Approxilyzer: Towards A Systematic Framework for Instruction-Level Approximate Computing and its Application to Hardware Resiliency

Radha Venkatagiri[†] Abdulrahman Mahmoud[†] Siva Kumar Sastry Hari [‡] Sarita V. Adve[†]
[†]University of Illinois at Urbana-Champaign [‡]NVIDIA
[†]{venktgr2,amahmou2,sadve}@illinois.edu [‡]shari@nvidia.com

***Abstract*—Approximate computing environments trade off computational accuracy for improvements in performance, energy, and resiliency cost. For widespread adoption of approximate computing, a fundamental requirement is to understand how perturbations to a computation affect the outcome of the execution in terms of its output quality.**

This paper presents a framework for approximate computing, called *Approxilyzer,* that quantifies the quality impact of a single-bit error in *all dynamic instructions* of an execution with high accuracy (95% on average). We demonstrate two uses of Approxilyzer. First, we show how Approxilyzer can be used to quantitatively tune output quality vs. resiliency vs. overhead to enable ultra-low cost resiliency solutions (with a single bit error model). For example, we show that Approxilyzer determines that a very small loss in output quality (1%) can yield large resiliency overhead reduction (up to 55%) for 99% resiliency coverage. Second, we show how Approxilyzer can be used to provide a first-order estimate of the approximation potential of general-purpose programs. It does so in an automated way while requiring minimal user input and no program modifications. This enables programmers or other tools to focus on the promising subset of approximable instructions for further analysis.

I. Introduction

The end of conventional technology scaling has led to two recent trends that consider systems that generate incorrect outputs. First, the emergent field of approximate computing considers deliberate, but controlled, relaxation of correctness for better performance or energy. Second, the increasing threat to hardware reliability [4] and high costs of traditional redundancy-based resiliency solutions has led to significant research in alternative low cost, but less-than-perfect, solutions. Specifically, software-anomaly based resiliency solutions use low cost hardware or software detectors that watch for anomalous software behavior as a symptom of a hardware fault [7], [9], [13], [18], [26], [29], [42]. These solutions are promising due to their ultra-low cost, but they occasionally let some errors escape as silent data corruptions or SDCs. Computations resulting in such SDCs must be protected using other techniques (e.g., instruction duplication or software assertion checks), thereby increasing cost. This additional cost can be eliminated for SDCs that produce inexact, but acceptable outputs.

For widespread adoption of both approximate computing and software-anomaly based error detection, a fundamental requirement is to understand how perturbations (deliberate approximations or unintentional errors) to a computation affect the outcome of the execution in terms of its output quality. We refer to this as the *perturbation-outcome* problem. A "holy grail" version of this problem is to be able to determine (1) *automatically* (without undue programmer burden), (2) for any *general-purpose application*, (3) for any *perturbation model*, (4) a *guaranteed* impact on the end-to-end output quality and the resulting improvement in performance, energy, and/or resiliency cost. With such knowledge, the system or end-user would have the ability to precisely make the desirable tradeoffs in quality, resiliency, performance, and resource usage.

Although solutions to the perturbation-outcome problem that satisfy all four of the above requirements – automation, general-purpose application, any perturbation model, and guarantees on impact – remain elusive, researchers have made significant progress by relaxing some of these requirements. For example, the signal and information processing communities have a long history of using approximations tailored for their application domains, but it is unclear how to extend this work to general-purpose computations [30]. EnerJ [33] abandons automation and guarantees – the programmer identifies approximable data and the system uses approximate hardware for such data without quality guarantees. Chisel [22] uses sophisticated compiler analyses and integer linear programming to determine when an instruction or data is approximable, but requires the programmer to specify the probability with which a function must execute correctly to generate acceptable output. The framework provides mechanisms to automatically generate such probabilities, but relies on limited error injection experiments which inherently cannot provide guarantees (because it is prohibitively expensive to generate an error injection experiment for every invocation of the function). SAGE [31] monitors output quality deviations at run-time, invoking more precise versions of the computation when quality is deemed too low. Although fully automated, SAGE is a reactive mechanism, where again it is difficult to provide

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.

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

quality guarantees for general-purpose applications. Section V provides a comprehensive review of the literature to show that past techniques make similarly limiting assumptions.

This paper proposes *Approxilyzer,* a framework that addresses the perturbation-outcome problem with assumptions complementary to past work. Approxilyzer addresses the above four requirements as follows. (1) It meets the automation requirement by relying on the programmer only for the metric to determine end-to-end output quality (and optional thresholds for acceptable quality). The quality metric is domain- and not program-specific and a minimal input for any solution to the perturbation-outcome problem. (2) Approxilyzer is applicable to general-purpose computations. (3) It provides guarantees on the impact of the studied perturbations on the end-to-end output quality with high accuracy, specifically targeting improvements in resiliency cost. (4) Approxilyzer's perturbation model is a single bit flip in an operand (both source and destination) register of a dynamic instruction. While this is a limited (albeit realistic) source of perturbations, Approxilyzer is far more ambitious than previous work in providing the quality impact of such a perturbation on every operand register bit in virtually every dynamic instruction in a program execution.

Approxilyzer builds on a recent resiliency-driven tool called Relyzer [12] which uses a combination of error injection and program analysis to predict the outcomes of errors, assuming all deviations from the error-free output are unacceptable. Relyzer also uses an instruction-level transient single bit error model, and determines outcomes for all such errors in the operand registers of all dynamic instructions. We use *error site* to refer to a specific bit in a specific operand register in a specific dynamic instruction. Using program analysis and some heuristics, Relyzer identifies error sites that behave similarly in the presence of a single-bit error and groups these together into an equivalence class. It then performs an error injection experiment on just one representative error site (called a *pilot*) and uses its result to predict the outcome (masked, detected, or SDC) for all the error sites in the equivalence class. Hence, Relyzer is able to predict the resiliency characteristics of virtually all the error sites in the application with relatively few error injection experiments and high accuracy.

Approxilyzer builds upon Relyzer by introducing the notion of quality, determining the quality degradation for each predicted SDC, and applying this information in the areas of low cost resiliency and approximate computing. It takes as input an unmodified program (with input), a quality metric, and an optional acceptable quality threshold, and produces a comprehensive *output quality profile.* This profile provides the outcome of a transient single bit error in each error site in an execution. The outcome can be masked (an output is produced and is the same as the golden error-free output), detected (e.g., a fatal trap was invoked), or an output corruption (an output is produced, but is different from the golden output). For the last case, Approxilyzer further categorizes the output as a detectable data corruption (outputs that are visibly incorrect and could be detected; e.g., a NaN) and SDC. The SDCs are further binned into buckets based on the output quality.

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

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

Overall, this paper makes the following contributions:

(1) We propose and implement Approxilyzer. To our knowledge, this is the first tool that quantifies the quality impact of a single-bit error in all dynamic instructions of an execution with high accuracy. Our validation results show that Approxilyzer can predict output quality at very fine granularities with high accuracy (95% on average).

(2) We demonstrate how Approxilyzer can be used to quantitatively tune output quality vs. resiliency vs. overhead to enable ultra-low cost resiliency solutions (with a single bit error model). For example, we show that Approxilyzer determines that a very small loss in output quality (1%) can yield large resiliency overhead reduction (up to 55%) for 99% SDC coverage.

(3) We demonstrate how Approxilyzer can provide a first-order estimate of the approximation potential of a general-purpose program for a general error model in an automated way (only requiring the knowledge of a quality metric), enabling programmers or other tools to focus on the promising subset of approximable instructions for further analysis. For example, we show that, on average, static instructions resulting in 27% (most conservative quality threshold) to 36% (no quality threshold) of dynamic instructions in our applications

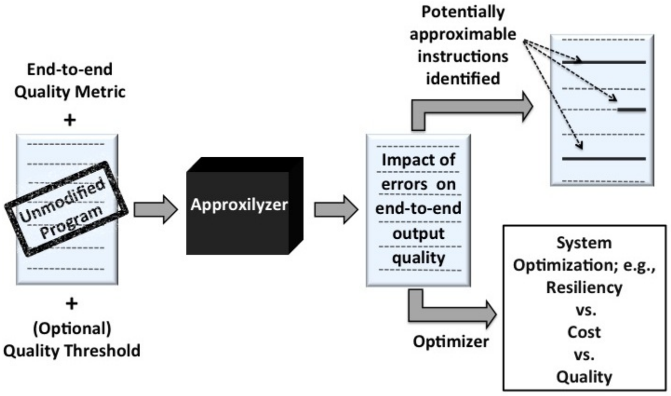

Fig. 1. Overview of the Approxilyzer framework and its usage.

have 32 continuous bits in their operand registers that are candidates for approximation.

While Approxilyzer does not claim to solve the perturbation-outcome problem, we believe that it provides a significant step forward and advances the state-of-the-art in the fields of approximate computing and low-cost resiliency. Section VI describes the limitations of Approxilyzer and how they can be partly overcome with future work and/or using Approxilyzer in conjunction with other techniques.

II. Approxilyzer and Its Usage

This section gives an overview of the Approxilyzer framework and its usage model, as illustrated in Fig 1.

A. Inputs to Approxilyzer

Underlying any approximate computing solution is the need to quantify output quality through an end-to-end quality metric. This metric is domain-specific [1], [23], [40] and Approxilyzer assumes that the programmer or user will supply it. Approxilyzer uses this metric to calculate the quality degradation of the erroneous output with respect to an error-free output.

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

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

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

B. Assessing Output Quality with Approxilyzer

Approxilyzer aims to quantify the impact of a single bit transient error on the program's end-to-end output quality, for error sites comprising each register bit of each dynamic instruction in an execution. To accomplish this, Approxilyzer builds upon Relyzer [12], a tool to predict the outcomes of errors in all of the above error sites. Relyzer's predictions, however, only consider whether an error results in being Masked, Detected, or a Silent Data Corruption (SDC). Relyzer does not consider output quality, marking all corruptions in an output (no matter how trivial) as an SDC outcome.

Relyzer uses a combination of error injection and program analysis to make its predictions for error outcomes. Its key insight is that errors that propagate through "similar" control and data flow paths in the program result in similar outcomes. It uses analysis and some heuristics to determine this similarity and groups resulting error sites predicted to have similar outcomes in an equivalence class. Using an error injection experiment on a representative from an equivalence class (the pilot), it predicts that all members of the class will have the same error outcome.

Approxilyzer refines the notion of an error outcome by including the quality of the erroneous output as part of this outcome. It assesses the quality of the corrupted/erroneous output by measuring its deviation from the error-free/precise output using the application-specific quality metrics provided by the user. Thus, an error that produces a quality degradation of (say) 20% is said to have a different outcome from one with a quality degradation of (say) 25%.

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

1) Quality-Aware Error Outcome Categorization: This section introduces a new categorization of error outcomes to incorporate the notion of output quality into the category of

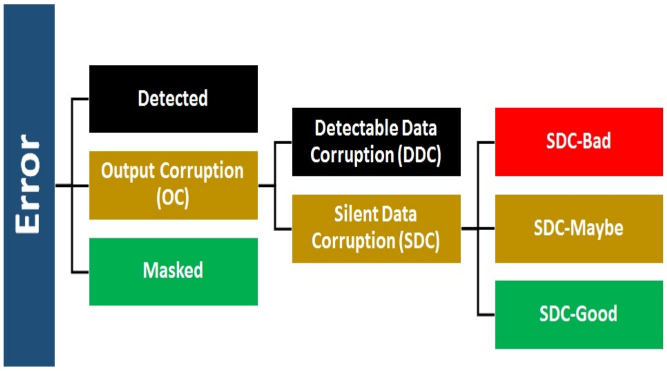

Fig. 2. A classification of errors.

errors traditionally known as Silent Data Corruptions (SDCs), illustrated in Fig. 2.

We use the term **Output Corruption (OC)** to indicate the outcome of an error where the execution runs to completion without crashing the program, but where the output does not match up identically to that of the golden output. In the literature, such outcomes have previously been uniformly referred to as SDCs. However, we observe that there is a subclass of these previously classified SDCs that is, in fact, detectable and not strictly silent. Such outcomes can be detected using a variety of low-cost mechanisms such as range detectors [11], [29]. We introduce additional detectors in Approxilyzer to catch NaNs, infinity values, negative outputs (if not expected by an application), and a check to see if the final output of the erroneous execution generates the same number of values as the golden output, irrespective of deviation. Our categorization refers to the output corruptions detected through the above means as **Detectable Data Corruptions (DDC)**. It refers to the remaining output corruptions, which are not detectable and truly silent, as Silent Data Corruptions (SDC).

The SDCs are further categorized as follows:

SDC-Good: These SDCs are *Highly Tolerable* SDCs which produce negligibly small quality degradations. This category also includes outcomes where the deviations from the golden output occur only in non-significant portions of the output (e.g., program related statistics and timing information).

SDC-Maybe: These are potentially tolerable SDCs. The entire class is not outright tolerable,but a subset of SDCs in this class may be tolerable based on user-provided application quality constraints – usually in the form of an acceptable quality degradation threshold.

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

The above categorization of the SDCs is dependent on the domain-specific quality metric (required) and acceptable quality thresholds (optional) provided by the user. If a quality threshold is provided by the user, then whether an SDC is tolerable or not is a binary decision based on whether the resulting output quality degradation falls below or above the quality threshold. In the absence of user-provided quality thresholds (e.g., in cases where the user wants to undertake program analysis or tuning), Approxilyzer classifies the SDC error sites into SDC-Good, SDC-Bad, and SDC-Maybe. Note that the classification into SDC-Good and SDC-Bad occurs only if the user chooses to apply common sense domain-specific thresholds provided by the tool (Section II-A); Otherwise all the SDC error sites are classified as SDC-Maybe. For each error site belonging to the SDC-Maybe error class, Approxilyzer also records its associated output quality degradation. Hence, the output quality for a given error site is characterized by its error outcome class and additionally, in the case of SDC-Maybe, by the amount of quality degradation introduced in the output.

TABLE I
ERROR OUTCOMES AND THEIR POTENTIAL FOR APPROXIMATION AND RESILIENCY OVERHEAD SAVINGS.

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

C. *Quality vs. Resiliency vs. Overhead*

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

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

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

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

D. Exploring Approximation Opportunities

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

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

Since each error site in the application contains the information regarding which dynamic instance of an instruction it belongs to, Approxilyzer can identify dynamic instructions that can be approximated. This can be useful to determine if the application will benefit from approximation techniques at the dynamic instruction granularity (e.g., task skipping [27] and loop perforation [37]). Sections III-E and IV-D show a case study for how Approxilyzer can be used to analyze applications for approximation potential.

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

III. Methodology

A. Workloads and Quality Measures

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

In the absence of specific domain studies and standardization [1], [23], [40], we have done our best to choose quality metrics that strike a balance between over- and under-estimating an application's tolerance to errors. For example, consider outputs with multiple components. Without further guidance, we must first determine a difference function for each component and then a method to aggregate across the components. Depending on the magnitude of the individual components, we use the absolute difference (small magnitude) or the relative difference (large magnitude) for the per-component difference function. To aggregate across components, we use the maximum instead of the average. In cases where there is an established common practice to analyze the output, we use the corresponding quality metric. For example, FFT produces a matrix and we use the relative difference in the bounded L2 norm to determine the quality.[1] More precisely, given a golden output G and a faulty output F, both having n components, where $n \geq 1$, Table II uses the following three quality metrics.

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

$$\textit{max-abs-diff} = max(|G_1 - F_1|, |G_2 - F_2|, \ldots, |G_n - F_n|) \quad (1)$$

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

$$rel\_err_i = \frac{|G_i - F_i|}{G_i} * 100 \quad (2)$$

$$\textit{max-rel-err} = max(rel\_err_1, rel\_err_2, \ldots, rel\_err_n) \quad (3)$$

(3) ***rel-l2-norm*****:** This metric is typically used in mathematics to directly compare two matrices. The metric estimates the relative difference in the bounded L2 norms (BL2N) of the

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

TABLE II
APPLICATIONS, QUALITY METRICS, THRESHOLDS, AND QUALITY BINS.

Application	**Input**	**Metric**	**DDC**	**SDC-Good**	**SDC-Bad**	**SDC-Maybe** **QB : $\{$Error range$\}$**
Blackscholes [3]	sim-large	max-rel-err max-abs-diff	$F_i >$ \$500 $F_i <$ \$0	max-abs-diff $< \$10^{-4}$	max-rel-err $> 100\%$	max-rel-err: 1: $\{10^{-4}\% \leftrightarrow 1\%\}$ 2: $\{1\% \leftrightarrow 2\%\}$ ... 99: $\{98\% \leftrightarrow 99\%\}$ 100: $\{99\% \leftrightarrow 100\%\}$
Swaptions [3]	sim-small	max-abs-diff	$F_i >$ \$500 $F_i <$ \$0	max-abs-diff $< \$10^{-4}$	max-abs-diff $>$ \$1	max-abs-diff: 1: $\{10^{-4} \leftrightarrow 10^{-3}\}$ 2: $\{10^{-3} \leftrightarrow 10^{-2}\}$ 3: $\{10^{-2} \leftrightarrow 10^{-1}\}$ 4: $\{10^{-1} \leftrightarrow 1\}$
LU [43]	512x512 matrix 16x16 blocks	max-rel-err	No App-Specific Detectors	max-rel-err $< 10^{-4}\%$	max-rel-err $> 100\%$	max-rel-err: same binning as Blackscholes
Water [43]	512 molecules	max-rel-err	No App-Specific Detectors	max-rel-err $< 10^{-4}\%$	max-rel-err $> 100\%$	max-rel-err: same binning as Blackscholes
FFT [43]	64K points	rel-l2-norm	No App-Specific Detectors	rel-l2-norm $< 10^{-4}\%$	rel-l2-norm $> 100\%$	rel-l2-norm: same binning as Blackscholes
Common to all apps			F_i = NaN F_i = Inf #F != #G	Errors in non-significant portions of the output		

golden and erroneous matrices. For any matrix A, having n elements, $a_1, a_2, \ldots, a_n$, we define the following,

$$\|A\|_{BL2N} = \frac{\|A\|_{L2}}{n} \tag{4}$$

where,

$$\|A\|_{L2} = \sqrt{\sum_{i=1}^{n} a_i^2} \tag{5}$$

The rel-l2-norm is thus defined as:

$$\textit{rel-l2-norm} = \frac{\|G - F\|_{BL2N}}{\|G\|_{BL2N}} * 100 \tag{6}$$

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

Finally, although output quality is a continuous function, for ease of analysis and comparison, we discretize it into multiple *Quality Bins (QB)*, fine-grained enough to capture small quality variations (last column of Table II). For example, with the *max-rel-err* metric, the bins are 1% wide (except at the boundary), and quality values of 12.1%, 12.6% and 13.3% are assigned a QB of 13, 13, and 14 respectively. The rest of the paper refers to QB values when quantifying quality. Thus, SDC-Maybe with QB10 refers to an SDC-Maybe outcome with an output quality degradation in the range specified by QB10.

B. Error Injection Framework and Speed

Our error injection simulation infrastructure is similar to that used for [12], based on Wind River Simics [41] and GEMS [21] running our applications on OpenSolaris and compiled to the SPARC V9 ISA.

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

We perform error injections only in the pilots of the generated equivalence classes. This can still lead to a large number of error injections, especially for longer applications. In order to reduce the simulation time, we only study 99% of the error sites in the application, thereby trading off simulation time for a modest loss in coverage [12]. The 1% of error sites not included in the study do not detract from the observations and gains reported. While identifying approximable instruction (Section II-D), these remaining error sites might introduce some false positives (in the event that they produce unacceptable errors). This is, however, consistent with our goal to tolerate some false positives, while minimizing false negatives, in the quest to uncover approximation opportunities in the application. For resiliency overhead tuning (Section II-C), these unexplored error sites might represent missed opportunity (in the event that they produce SDCs) for further overhead reduction using Approxilyzer.

Approxilyzer retains Relyzer's speed benefits, with negligible additional overheads. Compared to a (hypothetical) framework that would perform an error injection for each error site, Relyzer is able to prune error injections by 3 to 5 orders of magnitude [12]. The remaining error injections complete on our cluster of 200 machines in a few days. Approxilyzer adds a few hours to this process to perform quality calculations and error outcome categorizations. The analysis to generate quality vs. resiliency vs. overhead curves for an application (Section III-D) takes several minutes. Analyzing error outcomes and quality to identify approximable instructions (Section IV-D) takes a few seconds per application.

C. Approxilyzer Validation

1) Approxilyzer Baseline Validation: Approxilyzer relies on Relyzer's heuristics to group error sites that produce similar quality outcomes into an equivalence class. It predicts the quality of each element of an equivalence class based on the outcome of a fault injection experiment on its pilot. To validate these predictions, we perform experiments similar to those in [12].

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

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

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

2) Flexible Quality Window: Requiring the pilot's QB to exactly match the QB of the associated population is unnecessarily conservative and a tall order for any tool, especially for outcomes with quality at the QB boundaries. We therefore introduce a flexibility parameter, δ, that allows a fine-grained margin of error at QB boundaries. For the validation of pilot X described above, setting $\delta = x$ means that an error site in its population with QB of $12 \pm x$ would be considered as a correct prediction. Thus, pilot X's prediction accuracy with $\delta = 1$ is 92% and with $\delta = 2$ is 97%. The baseline validation described in Section III-C1 is the same as setting $\delta = 0$.

3) Equalizing Error Outcomes: We can further loosen our constraints on validation by considering the context in which Approxilyzer is used as follows.

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

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

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

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

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

D. Tuning Quality vs. Resiliency vs. Overhead

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

To produce the different tuning curves, we first identify the instructions that need resiliency protection for different output quality thresholds (range of QBs). Then we use a 0/1 knapsack algorithm to pick the instructions for resiliency protection that offer the specified coverage for the least overhead. A similar methodology is used in [12] to tune resiliency vs. overhead, but without relaxation of output quality.

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

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

E. Exploring Approximation Opportunities

As mentioned in Section II-D, Approxilyzer can be used as a tool to analyze the first order approximation potential of an unknown application along different dimensions. We show one such use case where we use Approxilyzer to analyze the approximation potential along the dimension of static instructions for our five workloads. We use the same technique used to identify which static instructions are potentially approximate, to also identify approximation along different static instruction granularities. For example, if all the error sites related to a particular register in a static instruction were deemed approximable, then we say that the register is approximable. In this case study we do this analysis for the following static instruction granularities:
(1) ***Full Instruction (FI)*****:** The entire static instruction (i.e., all register bits) is approximable.
(2) ***Partial Instruction, Full Register (PI_FR)*****:** At least one full register in the static instruction is approximable.
(3) ***Partial Instruction, Partial Register, x bits (PI_xb)*****:** At least one x bit long register chunk in the static instruction is approximable.

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

IV. Results

A. Output Corruption Distribution

Fig. 3 shows the distribution of outcomes for error sites in the studied applications.[2] Each application exhibits a unique distribution of error outcomes. At 68.8%, LU contains the highest percentage of Output Corruption (OC) causing error sites and Swaptions, at 15.6%, the lowest.

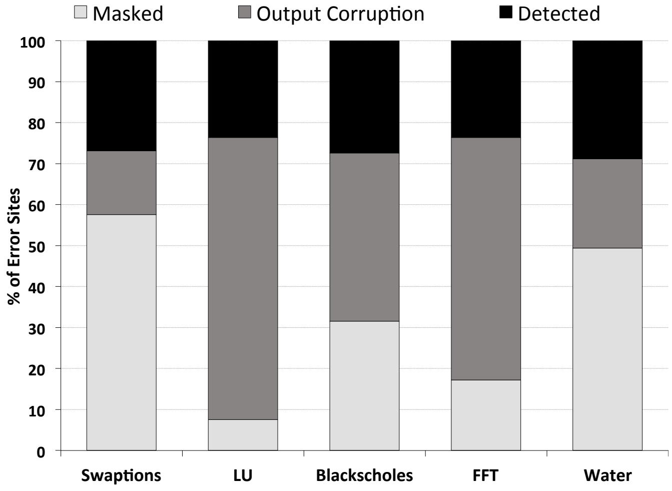

Fig. 3. Distribution of error outcomes for the applications studied.

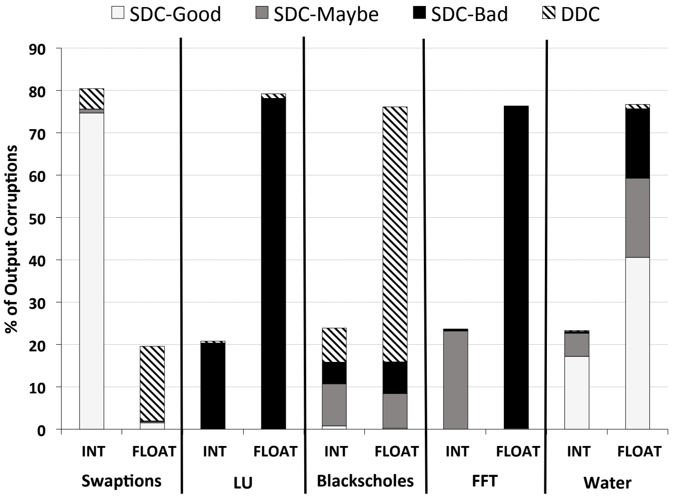

Fig. 4. Distribution of output corruptions (OC) in integer (INT) and floating point (FLOAT) registers.

Fig. 4 shows the different categories of output corruptions, separately for integer and floating point register error sites. Swaptions and Water show very high percentage of SDC-Good at 76% and 58% respectively while Blackscholes produces 68% DDC (for both Integer and Float combined). Swaptions and FFT show an interesting dichotomy in the behavior of errors in the integer vs. floating point registers, implying perhaps, a need for separate techniques for resiliency and approximation across the two different register classes. LU's OC error sites are almost exclusively (>98%) composed of SDC-Bad outcomes. This may either imply that LU is inherently not tolerant to errors or that the quality metric used to classify errors in the output of LU may not be the correct choice.

[2] The OC (originally SDC) rates reported in this work are different from the rates reported in previous work [11], [12] as our error model is different. We study errors in both integer and floating point architectural registers, while our prior work only considered integer registers.

These results illustrate how Approxilyzer can be employed to automatically analyze an application to gain insights into its behavior in the presence of perturbations.

B. Approxilyzer Validation

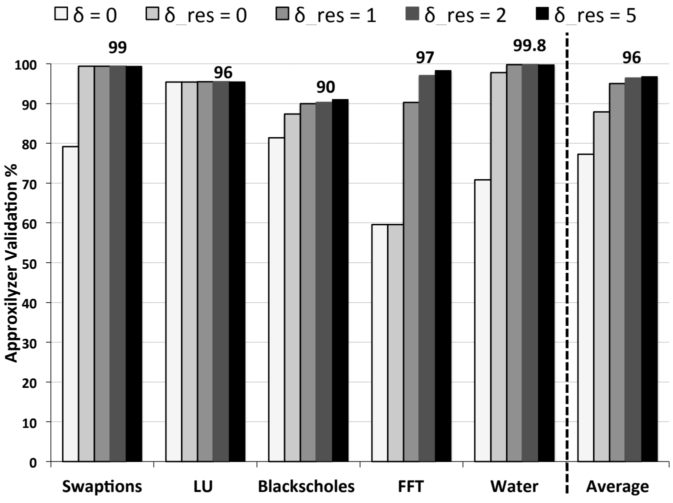

Fig. 5. Approxilyzer validation geared towards resiliency.

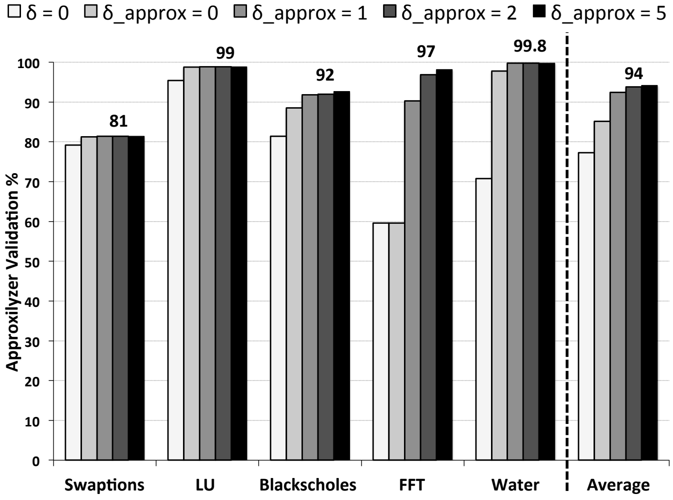

Fig. 6. Approxilyzer validation geared towards approximation.

We carry out validation experiments as described in Section III-C. We discuss the findings for validation for approximation and resiliency separately below.

Validation for Resiliency: The results for validation geared towards resiliency are shown in Fig. 5. All of the applications show very high pilot prediction rates with an average prediction rate of 96% across applications using a very fine quality window of 2 (bar corresponding to $\delta_{res} = 2$).

Swaptions and Water see big gains in validation just by applying the δ_{res} optimization. Both of these applications have high SDC-Good rates (Fig. 4) and therefore many pilots that are picked for validation belong to the SDC-Good outcome category. Some of these equivalence classes (with SDC-Good pilots) contain a mix of Masked and SDC-Good error sites, leading to lower overall validation for $\delta = 0$.

Water especially has a high rate of SDC-Good outcomes which are due to very small errors ($< 10^{-6}\%$) in the program statistics part of the output file. Approxilyzer heuristics (not surprisingly) combine these error sites with Masked outcomes into equivalence classes. As a result, applying the δ_{res} optimization causes the validation rate of Water to jump from 71% to 98%.

In addition to having equivalence classes with mixed SDC-Good and Masked outcomes (as described above for Water), Swaptions also has some pilots with DDC outcomes (Fig. 4) belonging to equivalence classes that feature a mix of DDC and Masked outcomes. These pilots represent error sites from a few floating point instructions that process randomly generated numbers. If the error causes the random number to exceed the (expected) range of 0 to 1, it causes floating point overflows which result in NaN values. Because Approxilyzer heuristics cannot accurately distinguish this special case, Swaptions contains some equivalence classes with a mix of Masked and DDC outcomes which results in poor validation for $\delta = 0$. Applying the δ_{res} optimization, causes the validation rate of Swaptions to go up from 79% to 99%.

While still high at 90% (for $\delta_{res} = 2$), Blackscholes shows the lowest validation accuracy of the applications studied. Further analysis shows that this is due to a few pilots whose equivalence classes have a mix of SDC-Maybe and SDC-Bad outcomes. This is why increasing the quality window size (δ) does not cause the prediction rate to increase. The reason behind the mixed equivalence class can be attributed to the fact that Blackscholes calculates the option price for a portfolio containing more than 64,000 options and hence, the same instructions produce OCs of different quality based on the input being processed at any given execution cycle. While range detectors to capture certain SDC-Bad outcomes and specialized heuristics to better capture variations in data patterns can be applied to handle some of these cases, we leave their implementation to future work. In spite of these special cases, Blackscholes shows high prediction rate.

Validation for Approximation: Fig. 6 shows the graph for Validation considering Approximation. On average, the validation percentage for $\delta_{approx} = 2$ is 94% across all applications. Swaptions shows lower validation predominantly due to poorly validated DDC pilots belonging to a few floating point instructions (validation accuracy for integer pilots with $\delta_{approx} = 2$ is 97%) operating on random numbers, as described above. While the δ_{res} optimization equalized these outcomes, the δ_{approx} considers DDC and Masked outcomes separately and hence the validation accuracy is not improved. Simple range detectors to check the range of the random numbers can resolve this issue and we leave its implementation to future work.

Overall, the average validation percentage, across $\delta_{approx} = 2$ and $\delta_{res} = 2$, for the applications studied is 95%. *Thus, we conclude that Approxilyzer can capture the Output Corruption* *quality – at very fine granularities – with high precision for the purposes of both Resiliency and Approximate Computing.*

C. Tuning Quality vs. Resiliency vs. Overhead

Fig. 7 shows the resiliency overhead cost vs. coverage for different levels of acceptable output quality degradation using the methodology in Section III-D. We show graphs for four of our five benchmarks (the fifth, LU, is discussed later). Each graph shows the following curves corresponding to different levels of acceptable quality degradation.

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

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

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

The gaps between the various curves for each point along the x axis represent the overhead/cost savings by not applying resiliency protection to those instructions that produce acceptable quality loss when perturbed. The benchmarks shown in Fig. 7 show significant overhead savings if the user can tolerate very small quality loss. For example, if the user can tolerate a 1% quality degradation in the output, then the resiliency overhead costs can be reduced by 20%, 55%, and 11% for Blackscholes, Water and FFT respectively, while still achieving 99% coverage (the difference between the *All Output Corruptions* and *All SDC-Bad + SDC-Maybe with QB>1%* curves at 99% on the x axis). Similarly, for a quality loss of less than one hundredths of a penny in final stock price (*All SDC-Bad + SDC-Maybe with QB>$0.001*), Swaptions achieves an overhead reduction of 26% while providing 99% coverage.

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

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

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

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

LU (not shown in the figure for brevity) shows no gains from either quality tuning or from not protecting SDC-Good and DDCs. This is because, as seen from Fig 4, LU produces only SDC-Bad corruptions and hence all of the instructions need protection.

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

D. Identifying Approximable Instructions

Figs. 8(a) and 8(b) show, for each application the worst and best case bound, respectively, on the number of static instructions, marked by Approxilyzer as candidates for approximation (as described in Section III-E). In order to understand the potential impact of approximating these static instructions, Figs. 8(c) and 8(d) show the proportion of dynamic instances produced by these static instructions in the full application. Note that while the static instruction percentage shown is for the fraction of static instructions studied, for a better insight, the dynamic instruction percentage reported is the fraction over the entire application, which includes dynamic instances of instructions we do not study (Section III-B).

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

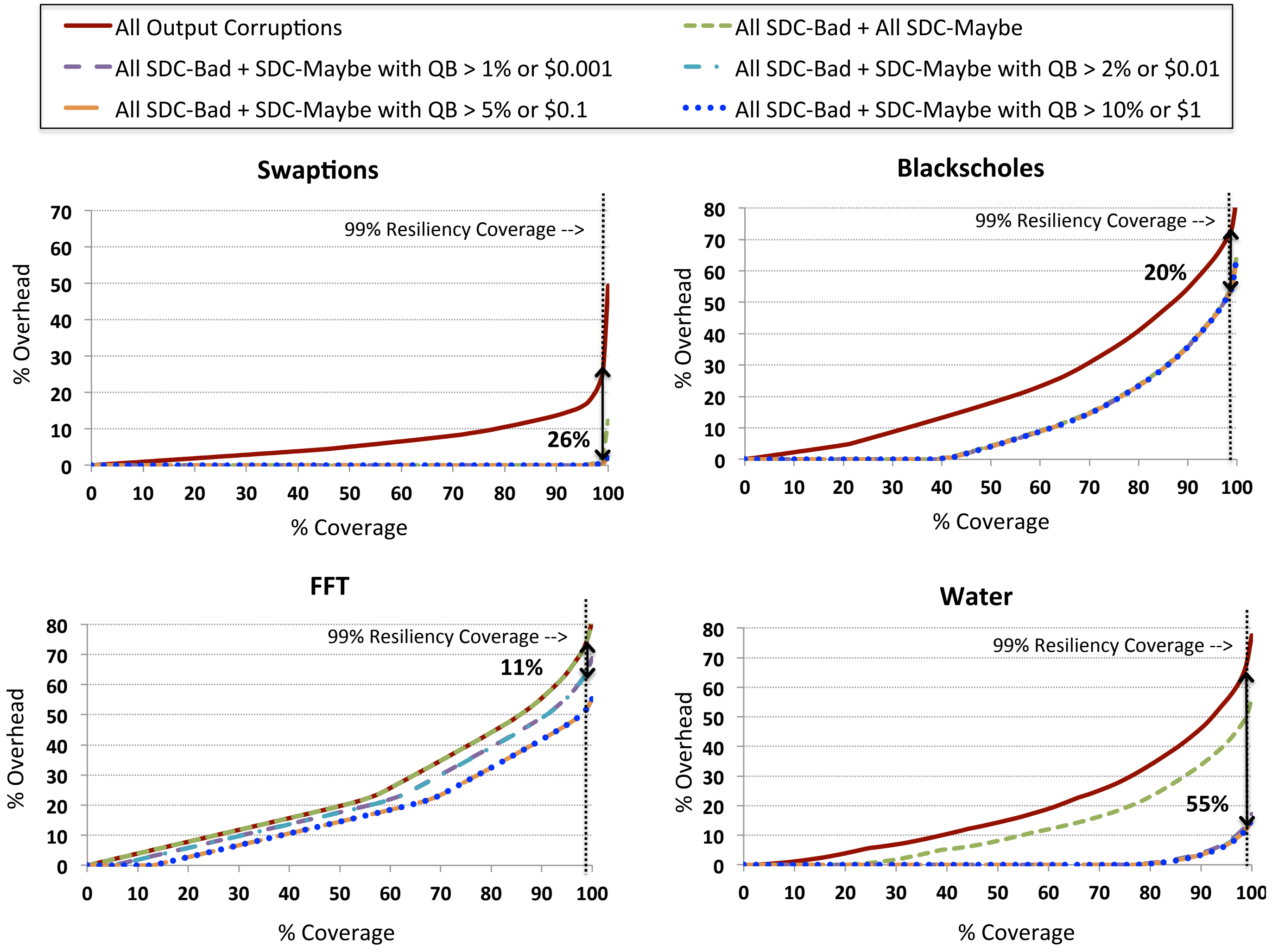

Fig. 7. Tuning resilience vs. execution overhead vs. output quality for different applications.

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

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

V. Related Work

Many techniques have been proposed that leverage approximate computing for improved performance, energy or reliability. Loop perforation [37], voltage scaling [14], [35], approximate ALU computations [10], [33], approximate kernels [2], [31], [36], neural hardware [8], and memory system approximations [34], [19] are all possible techniques that trade off accuracy for system benefits.

Programming language support, such as that in [6], [22], [24], [32], [33] helps programmers abstractly express approximations and check program correctness at the cost of increased programmer burden. Recent frameworks [5], [22], [25] build on these languages to automatically identify approximate regions of the code while providing some statistical [25] or probabilistic [22] guarantees on the final end-to-end error. While these frameworks advance the state-of-the-art to greatly reduce programmer burden, they still require the programmer

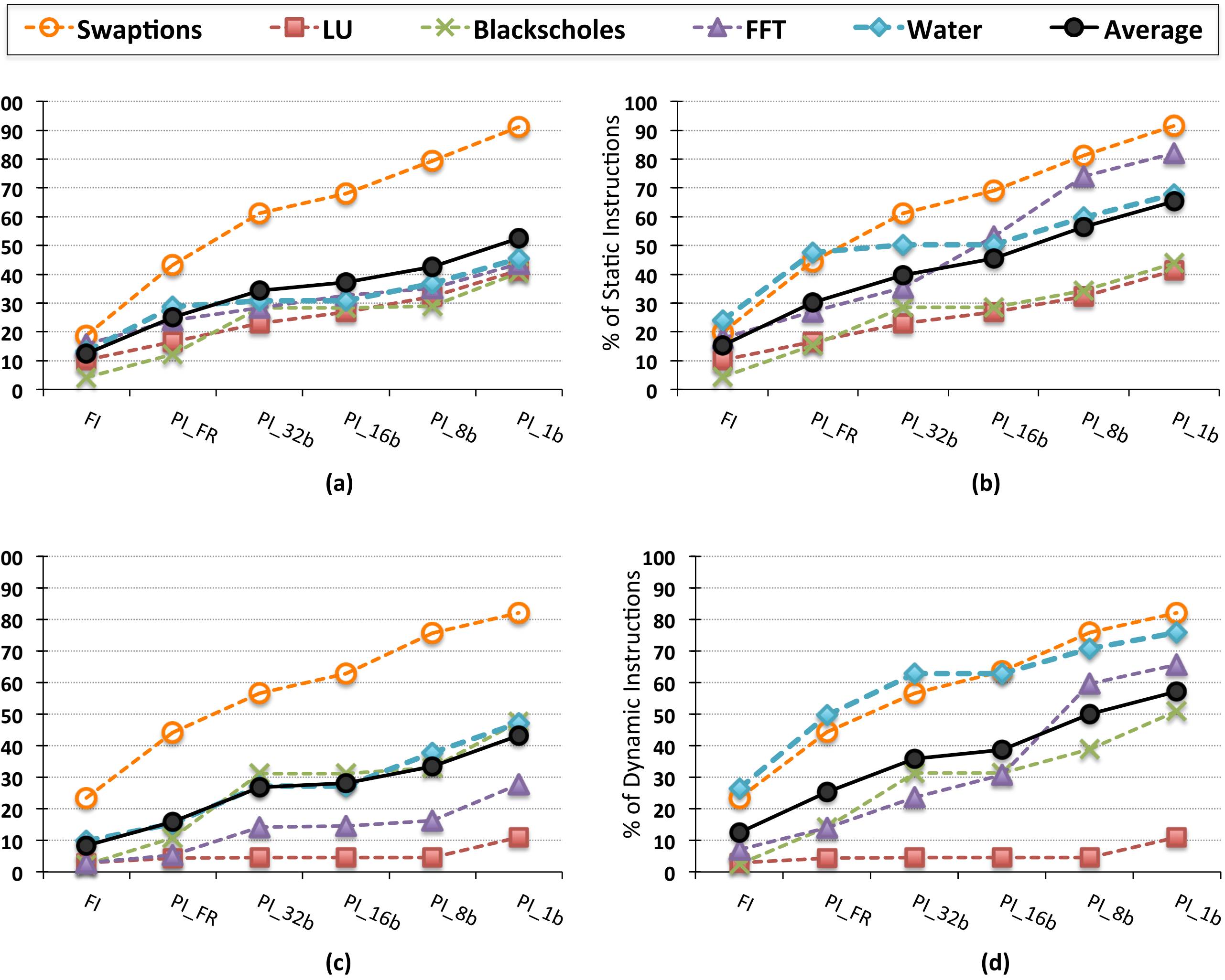

Fig. 8. Graphs (a) and (b) show the first order worst and best case bounds respectively for the percentage of static instructions (studied) that are approximable for each application at different granularities of approximation. Graphs (c) and (d) show the percentage of dynamic instructions (in the full application) generated by these static approximable instructions in the worst and best case respectively.

to adopt a new programming language and/or modify their source code. Thus, they cannot be used for very large multi-kernel programs (static analysis may be complicated and under-estimate the approximation potential) or for programs where the source code is not available (such as legacy code). We believe that Approxilyzer is a complementary technique and can be used as a front end plugin to these frameworks. A concurrent work [20] provides statistical guarantees on final output quality given an approximate kernel and accelerator configuration using compiler support and hardware binary classifiers. While this work focuses on coarse-gain approximation with accelerators, Approxilyzer is a general framework which studies approximation at the fine granularity of single instructions.

SAGE [31] automatically generates approximate kernels for GPUs but like other methods [2], [16] uses an online mechanism to catch unacceptable quality degradation in a reactive fashion. Approxilyzer on the other hand provides offline output quality information. Techniques such as [38] control output quality constraints by tuning various knobs in an approximate program. Approxilyzer solves the problem of identifying approximate code and as such is an orthogonal technique.

The idea of identifying unacceptable output corruptions for selective reduced-cost resiliency protection has previously been explored in the realm of soft computations. A combination of error injections and static analysis is used in [39] to identify Egregious Data Corruption (EDC) prone code and data segments in soft computations that can then be protected by detector placements. IPAS [17] uses machine learning to identify and protect only those Silent Output Corruptions (SOC) instructions that alter the output of scientific codes. Khudia et al. [15] use compiler analysis to identify critical variables in the application that are likely to generate Unacceptable Silent Data Corruptions (USDCs) in the presence of errors and only protect those using strategic expected value checks. Approxilyzer classifies error outcomes into categories based on approximation potential and predicts the impact errors in individual instructions with high accuracy. This allows for very fine tuning of resiliency protection schemes for different quality and overhead requirements.

Application of approximate computing to hardware resiliency has also been demonstrated in specialized domains such as bio-medical applications. Sabry et al. [28] study Electrocardiogram (ECG) monitoring wireless body sensor nodes and tradeoff inaccuracies inherent to the domain to achieve resiliency overhead savings. Approxilyzer also exploits accuracy loss for resiliency overhead savings but does so in a manner that can be used by any general-purpose program.

VI. Conclusions and Future Work

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

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

Although Approxilyzer has several limitations, we believe it moves the field of approximate computing forward in a significant way, providing a unique contribution to the "holy grail" problem of accurately predicting output quality impact of a general perturbation on a general computation without programmer burden. We list below some of the key limitations that we wish to address in our future work.

Error Model: We assume single-bit transient errors in an instruction's integer and floating point registers. Such an error model can be easily extended to cover the many microarchitecture level errors that also manifest as bit flips in architectural state read by an instruction. For example, a consequential bit flip in a pipeline latch can often be modeled as a bit flip in architectural state pertaining to the instruction that currently occupies the latch (if any). Validating Approxilyzer for such low level errors is part of our future work. (Relyzer already models and validates against errors in address generation units [12].) Admittedly, not all errors are single bit flips affecting only one instruction. We have already shown how given the context, we can derive useful information from single bit flip information to narrow down the subset of approximable instructions. We believe we can apply similar ideas to prune the analysis space for more general error models and use cases as well.

Input Dependence: Comprehensively analyzing the impact of errors on output quality even for a single program input is challenging and is the focus of this work. Considering different inputs is important and is our future work. Our goal (albeit ambitious) is to integrate Approxilyzer within the software testing workflow. We believe we can leverage the large body of work dealing with input dependence in software testing for our purposes.

Other Work: Unlike many prior works that aim to approximate data [32], [33], [34], Approxilyzer focuses on instructions. We are currently evaluating the effectiveness of a data- vs. instruction-centric approach to uncover approximation potential. Examining approximation potential at higher granularities (e.g., function or data structure level) and interaction with other granularities is also an interesting future direction. Finally, we would also like to validate Approxilyzer for multithreaded programs, likely with the heuristics modified to incorporate similarities in synchronization paths.

References

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

[2] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 198–209.

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

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

[5] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 470–487.

[6] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013, pp. 33–52.

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

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

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

[10] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design." in *Proc. of IEEE European Test Symposium*, 2013.

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

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

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

[14] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Designing a processor from the ground up to allow voltage/reliability tradeoffs," in *Proc. of International Symposium on High Performance Computer Architecture*, 2010.

[15] D. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *Proc. of International Symposium on Microarchitecture*, 2014.

[16] D. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *Proc. of International Symposium on Computer Architecture*, 2015.

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

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

[19] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," *SIGPLAN Not.*, vol. 46, no. 3, pp. 213–224, Mar. 2011.

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

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

[22] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[23] J. Park, E. Amaro, D. Mahajan, B. Thwaites, and H. Esmaeilzadeh, "Axgames: Towards crowdsourcing quality target determination in approximate computing," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 623–636.

[24] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, "Flexjava: Language support for safe and modular approximate programming," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 745–757.

[25] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik, "Expax: A framework for automating approximate programming," in *Technical Report, Georgia Institute of Technology*, 2014.

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

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

[28] M. M. Sabry, G. Karakonstantis, D. Atienza, and A. Burg, "Design of energy efficient and dependable health monitoring systems under unreliable nanometer technologies," in *Proceedings of the 7th International Conference on Body Area Networks*, 2012, pp. 52–58.

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

[30] F. Sala, C. Schoeny, and L. Dolecek, "Approximate and noisy computing: Connections to the information-theory world," in *Workshop on Approximate Computing Across the Stack (WAX)*, 2016.

[31] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 13–24.

[32] 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.

[33] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 164–174.

[34] J. San Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelganger: A cache for approximate computing," in *Proc. of International Symposium on Microarchitecture*, 2015.

[35] J. Sartori and R. Kumar, "Architecting processors to allow voltage/reliability tradeoffs," in *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2011, pp. 115–124.

[36] ——, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," *Multimedia, IEEE Transactions on*, vol. 15, no. 2, pp. 279–290, Feb 2013.

[37] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[38] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive control of approximate programs," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 607–621.

[39] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *Proc. of International Conference on Dependable Systems and Networks*, 2013.

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

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

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

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