Assessing The Impact of Timing Errors on HPC Applications

Chun-Kai Chang
University of Texas at Austin
chunkai@utexas.edu

Wenqi Yin
University of Texas at Austin
wqyin@utexas.edu

Mattan Erez
University of Texas at Austin
mattan.erez@utexas.edu

ABSTRACT

Timing errors are a growing concern for system resilience as technology continues to scale. It is problematic to use low-fidelity errors such as single-bit flips to model realistic timing errors. We address the lack of holistic methodology and tool for evaluating resilience of applications against timing errors. The proposed technique is able to rapidly inject high-fidelity and configurable timing errors to applications at the instruction level. Our implementation has no runtime dependencies on proprietary tools, enabling full parallelism of error injection campaign. Furthermore, because an injection point may not generate an actual error for a particular application run, we propose an acceleration technique to maximize the likelihood of generating errors that contribute to the overall campaign with speedup up to 7X. With our tool, we show that realistic timing errors lead to distinct error profiles from those of radiation-induced errors at both the instruction level and the application level.

ACM Reference Format:
Chun-Kai Chang, Wenqi Yin, and Mattan Erez. 2019. Assessing The Impact of Timing Errors on HPC Applications. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356184

1 INTRODUCTION

As semiconductor technology scaling slows, timing margins are likely to shrink to improve performance and power. As a result, large-scale systems may become increasingly susceptible to timing errors resulting from variations in supply voltage and temperature. Such errors can cause insidious application failures known as silent-data corruption (SDC). In this paper we evaluate the impact of timing errors on a set of scientific benchmarks. We develop a new, fast, high-fidelity timing-injection framework that uses just-in-time circuit-level analysis to account for both low-level functional-unit design details and the workload. To the best of our knowledge, our framework is the first that can be used to dynamically inject high-fidelity timing errors into running applications to account for the values used and their error-masking effects.

We are also the first to evaluate timing-errors specifically for scientific applications. This is in contrast to prior work in this field, which has either focused on particle-strike induced errors or used generic low-fidelity synthetic error models, such as single-bit flips (e.g., [1–6]). Previous work on timing errors has focused on developing statistical models for errors [7–9] rather than on incorporating injection in a way that fully accounts for both the circuit and its dynamic application-based inputs. We evaluate the impact of timing errors on functional units and find that their impact is quite different from that of particle-strike errors. Functional-unit timing errors can only occur in bit positions that change value during the execution of an operation. As a result, timing errors nearly always affect lower-significance bits, while particle-strike errors tend to be distributed more uniformly across bit positions. The implications are that timing errors nearly never lead to detected errors and are either masked or lead to an SDC failure.

Unlike previous work that characterizes timing errors to form abstract statistical error models, our work takes a direct approach in which timing errors are generated on demand at runtime based on the circuit state and its workload. We adopt the idea of hierarchical injection [6, 10, 11] where an instruction-level error injector invokes a circuit-level analysis routine just in time to generate an error at the injection point. In this way, the generated errors are high-fidelity, workload-dependent, and adaptive to dynamic behavior and different circuit implementations. At the same time, the whole error injection campaign is fast because the instruction-level injector has low overhead and the entire application often has to be executed to completion to determine the impact of each injected error. Our approach applies further acceleration by maximizing the likelihood that each of these expensive application execution trials yields an outcome that contributes to the overall campaign; because timing errors are particularly input-dependent, it is quite possible that an injection point simply does not generate an actual error for a particular application run. We call this technique Injection-Point Overprovisioning because it allows us to correctly maintain Monte Carlo statistical analysis while continuing to use an application run even if its initial injection point fails to generate an error.

Our approach of just-in-time timing-error injection is unique and challenging. Like prior work, we focus on injecting and evaluating the impact of errors, rather than on estimating the rate at which such errors occur. Prior approaches pursued a statistical modeling approach because simulating timing errors dynamically either requires the use of proprietary and expensive commercial circuit-design software or extremely-slow low-level circuit simulation. Injecting particle-strike errors is simpler and faster because it only requires simulating logic gates, rather than circuits. To address

SC '19, November 17–22, 2019, Denver, CO, USA

this challenge, we develop a new methodology and tool for high-fidelity just-in-time error generation and injection at the instruction level.

We rely on recently-developed static circuit timing analyses that are fast and are available as open source [12]. We adapt these analysis routines to evaluate dynamic timing for specific inputs to the functional unit and the magnitude of the *voltage droop* that causes a timing violation. We borrow the method of Cherupalli and Sartori [13]. We first use gate-level simulation to identify propagation paths through the circuit and then dynamically remove all paths that do not toggle before evaluating the new input-specific circuit timing constraints. We can then determine if an error is possible and which output bits are erroneous. The model parameters are dynamically configurable and include changing the circuit, operating conditions, and voltage variation signal profile. Because we inject at the instruction level, our model also accounts for timing violations that impact multiple consecutive instructions.

To the best of our knowledge, this is the first tool that is capable of evaluating the end-to-end effects of high-fidelity, yet configurable timing errors on applications. At the same time, the overhead of injection is low (6X on average relative to a native application run). The framework is compatible with existing instruction-level error injectors, and we rely only on free open-source components. We can thus release our tool and share it as open source as well. [1]

The contributions of this paper are summarized as follows:

- We develop the first fast, open-source, high-fidelity, instruction-level timing-error injection model and tool. It can be used with any injection framework (we use Hamartia [6]) and applications it supports (x86 binaries) and can be configured with different functional-unit implementations and operating conditions.
- We propose Injection-Point Overprovisioning to speed up an error-injection campaign. Depending on the likelihood that a given voltage variation leads to an error, Injection-Point Overprovisioning speeds up injection by up to 7X.
- We characterize the impact of timing errors on HPC applications at both the instruction level and the application level. We find that the impact of timing errors on application outcomes differs from the impact of particle-strike errors. At the same time, we show that two well-known error models (i.e., single-bit flips and previous values) fail to represent timing errors.

2 BACKGROUND

We define *faults* as physical events that affect hardware components. If a fault eventually changes the architectural state, it becomes an *error*.

2.1 Timing Errors of Digital Circuits

Timing faults result from variations in supply voltage, temperature, or device characteristics that change circuit timing.

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

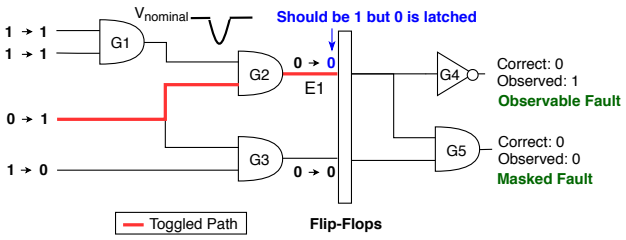

Figure 1: An example timing error caused by a voltage droop.

As a result, some inputs may take too long to propagate through the circuit and lead to an error. In a sequential circuit, a timing fault manifests as a timing error when at least one flip-flop latches an incorrect value and it eventually perturbs the architectural state. This process is complicated because it depends on the variation magnitude and duration, the circuit, its operating condition (including clock frequency, voltage, temperature, etc.), and the sequence of input values to the circuit; both current inputs and previous inputs are important. *In this work, we specifically focus on timing errors as a result of voltage droops.*

Although the circuit structure determines the delay of each path, input values control which paths are toggled (or *sensitized*). For example, in Figure 1, given the input pair, it is the red path that determines the timing of the endpoint E1, while the other paths are *false paths* because their timing is irrelevant given the inputs. Only variations that are strong enough can significantly delay the toggled paths such that old values or glitches are latched by flip-flops at the circuit endpoints. However, old values are not always different from the correct values. As a result, whether an instruction is affected by a timing fault depends on the input history of the circuit as well (i.e., circuit inputs at previous cycles).

In addition to the aforementioned *temporal masking*, for pipelined circuits, whether faults become errors also depends on *logical masking*. For instance, in Figure 1, even though the flip-flop of E1 fails to latch the new value produced by gate G2, the fault only affects the output of G4 but not G5 in the subsequent stage. This is because the other input to G5 is a controlling value (in this case, 0). Thus, one cannot faithfully model timing errors in pipelined circuits without modeling logical masking as well.

Moreover, timing errors may affect multiple instructions and corrupt multiple bits in output operands. For instance, consider variations that persist across multiple cycles or those that impact multiple pipeline stages simultaneously. Note that this is different from radiation-induced errors, which usually affect a single instruction. We refer readers to prior work [14, 15] for more details about timing errors.

```
// binary snippet
fpadd r1, r2, r3
  // r3 = r1 + r2 (previous inst)
fpadd r4, r5, r6
  // r6 = r4 + r5 (victim inst)
```

```
// error config
voltage: "0.85V"
frequency: "400MHz"
duration: "single_cycle"
occupancy: "low"
```

Figure 2: Example binary snippet and error configuration.

2.2 Impact of Timing Errors on HPC Systems

Modern systems apply substantial voltage and/or frequency guardbands in hardware to ensure that timing errors never occur, even at the worst operating condition. However, guardbands waste energy and performance because typical use cases do not lead to timing errors. Prior work has shown that such guardbands can account for 18% of total node power for IBM POWER7 [16] and ARMv8 systems [17].

Researchers have been advocating new cross-layer techniques to overcome the power constraints of future HPC systems without expensive guardbanding [15, 18, 19]. Nonetheless, due to dynamic variations (e.g., voltage droops and temperature fluctuations) and interference of supply voltage in many-core processors [15, 20, 21], the likelihood that timing errors occur in HPC systems may become non-negligible. Emerging technologies such as near-threshold voltage computing are even more susceptible to timing errors [22]. Thus, a methodology to evaluate the resilience of applications against timing errors is highly desirable.

3 TIMING ERROR INJECTION

We adopt the concept of hierarchical injection [6, 10, 11] where a faster injector invokes a detailed model at the injection point to accelerate injection and still generate high-fidelity errors. In this work, we integrate an instruction-level injector with a gate-level timing fault injector. At the injection point, the instruction-level injector provides the gate-level timing fault injector with: (1) the instruction type (used to determine which circuit to simulate) and (2) a pair of input vectors to the circuit for the previous and current cycle. A user-provided error configuration supplies the voltage droop profile (i.e., magnitude and duration). The gate-level timing error injector then returns a potentially corrupted output to the instruction-level injector. Figure 3 shows the overall flow of our timing error injector. We use the example in Figure 2 to explain the workflow of the injector.

3.1 The Instruction-Level Injector

The instruction-level error injector is based on the injector from Hamartia, an open-source error injection and detection suite [6]. It uses dynamic binary instrumentation to modify the architectural state of the instrumented application. However, the models we develop do not depend on Hamartia and are portable and open source; they should work with any instruction-level injector, including compiler-level ones.

As mentioned in Section 2.1, the manifestation of a timing error depends on the circuit's computation history. As a result, the role of the instruction-level injector is to collect the data necessary for reproducing circuit state. We refer to instruction instances affected by a timing fault as *victim instructions*. Specifically, the injector logs: (1) *input pairs*, which are input operands of victim instructions and those of the previous instructions that utilized the same circuit and (2) instruction types of those instructions (e.g., ADD, MUL, etc.). For the example in Figure 2, the input pair is `((r1,r2), (r4,r5))` and the instruction type is floating-point addition. These data along with the error configuration are passed to the gate-level injector.

3.2 The Gate-Level Injector

This injector produces the potentially corrupted output operand for each victim instruction. The core consists of a fault driver and multiple timing fault injectors. Each fault injector models temporal masking in one pipeline stage. The fault driver prepares input for the fault injectors which are chained to model logical masking across stages.

3.2.1 The Fault Driver. Based on the instruction type, it first looks up the circuit database to determine the circuit used by the victim instruction. If the circuit's pipeline depth is D, then D fault injectors are chained together. Next, the driver prepares input for each fault injector. It decides the operating condition of the circuit based on the error configuration. It also generates fault-enable signals to control whether a stage should be injected or not. These fault-enable signals are determined based on the fault duration and pipeline occupancy (explained in Section 5.1.1). Once all input to fault injectors are ready, the driver launches simulation of the first stage and waits for the result of the last stage.

For the example in Figure 2, the victim instruction uses the floating-point adder, which has three pipeline stages in our evaluation (see Table 1). Therefore, three fault injectors are chained together. Since the user specifies that the droop decreases the voltage to 0.85V, the driver fetches the characterization corner for that voltage level from the cell library database. Assuming the victim instruction is in the third stage when the single-cycle droop occurs, only the fault-enable signal to the third stage should be set (i.e., the results of first two stages are error-free). Finally, the input pair, `((r1,r2), (r4,r5))`, is sent to the first stage and the simulation begins. The output of the first stage becomes the input to the second stage, and so on. In the end, the fault driver sends the potentially corrupted output of the last stage back to the instruction-level injector, which modifies the value of `r6` accordingly.

3.2.2 Modeling Temporal Masking. Figure 4 shows how we model temporal masking within each pipeline stage. To this end, we need to know which endpoints (i.e., input of flip-flops or circuit's output pin) experience a timing violation

given the input pair to the circuit and its operating condition. We perform dynamic timing analysis to determine which endpoints encounter a timing violation using the method proposed by Cherupalli and Sartori [13]. The idea is to remove false paths before running static timing analysis (STA) such that only paths with gates toggled in the specific cycle determine signal arrival times at the endpoints (e.g., the red path in Figure 1). We derive the false paths by parsing the value change dump (VCD) generated by gate-level simulation. Although the original evaluation in [13] uses a proprietary STA tool, we use OpenTimer, an open-source STA tool [12]. For gate-level simulation, we use Icarus Verilog, an open-source Verilog simulator [23].

Once we know which output pins encounter a timing violation, the error-free output of the victim instruction, and error-free output of the previous instruction that used the same circuit, we can then derive the potentially corrupted output operand for the victim instruction (the timing fault generator box in Figure 4). For example, if the error-free output of the victim instruction is 0010, the output of previous instruction is 1111, and all bits except the least-significant bit encounter a timing violation, then the corrupted output operand is 1110.

3.2.3 Modeling Logical Masking. We model logical masking by chaining fault injectors. If a fault is injected at one stage, it may be logically masked by one of its following stages. We preprocess each pipelined circuit by splitting it into individual stages using Pyverilog, a hardware design processing toolkit for Verilog HDL [24]. We verify the resultant circuits to ensure that the circuit transformation does not break functionality. This is a one-time procedure and it is completely transparent to the user. At runtime, the fault driver provides the circuit of each pipeline stage to each corresponding fault injector.

3.3 Limitations

We assume that the magnitude of variations is constant within each cycle. Also, we assume that the execution order of instructions follows the program order. This is not always true for processors with out-of-order execution, which require detailed modeling of their micro-architecture for full fidelity. For such a study, our gate-level injector can be integrated with injectors at the micro-architectural level [25, 26]. We do not model metastability because the estimated mean time between metastability (using the models in [27]) is larger than 10^{40} years even for the worst operating condition evaluated in this work.[2] Although our evaluation focuses on timing errors as a result of voltage droops, the tool can be used to evaluate timing faults due to overclocking or temperature fluctuations as well.

[2]If the user specifies very small cycle time or very strong droop, the tool reports the mean time between metastability based on existing models and warns the user if the mean time between metastability is lower than 10^5 years (i.e., metastability rate is higher than 0.1 FIT).

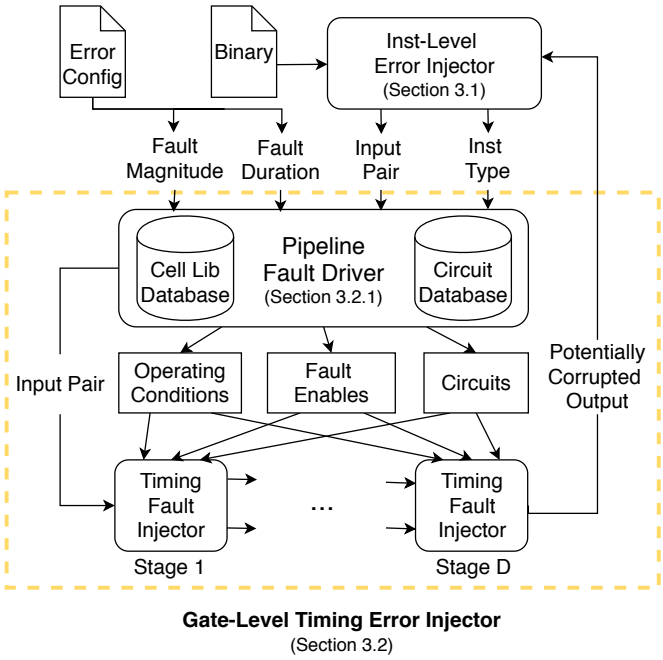

Figure 3: Instruction-level timing error injector for a pipeline of depth D. Each timing fault injector is the per-stage injector in Figure 4.

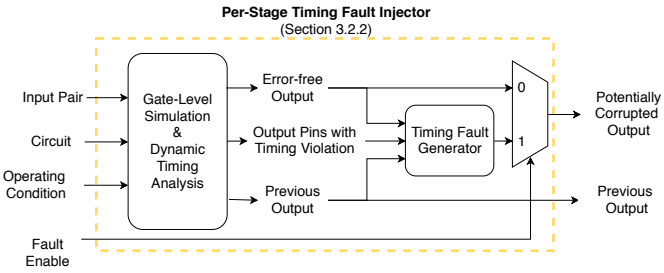

Figure 4: Per-stage timing fault injector.

4 ACCELERATING INSTRUCTION-LEVEL TIMING ERROR INJECTION

We assume that timing faults occur randomly and uniformly during execution because timing errors depend on a large number of factors that include dynamics of the system and variations of the operating environment. To evaluate the resilience of applications against timing errors, we adopt the Monte Carlo method because the error space is enormous. However, it takes a large number of Monte Carlo trials to collect statistics on errors that do affect applications. This is because many injected faults may be masked either temporally or logically by hardware and thus have no impact on the running applications. Although we utilize hierarchical injection to reduce per-trial evaluation time, the overall

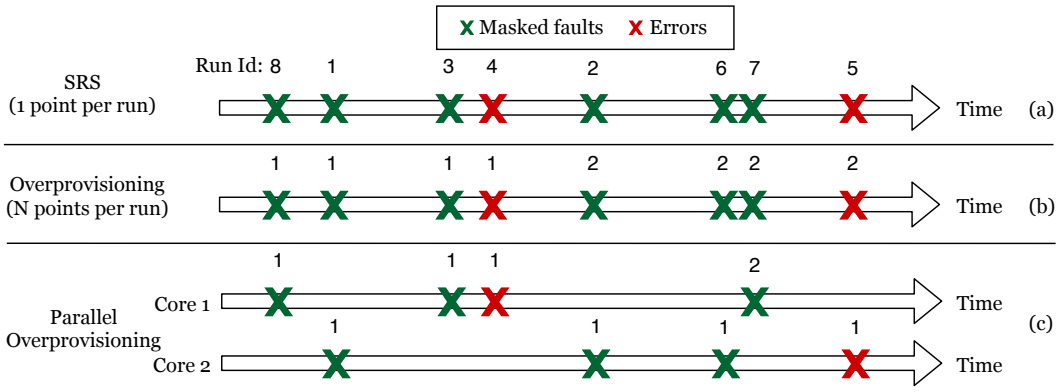

Figure 5: The proposed injection-point overprovisioning technique vs. simple random sampling. (a) simple random sampling, (b) injection-point overprovisioning, and (c) an example of applying injection-point overprovisioning on two cores in parallel. Green crosses denote masked faults and red ones are errors that change the architectural state.

evaluation is still slow because the number of trials remains large.

4.1 Inefficiency of Simple Random Sampling

The conventional sampling algorithm (*simple random sampling*) is embarrassingly parallel, yet is inefficient in terms of collecting errors that are observable at the applications level. Figure 5a shows an example of simple random sampling in which each application run is provided with one random fault injection point (a random dynamic instruction instance for instruction-level error injection). In this example, eight application runs end up collecting only two observable errors. In general, the higher the fault masking rate, the more Monte Carlo trials are needed, which enormously wastes resources. We propose an acceleration technique to reduce the number of application runs required for error injection campaigns.

4.2 Injection-Point Overprovisioning

This technique is based on the observation that masked faults do not change the architectural state (i.e., are invisible to applications). Unlike simple random sampling which provides only one injection point per application run, our proposed Injection-Point Overprovisioning technique provides multiple potential injection points, sorted by injection time. If a fault is masked at an early injection point, we evaluate the next point until an error is generated or until all injection points for the run are exhausted. Once injection succeeds in manifesting an architecturally-visible error, all remaining points are passed to the next run to ensure that the sampling is unbiased.

Figure 5b demonstrates the idea of injection-point overprovisioning with the same set of injection points as shown in Figure 5a for simple random sampling. Injection-point overprovisioning reduces the number of applications runs from 8 to 2 in this example. Note that injection-point overprovisioning increases the injection overhead per run because it performs multiple injections. However, as long as the per-run injection overhead is negligible compared to the overhead of running the application itself, the overall evaluation time will be improved. We demonstrate this analytically in Section 4.2.2.

Injection-point oversampling is easily generalized to machines with parallel architectures and still reaches the same level of parallelism as simple random sampling. Figure 5c shows an example of parallel overprovisioning on two cores with the same set of injection points. Note that dependencies only exist between application runs that execute back-to-back on the same core, while runs on different cores are independent. A synchronization mechanism among cores is needed to terminate the entire error injection campaign when it collects enough errors to claim statistical significance, but this synchronization is infrequent.

4.2.1 Implementation. We implement injection-point overprovisioning with the master-worker parallel pattern. The master thread is responsible for generating and updating the batch of injection points associated with each core and for synchronizing worker threads upon termination.

We adopt the two-phase error injection workflow consisting of a profiling and an injection phase as in prior work [5, 6]. The profiling phase simply determines the range of injection points, while the injection phase does fault/error injection with the injection-point overprovisioning technique.

In the injection phase, the master thread initializes each batch with N points randomly drawn within the range determined in the profiling phase. It then launches a worker thread for each core that performs fault injection to the specified application. Next, each worker thread injects faults at points specified in its batch until an error is generated or until the batch is exhausted, and then records the remaining points (if any). When a worker thread on core C_i is done, the master thread updates the batch associated with core C_i by either passing the remaining points or generating a new batch of

points. Once the total number of collected errors reaches the target, the master thread waits until all worker threads finish before terminating the injection phase.

Note that the efficiency of the proposed technique depends on the batch size N and other factors, which we discuss next.

4.2.2 Analytical Modeling. Our goal is to understand when injection-point overprovisioning outperforms simple random sampling. To this end, we derive a first-order analytical model for the cost of generating an error relative to application execution time.

Assume that at each injection point, the fault masking rate is an i.i.d. Bernoulli random variable with masking-probability P. Let α be the injection overhead normalized to the execution time of the application. The cost of using simple random sampling is then formulated as in Equation 1. The first term is the cost of injecting a fault for each application run and the second term is the mean number of fault injections (equal to the number of application runs) needed to generate an error. For instance, if P is 50%, it takes two runs on average to generate one error.

$$C_{SRS} = (1+\alpha) \times (\frac{1}{1-P}) \tag{1}$$

The cost of injection-point overprovisioning (with N points per run) can be formulated as in Equation 2, based on the law of total probability. The first term is the expected cost if the first injection leads to an error. $\frac{1}{N}$ is the expected relative cost of running until the injection point because the injection points are generated uniformly across time. The second term is the cost if the first injection is masked but the second injection results in an error, and so on.

$$\begin{aligned} C_{IPO} &= (1-P)(\frac{1}{N}+\alpha) + P(1-P)(\frac{2}{N}+2\alpha) \\ &\quad + \ldots + P^{N-1}(1-P)(\frac{N}{N}+N\alpha) \\ &= (\frac{1}{N}+\alpha) \times (\frac{1-(1+N-NP)P^N}{1-P}) \end{aligned} \tag{2}$$

We define the speedup of overprovisioning as the ratio of C_{SRS} to C_{IPO}. If speedup is greater than 1, then injection-point overprovisioning outperforms simple random sampling. We sweep the parameters in both equations and plot speedup in Figure 6. When injection overhead is small (e.g., $\alpha \leq 0.1$), injection-point overprovisioning outperforms simple random sampling by at least 4X in most cases, and speedup increases with the batch size (N). However, when the masking rate (P) is less than 0.2, or when the injection overhead is large (e.g., $\alpha = 1$), injection-point overprovisioning is worse than simple random sampling.

In reality, the fault masking rate depends on multiple factors (including circuits, operating condition, input data, etc.). Thus, it is not as simple as a Bernoulli random variable. We present the experimental results for the benefits of injection-point overprovisioning in Section 6.1.

4.2.3 Optimization with Checkpoint-Restart. Injection-point overprovisioning can be further improved with the aid of low-overhead checkpoint-restart. The idea is to reduce the overhead of running applications toward injection points by fastforwarding to the nearest checkpoint before each injection point. Note that this optimization helps only when the overhead of restarting is small compared to application execution time. We leave the analysis for future work.

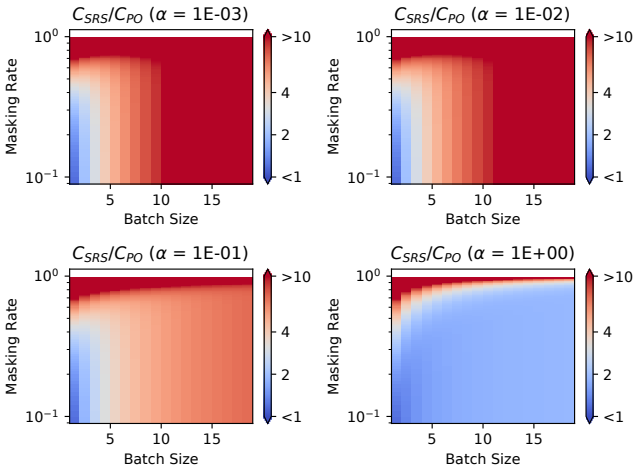

Figure 6: Speedup of injection-point overprovisioning as a function of the batch size, the masking rate, and injection overhead. X-axis is the batch size (N), and y-axis the masking rate (P). α is the injection overhead normalized to execution time of the application.

5 EVALUATION METHODOLOGY

Our evaluation consists of the following parts: (1) assessing the benefits of injection-point overprovisioning, (2) characterizing the timing error models at the instruction level, (3) evaluating the impact of error models on the resilience of applications, and (4) conducting sensitivity studies of application resilience to droop profiles and pipeline occupancy.

5.1 Error Models

In this work, we focus on timing errors resulting from transient voltage droops (i.e., $\frac{dI}{dt}$ droops), but the tool and framework can be used to study other types of timing errors as well. We compare the proposed high-fidelity error model using just-in-time error generation to two well-known low-fidelity error models.

5.1.1 High-Fidelity Timing Error Models. Recall that the error manifestation process of timing errors is very complex and depends on the following factors: (1) the circuit structure, (2) the operating condition, (3) the variation's profile, and (4) the history of input values to the circuit.

Circuits and Operating Conditions. We inject errors to arithmetic units because those circuits are more prone to timing errors according to previous work [28, 29]. We synthesize

Table 1: Circuits and their fault injection overhead.

Circuit	Depth	Mean	Std.
INT-ADD	1	0.39s	0.003
INT-MUL	3	4.80s	0.112
FP-ADD	3	1.59s	0.011
FP-MUL	3	3.95s	0.026
FP-DIV	1	4.15s	0.029

gate-level netlists of integer and floating-point adders and multipliers using Synopsys tools (Design Compiler and DesignWare Library) with Synopsys's SAED 32nm technology, optimized for performance. The pipeline depths are tuned to match Intel's Broadwell processors based on the latency data from [30]. After carefully verifying the netlists, we use OpenTimer to obtain the critical path of each circuit and set the clock frequency as 400MHz. The nominal voltage of the cell library is 1.05V and temperature is set to 25°C. Table 1 lists the circuits studied in this paper. Notice that these circuits can be different from those designed and optimized for commodity processors, but the developed tool is generic enough to evaluate other circuits as well.

Droop Profile. Other than the nominal 1.05V voltage, we use two other voltage levels (0.85V and 0.78V) to model droops. We model single-cycle droops and multi-cycle droops that persist for the maximum pipeline depths (i.e., 3 cycles for the circuits under test).

Pipeline Occupancy. The number of instructions affected by a voltage droop depends on the droop's duration and the occupancy of the affected pipeline. In general, for a pipelined circuit of depth D, the maximal number of instructions affected by a T-cycle droop is $D+T-1$. We model the extreme cases to understand whether accurate modeling of pipeline occupancy is necessary:

- **Low occupancy**: only one instruction in the pipeline throughout duration of the droop.
- **High occupancy**: the pipeline is fully utilized.

For each droop magnitude, the combination of droop duration and pipeline occupancy leads to four error groups:

- **Single-cycle Low-occupancy (SL)**: a single-cycle droop affects a low-occupancy pipeline.
- **Multi-cycle Low-occupancy (ML)**: a multi-cycle droop affects a low-occupancy pipeline.
- **Single-cycle High-occupancy (SH)**: a single-cycle droop affects a high-occupancy pipeline.
- **Multi-cycle High-occupancy (MH)**: a multi-cycle droop affects a high-occupancy pipeline.

Figure 7 illustrates the four error groups for a 3-stage pipeline. In this example, *SL* affects one instruction at a random stage. *ML* affects the same instruction in three consecutive cycles at different stages. *SH* affects three instructions simultaneously in one cycle. *MH* affects five instructions in total across three cycles. For each instruction, the error

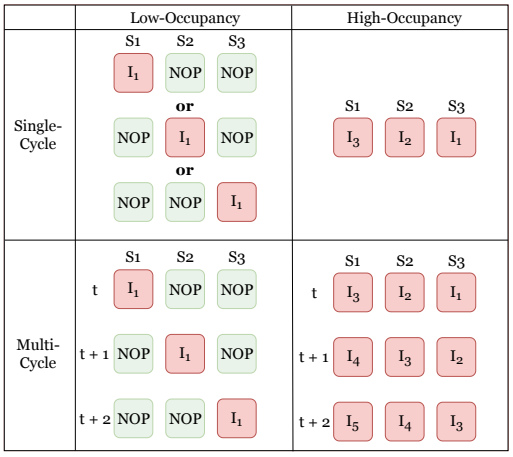

Figure 7: Illustration of error groups for a 3-stage pipeline. Data flow from stage 1 (S_1) to stage 3 (S_3).

group determines which stages are affected by the droop. Recall that in our design we use fault-enable signals to control which stages to inject (Figure 3). For instance, when the *ML* error group is specified, I_1 is affected by the multi-cycle droop in three consecutive stages. Thus, fault-enable signals corresponding to those stages are set.

The error models are:

- **Single-bit flip (RB1)**: randomly flips a single bit, shown to be a good approximation of high-fidelity arithmetic errors due to particle strikes [6].
- **Previous value (PREV)**: models a severe voltage droop which causes timing violation at all output pins and thus latches the output value of the previous instruction using the same execution unit [31].
- **Decreasing voltage to 0.85V (0.85V)**: generates a timing error resulting from a droop that decreases the voltage to 0.85V. It consists of four error groups (Figure 7).
- **Decreasing voltage to 0.78V (0.78V)**: generates a timing error resulting from a droop that decreases the voltage to 0.78V. It also consists of four error groups.

5.1.2 Experimental Settings. We assume that timing faults occur randomly and uniformly in this work (e.g., when hardware error-mitigation techniques fail due to an unexpectedly strong voltage droop). In each experiment, we inject an error into a random instruction's output operand with one of the four error models listed above. For each model, we use injection-point overprovisioning to collect 2000 random *errors*. This ensures a margin of measurement error around 2.2% for a confidence level of 95% [32]. Based on Figure 6, we set the batch size (N) to 10. For SIMD instructions, we inject timing faults to all SIMD lanes because they usually share the same power delivery network.

Table 2: Benchmarks, their input, average execution time, and how we classify SDCs.

	Input	Native Time	Time w/ Injection	SDC Criteria
FT	Class A	3.5s	23.4s	Failed verification
LU	Class A	29.3s	201.8s	Failed verification
MG	Class B	5.8s	46.7s	Failed verification
CG	Class A	1.2s	7.6s	Failed verification
CoMD	default	5.7s	33.8s	Potential energy $> 10^{-10}$ [34]
LULESH	default	22.1s	130.5s	MaxAbsDiff $> 10^{-8}$ [36]

5.2 Benchmarks and Outcome Classification

We use six serial scientific kernels and proxy applications: `FT` and `LU` and `CG` and `MG` from NPB [33], `CoMD` [34], and `LULESH` [35]. Table 2 summarizes the benchmarks, their input, native execution time per experiment, execution time with injection overhead, and how we classify injection outcomes as SDCs. We compile all benchmarks with gcc 4.8.5 using the original building scripts.

As in prior work [6], injection outcomes are classified into three 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.

5.3 Testbed

We develop the injector on a single-node machine that runs OpenSUSE 42.3 and then conduct error injection experiments on the Lonestar5 supercomputer at TACC.

6 EVALUATION RESULTS

6.1 Injection-Point Overprovisioning

Figure 8 shows the speedup of injection-point overprovisioning for different error groups under different droop magnitudes. Note that the maximum speedup is 7X (for `LULESH` in the setting with 0.85V and the SL error group). Speedup is highest for the SL error group (i.e., voltage droop affects only one instruction at a random stage). In this case, timing errors are more likely to be logically masked at later pipeline stages, and thus the fault-masking rates are higher. Also, speedup is higher when droop magnitude is lower (0.85V) because the fault-masking rate is higher (mostly due to temporal masking). On average, for the weaker droops (0.85V),

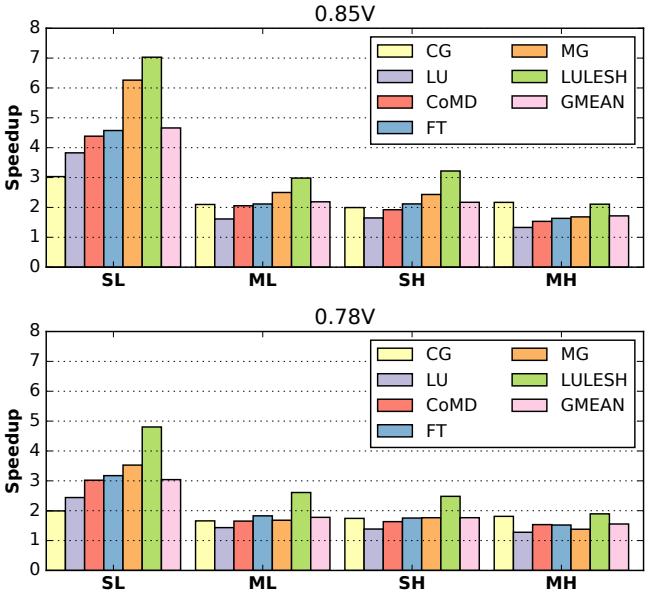

Figure 8: Speedup of injection-point overprovisioning for high-fidelity timing error injection. Speedup here is defined as the ratio of evaluation time using simple random sampling to injection-point overprovisioning. Top: 0.85V; Bottom: 0.78V. X-axis: four error groups (Figure 7).

speedup of injection-point overprovisioning is 4.7X, 2.2X, 2.2X, 1.7X for the SL, ML, SH, MH error groups, respectively. For stronger droops (0.78V), speedup is 3.0X, 1.8X, 1.8X, 1.6X, respectively.

In terms of resource savings due to injection-point overprovisioning, we save 2,943 core-hours for the entire evaluation in this work. The original resource requirement for simple random sampling was 5,585 core-hours. Notice that the savings can be even higher if one needs higher accuracy (i.e., observing more errors). In this work, our accuracy is around 2%. The projected savings for a 1% of accuracy target is 13,978 core-hours for the same set of experiments. The resource requirement would be 26,527 core-hours if simple random sampling is used.

6.2 Injection Outcome and Analysis

In this section, we present the characteristics of timing errors at the instruction level and the injection outcome at the application level.

6.2.1 Instruction-level error patterns. We characterize how many bits of the circuit output are flipped (Figure 9) and which bits are more likely to be flipped (Figure 10). Note that distributions are application-dependent.

Observation 1: timing errors that corrupt arithmetic circuits tend to flip multiple lower-significance bits.

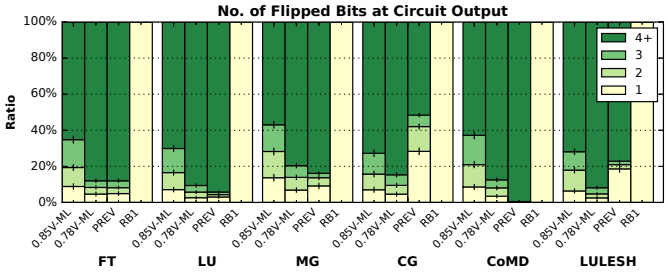

Figure 9: Distributions of the number of flipped bits at circuit output (simple error models vs. the high-fidelity ML timing error group).

According to Figure 9, timing errors flip multiple bits in most cases (> 80%). The fact that timing errors tend to flip lower-significance bits is due to value locality at the higher-significance bits. In other words, the higher-significance bits rarely transition between previous output and current output at the circuit level. Therefore, even though higher-significance bits are more likely to experience timing violation, the latched values are still correct.

Note that PREV in some cases (e.g., `CG` and `LULESH`) flips fewer bits than the high-fidelity model. This is due to the fact that errors generated by PREV exhibit a different instruction mix when compared with the high-fidelity model; PREV essentially injects errors into some instructions that never generate an error when using a higher-fidelity model. Although circuits with lower complexity (e.g., integer adders) are less sensitive to timing errors, PREV can still corrupt results generated by such simple circuits because it lacks timing information.

6.2.2 Application-level injection outcomes. Figure 11 shows the distributions of error injection outcomes using different error models. Recall that RB1 is shown to be a good approximation of high-fidelity arithmetic errors due to particle strikes [6]. Therefore, we can use it as a proxy to compare timing errors with errors due to particle strikes.

Observation 2: timing errors rarely lead to DUEs.

Compared to RB1 and PREV, high-fidelity timing errors rarely lead to DUEs. This observation is related to previous results at the instruction level: lower-significance bits are more likely to be flipped. On the other hand, since RB1 flips each bit with equal likelihood, it tend to flip higher-significance bits and causes more segmentation faults. `LULESH` is the only application in which high-fidelity timing errors result in some DUEs. It turns out that most DUEs are bus errors (i.e., unaligned memory accesses) instead of segmentation faults.

Observation 3: timing errors result in low SDC ratio.

High-fidelity errors cause fewer SDCs compared with RB1 and PREV. This is also related to the results at the instruction level. For instance, for floating-point instructions, flipping lower-significance bits leads to changes in the significand field, which has less impact compared to the exponent

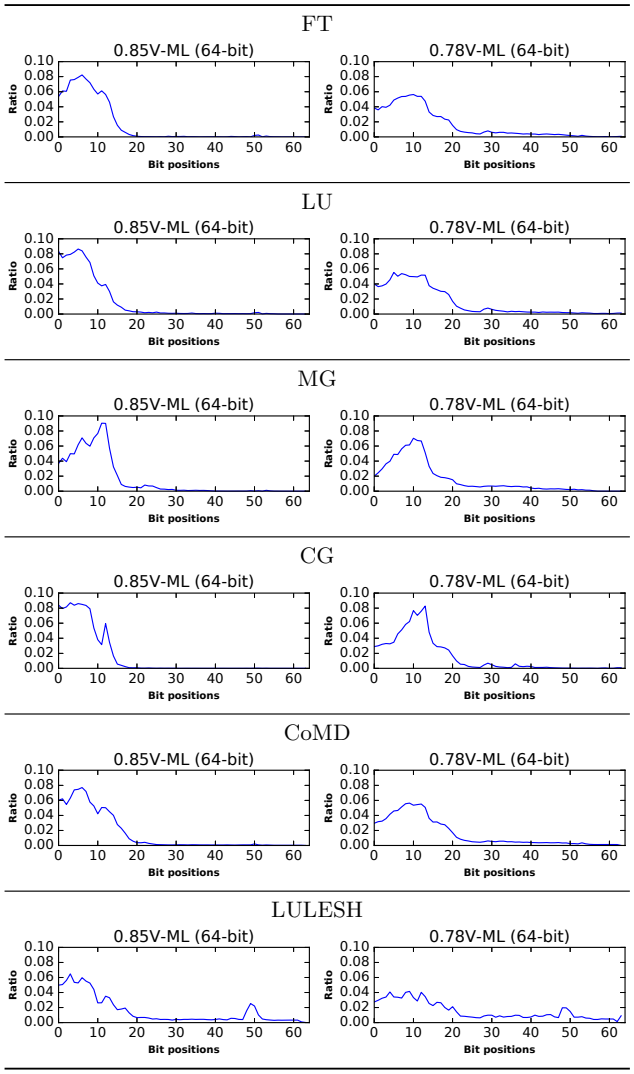

Figure 10: Distribution of bit-flip positions at circuit output with input from each benchmark under the ML error group. Note that these are not distributions of positions with timing violation.

field. We expect that SDC ratios would be even smaller for weaker droops (e.g., droops that decreases the voltage to 0.9V). If we compare the SDC ratios between RB1 and high-fidelity timing errors, the maximum difference occurs in `FT` where the SDC ratio of RB1 is 10% and those of 0.78V and 0.85V are 3% and 0%, respectively.

Observation 2 and observation 3 indicate that it is plausible to save power by reducing guardbands if the user can tolerate some DUEs and SDCs. However, such decisions must be made carefully as timing errors can occur when the processor is in kernel mode as well. The evaluation of kernel mode is beyond the scope of this work.

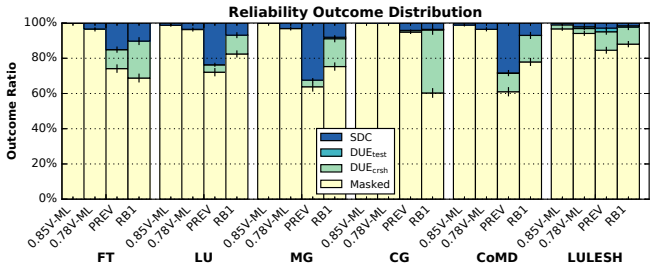

Figure 11: Injection outcome distributions (simple error models vs. the high-fidelity ML timing error group). Results of other error groups are in Figure 12.

Observation 4: neither RB1 nor PREV is a good approximation for high-fidelity timing errors.

Both RB1 and PREV result in pessimistic results compared with high-fidelity timing errors. RB1 overestimates DUE ratio by at least 7% in all cases and it overestimates SDC ratio by 5.7% and 3.6% for 0.85V and 0.78V, respectively. PREV greatly overestimates both DUE ratio and SDC ratio in all applications.

6.3 Sensitivity Studies

Figure 12 shows the sensitivity studies of injection outcome distributions to droop profiles and pipeline occupancy.

First, we observe that SDC ratio increases with droop magnitude since stronger droops are more likely to flip higher-significance bits. Second, for the same droop magnitude, three error groups (SL, ML, and SH) lead to similar distributions (error bars are overlapped). This also means that for single-cycle droops (SL and SH), injection outcomes depend only on droop magnitude. Note that MH leads to higher SDC ratio than ML since it not only affects more instructions but also affects them multiple times. The maximum difference of SDC ratio between MH and the other three groups is 3.5% in `MG` with 0.78V. Therefore, for evaluating the impact of multi-cycle droops on applications, more accurate modeling for pipeline occupancy is needed.

Observation 5: for single-cycle droops, injection outcomes depend only on droop magnitude, while for multi-cycle droops, injection outcomes depend on both droop magnitude and pipeline occupancy.

7 RELATED WORK

In terms of evaluating the resilience of HPC applications against hardware errors, much prior work assumes particle strikes as the fault model [1, 4–6, 37, 38], though researchers have pointed out that circuit timing uncertainty poses challenges on designing systems with high power efficiency [14, 15, 39]. In this section, we first summarize existing resilience evaluation techniques and draw an analogy between assessing the resilience of applications against timing faults vs. particle strikes. Then, we discuss state-of-the-art error models for timing errors. Finally, we compare the proposed injection-point overprovisioning technique with other acceleration techniques.

Hardware-based. Evaluation methodology such as beam tests can quantify the rate and the impact of radiation effects on applications [38, 40]. This methodology is highly accurate and more realistic because real devices are used. The analogous methodology for evaluating timing errors is directly undervolting or overclocking commercial processors [41–44]. Disadvantages of this methodology include high cost and inability to precisely control the fault locations of interest. The tool used in this work is not of this type.

Software-based. Methods that use software to perform fault/error injection at various abstraction layers fall in this category (including this work). These methods can target specific layers of interest with different fault/error models. For instance, faults/errors can be injected into applications [38], the operating system [45, 46], the architecture level [1, 3, 5], the micro-architecture level [25, 26], RTL gate level [2, 6, 11], etc. Also, they are more flexible because the injection substrate is decoupled from the fault/error models such that tools can be repurposed to model different types of faults.

However, software-based methods have a fidelity problem because (1) they can only inject faults/errors into the layer(s) modeled by software and (2) the fault/error model assumed may be different from reality. As a result, prior work adopts hierarchical injection to balance injection speed and accuracy by integrating injectors at different levels [6, 10, 11]. The tool used in this work spans across the instruction level, RTL gate level, and circuit level.

Timing-error modeling. There are two categories of timing error models: (1) pre-characterization and (2) just-in-time error generation. Models based on pre-characterization assume a specific set of circuit implementations and derive an error model based on circuit-level fault injection. The derived error model usually incurs low overhead but they suffer from at least one of the following issues: (a) inflexibility in terms of circuit implementations, (b) assuming timing errors are independent between output pins of circuits, (c) using pseudo-random input vectors to train an error model, or (d) high overhead due to circuit transformation.

CrashTest [47] emulates timing errors by adding additional logic to the original design and thus incurs high overhead. Therefore, usage of FPGA is required. Another issue of the technique is that the set of flip-flops affected by timing faults is not selected based on toggled paths (Figure 1). Authors of b-HiVE [7] point out the importance of taking computation history into account. They derive the fault probability of each output pin for ALU circuits of OpenSPARC T1. The work by Constantin et al. [8] is similar to b-HiVE but provides fine-grained control over fault probabilities. Both work use pseudo-random input to train their models and assume that timing errors between circuit endpoints are independent. CLIM [9] uses machine learning to build timing error models

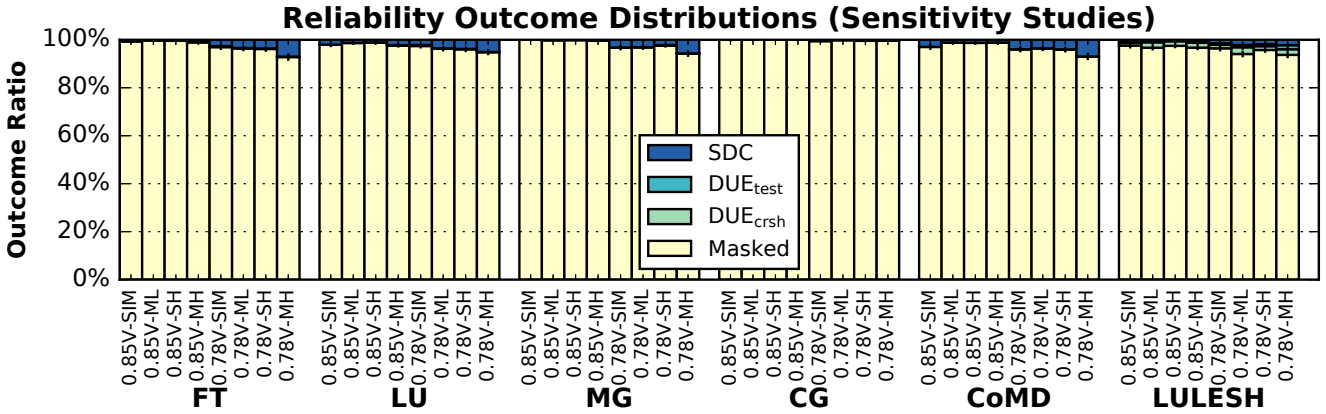

Figure 12: Injection outcome distribution for sensitivity studies of droop magnitude and error groups.

and shows high accuracy compared with gate-level simulation, but it has the inflexibility issue and using pseudo-random input to train their model.

On the other hand, models that rely on just-in-time error generation are more generic but they incur higher overhead [6, 11]. However, they do not suffer from the aforementioned issues because they perform just-in-time simulation with circuit input from applications to generate errors. Also, user has full control over the circuits, its operating condition, and the profiles of timing faults (e.g., magnitude and duration). The tool used in this work falls in this category. Note that although just-in-time error generation has been used to model hardware faults due to particle strikes, to the best of our knowledge, this work is the first that models timing errors and evaluates their end-to-end effects on applications thanks to recent open-source campaigns in the EDA community.

Acceleration techniques. The work by Chang et al. [6] proposes nested Monte Carlo to accelerate hierarchical injection by sampling different physical fault sites to generate an unmasked fault at an injection point. The method is not applicable for injecting timing errors hierarchically because there is only one fault site in our case. However, our injection-point overprovisioning can be used to accelerate their experiments. Also, our approach is orthogonal to techniques that prune faults potentially leading to the same injection outcome [26, 48] and techniques based on hardware acceleration [2, 47].

8 CONCLUSION

We show that injection-point overprovisioning saves evaluation time and resources for timing error injection experiments. The method can be applied to other experiments that use hierarchical injection. Additionally, we build an open-source instruction-level timing error injector that generates high-fidelity errors on demand at runtime with low overhead. Using the tool, we find that timing errors tend to flip multiple lower-significance bits due to value locality in applications. When timing errors corrupt the architectural state of HPC applications, they rarely lead to DUEs and SDCs, indicating the plausibility of saving power by reducing guardbands without significantly degrading the correctness of applications. We also demonstrate that two synthetic error models, single-bit flips and previous values, do not represent high-fidelity timing errors that affect arithmetic units.

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] 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.

[2] 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.

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

[4] 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.

[5] 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, 2017*, IEEE.

[6] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Evaluating and accelerating high-fidelity error injection for hpc," in *In the Proceedings of the ACM/IEEE International Conference on High-Performance Computing, Networking, Storage, and Analysis (SC18)*, (Dallas, TX), November 2018.

[7] G. Tziantzioulis, A. Gok, S. Faisal, N. Hardavellas, S. Ogrenci-Memik, and S. Parthasarathy, "b-hive: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 105, ACM, 2015.

[8] J. Constantin, Z. Wang, G. Karakonstantis, A. Chattopadhyay, and A. Burg, "Statistical fault injection for impact-evaluation of timing errors on application performance," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 13, ACM, 2016.

[9] X. Jiao, A. Rahimi, Y. Jiang, J. Wang, H. Fatemi, J. P. De Gyvez, and R. K. Gupta, "Clim: A cross-level workload-aware timing error prediction model for functional units," *IEEE Transactions on Computers*, vol. 67, no. 6, pp. 771–783, 2018.

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

[11] 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.

[12] T.-W. Huang and M. D. Wong, "Opentimer: A high-performance timing analysis tool," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 895–902, IEEE Press, 2015.

[13] H. Cherupalli and J. Sartori, "Graph-based dynamic analysis: Efficient characterization of dynamic timing and activity distributions," in *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*, pp. 729–735, IEEE, 2015.

[14] V. J. Reddi and M. S. Gupta, "Resilient architecture design for voltage variation," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 2, pp. 1–138, 2013.

[15] A. Rahimi, L. Benini, and R. K. Gupta, "Variability mitigation in nanometer cmos integrated systems: A survey of techniques from circuits to software," *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1410–1448, 2016.

[16] C. R. Lefurgy, A. J. Drake, M. S. Floyd, M. S. Allen-Ware, B. Brock, J. A. Tierno, and J. B. Carter, "Active management of timing guardband to save energy in power7," in *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–11, IEEE, 2011.

[17] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers, and S. Das, "Harnessing voltage margins for energy efficiency in multicore cpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 503–516, ACM, 2017.

[18] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. K. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. S. Rosing, M. B. Srivastava, *et al.*, "Underdesigned and opportunistic computing in presence of hardware variability," *IEEE Transactions on Computer-Aided Design of integrated circuits and systems*, vol. 32, no. 1, pp. 8–23, 2013.

[19] 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: C ross-l ayer e xploration for a rchitecting r esilience-combining hardware and software techniques to tolerate soft errors in processor cores," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 68, ACM, 2016.

[20] V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. D. Smith, G.-Y. Wei, and D. Brooks, "Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 77–88, IEEE Computer Society, 2010.

[21] R. Bertran, A. Buyuktosunoglu, P. Bose, T. J. Slegel, G. Salem, S. Carey, R. F. Rizzolo, and T. Strach, "Voltage noise in multi-core processors: Empirical characterization and optimization opportunities," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 368–380, IEEE, 2014.

[22] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, 2010.

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

[24] 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.

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

[26] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 241–254, ACM, 2017.

[27] D. Harris and N. Weste, "Cmos vlsi design," *Pearson Education, Inc*, 2010.

[28] A. Rahimi, L. Benini, and R. K. Gupta, "Analysis of instruction-level vulnerability to dynamic voltage and temperature variations," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1102–1105, IEEE, 2012.

[29] Y. Kim, L. K. John, I. Paul, S. Manne, and M. Schulte, "Performance boosting under reliability and power constraints," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pp. 334–341, IEEE, 2013.

[30] 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.

[31] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Notices*, vol. 46, pp. 164–174, ACM, 2011.

[32] 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.

[33] 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.

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

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

[36] 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.

[37] 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 *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, IEEE, 2015.

[38] D. Oliveira, L. Pilla, N. DeBardeleben, S. Blanchard, H. Quinn, I. Koren, P. Navaux, and P. Rech, "Experimental and analytical study of xeon phi reliability," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 28, ACM, 2017.

[39] J. Torrellas, "Extreme-scale computer architecture: Energy efficiency from the ground up," in *Proceedings of the conference on Design, Automation & Test in Europe*, p. 200, European Design and Automation Association, 2014.

[40] 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 *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 587–596, IEEE, 2014.

[41] A. Bacha and R. Teodorescu, "Dynamic reduction of voltage margins by leveraging on-chip ecc in itanium ii processors," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 297–307, ACM, 2013.

[42] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi, "Safe limits on voltage reduction efficiency in gpus: a direct measurement approach," in *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*, pp. 294–307, IEEE, 2015.

[43] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers, and S. Das, "Harnessing voltage margins for energy efficiency in multicore cpus," in *Proceedings of the 50th*

Annual IEEE/ACM International Symposium on Microarchitecture, pp. 503–516, ACM, 2017.

[44] K. Parasyris, P. Koutsovasilis, V. Vassiliadis, C. D. Antonopoulos, N. Bellas, and S. Lalis, "A framework for evaluating software on reduced margins hardware," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 330–337, IEEE, 2018.

[45] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Transactions on computers*, vol. 44, no. 2, pp. 248–260, 1995.

[46] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "Fail*: Towards a versatile fault-injection experiment framework," in *ARCS 2012*, pp. 1–5, IEEE, 2012.

[47] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "Crashtest: A fast high-fidelity fpga-based resiliency analysis framework," in *2008 IEEE International Conference on Computer Design*, pp. 363–370, IEEE, 2008.

[48] 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.