GangES: Gang Error Simulation for Hardware Resiliency Evaluation[*]

Siva Kumar Sastry Hari[†] Radha Venkatagiri[‡] Sarita V. Adve[‡] Helia Naeimi[§]

[†]NVIDIA [‡]University of Illinois at Urbana-Champaign [§]Intel Labs
[†]shari@nvidia.com [‡]{venktgr2,sadve}@illinois.edu
[§]helia.naeimi@intel.com

Abstract

As technology scales, the hardware reliability challenge affects a broad computing market, rendering traditional redundancy based solutions too expensive. Software anomaly based hardware error detection has emerged as a low cost reliability solution, but suffers from Silent Data Corruptions (SDCs). It is crucial to accurately evaluate SDC rates and identify SDC producing software locations to develop software-centric low-cost hardware resiliency solutions.

A recent tool, called Relyzer, systematically analyzes an entire application's resiliency to single bit soft-errors using a small set of carefully selected error injection sites. Relyzer provides a practical resiliency evaluation mechanism but still requires significant evaluation time, most of which is spent on error simulations.

This paper presents a new technique called GangES (Gang Error Simulator) that aims to reduce error simulation time. GangES observes that a set or gang of error simulations that result in the same intermediate execution state (after their error injections) will produce the same error outcome; therefore, only one simulation of the gang needs to be completed, resulting in significant overall savings in error simulation time. GangES leverages program structure to carefully select when to compare simulations and what state to compare. For our workloads, GangES saves 57% of the total error simulation time with an overhead of just 1.6%.

This paper also explores pure program analyses based techniques that could obviate the need for tools such as GangES altogether. The availability of Relyzer+GangES allows us to perform a detailed evaluation of such techniques. We evaluate the accuracy of several previously proposed program metrics. We find that the metrics we considered and their various linear combinations are unable to adequately predict an instruction's vulnerability to SDCs, further motivating the use of Relyzer+GangES style techniques as valuable solutions for the hardware error resiliency evaluation problem.

1. Introduction

Moore's law continues to provide increasing numbers of devices on chip, but with increasing vulnerability to failures. Transient hardware errors from high-energy particle strikes (or soft errors) are expected to become a dominant category of in-field processor failures [1, 2, 5]. Traditional resiliency solutions that rely on full hardware or software redundancy have intolerable area, power, and/or performance costs for most computing markets. Future systems must, therefore, deploy low cost in-field resiliency solutions to guarantee continuous error-free operation.

Error detection is an important component of a hardware resiliency solution and must especially be low cost because it is always on. Software anomaly based error detection has emerged as an attractive approach that detects only those hardware faults that propagate to software [7, 10, 13, 16, 21, 24, 27]. This approach places near-zero cost error monitors that watch for anomalous software behavior. Recent evaluations have shown that this approach is effective in detecting in-field transient errors [16, 13]. A small fraction of errors, however, still escape these detectors and silently corrupt application outputs, producing Silent Data Corruptions (SDCs).

For widespread adoption of software anomaly based detection approaches, we therefore need systematic error analysis techniques to evaluate their effectiveness. Such techniques must identify any remaining SDC producing software locations so they can be appropriately protected with additional anomaly monitors or other techniques, to achieve the SDC rate acceptable for the application. We identify three classes of error analysis techniques proposed in the literature, focusing on transient errors:

(1) Error injection: The most widely used evaluation technique is error injection, where an error is injected (typically one error at a time) in a given cycle in a real or simulated system and its impact studied [17, 16, 15, 23, 22, 19]. This technique can predict the impact of an error with high accuracy. Unfortunately, comprehensively injecting each error of interest at each cycle in an execution is prohibitively time consuming. Most work therefore performs statistical fault injection which injects selected error types in a randomly selected sample of execution cycles. While such a technique can provide statistical summaries such as average SDC rate, it does not indicate which remaining instructions in the unsampled set could result in SDCs and hence must be protected.

(2) Program analysis: Some evaluation techniques examine certain (static or dynamic) program properties to identify program locations that are susceptible to producing

[*] This work was supported in part by the National Science Foundation under Grants CCF-0811693 and 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.

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

SDCs [9, 20, 3, 25]. These techniques are much faster than error injection based techniques (running time average of approximately 5 CPU hours for the techniques and applications we study), but their accuracy in finding SDC-causing error sites has previously been hard to validate.

(3) Hybrid injection+analysis: A recent technique, called Relyzer, uses a hybrid of error injection and program analysis [12]. Analogous to previous work, it applies (simple) dynamic analysis to predict when errors in certain application locations will be detected. Unlike previous work, it also applies static and dynamic analyses (with some heuristics) to determine when multiple dynamic instruction instances will produce the *same outcome* (i.e., detection, masking, or SDC) for a given error in the instructions' operands. For a set of dynamic instructions that are shown to produce equivalent error outcomes, Relyzer performs error injection in only one of these instructions (referred to as the *pilot*) to determine that outcome. To search for equivalent-outcome instructions (referred to as an equivalence class), Relyzer examines (i) dynamic instances of the same static instruction and (ii) instructions that define a variable and first use it.

Relyzer was able to prune the number of error injections needed to identify the error outcomes for virtually *all* instructions by 99.78% and its heuristics showed an aggregate accuracy of >96% (for error outcome prediction) across all the techniques and errors studied [12]. Although Relyzer is 2 to 6 orders of magnitude faster than comprehensive error injection, it unfortunately still spends a significant amount of time in error injection for the pilot instructions. Specifically, unless the error is detected, an error injection experiment for a pilot requires simulating the application to completion. The simulated execution's output is then compared to that of the error-free execution output to determine if that error is masked or produces an SDC. Thus, while Relyzer is certainly more practical than comprehensive pure error injection, compared to program analysis based techniques, its accuracy comes at a significant cost in speed. For example, for a set of only 8 applications on our cluster of 188 nodes, Relyzer takes about 3.5 days of wall clock time to analyze about 95% of the error sites; about 90% of this time is spent on error injections.

1.1. Contributions of this Work

All three error analysis techniques described above are limited by speed or (potentially) accuracy. The ideal technique will have the speed of pure program analysis based methods and the accuracy of pure error injection based methods. This paper makes two contributions towards such an ideal technique.

GangES: A new hybrid error analysis technique:

We propose a new technique and tool called Gang Error Simulator or GangES[1] that takes a different, but complementary, approach to Relyzer to reduce error simulation time. GangES is based on the observation that a set or "gang" of error simulations that result in the same intermediate execution state (after their error injections) will produce the same error outcome. If such an intermediate state can be detected, we need only complete one simulation from such a gang – the others can be terminated early and use the outcome of the single completed simulation. This observation has the potential to significantly reduce overall error simulation time, but requires addressing two challenges: (i) *when* to compare the state of error simulations and (ii) *what* state to compare.

A judicious choice for when to compare (which execution points) is critical because the instruction sequences executed by multiple error simulations may temporarily diverge but merge again. A judicious choice of what to compare is also critical because naively comparing all register and memory locations can be prohibitively inefficient.

For when to compare, we select program locations where multiple error simulations are likely to reach even if the error (temporarily) exercised different system events and branch directions. We leverage the previously proposed program structure tree (PST) from the compiler literature for this purpose [14]. A PST organizes an application's control flow graph (CFG) into nested single-entry single-exit (SESE) regions (Section 2.2). If an execution exercises the entry edge of a SESE region, then it will also exercise the exit edge, as long as the dynamic control flow complies with the static CFG. Therefore, if an error is injected in a particular SESE region, the corresponding SESE exit edge will most likely be exercised. Such SESE exit points provide common program locations to compare execution states of error simulations. The SESE exit points also represent program locations where potentially few program variables are alive, limiting the amount of state to compare. We compare all the (potentially) live registers at these points and any memory locations to which there have been stores before these points.

GangES can be used to determine error-outcome equivalence for arbitrary sets of error simulations. Here we use it in conjunction with Relyzer; i.e., its input is error sites (pilots) that are not pruned by Relyzer. Without GangES, Relyzer would run error simulations for each of the input pilot instructions until the error was detected or the application completed. GangES instead checks for equivalence and terminates several of these simulations. Overall, we found that after applying GangES, only 36% of the error simulations in its input required running the application to completion and checking the output to determine the fault outcome. 92% of the error simulations that were terminated early by our approach required an average of only 3,025 instructions to be executed before termination; the next 7% required about 16,000 instructions on average. Overall, we found that GangES replaced Relyzer's error simulation time of 14,225 CPU hours (for analysis of 95% of all error sites) with a total time of 6,010 CPU hours, providing a wall-clock time savings of 57.7% for our workloads and error model.

Accuracy of pure program analysis methods:

Although GangES significantly reduces the error simulation

[1] Pronounced as gan-jeez.

time for Relyzer, it is still not as fast as pure program analysis based techniques. A legitimate question therefore is whether current program analysis based techniques obviate the need for tools such as GangES altogether. It has been previously difficult to test the accuracy of program analysis based techniques since there has not been a mechanism for comprehensive fault injection. The availability of Relyzer (and GangES) allows us to perform such a test.

Our second contribution, therefore, is an evaluation of the accuracy of several program analysis based techniques using Relyzer. We use program based metrics proposed by [20] and some derivatives as examples of pure program analysis based techniques.[2] Using Relyzer, we find that these metrics and their various derivatives and combinations that we study are unable to adequately predict an instruction's vulnerability to SDCs. Although it is possible that other analysis based techniques are more accurate, a comprehensive evaluation of all such techniques is outside the scope of one paper. Nevertheless, the negative results provided here do indicate that there is much work needed to develop accurate pure program analysis based techniques, and techniques like Relyzer+GangES provide a valuable solution for the error resiliency evaluation problem.

2. GangES: A hybrid error analysis technique

GangES presents a transient hardware error simulation technique that takes as input a set of errors to be simulated for an application and outputs the outcome for each error (masked, detected, or SDC). Each error in the error set specifies a dynamic instruction instance, a hardware resource used by that instruction instance, and the type of error to be injected in that hardware resource for that instruction instance. In our experiments, we focus on single-bit flips in the integer architectural registers used by the specified instruction instance (one bit flip per simulation). For error detection, we use detectors similar to those used in Relyzer (Section 3). In our experiments, the input set of errors for GangES consists of errors that Relyzer is not able to prune (i.e., pilots of instruction equivalence classes as categorized by Relyzer). Relyzer must perform error injections for all of these errors – for those not detected, Relyzer must execute the application to the end and compare the output with that of the error-free execution to determine masked or SDC outcomes.

GangES aims to reduce the overall evaluation time for its input error set by terminating as many error simulations as possible soon after error injection and well before the end of the application (including those simulations that would eventually be masked or produce SDCs). Our approach is to repeatedly compare execution states of a set or gang of multiple simulations in progress (hence the name Gang Error Simulator). Any simulations in the gang that reach identical states will produce the same outcomes and all but one of them can be terminated. Figure 1 illustrates the differences between Relyzer and GangES.

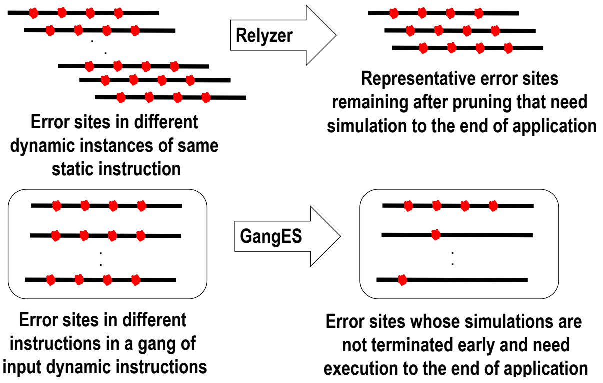

Relyzer
Error sites in different dynamic instances of same static instruction
Representative error sites remaining after pruning that need simulation to the end of application
GangES
Error sites in different instructions in a gang of input dynamic instructions
Error sites whose simulations are not terminated early and need execution to the end of application

Figure 1: Difference between Relyzer and GangES

A naive implementation would compare the entire system state (processor and memory state) at every cycle to identify the earliest point in the execution to terminate an error simulation. Since the entire system state may consist of megabytes to gigabytes of data, comparing it on every cycle can be prohibitively expensive in time. Moreover, such comparisons may not identify error equivalence effectively because even a single mismatch in temporarily divergent state (e.g., due to temporarily divergent control flow) or in dead values will flag non-equivalent error outcomes. Hence, the challenge in developing a time-effective simulation framework is in identifying what state to compare and when to compare it.

2.1. What state to compare?

The state to compare at an execution point can be divided into two components – processor register state and memory state. The size of the entire memory state at a given point in the execution is significantly larger than the memory state relevant to the processor. Ideally, we want to identify only the live memory state (the memory locations that will be read in the future before being overwritten) – this is potentially much smaller than the full memory state. However, the live memory state for different erroneous executions and for the error-free execution may be different. Moreover, identifying the live memory state is known to be a complex problem [18].

Our approach is to compare the memory addresses and data that are *touched* (written) by the multiple error simulations. This significantly reduces the amount of memory state to compare. Consider a set (gang) of error simulations that are being compared. Until the point of the first error injection, all these simulations will touch the same addresses. We therefore need to start comparing the touched addresses only after the point of the first error injection in the gang. This motivates considering errors that are "close-by" in execution time for grouping in one gang. Section 3.1.2 describes our specific methodology for ganging error sites together.

For processor register state, it would be efficient to compare the values of all the registers at comparison points. This,

[2] Although the work in [20] is motivated by error detection analysis, it also discusses the application to determining SDC vulnerability.

however, may not be effective in determining error equivalence. Comparing just the live architectural registers may provide more opportunity for simulation equalizations. The live register state at a given point in a program for an error-free execution can be obtained statically. However, an error in an execution may result in a different control flow changing the live state for that execution. Hence, we obtain a conservative live set of registers dynamically by fast forwarding the execution to hundreds to thousands of instructions and removing the registers that are written before being read from the set of all the registers in this fast-forward phase.

2.2. When to compare executions?

For when to compare execution states, our approach is to select program locations where all executions would reach even if different system events take place or different branch directions are exercised during error simulations. To select such program locations, we identify single-entry single-exit (SESE) regions from the control flow graph of the application. Formally, a SESE region is defined as an ordered edge pair (a,b) of distinct control flow edges, a and b, where a dominates b; i.e., every path from start to b includes a; b postdominates a; every path from a to exit includes b; and every cycle containing a also contains b [14]. For every execution that exercises the entry edge of a region, the exit edge will be exercised, assuming control follows the static CFG. Therefore, for nearly all simulations where errors are injected in a particular SESE region, the corresponding SESE exit edge would be exercised, providing a common program location to compare execution states. An exception to this is an error that changes the dynamic control flow such that it does not comply with the static CFG; in this case, our technique is still correct, but it may loose an opportunity to show equivalence of error simulations.

Our algorithm to identify the comparison points is inspired by and similar to the SESE regions identification algorithm by Johnson et al. [14]. They provide a linear-time algorithm for finding SESE regions and for building a hierarchical representation of program structure based on SESE regions called the program structure tree (PST). This algorithm works by reducing the problem to that of determining a simple graph property called cycle equivalence: two edges are cycle equivalent in a strongly connected component iff for all cycles C, C contains either both edges or neither edge. The algorithm is based on depth-first search for solving the cycle equivalence problem, thereby finding SESE regions in linear time.

In straight-line code, the region between any two points is a SESE region; we will ignore these regions and focus only on the block-level CFG where the straight-line code has been coalesced into basic blocks. However, we modify the CFG to potentially obtain more comparison (SESE exit) points for a given error site as follows. We split the non-SESE regions (with multiple entry and/or exit points) into two or three regions such that a new SESE region is created with as many static instructions as possible (and with many potential error sites) in it. For a basic block with multiple entry edges,[3] we split it into two blocks such that the first one with multiple entry edges has as few instructions as possible and the second block has a single entry edge. If the original block had only one exit edge, then the new second block becomes a new SESE region providing error sites in it with extra opportunity to compare state. Similarly, we also split the blocks with multiple exit edges such that the first block has a single exit edge and the second one has the minimal instructions. We then apply Johnson et al.'s algorithm to obtain the SESE regions [14]. Figure 2 shows a CFG and the SESE regions obtained by applying the above algorithm on it.

Once all the SESE regions are obtained, which are typically nested (see Figure 2), they are organized in a hierarchical representation to obtain a program structure tree (PST) using the algorithm proposed by Johnson et al. A SESE region that immediately contains other regions is considered parent for the containing regions; e.g., a becomes the parent for regions b and d from Figure 2.

We modified this tree by adding new leaf nodes as children to SESE regions that immediately contain non-SESE blocks (Figure 3). A new leaf node contains instructions that do not belong to any of its sibling SESE regions (SESE regions that are immediately contained in its parent). For example, a new leaf node containing instructions from blocks 1 and 16 is added as a child of a (Figure 3). This modified PST is utilized to identify the next comparison point during an error simulation. By traversing up the tree from a node where error injection is being considered, exit points of subsequent parent nodes become the comparison points.

In the example shown in Figure 2, state comparisons for simulations for errors in basic block 3 are performed when the exit edge of the current SESE region (c) is exercised and when exit edges of all ancestors of the current SESE region according to the modified PST are exercised (Figure 3); i.e., when edges 7 → 8, 8 → 16, and 16 → *end* are exercised. Similarly, for simulations for errors in basic block 13, the checks are performed at exit edges 14 → 15, 15 → 16, and 16 → *end*.

3. Methodology

3.1. GangES

We evaluate GangES by employing it to evaluate the resiliency of applications when using software anomaly based error detectors (e.g., fatal traps and application assertion failures). We used eight single-threaded applications in our study – four each (randomly selected) from the SPLASH-2 [29] and PARSEC [4] benchmark suites. Table 1 provides a brief description of these applications and inputs used. Extending GangES to multithreaded applications is one of our future directions.

We use an instruction-level transient error model. Our model injects a transient error in the form of a single bit flip

[3] Note that a basic block has a single entry instruction, but there may be multiple edges into the instruction.

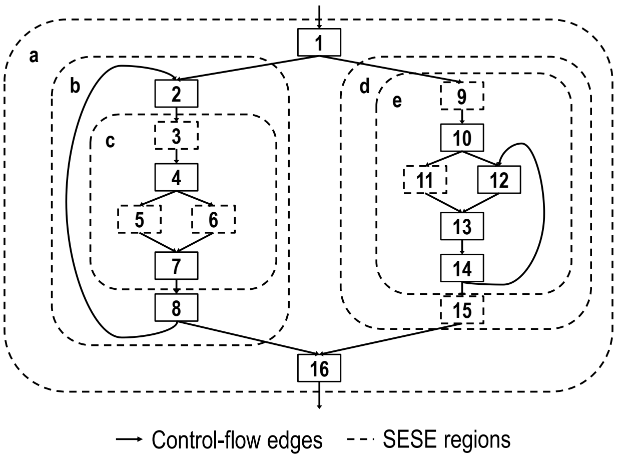


Figure 2: SESE regions. This example shows a program's control flow graph and identifies the SESE regions (enclosed by dotted lines). The exit points of the SESE regions form the comparison points.

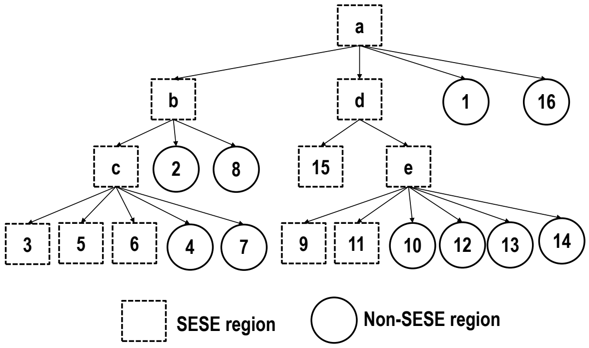


Figure 3: Modified program structure tree (PST) for the CFG shown in Figure 2. This example shows a hierarchical representation of nested SESE regions. It also adds non-SESE regions as new leaf nodes to non-leaf SESE regions; e.g., a non-SESE region 4 is added to c. These new nodes contain instructions that do not belong to any of the sibling SESE regions.

in a specified bit of a specified architectural integer register accessed by the specified dynamic instruction. Specifically, we identify an error with a tuple consisting of a dynamic instruction count in an execution, the program counter of the instruction that exercises the error, integer register operand, and bit location.

As mentioned earlier, although Ganges can accept any set of error sites as input, we focus on the sites that Relyzer is not able to prune. Even these require substantial time to determine their outcome through error injection simulations (the full simulations are required to determine Relyzer's wall-clock time and the speedup provided by Ganges). We therefore restricted our input to Ganges to be the minimal set of error sites that would provide 95% coverage; i.e., the outcomes of error injection simulations for this set enable Relyzer to determine the error outcomes for 95% of all our error sites (at least 92% for each application).

	Application	Input
PARSEC 2.1	Blackscholes	sim-large
	Fluidanimate	sim-small
	Streamcluster	sim-small
	Swaptions	sim-small
SPLASH 2	FFT	64K points
	LU	512×512 matrix, 16×16 blocks
	Ocean	258×258 ocean
	Water	512 molecules

Table 1: Applications studied.

We made one modification to Relyzer's pruning algorithm. Relyzer uses a control equivalence heuristic, where it decides certain equivalence classes based on the outcomes of the next 5 branches after an error [12]. We modify this to consider 50 upcoming branch outcomes since the first use of the injected error. Since examining the next 50 branches can potentially create too many equivalence classes, we limit the number of classes per static instruction to 50. This modification gave substantially better accuracy in predicting the outcome of certain error sites in our applications.[4]

The GangES implementation has two components: (1) static program structure identification and (2) a framework to perform dynamic error injections and state comparisons.

3.1.1. Static program structure identification: We implement our static program analyses at the binary level. Since our error injection infrastructure is developed for the SPARC V9 ISA model [28], we restrict our study to SPARC V9 binaries using an in-house static binary analyzer. The tool constructs a control flow graph from the binary and performs basic control flow analyses. We implemented intra-procedural SESE region identification and PST generation algorithms (Section 2.2) in this infrastructure.

3.1.2. Framework performing dynamic error injections and state comparisons: Once we identify when to compare execution states, the next steps are to (1) identify when to start an error simulation and how to group error simulations together for efficiency, (2) inject the error, and (3) collect and compare state at comparison points for early termination. Figure 4 explains how our technique works with an example. We use Wind River Simics [26] to implement our error injection and simulation state comparison algorithm.

Identifying when to start an error simulation and how to group injections together: We take several application checkpoints (using Simics' checkpointing feature) at periodic points (after every 100 million instructions) in the application. This allows us to start simulations from intermediate execution points(① in Figure 4), instead of starting from the beginning of the application for every simulation, saving running time. We group (gang) error injection sites such that each error site in a gang has the same checkpoint immediately preceding

[4] The previous validations for Relyzer reported aggregate accuracy statistics [12] – we additionally conducted a validation study (not reported here) specifically for instructions with outcomes predicted as SDCs.

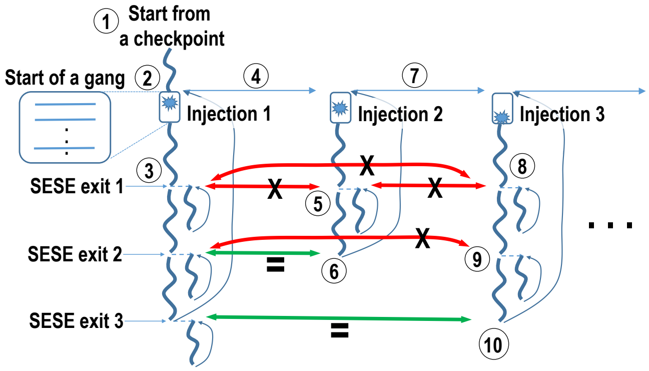


Figure 4: Error simulations in GangES.

it; we can therefore start all simulations in a gang from the immediately preceding checkpoint.

Further, we ensure that any pair of errors in a gang are separated by at most 100,000 instructions and there are at most 1,000 error sites in a gang. Since we need to compare touched addresses from the first error site in a gang of error simulations, the above parameters provide a bound on the amount of state that needs to be stored and compared for these simulations. We observe an average gang size of 279 with these parameters. When we lowered the maximum distance between errors in a gang from 100,000 to 1,000, the average gang size reduced rapidly to 79. When we increased the maximum gang size to 2,000 errors, the average gang size increased to 410 with only two applications (Blackscholes and Ocean) seeing a noticeable increase in their gang sizes.

Starting a gang of error injections: We start the simulations for a gang from an application checkpoint (① in Figure 4) and create a Simics bookmark just before the first error injection point (② in Figure 4). A bookmark set at a particular point in an execution allows Simics to move the simulation backwards to that point from anywhere in the application, restoring the execution state at that point. This feature allows us to move backwards in an execution to start a different error injection run from a particular gang. At an error injection point, we inject the error directly into the architecture register, according to our error model.

Collecting and comparing execution states: After an error injection, we continue simulation until a comparison point, the exit point of the (current) SESE region that contains the instruction where the error was injected, is reached. We set the breakpoint at the program counter of the instruction that immediately follows the current SESE region's exit edge (③ in Figure 4). We also set breakpoints at the ancestor SESE region exits according to the PST (e.g., we set breakpoints at the exits of regions *3*, *c*, *b*, and *a* for error injections in region *3* from Figure 3). Whenever a breakpoint is reached, we remove it to avoid further interrupts in the simulation (which may be caused by regions within loops).

At each comparison point, we compare the live register state and touched memory state with other simulations that reached this point previously; for example, the state at ⑧ is compared with the state at ⑤ and ③ in Figure 4. (The first error simulation only collects the state for future comparisons.) Since different error injection sites (in a gang) may belong to different sub-trees in a PST (and different SESE regions), we ensure that we only compare states when simulations reach the same program location by comparing the program counter of the breaking instruction.

For state collection, we identify live registers by listing all processor registers, executing the next thousand instructions, and removing the registers that are written before being read in this period of 1000 instructions (we use Simics' bookmark utility to execute forward and return to the same execution point). To determine touched memory state, we observe memory operations (reads/writes) through a memory module attached to Simics (which is added to the simulation just before the first injection, at ② in Figure 4).

We continue state collection and comparisons to all previously collected states from prior error simulations until one of the following occurs. If the compared states match (⑥ and ⑩ in Figure 4), we terminate that simulation and declare it as equivalent to the matching previous simulation and continue to the next error simulation, which involves rolling back to the bookmark at ② in Figure 4 and continuing to the next injection point (⑦ in Figure 4). We also terminate a simulation if the error is detected. We use detection techniques similar to those used by Relyzer and recent software anomaly based systems; e.g., a fatal processor exception, application assertion failure, application abort, out-of-bounds access, or timeout. If an error simulation continues for too long (defined below) without showing a state match or a detection, we mark the error as needing full simulation and move to the next error injection (④ in Figure 4). We consider a simulation to run for too long if it invokes 5 SESE exits or executes a threshold number of instructions. We set the latter threshold to 100,000 instructions by default to limit the memory footprint to <2GB. A small number of cases exceed this limit; we then adjust the threshold to 100 for the remaining injections in that gang.

At the end of the above process, for each error, we know that it either has an outcome equivalent to another error, or is detected, or must be fully simulated.

3.1.3. Evaluation metrics: To evaluate GangES, we first determine the wall clock time to identify the outcomes of all its input error sites by performing full error injection simulations (on all input error sites). We compare this with the wall clock time that GangES needs to identify the number of errors that need full simulations plus the time needed to run such simulations to completion to obtain the outcomes. An error simulation time is measured from the same application checkpoint for both sets of wall clock times.

We also show the fraction of full simulations that were *saved* by showing them equivalent to others and the fraction of error injections that *need full* simulation after applying GangES. In cases where equivalence was observed, we measure the number of instructions simulated until equalization.

3.2. Pure program analyses based techniques

To evaluate the accuracy of pure program analyses based techniques, we study metrics proposed by Pattabiraman et al. [20] and some derivatives. Specifically, we explore the following two metrics from [20] for a given static instruction, as an indicator of its vulnerability to producing SDCs. (1) *Fanout* is defined for a static instruction that writes to a register as the cumulative fanout of all the dynamic instances of the instruction. Fanout for a dynamic instruction that writes to a register *R* is defined as the number of dynamic uses of *R* before the next dynamic write to *R*. (2) *Av.lifetime* is defined for a static instruction that writes to a register as the average of the lifetimes of dynamic instances of the static instruction. Lifetime for a dynamic instruction I_d that writes to a register *R* is defined as the number of cycles from the execution of I_d to the last use of *R* before the next dynamic write to *R*.

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

We evaluate our five metrics using five of our applications – Blackscholes, Swaptions, FFT, LU, and Water. We collect the values of these metrics at the instruction level using Simics. We normalize all our metric values to one. Since lifetime and fanout for an instruction are derived from its destination register, we restricted our metrics evaluation to errors in destination registers.

For a given static instruction, we also obtain the number of SDCs it produces by employing Relyzer and use it as the golden metric (*sdc*), also normalized to one. As mentioned in Section 3.1, we limited Relyzer error simulation times to cover 95% of the errors. To ensure that the missing error information did not skew the metrics evaluation, we added more error simulations as follows. For each metric and application, we ensured we have Relyzer provided error outcome information for all error sites for the static instructions that cover at least the top 75% of the metric values (this increased the aggregate Relyzer coverage to over 97%). Any missing information, therefore, is from static instructions that have among the lowest metric values and the lowest dynamic instruction counts. This is unlikely to affect our results since these instructions are unlikely to contribute to a significant fraction of SDCs from the point of view of the metrics or Relyzer.

3.2.1. Metric evaluation: We use three different methods to evaluate our metrics. The first two methods quantify how accurately the individual metrics predict SDCs in isolation while the third evaluates combinations of our metrics.

Correlation coefficient: For each application, we measure correlation coefficients[5] between individual metrics and *sdc* (golden metric) to study the linear relationship between them.

Cost vs. SDC reduction: We note that the objective of estimating SDCs with metrics is to identify an optimal set of SDC-targeted error detectors. We therefore employ a 0/1 knapsack algorithm to find an optimal set of detectors that will provide the largest SDC reduction at a given cost – we assume duplication for detectors and charge one instruction as the cost of duplicating and comparing results for one instruction on average (similar to [11]). Thus, we obtain an SDC reduction vs. cost graph for each application using the known SDC count for each instruction from Relyzer. We call this the Relyzer curve (*RC*).

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

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

$$\begin{aligned} sdc_i = \beta_0 lifetime_i + \beta_1 fanout_i + \beta_2 av.lifetime_i + \\ \beta_3 av.fanout_i + \beta_4 dyn.inst_i + \varepsilon_i \end{aligned} \tag{1}$$

We also attempt to evaluate a nonlinear combination of our metrics. Since some nonlinear relationships between variables (or metrics) can be approximated using linear regression on polynomials,[6] we evaluated another linear regression to model the following:

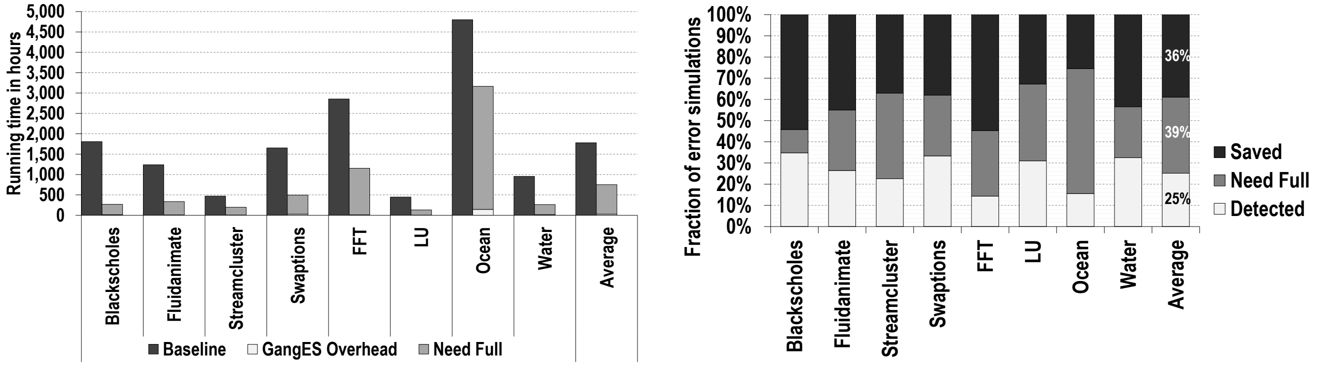
$$\begin{aligned} sdc_i = \beta_0 lifetime_i + \beta_1 (lifetime_i)^2 + \beta_3 (lifetime_i)^3 + \beta_4 fanout_i + \\ \beta_5 (fanout_i)^2 + \beta_6 (fanout_i)^3 + \beta_7 av.lifetime_i + \beta_8 (av.lifetime_i)^2 + \\ \beta_9 (av.lifetime_i)^3 + \beta_{10} av.fanout_i + \beta_{11} (av.fanout_i)^2 +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

4. Results for GangES

We evaluated a total of about 1.33 million application error sites identified by Relyzer. The error simulation experiments

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

[6] The more complex the nonlinearity, the higher the order of polynomials required.



(a)

(b)

Figure 5: Effectiveness of GangES in reducing the total wall clock time needed for error simulations and the number of full error simulations. For each application, the left bar in Figure (a) shows the total wall clock time needed for error simulations for all the (Relyzer-identified) input error sites (baseline). The right bar shows the time used by GangES to determine which errors need full simulation (*GangES overhead*) and the time needed to simulate the errors that need full simulations (*need full*). The bars in Figure (b) show the fraction of error simulations that GangES *saved* from full execution, that *need full* simulation, and that were identified as detections (*detected*).

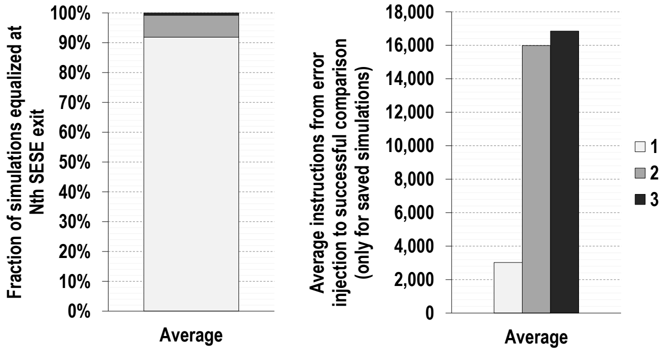
for these sites (to determine the Relyzer wall clock baseline) required approximately 14,225 hours of CPU time. Ganges seeks to reduce this time and the number of full simulations.

Figure 5a shows the effectiveness of GangES. For each application, the left bar shows the total wall clock time (in CPU hours) to identify the outcomes of all the input error sites by performing full error injection simulations on each such site as the *baseline*. The right bar shows the wall clock time to run GangES to identify the number of errors that need full simulations (*GangES overhead*) and to run such full simulations to completion (*need full*).

The figure shows that we obtain high simulation time savings of about 57%, averaged across our workloads. These savings translate to hundreds of CPU hours of simulation time savings for all our workloads (ranging from 272 CPU hours for Streamcluster to 1,700 CPU hours for FFT). This figure also shows that GangES consumes a small fraction of the total simulation time, $<$1.6% for our workloads. Specifically, the total wall clock time needed to determine which errors need full simulations (and which are saved) was only 225 hours for all our workloads together.

Figure 5b shows the fraction of the total error simulations that were *saved* from full execution, that *need full* simulation, and that result in a detection (these would be terminated early regardless of GangES). On average, approximately 36% of the total error simulations were saved; i.e., they were shown equivalent to another execution, saving the simulation time of running them to completion and comparing their output to the error-free output. Overall, 39% of the total input error set required full simulation.

Figure 6(a) shows when the equalization was performed for the *saved* simulations (from Figure 5b). It shows that on average about 92% of saved simulations were terminated at the first SESE region exit from the point of error injection. Approximately 7% and 1% of the saved simulations were equalized at the second and third SESE exits respectively.



(a)

(b)

Figure 6: When does simulation equalization occur? Part (a) shows the fraction of the *saved* simulations (from Figure 5b) that are equalized at the N^{th} SESE exit, averaged across our workloads. Part (b) shows the average distance of the successful comparison points (first, second, or third SESE exit) from the point of error injection in terms of number of dynamic instructions.

Figure 6(b) shows the average distance in the number of executed instructions from the point of error injection to the simulation state equalization (at a SESE exit). Specifically, it shows that the distance to first successful comparison (averaged across our applications) is approximately 3,000 instructions (where 92% of saved simulations are equalized). Distance to successful comparisons varied significantly with applications because the comparison points are identified according to the PST, which is application-specific. We do not show the application-specific breakdown here due to space limitations.

Figure 7 explains why the simulations categorized as *need*

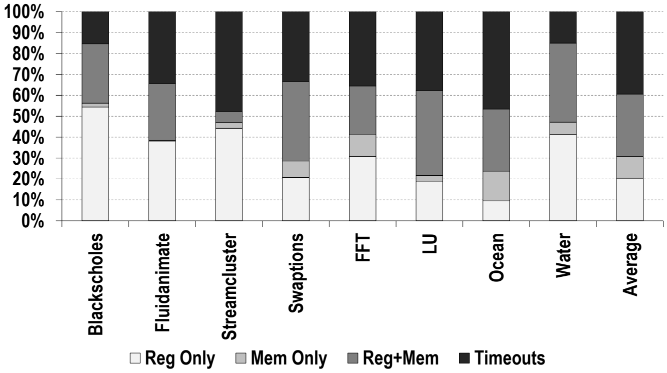


Figure 7: Categorizing the errors that need full simulations based on whether register state, memory state, or both mismatched during comparisons or whether no comparison was made before the timeout condition was met.

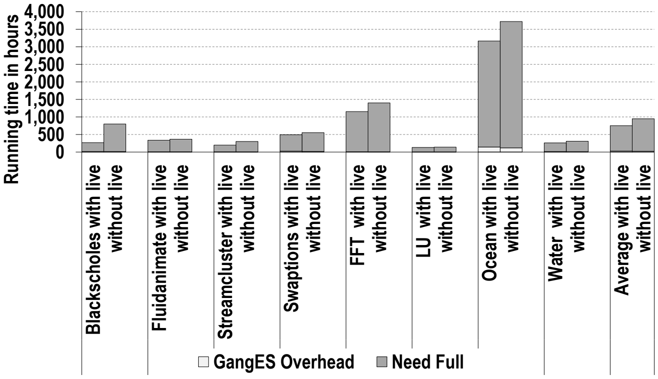


Figure 8: Comparing GangES wall clock time for comparing only live registers (left bar) vs. all registers (right bar) at SESE exits. Each bar shows the time for GangES overhead and for full error simulations.

full in Figure 5b were not equalized to other simulations. We categorize these simulations based on the four following criteria: (1) Register state mismatched at all exercised SESE exits (Reg Only). (2) Register state matched but memory state mismatched at all exercised SESE exits (Mem Only). (3) Combination of categories (1) and (2) occurred at different SESE exits; i.e., register state mismatched at some SESE exits and when it matched the memory state did not match (Reg+Mem). (4) No comparison was performed prior to the execution of a threshold number of instructions (Timeout). From Figure 5b, we observed that Ocean requires the highest number of full simulations, requiring 59% of all the simulations to be run until completion. From Figure 7, we note that over 46% of simulations that were marked as *need full* were never compared to any other simulation for Ocean. Hence, we need to identify more reachable comparison points and a compact way to store simulation state to allow more comparisons to increase the savings of GangES. Increasing our threshold for comparison should allow more comparison points and employing compression or encoding techniques should lower the simulation state storage overhead. These alternatives are a subject of our future research.

We also evaluated the benefit of comparing only the live registers vs. all registers at SESE exits by observing the impact on the wall clock time taken by GangES, shown in Figure 8. Recall that we fast forward 1000 instructions to obtain a conservative live processor register state at a comparison point (Figure 4). When we disallowed this step and compared all processor registers, the average wall clock time needed by GangES (including the overhead to determine which errors need full simulations and the full simulations themselves) increased by approximately 26% for our applications. This clearly shows that comparing live processor registers provides significant simulation savings over comparing all registers.

5. Results for pure program analyses based metrics

5.1. Correlation coefficients

Table 2 shows the correlation coefficients between *sdc* and individual metrics for all our metrics and applications. It shows that *av.lifetime* and *av.fanout* have virtually no correlation with *sdc* for our workloads. *Lifetime* displays weak to virtually no correlation and *fanout* exhibits moderate correlation for Blackscholes, FFT and LU. Although *dyn.inst* is the only metric that shows high correlation with *sdc* for a few applications (FFT and LU), there is no single metric that uniformly demonstrates a strong linear relationship with *sdc* for all our workloads. Furthermore, when correlation is calculated on the combined data points from all the benchmarks (represented by *All*), none of the metrics display a strong association with *sdc*.

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

Table 2: Correlation coefficients between *metrics* and *sdc* for different workloads.

5.2. Cost vs. SDC reduction

Here we compare optimal cost vs. SDC reduction curves for our metrics vs. Relyzer+GangES by plotting the *RC*, *PC*, and *AC* curves as described in Section 3.2. For brevity, we present the cost vs. SDC reduction curves for a representative subset

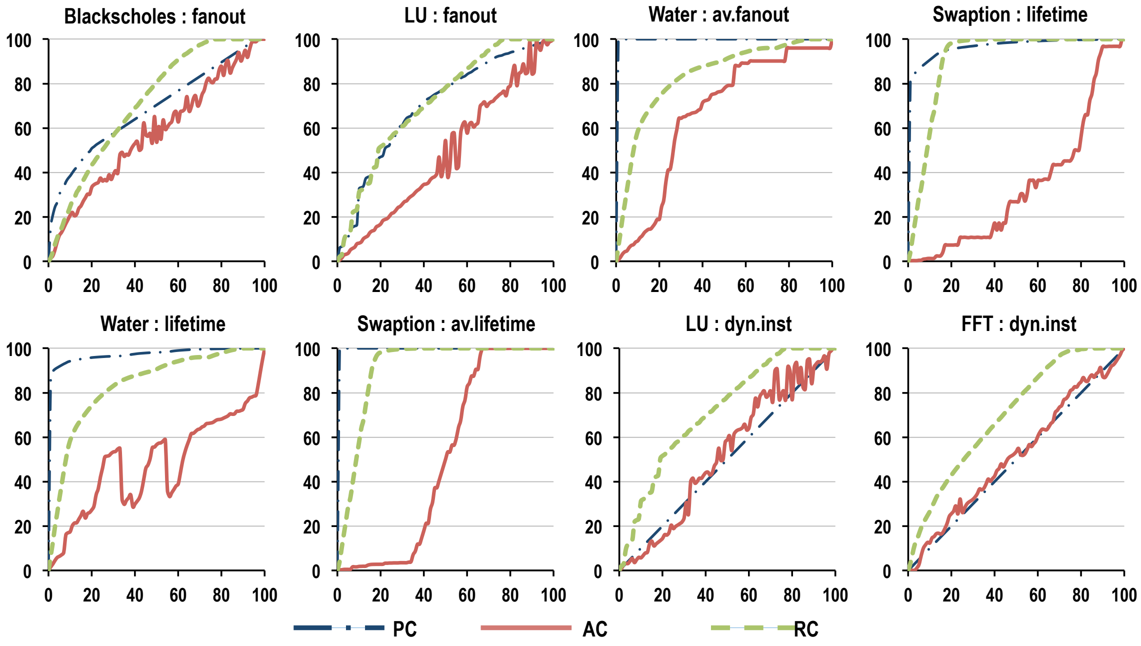


Figure 9: SDC reduction vs. execution overhead. The X axis plots *% execution overhead* **(in terms of increase in dynamic instructions) and the Y axis represents** *% reduction in SDCs*.

from our workload and metric combinations in Figure 9.[7]

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

Graphs for LU:*dyn.inst* and FFT:*dyn.inst* show a high correlation between PC and AC, which was expected based on Table 2. (The gap between the PC and AC curve – which shows the inaccuracy in the SDC coverage claimed by the corresponding metric – is lower for these graphs).

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

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

5.3. Combining multiple metrics

Table 3 shows the result of the linear regression (Equation 1) for our workloads. It shows the metrics that are significant[8] to the model and the model's adjusted R^2. The adjusted R^2 value estimates the percentage of variance in *sdc* that is explained by the metrics. If the adjusted R^2 is high then the derived model is considered robust. For example, 0.66 adjusted R^2 for LU implies that only 66% of the variance in *sdc* can be explained by the metrics, which leaves 34% as unexplained or caused by randomness. However, a low adjusted R^2 value can be interpreted either as *(a)* the model is missing key additional explanatory variables (other metrics), or *(b)* that a linear model is not sufficient to explain the relationship between the metrics and *sdc*. Overall, we make the following observations:

- No common model (formed by a linear combination of our metrics) that offers a best fit for all our workloads was identified. For different applications, different metrics were identified as being significant contributors. For metrics that prove to be significant for multiple applications, the respective regression coefficients (β_i) were different. For example, even though *fanout* is identified as a significant metric for Blackscholes, LU, and Water, the regression coefficients (β_1) were 0.60, -0.21, and -0.33 respectively.
- The adjusted R^2 values varied between 0.07 (for Swaptions) to 0.68 (for FFT) and were mostly lower than desired.
- The last three columns of Table 3 show the ratio of the Root Mean Square Error (RMSE) to the Mean for K-fold cross

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

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

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

Table 3: Linear regression summary. The significant metrics and the main number in each cell are the result of using linear regression based on Equation 1. The numbers in the square brackets are results using linear regression on polynomials based on Equation 2.

validations (CV_K)[9] with K = 10, 4 and 2. Even for models that have relatively high adjusted R^2 value (e.g., for LU or FFT), the cross validation showed high errors (values >1) in the predicted and observed SDCs. For example, for FFT, the average error for CV_4 is a very high 10.1 times the mean.

Our results from nonlinear regression (Equation 2 in Section 3.2) are presented in brackets in Table 3. They show a trend similar to that of linear regression – no common model for our studied workloads is identified. The adjusted R^2 has improved for all our workloads, which indicates that a nonlinear combination of the metrics can perform better than a linear combination in predicting SDCs. However, for several applications the adjusted R^2 value is still poor, indicating that other metrics and/or different nonlinear regression models are required. The last three columns show that the error from cross validation is also high. In the data sets for some of the applications, there are outliers that have significantly larger metric values than others. During *CV* when these outlier metric values are fed into the model, they produce large deviations in the output which result in higher RMSE. Regression on polynomials further exacerbates this problem as the model exponentially increases the error. For example, in FFT just one instruction accounts for approximately 96% of *av.fanout* of the entire application and removing it brings the *CV* error for polynomial regression down from approximately 10^7 to approximately 5. Although removing these outliers may improve the error rate, it also means that we are removing from our analysis instructions that the metrics identify as the most vulnerable. Since we are evaluating the predictive capacity of the metrics, we choose to not remove these instructions.

In conclusion, simple linear and nonlinear models using the pure program analyses based metrics we study do not uniformly explain or predict SDCs in our workloads.

6. Conclusions and future research

As technology scales, the hardware reliability challenge is expected to affect a broad spectrum of computing devices rendering traditional redundancy based solutions too expensive. Software anomaly based error detection has emerged as a low cost resiliency solution, but it suffers from silent data corruptions or SDCs. We therefore need mechanisms to accurately evaluate SDC rates of current resiliency solutions, and identify the remaining SDC producing program locations to develop cost effective software-centric resiliency solutions.

Relyzer evaluates an application's resiliency by analyzing its execution trace and carefully selecting a small set of error sites for thorough evaluation. This hybrid injection+analysis based approach is practical but still incurs significant run time (over 15,600 hours for our applications), most of which (about 90%) is spent in error simulations.

We have presented a new error evaluation framework called Gang Error Simulator or GangES that can significantly improve evaluation time of the hybrid injection+analysis approach. GangES observes that a set or gang of error simulations that produce the same intermediate execution sate after error injections would produce the same outcome. To check for similar intermediate execution state, GangES periodically compares states between multiple bundled simulations. Identifying *when* to compare executions and *what* state to compare can be challenging. GangES leverages the static structure of a program to identify when to compare simulations and compares limited live processor register state and touched memory addresses at these comparison points. Our results show that GangES saves 57% of the total error simulation time averaged across our applications (saving a total of about 8,200 CPU hours across all our workloads).

This paper also explores pure program analyses based techniques that are much faster and could potentially eliminate the need for tools such as GangES altogether. Relyzer+GangES allows us to systematically evaluate the effectiveness of previously proposed pure program analyses based metrics in predicting SDC. For the metrics we are able to study, our results show that these metrics and their various simple linear and non-linear combinations are unable to adequately predict an instruction's vulnerability to producing SDCs. While other program analysis based metrics may be more effective, our results provide evidence that developing such metrics is not straightforward and establish the value of Relyzer+GangES.

6.1. Limitations and future directions

We have evaluated GangES using an instruction-level transient error model. Prior work has performed resiliency evaluations using lower level error models [6, 15]. Extending our concepts

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

to such simulators is an interesting future direction. One of the main challenges in directly employing GangES on lower level error simulators (by collecting and comparing states at the architecture level) is handling latent errors – errors that are live at gate- or microarchitecture-level but not visible at architecture level – at comparison points.

A prior evaluation framework achieves speedup by repeatedly comparing the entire simulation state with the golden execution's state [8], terminating simulations early if the error is masked (not when the simulation becomes equivalent to another one). Our solution, on the other hand, compares all simulation states collected at a particular point in an application to find and terminate equivalent simulations. The solution in [8] uses CRC as a signature of the simulation state and compares just the CRC to limit the comparison overhead. Exploring such an approach for compacting the state we collect for comparisons (especially, touched memory addresses) is one of our future directions.

GangES performs state comparisons at program points that are identified using the modified PST to allow early terminations. This scheme, however, misses some opportunities to show equivalence (Figure 7). Prior research in software reliability has studied a program indexing scheme [30] mainly for software bug diagnosis. This scheme provides a mechanism to uniquely identify individual execution points so that the correlation between points in one execution can be inferred and correspondence between execution points across multiple executions can be established. This scheme can potentially provide GangES more comparison points; however, the overhead from more frequent state comparisons must be carefully traded off against the potential savings from early termination of simulations.

References

[1] *International Technology Roadmap for Semiconductors*, http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_Design.pdf, 2009.

[2] *Final Report of Inter-Agency Workshop on HPC Resilience at Extreme Scale*, http://institute.lanl.gov/resilience/docs/Inter-AgencyResilienceReport.pdf, 2012.

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

[4] 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.

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

[6] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proc. of International Design Automation Conference*, 2013, pp. 1–10.

[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] A. Evans, S.-J. Wen, and M. Nicolaidis, "Case Study of SEU Effects in a Network Processor," in *Proc. of IEEE Workshop on Silicon Errors in Logic - System Effects*, 2012.

[9] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[10] 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.

[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] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: computing control regions in linear time," *SIGPLAN Not.*, vol. 29, pp. 171–185, June 1994.

[15] M.-L. Li, P. Ramachandran, R. U. Karpuzcu, S. K. S. Hari, and S. V. Adve, "Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults," in *Proc. of International Symposium on High Performance Computer Architecture*, 2009.

[16] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[17] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *Proc. of International Symposium on Microarchitecture*, 2007.

[18] S. S. Muchnick, *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

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

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

[21] 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.

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

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

[24] 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.

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

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

[27] 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.

[28] D. L. Weaver and T. Germond, Eds., *The SPARC Arch. Manual*. Prentice Hall, 1994, version 9.

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

[30] B. Xin, W. N. Sumner, and X. Zhang, "Efficient program execution indexing," in *Proc. of International Conference on Programming Language Design and Implementation*, 2008, pp. 238–248.