Hamartia: A Fast and Accurate Error Injection Framework

Chun-Kai Chang¹, Sangkug Lym¹, Nicholas Kelly¹, Michael B. Sullivan¹,², and Mattan Erez¹

¹The University of Texas at Austin
{chunkai, sklym, nick.kelly, mattan.erez}@utexas.edu
²NVIDIA
misullivan@nvidia.com

Abstract—Single bit-flip has been the most popular error model for resilience studies with fault injection. We use RTL gate-level fault injection to show that this model fails to cover many realistic hardware faults. Specifically, single-event transients from combinational logic and single-event upsets in pipeline latches can lead to complex multi-bit errors at the architecture level. However, although accurate, RTL simulation is too slow to evaluate application-level resilience.

To strike a balance between model accuracy and injection speed, we refine the concept of hierarchical injection to prune faults with known outcomes, saving 62% of program runs at 2% margin of error on average across 9 benchmark programs. Our implementation of the hierarchical error injector is not only accurate but also fast because it is able to source realistic error patterns using on demand RTL gate-level fault injection. Our tool outperforms state-of-the-art assembly-level and compiler-based error injectors by up to 6X, while providing higher fidelity.

I. INTRODUCTION

Due to technology scaling, computer systems are becoming sensitive to hardware faults from a variety of sources (e.g., particle strikes, voltage droop, wear-out, etc.). Some hardware faults eventually cause soft errors that are hard to detect because they are random and transient in nature. Although some errors might be masked or detected by either hardware or software, other soft errors eventually propagate and silently affect the output of applications. Such events are known as Silent Data Corruption (SDC) and can lead to application and system failure.

Hardware-based fault detection mechanisms provide high coverage but they are costly in terms of area and power, such that they are only adopted by mission-critical systems. Recently, researchers have shown that software-based resilience techniques can also detect SDC with high coverage [1], [2], [3], [4]. However, such techniques incur performance overhead due to the software detector (e.g., instruction duplication and symptom-based detectors). To reduce performance overhead, software resilience techniques should be customized to specific applications and scenarios.

In order to evaluate the efficacy of a custom software resilience technique and tune its parameters, software-based error injection tools are widely used. However, current injectors are either slow, or focus on single bit-flip error model. Injecting at the RTL level or the circuit level are time-consuming but the generated errors are said to be higher-fidelity (i.e., closer to realistic hardware faults). On the other hand, instruction-level and compiler-level injectors are faster at the cost of losing fidelity because unrealistic bit-flip error models are used.

In this work, the main questions we answer are:
1) Is it important to have higher fidelity?
2) Can we make higher-fidelity injectors fast?

Prior work [5], [6] has shown that errors injected at different abstraction layers can affect application behavior differently. In this work, we perform RTL gate-level injection experiments and find that the error patterns at a circuit’s output (i.e., the bit-flip count and bit-flip positions) are different from those generated with the single bit-flip error model. Also, the error patterns depend on both circuit structure and input data.

To address the second question we ask, we propose a refined hierarchical error injection methodology to make higher-fidelity injector fast. Our proposed method allows instruction-level injectors to proceed at speeds close to native execution, and only launch RTL simulation at the target instruction instance to generate realistic faults.

Since a fault at the RTL level might be masked before propagating to the architecture level, an iterative fault-generation process is conducted until a fault manifests as an error that is visible to architectural components. We show that this methodology saves 62% of program runs at 2% margin of error on average across 9 benchmark programs.

Our implementation of this injection framework outperforms a state-of-the-art injector by 3X on average. Plus, our higher-fidelity RTL-level injector is designed to be pluggable such that it can be easily integrated with other instruction-level injectors to expedite injection without losing fidelity.

The contributions of this paper are summarized as follows:

- We show that errors generated with a realistic RTL gate-level fault model are different from random bit-flips at the instruction level and thus make applications behave differently.
- We refine the concept of hierarchical injection to accelerate fault injection by pruning faults with known outcomes.
- We implement an error injection framework capable of generating realistic errors and still outperforming existing...
in injectors by 3X on average\textsuperscript{1}.

II. BACKGROUND AND MOTIVATION

In this paper we denote 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.

A. Fault Masking Assumptions

Soft errors that affect storage elements and combinational logic are triggered by random radiation events such as particle strikes. However, the impact of these events can be masked due to electrical masking, logical masking, and timing masking. In this paper, we are concerned with the impact of errors, but not the rate of errors. We therefore assume faults escape electrical masking and timing masking; that is, we only consider particle strikes that carry sufficient charge and result in faulty signals that arrive on time at the next latch. We respect logical masking because it depends not only on the circuit but also on application.

B. Mapping Existing Error Models to Hardware Soft Errors

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

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 manifest as an error. On the other hand, when a fault occurs in the input buffer, it can be masked by the circuit. 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.

Next, we consider the case where the fault site at a logic gate within the combinational logic component and assume the fault induces a pulse that flips the gate’s output. 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 combinational logic.

The same is true for latches within the circuit, which cannot be directly observed or manipulated at the architecture level. To quantify the impact of this modeling gap, we study characteristics of soft errors from an internal circuit node via RTL gate-level fault injection.

\textsuperscript{1}Available at \url{http://lph.ece.utexas.edu/users/hamartia/}

C. Characteristics of Soft Errors from Combinational Logic

Soft errors in combinational logic are a growing concern because logic error rate increases with clock frequency due to less timing masking. Researchers have reported the error rate for combinational logic can be as high as 30% of that for latches at 32-nm and 22-nm nodes \cite{7}, \cite{8}, \cite{9}. To estimate the impact of these soft errors, we perform RTL gate-level injection to arithmetic and logic units (details in Section IV).

Figure 2 shows the distribution of how many bits at the circuit output are flipped if a fault from a circuit internal node 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 (Figure 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 \textit{LULESH}, the floating-point adder has strong correlation between bits in the exponent field (bit 52-62), while we do not observe such phenomena with input from \textit{NAS-IS}. 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 correlation is not modeled by random bit-flipping models.

Although we have shown that realistic error patterns are...
different from random bit-flipping at the instruction level, the impact on application 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.

III. AN ACCURATE AND FAST INJECTION FRAMEWORK

A. Efficient Hierarchical Error Injection with Instruction-level Error Detection

Although RTL-based error injectors offer high fidelity, they are too slow to evaluate software resilience techniques in a timely manner (seven orders of magnitude slower than application-level injection [5]). To strike a balance between injection speed and fidelity, previous work proposes hierarchical injection where a faster injector invokes another detailed injector [6], [10], [11], [12]. For instance, SWAT-Sim [6] couples a microarchitecture-level injector with a gate-level simulator to accelerate injection and gain fidelity at target injection point. However, microarchitecture-level injectors are still too slow to evaluate fast-changing software resilience techniques. For instance, it takes days to evaluate the original MiBench benchmark [13] (a very small embedded benchmark) using a state-of-the-art microarchitecture-level injector [14]. Evaluating the software-protected versions and applications at larger scale (e.g., long-running HPC applications that are more susceptible to soft errors) is infeasible. It is thus necessary to further shorten the time taken by error injection experiments.

We refine hierarchical injection to boost its speed by orders of magnitude. First, we focus on faults that directly affect architectural state (i.e., faults directly relating to instructions). Second, we use a novel technique to filter out useless injections. Because injecting a fault into a gate does not always lead to an error due to logical masking, we first record the correct circuit output and then repeatedly perform gate-level fault injection until a fault manifests as an error that corrupts the output. By doing so, we guarantee an error is injected, significantly saving time by pruning faults known to be masked without simulating the application to completion. Figure 4 shows the flow of our RTL gate-level methodology.

Moreover, if an instruction-level hardware error detector is adopted (e.g., arithmetic residue checkers and codeword-based checkers for control logic), errors that would be detected should also be pruned since the injection outcome is known at this point (detected). Thus, if an error is detected, we should also keep injecting until an undetected one is generated to save time. Otherwise, similar to the cases of masked faults, it is wasteful to wait for the application running to completion because we collect no information during this waiting period. This filtering notion can be extended to software detectors that integrate fine-grained software state-recovery mechanisms such as Containment Domains [15].

B. Injecting Errors with Dynamic Binary Instrumentation

Our hierarchical injection framework includes an assembly-level injector which drills down to the RTL gate-level at a specific instruction instance. Specifically, we implement the assembly-level injector using Pin, a dynamic binary instrumentation tool [16]. 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 injection point to run at native speed, (b) it is more accurate than a compiler IR-level injector [17], (c) it allows mapping injected instructions back to the program source lines (i.e., directly pointing out which lines are problematic), and (d) it allows injection into specific binary and source code regions. Overall, dynamic binary instrumentation allows us to inject errors at a lower level (i.e., higher-fidelity) with acceptable execution overhead (see Section IV-B), at the same time providing application developers useful feedback regarding resilience from a higher-level point of view. Note that our methodology should work for other instruction-level injectors [18], [19], [20], [21], [22].

C. 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 using Pyverilog [23]. 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 simulate a fault. The real RTL gate-level simulation is done by Icarus Verilog, an open-source Verilog simulation tool [24]. We also support customized instruction-level error detection (Figure 4).

Note that our fault injection process has no dependency on any proprietary tools and the entire RTL gate-level injector is wrapped into a set of Python modules. With the help of the error context API, our RTL gate-level injector can be easily integrated with other instruction-level injectors as well.

D. Injection Workflow

The overall injection workflow consists of two phases: profiling and injection. The profiling phase derives the upper bound of dynamic instructions in the program. It also outputs the statistics for each binary and instruction type, which are useful for targeting specific binaries and instructions and necessary for modeling uniform-random particle-strike time.

At the injection phase, we adopt the Monte Carlo method to evaluate the impact of errors at the application level. Assuming soft errors affect each instruction with equal likelihood, in each

2See http://lph.ece.utexas.edu/users/hamartia/docs/pintool for all features.
Fig. 3: Correlations between bit-flip positions with RTL-level particle-strike model. Axes denote bit locations of circuit outputs. Breakdown into averaged correlations among all circuits (all) and individual circuits (integer adder, floating-point adder, and floating-point multiplier).

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 [26]. Note that these circuits are not pipelined (i.e., no internal latches). Although they can be different from those designed and optimized for commodity processors, we have shown that they lead to errors different from single-bit flips at the instruction level.

For this paper, we assume on-chip SRAM and system DRAM are protected by ECC and only inject errors to instructions that use arithmetic units. We evaluate the serial version of 9 benchmark programs (Table I). For each application and each error model, we perform 3000 injection experiments, which ensures a margin of error $<2\%$ for a confidence level of 95\% [27]. Therefore, 81,000 experiments in total are conducted. Note that our tool is capable of injecting MPI programs as well, but the evaluation is out of the scope of this work.

Injection outcomes are classified into these categories:

- **Masked**: the injected error is masked by application, with outputs identical to the error-free run.
- **Detected Uncorrectable Error (DUE)**: the injected error makes the program crash or hang (denoted as $DU_{crsh}$) or results in obviously erroneous application output such as infinite output or mismatch in matrix size (denoted as $DU_{test}$).
- **Silent Data Corruption (SDC)**: the program ends normally with hard-to-detect output error.

**E. Limitations**

We are not currently modeling microarchitecture-level error masking in our injector because the simulation is too slow. Although our Pin-based implementation limits us to x86 platforms, the framework can be easily generalized to others.

**IV. Evaluation**

**A. Experimental Settings**

In each experiment, we inject an error to a random instruction’s output operand with the following error models.

- **Single-bit flip (RB1)**: randomly flips a bit.
- **Double-bit flip (RB2)**: randomly flips two bits.
- **RTL gate-level model (RTL)**: generates an error pattern using the methodology described in Section III-A.

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 [26]. Note that these circuits are not pipelined (i.e., no internal latches). Although they can be different from those designed and optimized for commodity processors, we have shown that they lead to errors different from single-bit flips at the instruction level.

For this paper, we assume on-chip SRAM and system DRAM are protected by ECC and only inject errors to instructions that use arithmetic units. We evaluate the serial version of 9 benchmark programs (Table I). For each application and each error model, we perform 3000 injection experiments, which ensures a margin of error $<2\%$ for a confidence level of 95\% [27]. Therefore, 81,000 experiments in total are conducted. Note that our tool is capable of injecting MPI programs as well, but the evaluation is out of the scope of this work.

Injection outcomes are classified into these categories:

- **Masked**: the injected error is masked by application, with outputs identical to the error-free run.
- **Detected Uncorrectable Error (DUE)**: the injected error makes the program crash or hang (denoted as $DU_{crsh}$) or results in obviously erroneous application output such as infinite output or mismatch in matrix size (denoted as $DU_{test}$).
- **Silent Data Corruption (SDC)**: the program ends normally with hard-to-detect output error.

**B. Injector Overhead**

Recall that we disable instrumentation after the injection point, so the injector overhead depends on the injection point (i.e., a dynamic instruction). Thus, we measure the injector overhead at the median instance of all dynamic instructions. 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 quad-core Intel i5-6500 CPU and 16GB DRAM.
Per-experiment overhead compared with native runs are shown in Table I (averaged over five runs). In general, smaller applications can finish in a couple of seconds while larger HPC proxy applications within two minutes. Figure 5 shows the injection overhead compared with an optimized PINFI, another PIN-based error injector shown to incur less overhead than compiler-based injectors [17]. Our implementation outperforms PINFI even with RTL-level injection except for applications that can finish in 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 3X.

C. Execution Savings by Iterative Fault Generation

With Monte Carlo method, we need to collect at least 600, 1067, and 2400 injection outcomes to ensure the margin of error below 4%, 3%, and 2% respectively [27]. However, due to logical masking at the RTL-level, the required numbers of programs runs are actually higher. Figure 6 shows the saved execution runs owing to our RTL-level iterative injection method (Section III-A). Execution savings are calculated using \( \Sigma_{n=1}^{N} \text{iter}_n \times \text{dyninst}_n / \text{total dyninst} \), where \( N \) is the number of injection experiments, \( \text{iter}_n \) is the number of local iteration for the \( n \)th experiment, \( \text{dyninst}_n \) is the dynamic instance number of the \( n \)th experiment, and \( \text{total dyninst} \) is the dynamic instruction count of the application. On average, the iterative injection method saves 389, 662, 1476 program runs (or 65%, 62%, 62% of the required runs) for 4%, 3%, and 2% margin of error respectively. Notice that the stricter the margin of error is (i.e., more injections), the more benefits in terms of actual saved time. Furthermore, our preliminary experiments with detectors show orders-of-magnitude runtime improvement.

D. Application Behavior with Various Error Models

1) Outcome Distribution: Figure 7 shows the outcome distribution with the three error models. First, the difference of SDC rates between RB1 and RTL is small (the largest difference is 1.9% in CoMD), while the difference of DUE rates between the two models is higher (the largest difference is 3.9% in CG). Note that in LULESH the RTL model also leads to more DUE_{est}, which in this case are application output values that are not finite. Secondly, RB2 results in quite different distributions compared with RB1 and RTL. This is because RB2 constantly flips two uncorrelated bits as opposed to correlated bit-flipping with RTL. Therefore, RB2 is not a good approximation for RTL error model.

In summary, although on average 22% of errors from RTL model are correlated multi-bit flips (Figure 2), we observe similar SDC rates between single-bit flip and our RTL model. Thus, in terms of SDC rates, single-bit flip remains a good approximation of errors from arithmetic circuits. However, when the DUE rate is of interest (e.g., evaluating resilience techniques that aim at reducing the DUE rate), the high-fidelity errors should be considered.

Authorized licensed use limited to: Nvidia Corp. Downloaded on July 02,2020 at 16:50:46 UTC from IEEE Xplore. Restrictions apply.
TABLE II: p-values of chi-square tests for application output quality between error models ($\alpha = 0.05$).

<table>
<thead>
<tr>
<th>Program</th>
<th>RB1 vs. RB2</th>
<th>RB1 vs. RTL</th>
<th>RB2 vs. RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>0.00</td>
<td>0.60</td>
<td>0.00</td>
</tr>
<tr>
<td>cjpeg</td>
<td>0.02</td>
<td>0.73</td>
<td>0.04</td>
</tr>
<tr>
<td>LU_c8</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Swaptions</td>
<td>0.25</td>
<td>0.03</td>
<td>0.31</td>
</tr>
<tr>
<td>CG</td>
<td>0.36</td>
<td>0.05</td>
<td>0.74</td>
</tr>
<tr>
<td>MG</td>
<td>0.79</td>
<td>0.71</td>
<td>0.83</td>
</tr>
<tr>
<td>CoMD</td>
<td>0.27</td>
<td>0.88</td>
<td>0.23</td>
</tr>
<tr>
<td>LULESH</td>
<td>0.25</td>
<td>0.05</td>
<td>0.31</td>
</tr>
</tbody>
</table>

2) Application Output Quality: Recently researchers introduced tuning resilience overhead with application output quality [4], [28]. The idea is that if users can tolerate degradation of quality up to a certain degree, one can lower the cost of protecting applications against soft errors.

We thus compare the application output quality of SDC cases with different error models. To compare the distributions objectively, we first group the SDC output qualities into bins (Table I) and then perform chi-squared tests for each pair of error models and each application. 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 means the choice of error model is critical for evaluating application output quality. We choose the significance level ($\alpha$) to be 0.05. As a result, if the calculated p-value from a chi-squared test is less than 0.05, the observed data rejects the null hypothesis and indicates output quality differs significantly between the pair of error models. The results are shown in Table II.

We find that RB1 and RTL generate statistically different quality distributions with Swaptions and LULESH. The corresponding histograms are shown in Figure 8. This indicates the complex multi-bit error patterns can have impact on application output quality.

Nevertheless, we want to emphasize that the results may change depending on the choice of quality metric and how the qualities are binned. Also, some SDC samples may be viewed as $\Delta E_{out}$ in some use cases. In summary, these results show that when trading output quality for resilience overhead using techniques such as Approxilizer [28], one has to pay attention to how injection results are categorized and consider different error models accordingly.

V. RELATED WORK

Previous work also studies multi-bit errors and their corresponding protection techniques [34], [35], [36], [37]. Most of them focus on errors occur in on-chip SRAM. In this work, we assume those parts are already protected and thus focus on errors resulting from combinational logic of execution units. Note that our high-fidelity fault model for combinational logic is fundamentally different from those for on-chip SRAM errors.

Current fast injectors are based on assembly-level [38], [25], [17], compiler IR-level [18], [19], [39], [20], emulator-based [21], [40], and debugger-based [41], [42], [43] injection, but most of them focus on the single bit-flip error model. Sangcholje et al. [44] investigate the impact of data-independent multi-bit errors on applications. They find that single bit-flips result in SDC rates reasonably close to 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 applied to arithmetic circuits.

Prior work also tries to make high-fidelity injector faster [45], [5], [46], [47], [6], [11], [10]. For instance, GeFIN [46] proposes several acceleration features for fault injection at the micro-architecture level. Our iterative error generation methodology can be viewed as the combination of the checkpointing and early stop on invalid features in that work. In our case, we early stop when a fault is logically masked or when an error is detected, and then we rollback to the previous injection point (i.e., checkpoint) for next injection. Our methodology is orthogonal to acceleration methods that prune faults potentially leading to the same injection outcome [48], [49], [14].

Combining resilience techniques across multiple abstraction layers to achieve minimal cost has also been an active research topic [50], [51], [52]. Our methodology demonstrates how to take advantage of interaction between error injection and detection to save time in injection campaigns, enabling efficient evaluation of cross-layer resilience techniques.

VI. CONCLUSION

To strike a balance between injection speed and fidelity, we refine the concept of hierarchical injection to prune faults with known outcome, saving evaluation time significantly. Our implementation of the hierarchical error injector is not only accurate but also 3X faster than state-of-the-art error injectors. We show that single bit-flip remains a good approximation of a more realistic RTL model when SDC rate is the major concern. However, when trading off output qualities vs. resilience overhead, one should consider multiple error models, especially for a more realistic one. All in all, this work enables us and future researchers to rapidly evaluate application resilience with a higher-fidelity error injector.


