Evaluating and Accelerating High-Fidelity Error Injection for HPC

Chun-Kai Chang[†] Sangkug Lym[†] Nicholas Kelly[†] Michael B. Sullivan[†‡] Mattan Erez[†]
[†]University of Texas at Austin [‡]NVIDIA Corporation
{chunkai, sklym, nick.kelly, mattan.erez}@utexas.edu misullivan@nvidia.com

***Abstract*—we address two important concerns for the analysis of the behavior of applications in the presence of hardware errors: (1) when is it important to model how hardware faults lead to erroneous values (instruction-level errors) with high fidelity, as opposed to using simple bit-flipping models, and (2) how to enable fast high-fidelity error injection campaigns, in particular when error detectors are employed. We present and verify a new nested Monte Carlo methodology for evaluating high-fidelity gate-level fault models and error-detector coverage, which is orders of magnitude faster than current approaches. We use that methodology to demonstrate that, without detectors, simple error models suffice for evaluating errors in 9 HPC benchmarks.**

I. Introduction

As high-performance computing (HPC) applications continue to scale, the likelihood of hardware faults silently corrupting results is increasing. Understanding such silent-data corruption (SDC) errors and mitigating them is therefore a critical research topic [1]. In this paper we address two important concerns in the analysis of the behavior of applications in the presence of hardware errors: (1) when is it important to model how hardware transient faults lead to instruction-level errors with high fidelity, as opposed to using simple bit-flip models, and (2) how to accelerate high-fidelity error injection campaigns, in particular when error detectors are employed.

There are three main reasons for why these concerns are important. Firstly, much prior work in the field of HPC has relied on simple bit-flip error models (e.g., [2]–[7]), despite the fact that it is already known such models do not match the error patterns of actual hardware faults [8]–[11]. Secondly, the overall methodology for evaluating the impact of hardware errors on applications is to use a Monte Carlo fault or error injection campaign, which requires massive resources for even toy scientific applications. Thirdly, as errors become more common, both hardware and software error detectors will likely become more common as well. Evaluating the coverage and impact of a good detector potentially requires orders of magnitude more Monte Carlo trials.

We present a new methodology and tool for the rapid evaluation of error detectors and high-fidelity instruction-level error models. Specifically, we adopt the hierarchical injection methodology from [12], which significantly accelerates the use of high-fidelity circuit logic-level fault injection, to the point that using the high-fidelity error model outperforms current state-of-the-art bit-flip error injectors. The main idea is that the majority of logic-level injection attempts (to either gates or latches) are *logically masked* by the circuit itself (e.g., a logical AND gate where the non-erroneous input is a 0). Completing the application for a Monte Carlo trial in such cases is wasteful. Instead, a new fault may be immediately injected once the masking is identified. In [12], logical masking is identified at the instruction level and re-injection occurs before the application proceeds such that no application run is "wasted" for evaluating logical masking. We extend this idea to also accelerate the evaluation of the impact of error detectors, where our method is $20-100$ times faster than the conventional approach. Furthermore, our acceleration methodology is orthogonal to acceleration methods that prune fault-injection paths that are unimportant [13]–[15].

Using our tool, we confirm prior studies showing statistically-significant differences in the outcomes when using high-fidelity circuit logic-level injection vs. simple bit-flip models. However, we develop new analyses and metrics for interpreting these statistical differences in the context of scientific applications. Surprisingly, *we find that there is no meaningful difference between modeling errors in arithmetic operations with high-fidelity logic-level injection or by randomly flipping a single bit in an instruction's output when hardware error detectors are not used.* We do this by evaluating the impact on execution efficiency when using end-to-end checkpoint-restart resilience and impact on accuracy when interpreting final results as part of an ensemble computation. We conclude that prior insights of work in the HPC community that uses the simple bit-flip model hold.

When detectors are introduced, however, using a high-fidelity error model has a much more substantial impact. We demonstrate that our tool and methodology allow for fast and high-fidelity evaluation of detectors and evaluate the impact of detectors on both the rate and effect of those errors that escape detection and silently corrupt data.

The contributions of this paper are summarized as follows:

- We propose and verify a nested Monte Carlo methodology that couples error injection and detection to reduce the overhead of high-fidelity injection and the evaluation of error detector coverage by $1-2$ orders of magnitude.

- We demonstrate an open-source tool[1] that requires no proprietary software to perform gate-level fault injection for the error-impact analysis of HPC applications at reasonable $3 - 10\times$ native slowdowns, which is faster than current state of the art injectors.
- We use this tool to conclude that without error detectors, the widely-adopted single bit-flip error model is representative of high-fidelity gate-level fault injection for evaluating HPC scientific applications; this is despite statistically-significant differences in the types of errors the two models generate.
- We show that the single bit-flip error model cannot be used for the evaluation of a specific error event if hardware-based error detectors are introduced.
- With further analysis, however, we conclude that estimating the impact of errors on the overall performance of a resilience scheme (specifically, checkpoint-restart of ensemble methods) can be done with a simple error model and any insights drawn are equivalent to those when using a high-fidelity gate-level fault model, whether hardware detectors are employed.

II. Background and Motivation

We define *faults* as physical events that affect hardware components. If a fault eventually changes the architectural state, it becomes an *error*. We focus on *soft/transient errors* in particular since they are random and transient in nature and thus hard to detect. In contrast, permanent faults that frequently lead to errors are typically detected and do not corrupt any results silently.

A. Fault Propagation and Masking

Soft errors that affect storage elements and combinational logic are triggered by random radiation events such as particle strikes. Only those strikes that are energetic enough can potentially lead to data corruption; otherwise, they are said to be electrically masked. When a strong particle strike hits a memory component (e.g., latches and flip-flops), the data is corrupted until new data is written into it. On the other hand, if a logic gate is hit, a transient pulse known as a *single-event transient* is created. The faulty signal (i.e., the single-event transient or the corrupted data signal from a memory component) might be masked logically before propagating to a following latch (e.g., it enters an AND gate with the other input being 0). If the faulty signal reaches the next latching window, erroneous data can be written into a latch. The propagation process continues and might eventually corrupt data in architectural components such as the register file. At this point, the fault manifests as an error.

We assume that faults escape electrical masking and timing masking in this paper; that is, we only consider particle strikes that carry sufficient charge and result in faulty signals that arrive on time at the next latch. However, we respect logical masking because it depends not only on the circuit but also on the application. In other words, *we are concerned with the impact of errors, but not the rate of errors*. Also, we focus on soft errors that affect execution units of the processor because errors occurring within those circuits directly affect application data and thus more likely lead to SDC. The same is assumed in previous work [7], [16], [17].

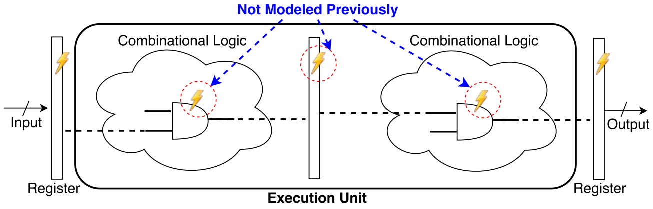

Fig. 1: Hardware faults that are not modeled by high-level error injection.

B. The Modeling Gap of Existing Error Models

Consider a fault that occurs at a random location within a circuit module consisting of an input buffer, combinational logic, internal pipeline buffers, and an output buffer (Fig. 1). Since we are interested in injecting errors, we assume the circuit has inputs and/or outputs associated with architectural state (e.g., an ALU or address generation unit).

Note that soft errors can be grouped based on their initial fault site within the circuit. First, consider the faults that occur at either the input buffer or the output buffer. When a fault happens at the output buffer, it directly manifests as an error. On the other hand, when a fault occurs in the input buffer, it can be masked by the operation (e.g., erroneous bits are shifted out). Such errors can be modeled by bit flips because they either directly affect the output or logical masking can be modeled by running the operation with erroneous inputs. These soft errors are already modeled in previous work via injecting errors into instruction operands [2]–[7].

Next, we consider the case where the fault site is at either a logic gate within combinational logic or internal pipeline buffers. Here we assume the fault induces a pulse that flips the output of the affected unit. Although this faulty signal may be masked logically before propagating to output buffer, it is possible that it leads to a soft error that corrupts multiple bits of the output buffer. Because the exact impact of the soft error on the output buffer depends on the initial fault site, the circuit, and the input data vector, there is no corresponding simple architecture-level modeling for soft errors originating from these internal circuit nodes. These errors are termed *hidden soft errors* in this work. To quantify the impact of this modeling gap, we perform gate-level fault injection to study characteristics of these hidden soft errors. This gate-level injection is also called *RTL injection* because the *register-transfer level* (RTL) description of a circuit specifies the circuit's gates and latches.

C. Characterization of Hidden Soft Errors

Fig. 2 shows the distribution of how many bits in the circuit output buffer are flipped if a fault from a circuit internal node

[1] Available at http://lph.ece.utexas.edu/users/hamartia/

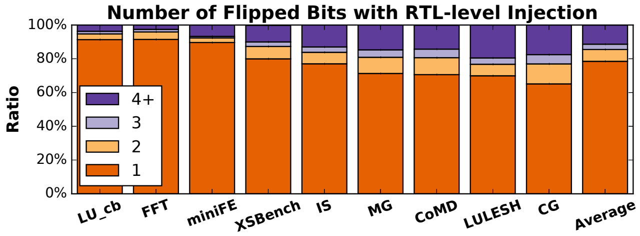

Fig. 2: Distribution of bit-flip count at circuit output with RTL-level particle-strike model.

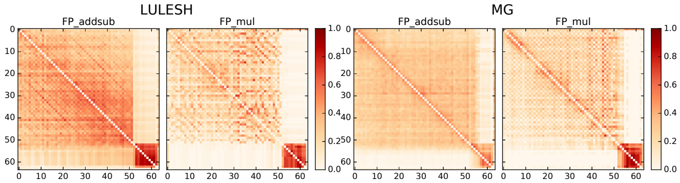

Fig. 3: Correlations between bit-flip positions with RTL-level injection. Breakdown into two applications (LULESH and MG) and two circuits (64-bit floating-point adder and multiplier). Axes denote bit locations of circuit outputs.

is not logically-masked. On average, 78% of errors manifest as single-bit flips. Hence, the single-bit flip model does not accurately reflect 22% of errors, and these errors potentially cause more severe data corruption. Note that the distribution varies across applications since logical masking depends on circuit input.

Modeling multi-bit errors is challenging because of correlations between bit-flip positions at a circuit's output (Fig. 3). First, correlations vary across circuits because the circuit's structure determines its logic operations, which in turn affect logical masking. Second, correlations are related to input data. For example, using input data from LULESH, the floating-point adder has strong correlations between bits in the exponent field (bit 52-62), while we do not observe such phenomena with input from MG. This is because logical masking depends on input data. Hence, not only do correlations vary across circuits, they also depend on input data and thus on applications. Such complex and data-dependent correlation is not modeled by existing random bit-flipping models.

Although we have shown that realistic error patterns are different from single-bit errors at the instruction level, the impact on applications is still unknown due to application-level error masking. Thus, we need to perform error injection to evaluate the end-to-end effects of hardware faults on applications.

D. Inefficiency of The Monte Carlo Method due to Fault Masking and Error Detection

The Monte Carlo method is widely adopted by existing error injectors to reduce evaluation time as a result of its high parallelism [2], [7]. The overall injection workflow usually consists of two phases: *profiling* and *injection*. The methodology assumes that soft errors affect each instruction with equal likelihood to model uniform-random particle strike time. The profiling phase is used for this purpose and derives the upper bound of dynamic instructions in the program. The injection phase uses the Monte Carlo method to evaluate the impact of errors at the application level. In each experiment, the injector chooses a random instruction instance (based on the profile), injects an error into the instruction, and then classifies the injection result.

However, in the presence of logical masking at the gate level, many injection experiments end up as masked faults that do not propagate to the instruction level and thus do not affect the application. The fraction of unnecessary runs is even worse when evaluating error detectors, because any detected errors can be immediately classified without waiting for the application to complete. The expected number of Monte Carlo experiments to obtain an unmasked fault (or an undetected error) increases with the fault-masking rate (or the detection rate), and it can be formulated as the mean value of a Geometric distribution with a $(1-masking\_rate)$ probability of success. This means when the masking rate is high, one needs a sufficiently large number of samples to generate an error sample given gate-level fault masking or to discover an undetected error given error detection.

E. Hierarchical Injection for High-Fidelity Error Modeling

One option for modeling errors with high fidelity is to perform each Monte Carlo experiment while simulating the entire hardware at the gate level. Such RTL-based error injectors offer high fidelity but are too slow to evaluate software resilience techniques in a timely manner (seven orders of magnitude slower than application-level injection [10]). To strike a balance between injection speed and fidelity, previous work proposes hierarchical injection where a faster injector invokes another detailed injector at the injection site (e.g., [9], [18]–[20]).

Despite offering orders-of-magnitude improvement over RTL-only injection, hierarchical injection is still too slow for evaluating the impact of errors on applications because of the inefficiency described in Section II-D. Techniques have been proposed to further accelerate injection campaigns by a form of importance sampling. Such techniques prune all possible fault/error injection sites, which may number in the millions, to a smaller list of perhaps thousands of important instructions [13] or hardware blocks [15] that dominate the likelihood of error occurrences. Still, such techniques do not fully account for data-dependent logical masking effects and do not address the magnified injection inefficiency with detectors. To accelerate evaluation in these scenarios, we introduce a novel error injection framework in the next section.

III. A Fast and Accurate Injection and Detection Framework

A. Coupling Error Injection with Detection for Saving Evaluation Time

We adopt a refined hierarchical injection methodology [12] and extend it to also be used for evaluating detectors. This methodology focuses on faults that directly affect architectural state (i.e., faults directly relating to instructions). The idea is to use a form of nested Monte Carlo methodology. The outer portion follows the traditional instruction-level error injection campaign described in Section II-D. However, within each Monte Carlo experiment a nested Monte Carlo is performed at the gate level. In each outer iteration, we first record the correct instruction (arithmetic circuit) output and then, a single random fault is repeatedly and immediately injected at the gate level until a fault manifests as an error that corrupts the output. By doing so, only an actual error that has not been logically masked is injected into the outer Monte Carlo trial, saving significant time by avoiding complete application runs when the outcome has already been determined. This injection flow is depicted in Fig. 4, which also shows how we extend the methodology to include detectors.

Evaluation using simple random sampling is even worse when error detectors are introduced. Numerous Monte Carlo runs and RTL gate-level fault injections are necessary to generate faults that truly affect the application because good error detectors have high coverage and detect most errors that are not logically masked. As a result, we apply the idea of iterative fault generation at the RTL level to filtering out detected errors. That is, errors that would be detected should also be pruned since the injection outcome is known at this point (*detected*). Thus, we should also keep injecting until an undetected error is generated to save time. Otherwise, similar to the cases of masked faults at the RTL level, it is wasteful to wait for the application running to other injection sites.

This filtering notion can be extended to resilience techniques at different abstraction layers by cascading error detectors starting from fine granularity to coarse granularity. For instance, only errors that are not detected by instruction-level hardware detectors are sent to fine-grained software state-recovery mechanisms [21]. Note that the longer the detector chain, the greater the savings that can be achieved. However, in this paper we only evaluate the impact of hardware instruction-level detectors.

An important aspect of this nested Monte Carlo methodology is that it requires multiple faults that propagate to actual errors to be identified in order to establish statistical bounds on the logical fault-masking rates for different instructions and applications. In that way one trial at the outer level is indeed equivalent to a flat methodology for the purpose of evaluating the logical masking rate and detector coverage. Note that one also has to set a limit on injection attempts to avoid practically infinite loop when the masking rate of the circuit is high. The selection of the limit can affect the tradeoff between evaluation time and accuracy. While the nested algorithm provides the same statistics as a traditional Monte Carlo, the specific instructions into which actual errors are injected will differ from a traditional injection campaign where each trial randomly selects an instruction. We verify that the nested methodology is fully equivalent to a traditional campaign despite this potential difference (Section V-A).

B. Pluggable RTL-level Injector and Detector

One of the main challenges of integrating an RTL gate-level injector with a higher-level injector is software compatibility across abstraction layers. To solve the compatibility problem, we design a generic error context API as an interface between abstraction layers. The error context API communicates essential information regarding a dynamic instruction instance. For instance, instruction operation type is used to select the target circuit for fault injection and instruction input operands are fed as input patterns to the circuit. Once a fault manifests as an error, the corrupted output is then sent back to the instruction level for modifying the target instruction instance.

To inject faults at the gate level, we insert an additional gate at each node of the circuit by modifying the RTL source code with Pyverilog [22]. For instance, to model a particle strike that flips a node's value, we augment the circuit with XOR gates, each of which has one input connecting to an existing node and the other input as a trigger signal. At runtime, we randomly trigger an XOR gate (by setting its trigger signal to 1) to emulate a fault. The real RTL gate-level simulation is done by Icarus Verilog, an open-source Verilog simulation tool [23]. We also support customized RTL-level error detectors to avoid the communication overhead of passing the error context between C++ (Pin) and Python (RTL simulation).

Importantly, our fault injection process does not depend on any proprietary tools and software. The entire RTL gate-level injector is wrapped into a set of Python modules and we have tested it with an open source RTL simulator.

C. Injecting Errors with Dynamic Binary Instrumentation

The hierarchical injection framework we use in this paper includes an assembly-level injector which drills down to the RTL gate-level at a specific instruction instance (see Fig. 4). Specifically, we implement the assembly-level injector using Pin, a dynamic binary instrumentation tool [24]. We select Pin (and in general dynamic binary instrumentation) for the following reasons: (a) it is much faster than microarchitecture-level and lower-level injectors since we can disable instrumentation after the injection point to run at native speed, (b) it is more accurate than compiler IR-level injectors [2], and (c) it allows injection into specific binary and source code regions. Using Pin, we can still map injected instructions back to the program source lines (i.e., directly pointing out which lines are problematic). Overall, dynamic binary instrumentation can inject errors at a lower level (i.e., higher-fidelity) with acceptable execution overhead (see Section V-B), at the same time providing application developers useful feedback regarding resilience from a higher-level point of view.

Our methodology and pluggable error models will work for other instruction-level injectors [16], [25]–[28] with the help of the error context API.

D. Limitations

Although our Pin-based implementation limits us to x86 platforms, the framework can be easily generalized to others.

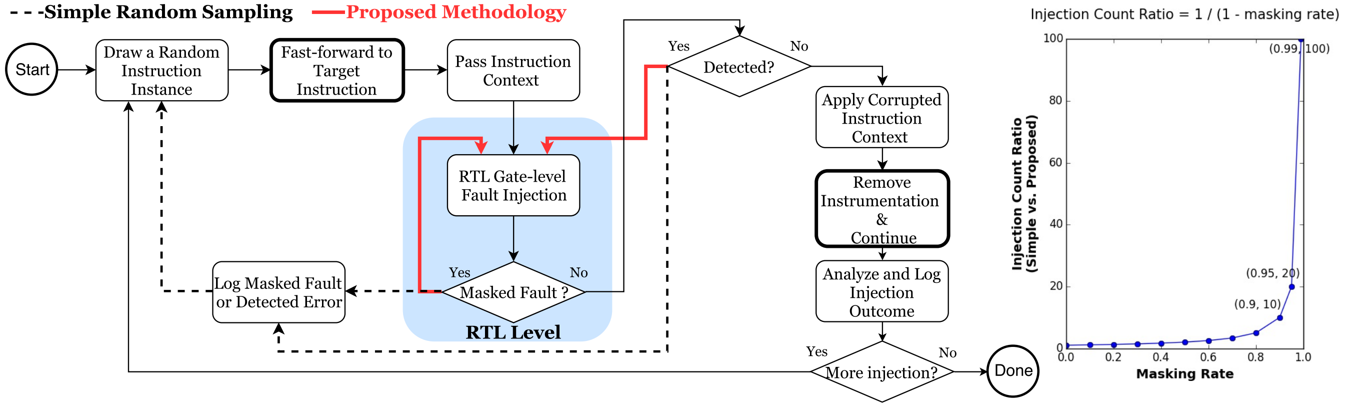
- - -Simple Random Sampling
Proposed Methodology
Start
Draw a Random Instruction Instance
Fast-forward to Target Instruction
Pass Instruction Context
RTL Gate-level Fault Injection
Masked Fault ?
Yes
No
RTL Level
Log Masked Fault or Detected Error
Detected?
Yes
No
Apply Corrupted Instruction Context
Remove Instrumentation & Continue
Analyze and Log Injection Outcome
More injection?
Yes
No
Done
Injection Count Ratio = 1 / (1 - masking rate)
Injection Count Ratio (Simple vs. Proposed)
Masking Rate
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100
(0.99, 100)
(0.95, 20)
(0.9, 10)

Fig. 4: (*Left*) Proposed error injection flow: refined hierarchical injector coupled with instruction-level error detection. Bold boxes denote time-consuming steps. (*Right*) Execution savings by nested Monte Carlo as a function of the masking rate (i.e., the probability that a fault (error) is masked (detected)).

We are not currently modeling microarchitecture-level error masking (e.g., masking due to speculative execution in modern processors) in our injector because the simulation is too slow for HPC applications. Faults that potentially affect multiple instructions are not modeled as well at this time. This paper focuses only on arithmetic errors and we leave high-fidelity models for memory errors as future work.

IV. Evaluation Methodology

Our evaluation consists of six main parts: (a) verifying the proposed nested Monte Carlo methodology, (b) evaluating the cost of our tool for running an injection campaign and the benefits of nested injection, (c) evaluating the impact of the error model on the reliability of applications, (d) evaluating the impact of error detection on the reliability and error types of applications, (e) evaluating the impact of the error model on application output quality in the context of HPC scientific applications, and (f) evaluating the impact of the error model on the overhead of an overall resilience scheme (specifically, checkpoint-restart).

Before discussing the evaluation results, we first describe which error models we are evaluating, how we evaluate the outcome of each injection trial, and which applications and quality metrics we use.

A. Detector Models

To study the impact of hardware detectors on application resilience, we choose arithmetic residue checkers. They are shown to provide high coverage for errors from execution units with relatively low cost [29], [30], and are also adopted by commodity processors (e.g., POWER6 [31]) and previous work on hardware resilience design [32], [33].

Residue checkers detect errors by comparing output of the arithmetic unit with that of a relatively low-cost datapath. The checking can be described by Equation 1 where $\oplus$ denotes integer addition, subtraction, or multiplication and $|x|_m$ denotes $x \bmod m$.

$$|a \oplus b|_m \overset{?}{=} |(|a|_m \oplus |b|_m)|_m \qquad (1)$$

We assume that no errors affect the final equality checker, so these residue checkers have perfect coverage for single-bit datapath errors. Specifically, we implement a residue checker with modulus 3 and another with two moduli, 3 and 5. The latter provides higher coverage because an error is not detected only when both checks fail.

Note that we only evaluate integer addition, subtraction, and multiplication operations with residue checker protection.

B. Error Models

We assume on-chip SRAM and system DRAM are protected by ECC and only inject errors to instructions using arithmetic and logic units. In each experiment, we inject an error into a random instruction's output operand with one of the four error models below. By injecting into the output register we model errors in the arithmetic units rather than register file errors; register file errors require a memory error model. The four error models are:

- **Single-bit flip (RB1)**: randomly flips a single bit, as commonly done in prior work in the HPC community.
- **Double-bit flip (RB2)**: randomly flips two bits.
- **Random (RND)**: replaces the output with a random value, possibly mimicking worst-case errors.
- **RTL gate-level model (RTL)**: generates an error pattern using the methodology described in Section II-E. We use *RTL-G* and *RTL-L* to denote injection into gates only and latches only, respectively; latch injection is only done for pipelined floating-point units.
- **model+**: uses *model* on a design with a single-modulus residue checker where *model* is one of the error model described above.
- **model++**: resembles **model+** but with a double-modulus residue checker (i.e., stronger detection).

For the RTL model, we synthesize gate-level netlists of integer and floating-point execution units using Synopsys tools (Design Compiler and DesignWare Library) with the 45nm Nangate Open Cell Library [34], optimized for performance

TABLE I: Circuits used in the RTL error model, their pipeline stages, and the fault-masking probabilities (average of all applications studied in this work) of injecting a logic gate and a latch, respectively.

Name	Stages	Gate Masking Rate	Latch Masking Rate
INT_add_sub	1	0.17±0.02	n/a
INT_mult	1	0.09±0.05	n/a
Shift	1	0.30±0.08	n/a
FP_add_sub	3	0.28±0.03	0.27±0.04
FP_mult	3	0.41±0.02	0.33±0.02
FP_div	10	0.49±0.09	0.54±0.12
FP_sqrt[2]	1	0.46±0.10	n/a

(as also done in [12]). Because the DesignWare Library does not include pipelined floating-point units, we use the register retiming feature of Design Compiler to pipeline the circuits. The pipeline stages are tuned to mimic those used by Intel Broadwell processors[2] based on the latency data from [35]. Tab. I lists the circuits used by the applications studied in this paper. Note that the synthesized circuits can be different from those designed and optimized for commodity processors, but we have shown that they lead to errors different from single-bit errors at the instruction level.

C. Outcome Classification

As in prior work, injection outcomes are classified into these primary categories:

- **Masked**: the injected error is masked by application, with output identical to the error-free run.
- **Detected Uncorrectable Error (DUE)**: errors that crash or hang the program are categorized as DUE_{crsh}. Errors that result in obviously erroneous application output (e.g., output is not finite or mismatch in matrix size) are also in this category and denoted as DUE_{test}.
- **Silent Data Corruption (SDC)**: the program ends normally with output errors that are hard to detect.

D. Experimental Settings and Output Quality

We evaluate the serial version of 9 HPC benchmark programs and applications (Tab. II).[3] For each application, we perform 3000 injection experiments, which ensures a margin of error <2% for a confidence level of 95% [36], on 10 combinations of error models and detector models. Therefore, 270,000 experiments in total are conducted. Note that this work focuses on whether high-fidelity errors would significantly affect application resilience and we do not evaluate parallel programs at this time; however, our injector and methodology can be used for MPI programs and runs on the Lonestar5 supercomputer at TACC.

For the output quality, we adopt the same metric shipped with the application or suggested by previous work [37], [38].

V. Experimental Results

A. Verification of the Nested Monte Carlo Methodology

To verify the statistical equivalence between the nested Monte Carlo we develop and a traditional fault-injection campaign, we compare the outcome distributions of each methodology. For brevity, we do not include figures for this verification experiment and simply note that any differences between the two methodologies, for all experiments we conducted with RTL injections are within the 95% confidence intervals of each experiment. In other words, the specific numbers obtained are not identical, but the 95% confidence intervals of the two methodologies overlap. We note that the nested approach has narrower (better) confidence intervals for a given number of trials because each outer Monte Carlo trial identifies multiple non-masked errors in the inner stage. Evaluation of output quality shows similar statistical equivalence.

Notice that there is a pitfall when comparing the RTL error injection results of our methodology with simple random sampling: *the proportion of each instruction type among the error samples is different between the two methods because each circuit has different logical masking rate*. Our methodology collects error samples following the instruction mix of the application since we keep fault injection until an error is generated for each experiment, while simple random sampling collects more samples from circuits with lower masking rate since the experiments end as soon as the injected fault is masked. Thus, it is necessary to normalize the proportion of each instruction type among the error samples before comparing the injection results. Such normalization is performed in the aforementioned validation campaign.

Conclusion 1: *the nested Monte Carlo approach is equivalent to traditional injection campaigns, despite significant potential benefits in execution time.*

B. Injector Overhead

Tab. II compares the execution time overhead of using our injector with a simple bit-flipping error model, RTL injection, and RTL injection with detectors, to the time each application runs natively. Recall that we disable instrumentation after the injection point, so the injector overhead depends on the injection point (i.e., a dynamic instruction). We therefore measure the injector overhead at the median instance of all dynamic instructions, at which we go through the entire injection phase but did not apply the corrupted value at the instruction level to avoid error affecting measurement. Evaluation is performed on a machine with an Intel i5-6500 CPU and 16GB DRAM.

We make three interesting observations. First, very short-running applications that require a fraction of a second to run natively incur a significant relative overhead for injection because starting up the Pin injector and invoking the error model require a fairly fixed, but comparatively high, amount of time. Second, longer running applications incur a reasonable injection overhead with a slowdown of $3-5\times$ without detectors and $5-10\times$ with detectors, even when RTL injection is used. This is because the relative time to bring up the injection

[2] The RTL simulation hangs with a 15-stage square-root unit (as used by Broadwell), so we use the unpipelined DesignWare version instead.

[3] We use a more resilient IS in which we insert an assertion at the end of `randlc()` to check if the output falls within $[0,1]$.

TABLE II: Benchmark, input, injection overhead per experiment, and output quality related information.

Program	Input	Native Time	Simple Time	RTL Time	RTL+ Time	RTL++ Time	Quality Metric	DUE_{test} Criteria
FFT [39]	-m 16	0.01s	1.3s	1.7s	5.2s	19.1s	Rel-L2-Norm	Infinite values
miniFE [40]	nx=18 ny=16 nz=16	0.04s	2.1s	6.0s	8.1s	12.1s	Resid-Norm	Resid-Norm > 1
LU_cb [39]	default	0.04s	1.2s	1.8s	4.8s	11.2s	MaxAbsDiff	Infinite values
IS [41] [3]	A	1.2s	3.0s	4.6s	7.4s	21.3s	n/a	Failed verification
CG [41]	A	1.2s	7.1s	7.8s	8.8s	10.9s	Zeta	Infinite values
MG [41]	A	1.5s	11.9s	15.0s	18.0s	51.7s	L2-Norm	L2-Norm > 1
CoMD [42]	default	5.7s	28.5s	29.1s	32.2s	52.4s	Potential energy	Lost atoms, infinite energy, or potential energy > 0
LULESH [43]	default	22.1s	103.7s	109.7s	112.3s	121.9s	Measure of symmetry (MaxAbsDiff)	[37]
XSBench [44]	small	30.9s	91.7s	92.5s	93.4s	96.3s	n/a	n/a

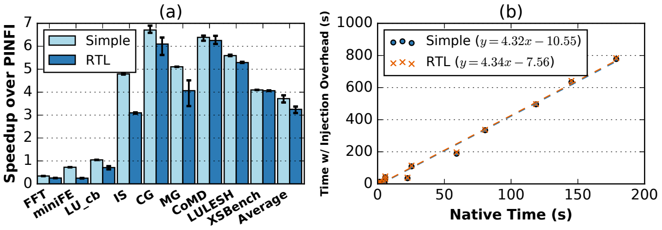

Fig. 5: (a) Speedup over PINFI with simple error models (RB1, RB2, and RND) and our RTL gate-level model. Error bars denote maximum and minimum speedups. (b) Scaling of per-experiment execution time as native time increases.

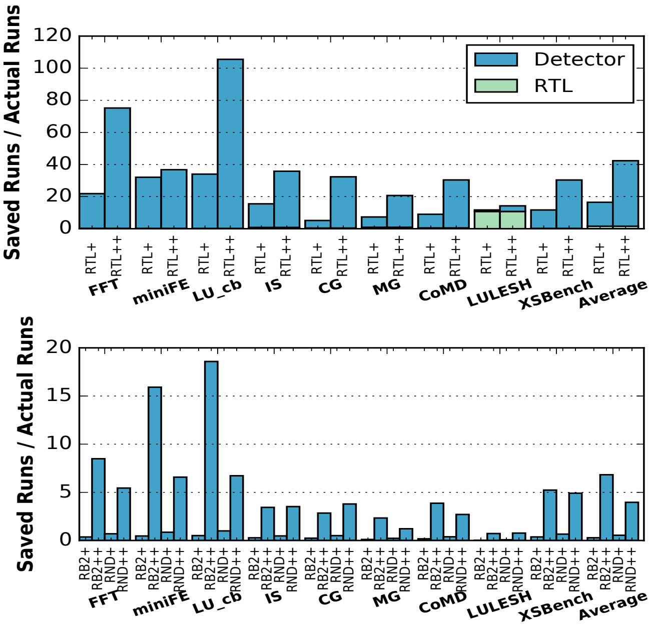

Fig. 6: Execution savings (relative to actual runs, 3000 in this work) by nested Monte Carlo for RTL and injection-detection coupling. Top: RTL error model with residue checkers. Bottom: Simple error models with residue checkers.

infrastructure is small relative to the execution time of these applications. Third, the overhead of RTL injection and detector evaluation varies between applications because the masking factor and detector coverage is data dependent—the higher the masking or coverage, the more iterations are required within the inner Monte Carlo step.

Fig. 5 (a) shows that the overhead of our injector, even with RTL injection, is actually lower than PINFI [7], which is one of the fastest injectors currently available. We compare to PINFI because recent work has shown that it has lower overhead than compiler-based injectors [2]. Our implementation outperforms PINFI even with RTL-level injection, except for applications that can finish in under two seconds. The performance loss is due to the additional features for injecting specific binaries and instructions, but the overhead is amortized for larger applications. On average, our error injector is faster than PINFI by a factor of 3.

Since full-scale HPC applications have much longer execution time, we therefore show the tool's execution time vs. problem size in Fig. 5 (b). We increase the input size of LULESH to mimic native execution time of larger HPC applications. Note that the overhead increases linearly due to instrumentation overhead of the Pin tool. When the problem size is large enough, the difference of overhead between simple models and the RTL model is negligible (4.32X and 4.34X slowdown for simple and RTL model, respectively). The overhead can be further reduced by taking checkpoints during the profiling phase and starting each injection experiment at the checkpoint closest to the target injection point as in [45].

Next, we measure the execution savings of the error injection experiments for each application. The savings are presented as the ratio of runs saved due to nested Monte Carlo: $\sum_{n=1}^{N} iter_n \times dyninst_n / total\_dyninst$, where N is the number of injection experiments, $iter_n$ is the number of local iteration for the nth experiment, $dyninst_n$ is the dynamic instance number of the nth experiment, and $total\_dyninst$ is the total dynamic instructions of the application. We assume the overhead of actual error injection at the injection site is small relative to the overhead of pre-injection and post-injection period, which we verify true for all applications that are not very short. Fig. 6 shows that the overall savings are higher when the RTL model is used since we keep injecting faults until one manifests as an error. Also the savings increase with the strength of the detector used. Note that LU_cb has the highest savings due to most of the instructions can be protected by the residue checkers (Tab. III). In contrast, LULESH has a small portion of instructions protected by residue checker but its savings at the RTL level is significant because of the input data result in high logical masking.

Conclusion 2: *with the nested approach, the overhead of injection, even when using detailed error models and high-coverage detectors, can be kept low and roughly match that of current injectors with simple error models.*

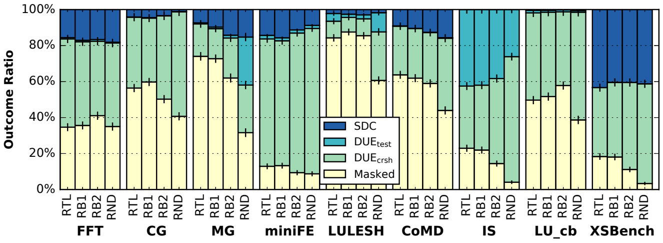

Fig. 7: Injection outcome distribution. Error bars are 95% confidence intervals.

C. Reliability Outcome Distribution

We now evaluate the impact of the error model on expected outcomes of applications that experience an error.

1) RTL vs. Simple Error Models: Fig. 7 shows the outcome distribution of all combinations of error models without detectors. The most surprising result here is the small difference between RTL and RB1 for all applications. While the small confidence intervals demonstrate that the outcome distribution is sometimes statistically different, the absolute difference is very small. The largest difference of SDC rates between the two models is only 4% (XSBench), and that of DUE rates is also only 4% (CG). This shows that even though the two models are quite different at the instruction level (Section II-C), the impact on application is likely unimportant due to high proportion of single-bit errors, correlated error patterns, and application-level error masking. However, neither RB2 nor RND can closely approximate the RTL model because of aggressive (uncorrelated) bit-flips in these two models. See Appendix for sensitivity study of input sizes and circuit implementations.

2) Gate Soft Error vs. Latch Soft Error: Next, we study the impact of gate soft errors (i.e., a logic gate output is flipped) vs. latch soft errors (i.e., the output of a latch (or flip-flop) is flipped). We only report results of six applications here because compiled binaries for the others do not use pipelined circuits. The top of Fig. 8 compares the distribution of bit-flip count at the instruction level between gate and latch soft errors. One can see that latch soft errors have more multi-bit patterns than gate soft errors. However, we observe similar final impact on application resilience for both error types (bottom of Fig. 8) due to application-level error masking.

Conclusion 3: *RB1 is a good approximation of RTL injection, whether experiencing gate or latch faults.*

D. Impact of Hardware Residue Checkers on Application Reliability Outcome

The impact of residue checkers on final outcome distribution is shown in Fig. 9. We only show the distribution of protected instructions (i.e., integer add/sub/mul instructions) so as to decouple protection ratio from the result. To account for the fact that each instruction is injected multiple times until an undetected error is generated, we assume a detected error triggers an exception that leads to DUE_{crsh} and thus compute the distribution as following. First, we construct four empty bins corresponding to each outcome category. Next, for each of the 3000 injection outcomes, it contributes c_e to the DUE_{crsh} bin and $(1 - c_e)$ to the outcome category of the undetected error where c_e is the averaged detector coverage of error model e in Fig. A.2 (see Appendix). In the end, the outcome distribution and confidence intervals are computed with data in the four bins.

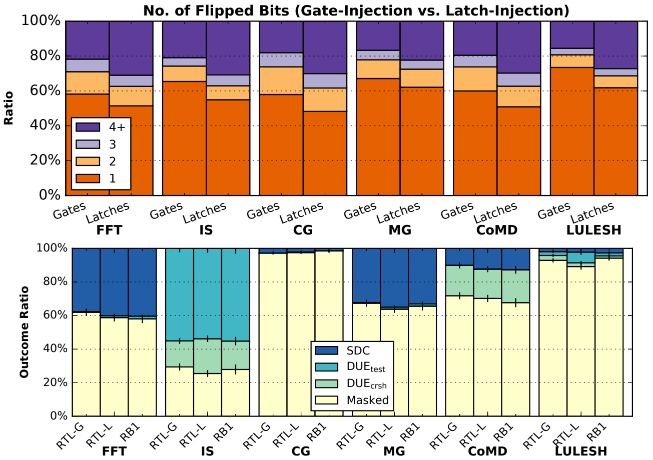

Fig. 8: Comparison of bit-flip count distribution at the instruction level (top) and injection outcome distribution (bottom) between gate soft error and latch soft error. Only floating-point instructions are injected here because they are pipelined (i.e., having both gates and latches).

Since residue checkers have perfect coverage for RB1 and good coverage for the RTL model, we observe higher improvement in SDC rate compared with RB2 and RND. However, with the RTL model, the additional improvement in SDC rate by adding another modulus to the residue checker is marginal because coverage of single-modulus residue checker is already high (except for XSBench).

It is important to note that SDC improvement due to residue checkers on *all* instructions is highly dependent on the protection ratio (Tab. III). For example, although SDC rate is improved by nearly 9% for protected instructions in MG, the actual SDC rate improvement is only 2% as only 25% of instructions are protected. The injection outcome results can guide the adoption of software error detectors and how they should be tuned. When the hardware detector can detect most critical errors such that the resultant SDC rate already meets the resilience target, software detector is not necessary and thus performance would not be degraded. On the other hand, the software detector should be tuned to target those errors left by hardware detectors to minimize impact on performance.

Conclusion 4: *When detectors are introduced, the impact of the error model is large when the fraction of instructions protected is also large.*

E. Impact of Error Models and Detector Models on Application Output Quality

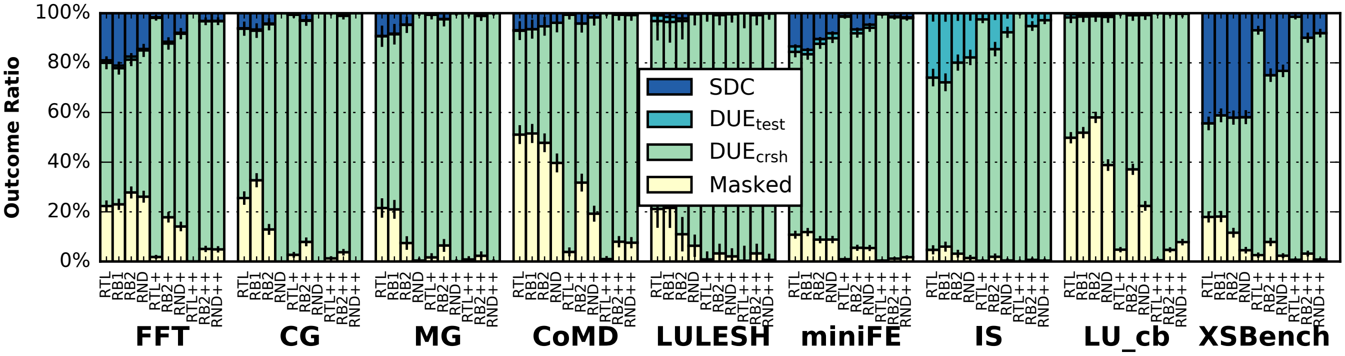

Fig. 9: Injection outcome distribution for instructions protected by residue checker (i.e., integer addition, subtraction, and multiplication operations). See Section IV-B for the definitions of error models and detector models.

TABLE III: Ratio of dynamic arithmetic and logic instructions protected by residue checkers.

Program	Ratio	Program	Ratio	Program	Ratio
FFT	0.78	miniFE	0.88	LU_cb	0.99
IS	0.38	CG	0.59	MG	0.25
CoMD	0.35	LULESH	0.09	XSBench	0.65

1) Perturbed Application Output Quality due to SDC: Previous work demonstrates that reliability and output quality may be traded off (e.g., [38], [46]). With a bounded degradation of quality, the cost of protecting the application against soft errors can be reduced. We thus compare the application output quality of SDC cases between error models. We use two methods to objectively compare output quality degradation distributions from different error models. First, we compare the quality metric of each SDC sample to the golden value to identify the most significant decimal position of the outcome error. We construct a histogram of these positions for each application and model. We then compare the histograms between each pairs of error models using chi-squared test, which is widely used to test the similarity of one set of binned data against another [47].

Secondly, we treat the perturbed quality of each error model as a continuous random variable. Thus, for each error model, the perturbed quality can be uniquely described as a cumulative distribution function (CDF). To compare similarity between two CDFs of continuous data, we perform Kolmogorov-Smirnov test.

For both hypothesis tests, our null hypothesis H_0 is no difference in output quality between a pair of error models; the alternative hypothesis H_1 is there is difference in output quality between error models. Thus, if the null hypothesis is rejected, it indicates output quality differs significantly between the pair of error models. We choose the significance level (α) to be 0.05. As a result, if the calculated p-value from a test is less than 0.05, the observed data rejects the null hypothesis. The results of both tests are shown in Tab. IV.

Both tests show that there is no significant difference in output quality between RB1 and RTL when there are no hardware residue checkers. However, neither RB2 nor RND results in similar output quality as RTL. When residue checkers exist, the perturbed output quality of RTL is statistically different from bit-flipping models for applications in which the detectors protect a reasonable fraction of instructions. This is because multi-bit errors generated with RTL model may not be detected by residue checkers and thus affect the output quality.

TABLE IV: p-values of Chi-squared and Kolmogorov-Smirnov tests for application output quality between error models. Bold fonts represent output qualities are significantly different.

Program	p-values (Chi-squared / Kolmogorov-Smirnov)				
	RTL vs. RB1	RTL vs. RB2	RTL vs. RND	RTL+ vs. RB1+	RTL++ vs. RB1++
FFT	0.87 / 0.36	**0.00 / 0.00**	**0.00 / 0.00**	**0.00 / 0.00**	**0.00 / 0.00**
miniFE	0.77 / 0.99	**0.03 / 0.01**	0.06 / 0.09	**0.00 / 0.00**	**0.00 / 0.00**
CG	0.05 / 0.66	0.74 / 0.22	**0.01** / 0.18	**0.00** / 0.16	0.21 / 0.50
MG	0.76 / 0.32	0.23 / 0.51	0.10 / 0.60	0.95 / 0.76	0.74 / 0.89
CoMD	0.74 / 0.69	0.38 / 0.49	**0.00** / 0.45	0.15 / **0.04**	0.91 / 0.33
LULESH	0.67 / 0.86	**0.01** / 0.07	0.05 / **0.00**	0.24 / 0.51	0.69 / 0.78

2) Effective Application Output Quality: So far we only show how error models and detector models affect application output quality *given an error always results in SDC*. Nonetheless, the effective output quality actually depends on SDC rate and the coverage of the detector (if any). The effective output quality can be computed as

$$(1-c)*((1-P_{SDC})*Mean_{errFree}+P_{SDC}*Mean_{SDC}) \\ +c*(Mean_{errFree}) \tag{2}$$

where c is detector coverage, P_{SDC} the SDC rate of the error model, $Mean_{errFree}$ the expected output without error, and $Mean_{SDC}$ the mean perturbed output due to SDC. Here detected errors contribute $Mean_{errFree}$ because they can be corrected either by checkpoint-restart, or by restarting the whole application from the begining.

Using this metric, we find that: (1) both RB1 and RTL lead to very similar effective output quality because their SDC rates and $Mean_{SDC}$ are similar, (2) residue checkers fail to improve effective quality of `miniFE` and `LULESH` because

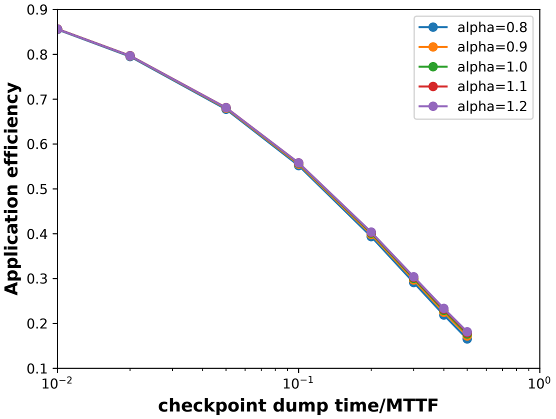

Fig. 10: Application efficiency due to checkpointing overhead vs. the ratio of dump time to MTTF where *alpha* is the ratio of estimated MTTF to true MTTF. This implies that slight estimation error of DUE rates has negligible impact on achieving minimal checkpointing overhead.

the $Mean_{SDC}$ is significantly different from $Mean_{errFree}$, and (3) the impact of error models on effective quality is negligible for the applications whose effective quality is improved by the residue checkers (e.g., RND++ and RTL++ have similar effective quality for FFT, CG, MG, and CoMD).

Conclusion 5: *While the specific output quality degradation of a specific run with detectors requires a high-fidelity error model, the overall impact on ensemble methods can be reasonably estimated with RB1.*

F. Impact of Error Models on Resilience Overhead

Recall that the DUE rates obtained with RB1 and RTL are slightly different (max difference is 4% for CG in Fig. 7). With the DUE rate from error injection results, one can tune the checkpointing interval for minimal overhead incurred by checkpointing and rollback. To quantify the impact of tuning the checkpointing interval based on RB1 instead of the RTL model, we compute the application efficiency (i.e., the fraction of time for real computation as opposed to checkpointing) using the higher-order formula in [48], [49]. Note that the 4% difference of DUE rates for CG (39.4% vs. 35.5%) leads to around 10% estimation error of the mean time to failure (MTTF). However, such difference has negligible impact on application efficiency (Fig. 10). This is also true when we consider the residue checkers studied in this work. The maximal difference of the DUE rates between RB1 and RTL with detectors is XSBench (100% for RB1+ vs. 90.5% for RTL+), which also leads to 10% estimation error for MTTF. Thus, tuning the workload-specific checkpointing interval through single-bit error injection is good enough.

Conclusion 6: *RB1 is sufficient for evaluating the overall performance efficiency impact of errors (on checkpoint-restart) whether detectors are used or not.*

VI. Related Work

Recent error injectors targeting HPC are based on compiler-level [2], [3], assembly-level [6], [7], emulator-based [4], [27], and debugger-based [5] injection. This work achieves fast injection by coupling an assembly-level error injector with an RTL-level fault injector because the former supports injecting HPC applications while the latter improves error fidelity. Moreover, given the high coverage of error detectors, we further apply the nested Monte Carlo idea introduced in [12] to filtering detected errors, saving evaluation time by orders of magnitude.

Combining resilience techniques across multiple abstraction layers to achieve minimal cost has also been an active research topic [50]–[52]. Orthogonal to prior work, this paper demonstrates how to take advantage of the interaction between error injection and detection to reduce the evaluation time of cross-layer resilience techniques.

Prior work also studies acceleration of error injection experiments [13]–[15], [45], [53]. Our methodology is orthogonal to acceleration methods that prune faults potentially leading to the same injection outcome [13]–[15].

A previous resilience study of the POWER6 processor at IBM [31] finds that when running *random instruction mixes*, injecting single bit-flips to latches at the RTL level results in an outcome distribution close to beam experiments. However, soft errors resulting from combinational logic were not considered in that work. These errors are a growing concern at newer technology nodes [54]–[56]. We find that even if these errors are considered, single-bit errors remain a good approximation of realistic errors, though only if hardware detectors are not used.

Recently, Sangchoolie et al. [57] find that single bit-flips result in SDC rates reasonably close to *data-independent* multi-bit flips in many cases. In this work, we further observe that this is also true for a more realistic *data-dependent* RTL error model, meaning that conclusion drawn by prior work using single-bit errors should still hold even if realistic errors are injected.

VII. Conclusion

We show how the nested Monte Carlo method saves evaluation time for error injection campaign. Since the benefit increases with the aggregate detection rate of the resilience technique under test, such methodology can significantly accelerate designs of cost-effective resilience techniques that span across abstraction layers. Additionally, to minimize performance overhead, existing and future software resilience techniques should take into account the effect of hardware detectors. Most importantly, by demonstrating the outcome distribution, application output quality, and the impact on checkpointing overhead, we conclude that, although different at the instruction level, single-bit errors remain a good approximation of realistic soft errors from arithmetic and logic circuits when hardware detectors are not considered. If hardware detectors are known to exist, a more realistic error model should be used to evaluate and tune the software resilience techniques.

Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0014097 and DE-SC0014098, program manager Lucy Nowell. The authors thank the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for supporting the simulation environment.

References

[1] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[2] G. Georgakoudis, I. Laguna, D. Nikolopoulos, and M. Schulz, "Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed," in *Proceedings of SC17, Denver, CO, USA, November 1217, 2017*, IEEE.

[3] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 72, ACM, 2015.

[4] Q. Guan, N. BeBardeleben, P. Wu, S. Eidenbenz, S. Blanchard, L. Monroe, E. Baseman, and L. Tan, "Design, use and evaluation of p-fsefi: A parallel soft error fault injection framework for emulating soft errors in parallel applications," in *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, pp. 9–17, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

[5] D. Oliveira, V. Frattin, P. Navaux, I. Koren, and P. Rech, "Carol-fi: an efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators," in *Proceedings of the Computing Frontiers Conference*, pp. 295–298, ACM, 2017.

[6] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 57, IEEE Computer Society Press, 2012.

[7] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pp. 375–382, IEEE, 2014.

[8] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, "An experimental study of soft errors in microprocessors," *IEEE micro*, vol. 25, no. 6, pp. 30–39, 2005.

[9] M.-L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 105–116, IEEE, 2009.

[10] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference*, p. 101, ACM, 2013.

[11] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller, "Understanding soft error resiliency of blue gene/q compute chip through hardware proton irradiation and software fault injection," in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pp. 587–596, IEEE, 2014.

[12] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Hamartia: A fast and accurate error injection framework," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2018.

[13] 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 *ACM SIGPLAN Notices*, vol. 47, pp. 123–134, ACM, 2012.

[14] S. K. S. Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "Ganges: Gang error simulation for hardware resiliency evaluation," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pp. 61–72, IEEE, 2014.

[15] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 241–254, ACM, 2017.

[16] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, pp. 41–50, IEEE, 2013.

[17] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pp. 1–12, IEEE, 2012.

[18] H. Cha, E. M. Rudnick, J. H. Patel, R. K. Iyer, and G. S. Choi, "A gate-level simulation environment for alpha-particle-induced transient faults," *IEEE Transactions on Computers*, vol. 45, no. 11, pp. 1248–1256, 1996.

[19] Z. Kalbarczyk, R. K. Iyer, G. L. Ries, J. U. Patel, M. S. Lee, and Y. Xiao, "Hierarchical simulation approach to accurate fault modeling for system dependability evaluation," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 619–632, 1999.

[20] S. Mirkhani, M. Lavasani, and Z. Navabi, "Hierarchical fault simulation using behavioral and gate level hardware models," in *Test Symposium, 2002.(ATS'02). Proceedings of the 11th Asian*, pp. 374–379, IEEE, 2002.

[21] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *the Proceedings of SC12*, (Salt Lake City, UT), pp. 58:1–11, November 2012.

[22] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *International Symposium on Applied Reconfigurable Computing*, pp. 451–460, Springer, 2015.

[23] S. Williams, "Icarus verilog," 2006. http://iverilog.icarus.com/.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, pp. 190–200, ACM, 2005.

[25] A. Thomas and K. Pattabiraman, "Llfi: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

[26] S. K. Vishal Chandra Sharma, Ganesh Gopalakrishnan, "Towards reseiliency evaluation of vector programs," in *21st IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, 2016.

[27] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1245–1254, IEEE, 2014.

[28] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, pp. 249–258, IEEE, 2017.

[29] A. Pan, J. W. Tschanz, and S. Kundu, "A low cost scheme for reducing silent data corruption in large arithmetic circuits," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*, pp. 343–351, IEEE, 2008.

[30] K. A. Campbell, P. Vissa, D. Z. Pan, and D. Chen, "High-level synthesis of error detecting cores through low-cost modulo-3 shadow datapaths," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 161, ACM, 2015.

[31] P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones, "Soft-error resilience of the ibm power6 processor," *IBM Journal of Research and Development*, vol. 52, no. 3, pp. 275–284, 2008.

[32] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 210–222, IEEE, 2007.

[33] J. Carretero, P. Chaparro, X. Vera, J. Abella, and A. González, "End-to-end register data-flow continuous self-test," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 105–115, ACM, 2009.

[34] J. Knudsen, "Nangate 45nm open cell library," *CDNLive, EMEA*, 2008.

[35] A. Fog, "Optimization manual 4 instruction tables," *Copenhagen University College of Engineering, Software Optimization Resources, http://www.agner.org/optimize,(1996-2017)*, pp. 215–230.

[36] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 502–506, European Design and Automation Association, 2009.

[37] B. Fang, P. Wu, Q. Guan, N. DeBardeleben, L. Monroe, S. Blanchard, Z. Chen, K. Pattabiraman, and M. Ripeanu, "Sdc is in the eye of the beholder: A survey and preliminary study," in *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pp. 72–76, IEEE, 2016.

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

[39] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pp. 24–36, IEEE, 1995.

[40] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[41] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, *et al.*, "The nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[42] ExMatEx, "Comd: Classical molecular dynamics proxy application." https://github.com/ECP-copa/CoMD.

[43] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[44] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, (Kyoto).

[45] A. Chatzidimitriou and D. Gizopoulos, "Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy," in *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pp. 69–78, IEEE, 2016.

[46] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 319–330, IEEE, 2014.

[47] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C*, vol. 2. Cambridge university press Cambridge, 1996.

[48] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future generation computer systems*, vol. 22, no. 3, pp. 303–312, 2006.

[49] W. M. Jones, J. T. Daly, and N. DeBardeleben, "Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 276–279, ACM, 2010.

[50] K. Swaminathan, N. Chandramoorthy, C.-Y. Cher, R. Bertran, A. Buyuktosunoglu, and P. Bose, "Bravo: Balanced reliability-aware voltage optimization," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 97–108, IEEE, 2017.

[51] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, *et al.*, "Clear: Cross-layer exploration for architecting resilience: Combining hardware and software techniques to tolerate soft errors in processor cores," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[52] V. B. Kleeberger, C. Gimmler-Dumont, C. Weis, A. Herkersdorf, D. Mueller-Gritschneder, S. R. Nassif, U. Schlichtmann, and N. Wehn, "A cross-layer technology-based study of how memory errors impact system resilience," *IEEE Micro*, vol. 33, no. 4, pp. 46–55, 2013.

[53] R. Balasubramanian and K. Sankaralingam, "Understanding the impact of gate-level physical reliability effects on whole program execution," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 60–71, IEEE, 2014.

[54] N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Q. Shi, R. Allmon, and A. Bramnik, "Soft error susceptibilities of 22 nm tri-gate devices," *IEEE Transactions on Nuclear Science*, vol. 59, no. 6, pp. 2666–2673, 2012.

[55] B. Gill, N. Seifert, and V. Zia, "Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node," in *Reliability Physics Symposium, 2009 IEEE International*, pp. 199–205, IEEE, 2009.

[56] N. Mahatme, N. Gaspard, T. Assis, S. Jagannathan, I. Chatterjee, T. Loveless, B. Bhuva, L. W. Massengill, S. Wen, and R. Wong, "Impact of technology scaling on the combinational logic soft error rate," in *Reliability Physics Symposium, 2014 IEEE International*, pp. 5F–2, IEEE, 2014.

[57] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pp. 97–108, IEEE, 2017.

[58] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, pp. 18–27, July 2011.

Appendix

A. *Reliability Outcome Distribution with Larger Inputs and Different Circuit Implementations*

The conclusion that RB1 is a good approximation of high-fidelity models should still hold for larger benchmarks and different circuit implementations. To further support this, we increase the input size of `MG` and `CG` from class A to class B and perform error injection with the same RTL model. Plus, we construct another RTL model (called *RTL-F*) with a different circuit implementation from the FloPoCo circuit generator [58]. Circuits are generated with the default unpipelined implementation and synthesized with the same cell library as the original RTL model. Fig. A.1 shows that the conclusion holds even with larger input sizes and another high-fidelity RTL model.

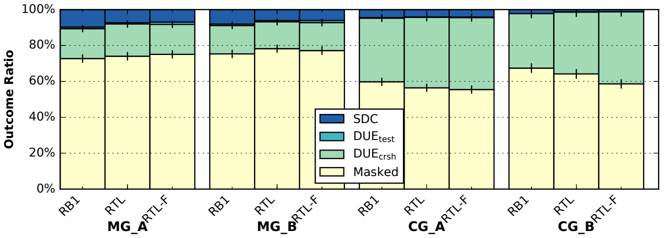

Fig. A.1: Injection outcome distribution of CG and MG with different inputs. Error bars are 95% confidence intervals.

B. *Characteristics of Hardware Residue Checkers*

1) Detector Coverage: Fig. A.2 shows the coverages (i.e., error detection rates) of the checkers with different error models applied. First, coverages are application-dependent when RTL error model is used. The residue checker with a single modulus can detect 88% of errors on average, while the coverage increases to 97% with two moduli. Second, when RB2 is used, the coverage of the single-modulo checker is slightly application-independent, while that of the double-modulus checker has larger variation across applications. Finally, when RND is used, coverages of both detectors are nearly the same across applications but significantly lower than RTL model.

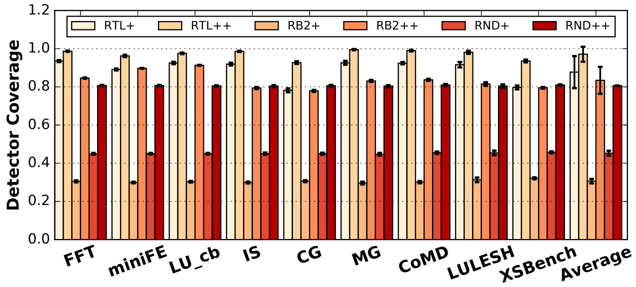

Fig. A.2: Coverage of residue checkers with different error models. Coverage for RB1 (not shown) is always 100%. Error bars are 95% confidence intervals.

2) Profile of Undetected High-Fidelity Errors: Recall that residue checkers can detect all single-bit errors. Fig. A.3 shows what errors are not detected by residue checkers when the RTL error model is used. We observe that: (1) the profile of undetected errors in terms of bit-flip counts is application-dependent; in comparison (not shown), RB2 always has 2 erroneous bits and RND nearly always exhibits 4+ bit errors, and (2) two-bit errors still account for a portion of errors undetected by the single-modulus checker (20% in most case), while few errors undetected by the double-modulus checker are two-bit.

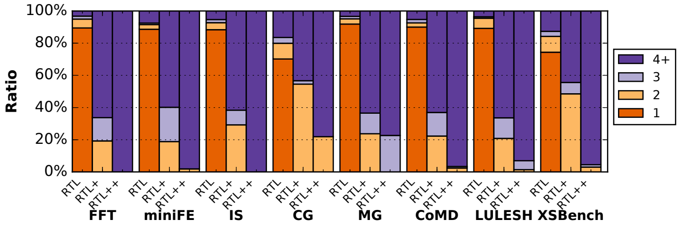

Fig. A.3: Distribution of bit-flip count of RTL errors undetected by residue checkers.