.**

Approxilyzer is a state-of-the-art tool that enables an accurate, efficient, and comprehensive analysis of how errors in almost all dynamic instructions in a program's execution affect the quality of the final program output. While useful, its adoption is limited by its implementation using the proprietary Simics infrastructure and the SPARC ISA.

We present gem5-Approxilyzer, a re-implementation of Approxilyzer using the open-source gem5 simulator. gem5-Approxilyzer can be extended to different ISAs, starting with x86 in this work. We show that gem5-Approxilyzer is both efficient (up to two orders of magnitude reduction in error injections over a naïve campaign) and accurate (average 92% for our experiments) in predicting the program's output quality in the presence of errors. We also compare the error profiles of five workloads under x86 and SPARC to further motivate the need for a tool like gem5-Approxilyzer.

I. Introduction

The end of conventional technology scaling has led to two recent trends that consider incorrect outputs. First, the emergent field of approximate computing [1]–[4] considers a deliberate, but controlled, relaxation of correctness for better performance or energy. Second, the increasing threat to hardware reliability [5] and the high costs of traditional resiliency solutions have led to significant research in alternative low-cost, but less-than-perfect, solutions that let some hardware errors escape as (user-tolerable) output corruptions [6]–[13].

The common underlying theme of both methods is to improve system efficiency by accommodating controlled errors (deliberate approximations or unintentional hardware errors). Such computing paradigms have the potential to significantly change how we design hardware and software (as current systems are designed for exact computations). Their widespread adoption, however, requires an understanding of how errors in computation affect the outcome of the execution. In this work, we focus on software-driven solutions and we use *error analysis* to mean the process of characterizing the effects of a given set of errors on the execution and final output of a given piece of software. Error injection – where an error is deliberately injected in the execution of a given workload to observe its effect – is a widely used error analysis technique.

A naïve error analysis would perform an error-injection campaign that injects errors in the executions of all applications of interest, in every possible execution cycle, using all error models of interest. Each erroneous execution would be monitored for anomalous behaviour and any output produced would be examined for possible corruption. Such a naïve campaign would provide a highly accurate error analysis but would be impractical. More practical solutions (both with and without error injections) have been proposed in the literature [2], [4], [14]–[24] with varying degrees of accuracy (guarantees on output quality), generality (applicable to any application), comprehensiveness (estimating *all* possible errors in execution), and automation (placing no undue burden on programmer, such as code annotations to identify error-tolerant regions).

Our recent tool called Approxilyzer [25] (which builds on our previous tool called Relyzer [26]) has furthered the state-of-the-art in error analysis by providing the impact (execution anomalies and output quality) of single-bit transient errors on *every* operand register bit in *virtually every* dynamic instruction in a program execution. It uses a hybrid technique of program analysis and error injections to perform this comprehensive analysis with high accuracy while performing relatively few error injections. Furthermore, Approxilyzer can analyze general-purpose applications while placing minimal burden on the programmer.

Approxilyzer's unique features enable new avenues of research, but limitations in its current implementation hinder its usability. The tool currently relies on Wind River Simics [27], a proprietary full-system simulator. The current implementation is also designed to handle only applications compiled for the SPARC ISA. The restrictions imposed from both the simulator and ISA make a wide adoption of the tool challenging.

The main contribution of this paper is the development of *gem5-Approxilyzer*, an open-source[1] implementation of Approxilyzer that enables support for more ISAs, beginning with x86 in this work. We build our new tool using the open-

*The first two authors led this work with equal contributions.

This material is based upon work supported by NSF Grant No. CCF-1320941, CCF-1421503, CCF-1703637, and CCF-1725734; and by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

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

source gem5 simulator [28] which facilitates (with relative ease) the future inclusion of more ISAs into the tool. Building gem5-Approxilyzer required significant engineering effort to support x86 error analysis on gem5. Approxilyzer's original implementation for SPARC assumes constant register size and instruction encoding length, which is not the case for x86.

This work is the first to show Approxilyzer's effectiveness/accuracy with a different ISA, namely x86. We show that our x86 implementation is effective in reducing the number of error injections required by up to two orders of magnitude over a naïve campaign and predicts the impact of errors on the program's output quality with high accuracy ($>$92% on average and up to 99.9% for some applications).

We also compare the comprehensive error profiles of our workloads under the two ISAs — SPARC and x86. We show that the error profiles of the same application can be rather different under different architectures, which in turn can require customized resiliency and approximation solutions. This result further motivates the need for open-source tools such as gem5-Approxilyzer that can enable such comparisons and aid in building better solutions and exploring new avenues of research.

II. (gem5-)Approxilyzer Overview

gem5-Approxilyzer is an implementation of Approxilyzer [25] using the open-source gem5 simulator. Hence, its interface, high-level design and techniques are the same as those described in prior work [25], [26]. This section provides a brief overview of the objectives, user interface, and techniques for Approxilyzer and hence, for gem5-Approxilyzer.

A. High-Level Objective

The goal of gem5-Approxilyzer is to characterize the impact of *any* single-bit transient error in a program's execution with high accuracy. The different ways in which an error can impact execution are described in Section II-B. We use the term *error site(s)* to refer to specific points in the application's execution where an error could be encountered. The error model we use is single-bit transient errors in architectural registers. Hence, we use error site to refer to a specific bit in a specific operand register in a specific dynamic instruction.

gem5-Approxilyzer uses program analysis to systematically analyze all error sites in the program and carefully picks a small subset to perform selective error injections. gem5-Approxilyzer employs error-pruning techniques (Section II-D) to prune errors that need no detailed study by either predicting their outcomes or showing them equivalent to other errors. Thus, it can perform fewer error injections than a naïve error injection campaign while maintaining high accuracy in predicting the impact of almost all errors in the program.

B. Inputs and Outputs of gem5-Approxilyzer

Figure 1 shows the inputs and outputs of gem5-Approxilyzer. gem5-Approxilyzer takes as inputs (1) an application, (2) inputs to the application, and (3) quality metrics to quantify the deviation of the erroneous output from the error-free program output. As an optional input, users can specify a quality threshold that quantifies the maximum acceptable degradation in output quality.

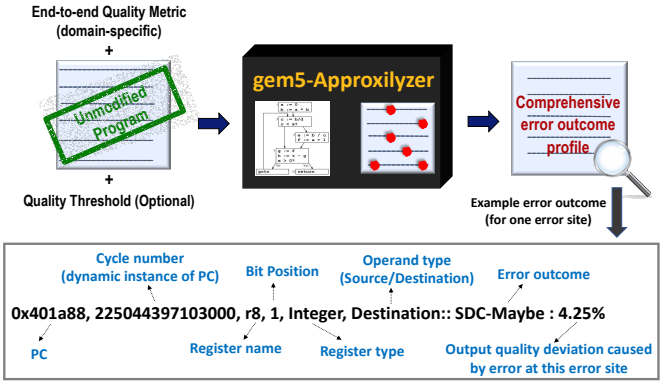

Fig. 1. Overview of gem5-Approxilyzer inputs and outputs.

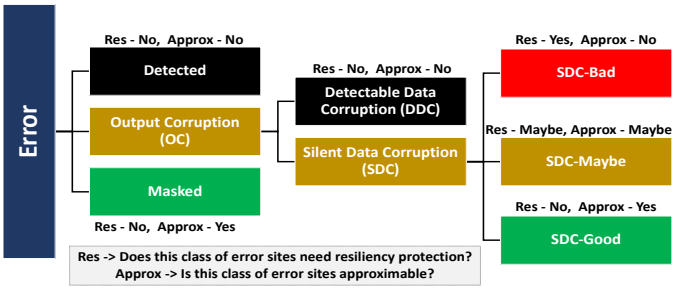

Fig. 2. A classification of error outcomes [25] and their implications for approximation and resiliency.

gem5-Approxilyzer outputs a comprehensive error outcome profile (simply referred to as an *error profile*) for the program. It lists all the error sites in the program as well as the outcome of an error (injected) at this error site, referred to as *error outcome*. gem5-Approxilyzer distinguishes error outcomes as masked, detected, or output corruptions (**OCs**). Instead of considering all OCs as silent data corruptions (SDCs), gem5-Approxilyzer analyzes the quality (degradation) of the corrupted outputs to separate outcomes that are tolerable to the user from those that are not. Figure 2 shows the various error outcome categories: **(1) Detected**: An error that raises observable symptoms (e.g., fatal traps, segmentation faults, timeout etc.) that can be caught using various low-cost detectors [6], [29]–[34] before the end of execution. **(2) DDC**: An OC that is detectable via low-cost mechanisms such as range detectors [35]. **(3) SDC-Bad**: An OC with very large (unacceptable) output quality degradations. **(4) SDC-Maybe**: An OC that may be tolerable if the output quality degradation is within a user-provided quality threshold (if no threshold is provided, all SDC-Maybe's default to SDC-Bad). **(5) SDC-Good**: An OC that produces negligibly small (acceptable) output quality degradation. **(6) Masked**: Errors that produce no OC.

Approxilyzer uses two different types of quality thresholds: (1) obvious domain-specific thresholds for SDC-Good/SDC-Bad/DDC categorization [25] and (2) user-specified quality threshold to decide if an SDC-Maybe is acceptable. For example, for financial applications, dollar outputs of negative value are obvious DDC, and quality differences of less than

one-hundredth of a cent could be assumed to be SDC-Good. Beyond this, a user can optionally specify that for their use-case, outputs within, say, 1 cent of the error-free value are tolerable which will define if an SDC-Maybe is acceptable.

Thus, the error outcome associated with each error site is composed of the error-outcome category as well as the output quality degradation (*QD*) for OC class of errors. For the example error site output shown in Figure 1, the error produces an output corruption that is measured (using the quality metric provided, say, for example, average relative error) to have a quality degradation of 4.25% from the error-free output. Because this value is neither small enough to be labeled SDC-Good nor large enough for SDC-Bad, the error outcome is categorized as SDC-Maybe (in the absence of a user-specified quality threshold).

C. Use Cases of gem5-Approxilyzer

The application error profile that gem5-Approxilyzer produces can be used by different techniques that trade accuracy in the program output for gains in other system parameters. We discuss two such techniques: low cost resiliency and approximate computing. Based on the knowledge of the output quality produced by perturbing each error site in the program, gem5-Approxilyzer can help decide which error site needs protection from transient errors, or alternatively, which error sites could be approximable.

Figure 2 shows how various error categories can be can be grouped or *equalized* based on whether the context is resiliency analysis (*Res*) or approximate computing (*Approx*). Masked (no quality degradation), SDC-Good (acceptable quality degradation), DDC (caught by low-cost detectors of output), and Detected (caught by low-cost detectors of execution) error sites can be equalized together as error sites that do not need expensive resiliency protection. SDC-Bad error sites, on the other hand, need to be protected (hardened). Similarly, in the context of approximate computing, Masked and SDC-Good can be equalized together and considered approximable, while DDC, Detected, and SDC-Bad error sites produce egregious outputs and so are not approximable [25].

Decisions for SDC-Maybe outcomes can be made based on the quality thresholds provided by the user. For example, if the quality degradation of the SDC-Maybe is below the acceptable quality threshold, then the error site need not be protected and is considered approximable. In the absence of quality thresholds, SDC-Maybe can be conservatively equalized with SDC-Bad. The reasoning about individual error sites can be extended to instructions (both dynamic and static) based on the error outcome of their constituent error sites.

D. Pruning Techniques

Compared to a naïve campaign that performs an error injection for every error site, gem5-Approxilyzer dramatically reduces the number of error injections to predict the error outcome for all error sites. gem5-Approxilyzer implements two sets of pruning techniques from prior work [25], [26]: *equivalence-based* and *known-outcome*. This section briefly describes these techniques; detailed explanations and examples can be found in prior work [26].

Equivalence-based pruning techniques use program analysis (static and dynamic) and heuristics to prune errors that are likely equivalent to others. These techniques partition error sites into *equivalence classes*, where each class requires an error injection into just a single representative error site (called *pilot*) to predict the error outcome for all other error sites in the class. gem5-Approxilyzer implements two equivalence-based pruning techniques. **Control equivalence** groups error sites based on the observation that errors that propagate through similar code sequences are likely to have similar error outcomes. This technique records the next *N* branches for dynamic instances of a given static instruction in the original execution (with no error injection). Corresponding error sites of dynamic instances that share the same control path (up to depth *N*) are grouped in an equivalence class. **Store equivalence** is used to equalize dynamic instances of store instructions (and instructions that a store depends on within a basic block) based on the observation that errors in a store instruction propagate through the loads that read the erroneous value. This technique records the subsequent loads that read from a store address and groups (corresponding error sites of) dynamic instances of store instructions that have the same list of subsequent loads together.

Known-outcome pruning techniques largely use static (and some dynamic) program analyses to determine the outcome of an error. gem5-Approxilyzer uses two known-outcome pruning techniques. **Address-bound pruning** uses the observation that single-bit errors that appear outside the address range of an application result in Detected outcomes. Thus, their outcomes are known a priori and these errors can be pruned. **Def-use pruning** uses the observation that an injection in a def is equivalent to an injection in the first use at the same register and bit position, so only one needs to be explored.

By combining all these pruning techniques, gem5-Approxilyzer can dramatically reduce the total number of error sites that need error injections. In Section V-B, we validate the effectiveness of these heuristics for both the resiliency and approximate computing use cases.

III. GEM5-APPROXILYZER: IMPLEMENTATION

This section describes the implementation details and associated challenges of gem5-Approxilyzer. We also briefly discuss future extensions to the tool.

A. Error Model used in gem5-Approxilyzer

The error model we use is single-bit transient errors in architectural registers. We study errors in bits of both source and destination registers of instructions.

In this work, we undertake error injection in registers of x86 macro-instructions. Modern CISC implementations like x86 often implement the complex machine instructions (macro-instructions) using low-level instructions called micro-instructions or micro-operations. Micro-instructions are generally specific and proprietary to the micro-architecture and not faithfully recreated in publicly available simulators. Hence, we restrict our analysis to macro-instructions.

For this study, we only consider general-purpose registers and SSE[2] registers in x86. We do not inject errors in special-purpose, status, and control registers (e.g., %rsp, %rbp, rflags) to simplify our error model and reduce the number of error injections required for a first-order analysis. We assume that these always need protection and can be hardened in hardware (e.g., with ECC). We also do not inject in implicit[3] registers in this work. Extending gem5-Approxilyzer to support these registers is relatively straightforward and we leave it to future work.

B. Implementation Details

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

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

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

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

(3) Phase 3 performs the error-injection experiments using our *error injector* module built for gem5. The error injector takes as input the error-site description: dynamic instruction described using the cycle number of the simulation, register information (register name and whether it is used as a source/destination operand) and register bit number. The error injector pauses the simulation at the specified dynamic instruction and flips the bit in the register. For source registers, the bit flip is performed before the instruction execution. For destination registers, the error is simulated by performing the bit flip after the instruction execution (otherwise the error would be overwritten by the instruction execution). The simulation then proceeds, checking for any hangs and crashes, or other symptoms to identify Detected outcomes. If no Detected symptoms are encountered before the simulation ends, gem5-Approxilyzer compares the generated output with the error-free execution's output. If there is an OC, gem5-Approxilyzer uses the user-provided quality metric to evaluate the output quality.

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

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

C. x86 Implementation Challenges

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

D. Extensions to gem5-Approxilyzer

We designed gem5-Approxilyzer to be reasonably modular (e.g., each phase in Section III-B is a separate module) to enable future extensions to support different ISAs, error models, and pruning techniques. This section briefly elaborates on some details for future extensions.

The gem5 simulator currently supports many ISAs, and gem5-Approxilyzer could support them with the following modifications. 1) The instruction parser in Phase 1 must be modified to capture the semantics of the new ISA. 2) ISA-specific behaviors that affect control flow (e.g., branch delay slots for SPARC) should be incorporated into the

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

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

control-equivalence algorithm accordingly. 3) Register aliasing must be captured correctly to track def-use pairs.

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

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

IV. Experimental Methodology

A. Benchmarks and Quality Metrics

To evaluate gem5-Approxilyzer, we select five benchmarks from three different benchmark suites spanning multiple application domains. Table I shows the applications and inputs used in our evaluation. We chose these inputs because recent work has shown that performing error analysis on these inputs is much faster and at least as accurate as for larger reference inputs [36]. We use gem5 to simulate an Ubuntu-16.04 system, and we use GCC 7.3 with -O3 to compile the applications.

To evaluate the quality degradation of the application's final output we use the following quality metrics: (1) *Absolute Maximum Difference* [25] (in the dollar value outputs) for the financial applications Blackscholes and Swaptions, (2) *Maximum Relative Error* [25] for LU and Blackscholes, (3) *Relative L2 Norm* [25] for FFT, and (4) *Root Mean Square Error* [3] for Sobel. We use the same quality thresholds to determine SDC-Good, SDC-Bad, and DDC as prior work [25].

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

Application	Input	Error Sites		Pruned Error Sites (%)
		Total	Remaining	
Black-scholes [37]	21 options	232K	100K	C: 12.24 S: 9.45 C+S+K: 56.77
Swaptions [37]	1 option 1 simulation	10.3M	720K	C: 52.47 S: 7.85 C+S+K: 93.01
LU [38]	16x16 matrix 8x8 blocks	1.2M	268K	C: 23.49 S: 22.72 C+S+K: 77.91
FFT [38]	2^8 data points	4.4M	215K	C: 43.99 S: 21.50 C+S+K: 95.05
Sobel [3]	81x121 pixels	85.3M	300K	C: 62.74 S: 20.94 C+S+K: 99.65

B. Pruning Effectiveness

We measure the effectiveness of gem5-Approxilyzer by observing how many error sites were pruned using the various pruning techniques described in Section II-D. For each application, we measure first the number of error sites in the application's region of interest and then the number of error sites remaining after the pruning to calculate the number of error sites that have been pruned. This metric evaluates the tool's effectiveness since the number of error sites pruned directly reduces the number of error-injection experiments needed to analyze the application. For the control heuristic, we set depth to N=50, as in prior work [39].

C. gem5-Approxilyzer Validation

The control- and store-equivalence based pruning techniques use heuristics and require validation. For a given equivalence class, these pruning techniques choose a single pilot to represent the error outcomes of all error sites in the equivalence class (Section II-D). Similar to the methodology used in prior work [25], [26], we quantify the validity of these techniques by measuring the extent to which the pilots correctly represent their equivalence classes.

The validation attempts to answer the following question: how accurately does the error outcome of the pilot predict the error outcome of the other error sites in its equivalence class? For validating a single equivalence class, we perform error injections in a sample of error sites (not including the pilot) – called *population* – chosen randomly from the equivalence class. We refer to individual error sites within a population as *population member(s)*. We calculate the prediction accuracy of an equivalence class's pilot, by measuring the number of its population members that produce the same equalized error outcome as the pilot.

Because the equalization of error outcome categories (Section II-C) depends on whether the context of the analysis is resiliency (Res) or approximation (Approx), we show the validation outcomes for both these contexts separately. For example, consider an equivalence class whose pilot X generates a DDC. Suppose 86% of its population is DDC, 6% is Masked, 5% is SDC-Maybe, and 3% is Detected. Then the prediction accuracy of pilot X for *Res* is 95% and for *Approx* is 89%.

SDC-Maybe error sites are equalized based on if their output quality degradations (QD) are above or below user specified output quality thresholds (QT). In the absence of a quality threshold, prediction accuracy measurements for any SDC-Maybe pilot with a quality degradation of, say, Q measures the number of its population members that also result in SDC-Maybe with the same quality degradation Q. Requiring the output quality degradation of different SDC-Maybe error sites within an equivalence class to exactly match is too strict, so we use a flexibility parameter, δ, that allows a fine-grained difference in the measured quality degradation [25]. Hence, if the absolute difference in the quality degradation of an SDC-Maybe pilot and an SDC-Maybe population member is less than or equal to δ, we count it as a correct prediction. In the presence of a user provided output quality threshold (QT), however, SDC-Maybe error outcomes of the pilot and population members can be appropriately equalized for resiliency and approximation.

To illustrate all of the above with an example, consider a pilot Y that generates an SDC-Maybe with a QD-6 (quality degradation of 6% using, for example, average relative error as the quality metric). Suppose the error outcomes of its population are as follows: (a) 84% of its population is SDC-Maybe with QD-6, (b) 6% is SDC-Maybe with

QD-3, (c) 5% is SDC-Maybe with QD-8, (d) 3% is Masked, and (e) 2% is DDC. Then the validation accuracy of Y for the different validation strategies is as follows: (1) $Res_{\delta=2}$ is 89%=84%+5%, (2) $Approx_{\delta=2}$ is 89%=84%+5%, (3) $Res_{\delta=2,QT=7}$ is 100% (all are correct, even (c), which lies above the QT, because its QD falls within the $\delta=2$ margin of the pilot), and (4) $Approx_{\delta=2,QT=7}$ is 98% ((e) is incorrect).

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

We randomly pick 750 equivalence classes to validate each of the control- and store-equivalence techniques (1500 equivalence classes for combined control+store). This gives us a 99% confidence interval with a 5% error margin [40]. For each equivalence class we set the population size to 750. If the equivalence class size is less than 750 (error sites), then the pilot is validated using all the remaining error sites in the equivalence class. This again gives us a 99% confidence interval with a 5% error margin [40]. In all, we perform approximately 1.6 million error-injection experiments to validate gem5-Approxilyzer.

D. Error Analysis of Applications

As mentioned in Section II-C, an application's error outcome profile can be used for different purposes. For example, gem5-Approxilyzer can be used to extract the set of static instructions that need resiliency protection [36] or the set of static instructions that are approximable under different output quality requirements. Information about a static instruction (called a PC) is derived by observing the error outcomes of all the error-sites for all the dynamic instances of that static instruction. For example, if all the error sites for a PC result in Detected, Masked, or SDC-Good outcomes, we say that the PC needs no resiliency protection against single-bit transient errors. Similarly, if the PC has Detected error sites, we can conclude it is not a candidate for approximation [25]. In our evaluation, we assume a very conservative classification of the PCs in our workload. If even a single error site for the PC needs resiliency protection, or is not approximable, then the entire PC is labeled as needing resiliency protection, or not approximable, respectively. Different methodologies for composing error sites within a PC are possible, e.g., error-site information can be used to derive the probability of an output with unacceptable quality degradation being generated by individual PCs.

For our workloads, we show the distribution of error sites in the application by error outcomes. Furthermore, we compose the error sites for a PC as described above to show the percentage of PCs in the application that need resiliency protection against SDCs and the percentage of PCs that are candidates for approximate computing under given output quality thresholds.

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

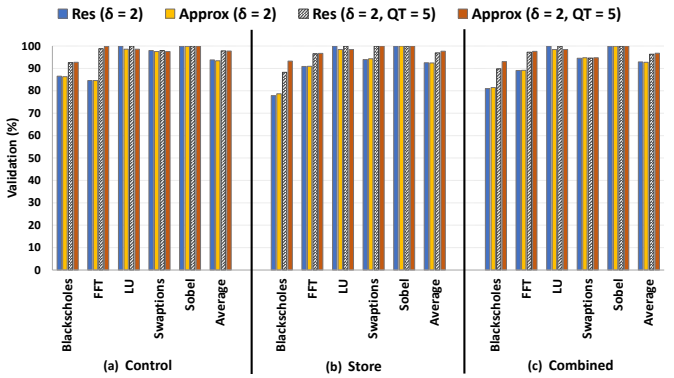

Fig. 3. gem5-Approxilyzer validation for (a) control equivalence, (b) store equivalence, and (c) combined (control + store) equivalence.

V. Results

A. Pruning effectiveness

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

B. gem5-Approxilyzer Validation

Figures 3(a), 3(b), and 3(c) show the validation accuracy for the control equivalence, store equivalence and their combination respectively. On average, both control and store equivalence techniques show high accuracy ($>92\%$) for Res and Approx with a flexible quality parameter $\delta=2$ that uses 2% for Blackscholes, FFT, LU, and Sobel, while for the financial applications, Blackscholes and Swaptions, uses the absolute difference in the dollar value set to \$0.01 (difference of 1 cent or less). In brief, gem5-Approxilyzer is able to correctly predict the output quality of the x86 application error sites with very fine granularity (2% or within a single cent). Swaptions ($>$ 94%), LU ($>$ 98.5%), and Sobel ($>$ 99.9%) show very high validation accuracy across the board.

While still relatively high, Blackscholes shows the poorest validation accuracy for both control ($Res_{\delta=2}$ = 87%, $Approx_{\delta=2}$ = 86%) and store ($Res_{\delta=2}$ = 78%, $Approx_{\delta=2}$

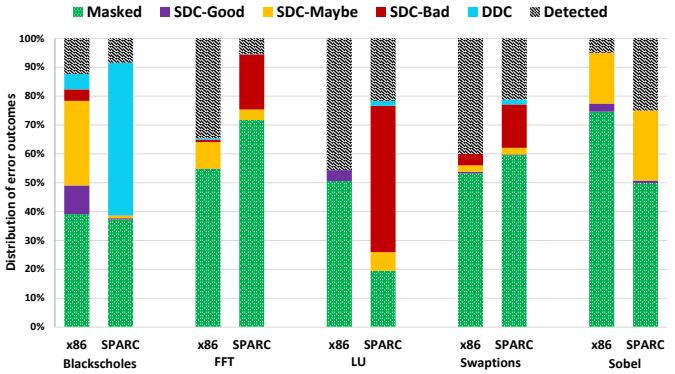

Fig. 4. Distribution of error outcome categories for the applications studied using the x86 and SPARC ISAs.

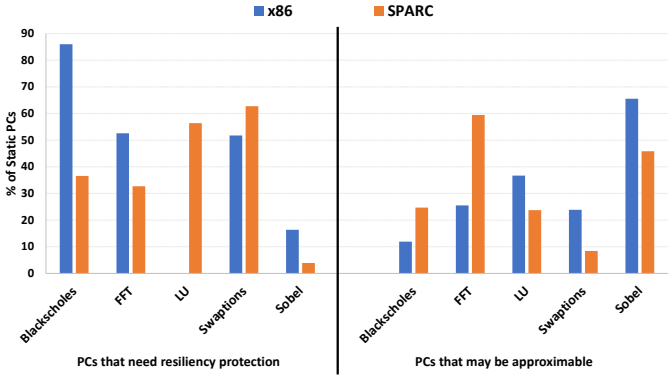

Fig. 5. Percentage of static PCs in the application that need resiliency protection and percentage of PCs that are approximable across x86 and SPARC. We use the same QT across both ISAs: 5% for Blackscholes, Sobel, FFT, and LU; \$0.001 for Blackscholes and Swaptions.

= 79%). As mentioned before, Blackscholes has small equivalence classes which can lead to poor prediction accuracy even if a single error site is predicted incorrectly.

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

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

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

C. Error Profiles for Different ISAs

Figure 4 compares the distribution of error outcomes (for all the error sites) in each application for the x86 and SPARC ISAs. The error outcome profiles of the same application look rather different for the different ISAs. We note, however, that some differences are expected due to the CISC vs. RISC nature of the instructions as well as the fact that x86 uses many more implicit registers (that we do not inject into) compared to SPARC. The graph shows that SPARC has a higher percentage of more egregious outcomes. For example, while Blackscholes-x86 has many error sites that lead to SDC-Good, SDC-Maybe, and SDC-Bad outcomes, the error outcomes in Blackscholes-SPARC produce such bad quality output that they become DDCs. We leave a deeper analysis of the causes for these differences to future work.

Figure 5 further shows the percentage of static instructions that need resiliency protection and those that are approximable for the same QT across the two ISAs. The wide differences across the two ISAs and the lack of a clear trend further underscore the importance of resiliency analysis tools that can analyze applications at the binary level to devise customized resiliency and approximation solutions for different architectures. Source-code or IR-level error analysis may not lead to the most optimized solutions.

VI. Related Work

gem5-Approxilyzer re-implements the concepts from Approxilyzer [25] and Relyzer [26] in gem5 [28] and the x86 ISA. Other tools for resiliency analysis have used gem5 as their base simulator. For instance, MeRLiN [23] uses GeFIN [41] (which is also a built on top of gem5) to simulate micro-architectural injections in an x86 O3 CPU. It performs fault pruning to accelerate statistical micro-architectural fault injections and can provide fine-grained reliability estimates for hardware structures as well as SDC vulnerability estimates for software. gem5-Approxilyzer's analysis is at the architectural level, and its primary goal is not a statistical average or probability but to determine precisely if/how an error in an instruction impacts the final output.

GemFI [42] is another error-injection tool that operates at the micro-architectural level, is built on top of gem5 and supports both Alpha and x86 ISA. Other error-injection tools, such as LLFI [19], analyze applications at the compiler intermediate representation (IR) level. IR is ISA-independent by design, so such an analysis would ideally hold regardless of the hardware architecture. However, there may be a loss in error site accuracy because IR still requires additional transformations before producing the assembly [43].

FAIL* [20] performs ISA-level analysis. A benefit of performing ISA-level injections is that the results provide

instruction-level resiliency information. System designers can then utilize these results to create soft error protection schemes at the instruction level [44], [45]. FAIL* also uses gem5 and supports ARM but is limited to one pruning technique: def-use analysis.

VII. Conclusion and Future Work

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

References

[1] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *SIGSOFT FSE*, 2011.

[2] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels," in *OOPSLA*, 2014.

[3] 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.

[4] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation," in *PLDI*, 2011.

[5] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, 2005.

[6] S. Sahoo, M.-L. Li, P. Ramchandran, S. V. Adve, V. Adve, and Y. Zhou, "Using Likely Program Invariants to Detect Hardware Errors," in *DSN*, 2008.

[7] G. Lyle, S. Cheny, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "An End-to-end Approach for the Automatic Derivation of Application-Aware Error Detectors," in *DSN*, 2009.

[8] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *MICRO*, 2007.

[9] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based Fault Screening," in *HPCA*, 2007.

[10] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLFIED: Symbolic program-level fault injection and error detection framework," in *DSN*, 2008.

[11] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic Soft Error Reliability on the Cheap," in *ASPLOS*, 2010.

[12] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra, "CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores," in *DAC*, 2016.

[13] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding Error Propagation in Deep-Learning Neural Networks (DNN) Accelerators and Applications," in *SC*, 2017.

[14] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors," in *ISCA*, 2008.

[15] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "Measuring Architectural Vulnerability Factors," *IEEE Micro*, 2003.

[16] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *MICRO*, 2003.

[17] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *HPCA*, 2009.

[18] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis," in *DSN*, 2016.

[19] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," in *DSN*, 2014.

[20] 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 *EDCC*, 2015.

[21] J. Calhoun, L. Olson, and M. Snir, "FlipIt: An LLVM based fault injector for HPC," in *Euro-Par*, 2014.

[22] J. Li and Q. Tan, "SmartInjector: Exploiting intelligent fault injection for SDC rate analysis," in *DFTS*, 2013.

[23] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "MeRLiN: Exploiting Dynamic Instruction Behavior for Fast and Accurate Microarchitecture Level Reliability Assessment," in *ISCA*, 2017.

[24] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning Approximation for Graphics Engines," in *MICRO*, 2013.

[25] 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 *MICRO*, 2016.

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

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

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, 2011.

[29] M. Dimitrov and H. Zhou, "Unified Architectural Support for Soft-Error Protection or Software Bug Detection," in *PACT*, 2007.

[30] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-Error Detection Using Control Flow Assertions," in *DFT*, 2003.

[31] S. K. S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve, "mSWAT: Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems," in *MICRO*, 2009.

[32] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design," in *ASPLOS*, 2008.

[33] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer, "Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware," in *EDCC*, 2006.

[34] N. Wang and S. Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," *IEEE TDSC*, 2006.

[35] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost Program-level Detectors for Reducing Silent Data Corruptions," in *DSN*, 2012.

[36] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve, "Minotaur: Adapting Software Testing Techniques for Hardware Errors," in *ASPLOS*, 2019.

[37] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA*, 1995.

[39] S. K. Sastry Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "GangES: Gang Error Simulation for Hardware Resiliency Evaluation," in *ISCA*, 2014.

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

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

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

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

[44] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-directed instruction duplication for soft error detection," in *DATE*, 2005.

[45] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-scalar Processors," *IEEE Transactions on Reliability*, 2002.