Fast Cross-Layer Vulnerability Analysis of Complex Hardware Designs

Joseph Paturel
Univ Rennes, Inria, CNRS, IRISA

Angeliki Kritikakou
Univ Rennes, Inria, CNRS, IRISA

Olivier Sentieys
Univ Rennes, Inria, CNRS, IRISA

***Abstract*—Simulation-based fault injection is commonly used to estimate system vulnerability. Existing approaches either partially model the fault masking capabilities of the system under study, losing accuracy, or require prohibitive estimation times. This work proposes a vulnerability analysis approach that combines gate-level fault injection with microarchitecture-level Cycle-Accurate and Bit-Accurate simulation, achieving low estimation times. Single and multi-bit faults both in sequential and combinational logic are considered and fault masking is modeled at gate-level, microarchitecture-level and application-level, maintaining accuracy. Our case-study is a RISC-V processor. Obtained results show a more than 8% reduction in masked errors, increasing more than 55% system failures compared to standard a fault injection approach.**

***Index Terms*—Fault injection, vulnerability analysis, RISC-V**

I. Introduction

Over the past few years, the Soft Error Rate (SER) of hardware designs has significantly increased due to reduced technology node sizes and lower supply voltages [1]. Circuits designed using modern deep-submicron technologies have become more and more sensitive to environmental sources [2], such as ionization, radiation, and high-energy electromagnetic interferences, leading to temporary reliability violations, called soft-errors. These sources can affect a memory cell of a design, corrupting its contents (flip the bit it contains). This type of fault is known as Single-Event-Upset (SEU). As technology node sizes have significantly reduced, several memory cells can be corrupted at the same time, leading to Multi-Bit-Upsets (MBUs) [3]. Moreover, when a combinational cell is affected, a logic transient, known as Single-Event-Transient (SET), is generated. The SET is propagated in the forward cone of the impacted cell, and it can eventually reach one or several flip-flops and corrupt their values. Until recently, soft errors occurring in memory circuits have been considered as the major contributors to the SER due to their large sizes and high-density, making them statistically more likely to be affected. However, soft errors occurring within logic circuits have become more and more important and combinational logic cannot be considered negligible anymore [4], [5].

As a result, the vulnerability of a design towards faults has to be evaluated as part of the development process. To do so, techniques that force the appearance of faults (fault injection) in different structures of the design are often employed. To obtain an accurate vulnerability analysis, the masking effects occurring at each layer of the design must be taken into account: from the *logical and timing masking* occurring at the gate-level, the *microarchitectural masking* introduced by the way computational blocks are arranged, up to the *application masking* resulting from the application tolerance to errors. To achieve that, fault injection has to be performed at a low layer, such as the gate-level. However, when complex designs are considered, prohibitive estimation times are required. As shown by our experimental results (Table VI-Section III), a gate-level fault injection on a RISC-V processor running an 4x4 matrix multiplication requires more than four months CPU time for a statistically representative vulnerability analysis.

To speed up the process, some approaches ignore the implementation details of the design. For instance, fault injection at the application level [6] is only able to alter the data held in the variables of the application under study. However, by ignoring the underlying executing hardware, this leads to less accurate vulnerability analysis [7]. The majority of approaches considering the hardware of the design focus only on single-bit faults occurring in the sequential logic [8]–[11]. For example, in the cross-layer vulnerability analysis flow of [11], the appearance and propagation of faults are modelled as SEUs, occurring in selected design regions. The flow uses an RTL simulator for a predefined number of cycles. Then, the state of the design is handled to a functional simulator for faster analysis. However, these approaches discard the presence of faults in combinational logic and the possibility that several bits of a design can be affected at the same time. In [12], several flip-flops of the design are simultaneously flipped through an almost exhaustive fault injection in the design registers, which requires high estimation time. Applying this approach to estimate the vulnerability of only the register file of the RISC-V processor for the 4x4 matrix multiplication application requires 4 hours 18 minutes. The technique presented in [13] considers faults occurring both in the sequential and the combinational logic. Authors use an Instruction-Set Simulator (ISS) to execute the application. To inject a fault, the ISS invokes a gate-level simulator, that injects the fault in a cell of the design. The corruptions, induced by the fault, are obtained and given back to the ISS. As a gate-level simulation is not used for each application cycle, the estimation time is reduced. However, ISSs do not consider the hardware implementing the instruction set and are unaware of the design registers. Therefore, they cannot model the microarchitecture-level masking of the design, reducing accuracy. Table I summarises the aforementioned representative works.

This work addresses the above limitations by proposing

Reference	Masking type			Fault model	
	Gate-level	µArch	Application	SEU	MBU
[6]	-	-	✓	✓	✓
[8]–[11]	-	✓	✓	✓	-
[12]	-	✓	✓	✓	✓
[13]	✓	-	✓	✓	✓
Proposed	✓	✓	✓	✓	✓

TABLE I: Comparison of representative state-of-the-art

a fast cross-layer vulnerability analysis method, applicable to entire complex hardware designs, providing statistically representative vulnerability analysis. It considers faults both in the sequential and combinational logic, as well as masking at the gate-level, microarchitecture-level and application-level. More precisely, the contributions of this work are:

- A vulnerability analysis method combining two fault injection campaigns at different hardware levels, i.e, gate-level and microarchitectural-level. The gate-level campaign is applied once per design. It statistically analyses realistic faults both at the combinational and sequential logic, providing a higher level of accuracy. It produces error patterns that depict the impact of gate-level masking. The microarchitectural-level campaign is applied per application under study. It is based on a fast Cycle-Accurate and Bit-Accurate (CABA) simulator, able to evaluate microarchitectural-level and application-level masking. The injection is driven by the gate-level error patterns.
- The population of SEUs and MBUs occurring due to realistic SETs on the combinational and sequential logic is analyzed by tuning the design operating frequency and SET pulse width. Results highlight the notable impact of MBUs on vulnerability analysis.
- A vulnerability analysis of a RISC-V processor is provided. The obtained results show, on average, a 8.32% reduction in masked faults, which leads to an increase in the number of crashes (57.81%), hangs (24.76%), and data corruptions (5.78%) compared to standard SEU-based approaches.

The paper is organized as follows. Section II presents the proposed vulnerability analysis method. Section III analyzes the experimental analysis results on a RISC-V processor. Finally, a discussion on the accuracy of this work as well as a conclusion and future works are presented in Section V.

II. Cross-Layer Vulnerability Analysis

To obtain a fast and accurate vulnerability estimation, we propose a cross-layer method based on two fault injection campaigns, at different hardware abstraction levels of the Design Under Test (DUT). Figure 1 depicts the overview of our method. The first campaign is performed at the *gate level* to statistically obtains error patterns independently of the application to be characterized. The error patterns describe the number of occurrences and the geometry of errors, due to SETs in the combinational logic and SEUs directly on the flip-flops of the DUT, considering gate-level masking. To evaluate the impact of these patterns on an application, considering microarchitectural-level and application-level masking, the second campaign is performed at the *microarchitecture level*, using a fast CABA simulator of the DUT microarchitecture.

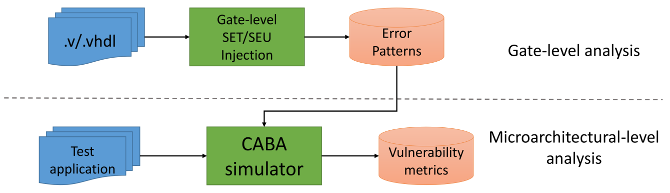

Fig. 1: Overview of proposed cross-layer vulnerability analysis

A. Single-Cycle Gate-Level Analysis

The aim of the gate-level analysis is to create statistical models of the SEU and MBU occurring due to soft-errors, in both combinational logic and flip-flops of the DUT. To do so, the netlist of the DUT is enhanced with an *injection block* (built with a XOR gate and a multiplexer), attached to the output of each netlist cell, allowing the modification of the cell's outputs. These injection blocks allow for the appearance of an SET anywhere and at any time in the netlist. To not affect the timing characteristics of the DUT, the propagation delays of all the injection blocks are set to zero.

Once the netlist modified, it is used by the fault injection campaign described in Algorithm 1. Since the emergence of SEUs and MBUs, propagated from SETs through the combinational logic, depends on the values of the DUT inputs, we explore N_{input} different sets of values as inputs. For instance, assuming that the DUT is the execution pipeline stage of a processor, these sets of values consist of instructions randomly selected from the processor's Instruction Set Architecture with random operands. For each new set of values (L.2), the DUT is executed without any fault (L.3) in order to obtain the golden reference (L.4), which will be used to detect errors. Then, N_{gl} faults are forced into the netlist for this input set (L.5). The number of faults N to be injected is defined based on the required confidence level of the statistical analysis as

Algorithm 1: Gate-level fault injection campaign

```
L1  for inputData in inputDataSets[0...$N_{input}$-1] do
L2      newInputSet();
L3      executeClockCycle();
L4      goldenReference = collectOutput();
L5      for ninj in [0...$N_{gl}$-1] do
L6          targetCell = selectNextCell();
L7          offset = random(0, Tclk);
L8          duration = getWidthFromDistribution(targetCell);
L9          wait(offset);
L10         flipCellOutput(targetCell);
L11         if isCellCombinational(targetCell) == true then
L12             wait(duration);
L13             flipCellOutput(targetCell);
L14         else
                // The target cell is a flipflop, the
                   error is latched
L15         waitForClockRisingEdge();
L16         error = compareDUTOutput();
L17         log(error);
```

Confidence	95% (t=1.96)	99% (t=2.5758)	99.8% (t=3.0902)
e = 5%	385	664	955
e = 1%	9604	16587	23874
e = 0.1%	960400	1658687	2387335

TABLE II: Number of faults required to be injected.

$N = \frac{t^2 \times p \times (1-p)}{e^2}$ (1), where t is the critical value related to the statistical confidence interval, e the error margin, and p the percentage of the possible fault population individuals that are assumed to lead to errors. p is usually set to 0.5 to maximize the sample size N. Eq. (1) is derived from [14], where the population size is set to $+\infty$ [15]. Table II presents the evolution of the number of faults to inject in the DUT in order to meet certain statistical characteristics. Hence, at the gate level, the fault injection campaign performs $N = N_{input} \times N_{gl}$ injections. During a fault injection cycle, the netlist cell (L.6), the time offset (L.7), and the duration of the fault to be injected (L.8), are selected. The cell that will be subject to injection is chosen function of its area – the bigger the cell, the higher its probability to be selected. The SET start time is selected randomly inside one clock cycle. The duration can be fixed, random, or provided by tools, that characterize the particles' impact over the netlist cells, e.g., MUSCA [16]. The campaign waits until the SET start is reached (L.9) and the injection block of the selected cell is activated, flipping its output (L.10). If the selected cell is combinational (L.11), an SET has been injected and has to remain active for the selected duration (L.12). After this duration has passed, it is deactivated (L.13). Otherwise, the injected fault is an SEU affecting directly a flip-flop cell (L.14). At the end of the cycle (L.15), the contents of the flip-flops provide the output, which is compared (through a XOR operation) with the golden reference (L.16). The errors and the number of times they have been observed are logged (L.17). With this information, we obtain the error patterns and their frequencies. They will be used at the microarchitecture-level injection campaign to perform an accurate vulnerability analysis. If the affected cell has several flip-flops in its forward cone, the SET propagation can lead to MBUs.

The gate-level analysis models the propagation of SETs during a single cycle. Hence, complex designs, that behave on a multi-cycle basis (such as pipeline circuits), have to be split into sections. Each section is a part of the design, where the output of combinational logic is computed and stored to flip-flops in a single cycle. To estimate the vulnerability of the complete design, the impact of the gate-level error patterns needs to be simulated on a platform, that is aware of the microarchitecture and able to accurately execute the application. To achieve that, we perform fault injection campaign at microarchitecture level, as described in next section.

B. Microarchitecture-Level Analysis

This analysis estimates the vulnerability of the complete DUT, taking into account microarchitecture-level and application-level masking. To achieve that, we extend a CABA simulator in order to inject the error patterns, collected at the gate level per DUT section, at the microarchitecture level.

The microarchitectural fault injection campaign is summarized in Algorithm 2. Initially, the application under study is executed on the DUT without faults in order to collect a set of golden references (L.1): i) application output, ii) system state (memory and registers), and iii) number of execution cycles. The campaign is executed N_{ml} times per application (L.2), each time injecting a fault in the DUT. In order to obtain statistically correct vulnerability estimations, the number of faults required to be injected is computed using Eq. (1).

During the application execution, the cycle to inject the fault is chosen randomly between the first cycle and the total number of cycles considering the fault-free execution (L.3). Then, the type of logic, where the fault is injected, is selected based on the area of the logic (L.4). For instance, if 40% of the DUT area is combinational logic, then it has a 40% chance of being chosen, and thus a 60% chance for the sequential logic. If the selected logic is sequential (L.5), an SEU is injected in a random flip-flop of the complete DUT (L.6), similar to common fault injection campaigns [8]–[10]. If the selected logic is combinational (L.7), the injected fault is a gate-level error pattern of a DUT section. The selection of the DUT section is area driven (L.8) – the larger the DUT section, the more chances it has to be selected. Then, an error pattern is selected as a function of its frequency – the more often a pattern has been observed during gate-level analysis, the higher the chances of being selected. The selected error pattern is injected (L.9) and the simulation is resumed. When simulation ends (L.10), the outcome belongs into one of the following error classes (L.11):

- Masked: The application has executed without any error or mismatch compared to the golden reference.
- Hang: The application has entered an infinite loop. A cycle counter is used to stop the current injection simulation, if the counted cycles exceed a threshold.
- Crash: The execution of the application has terminated unexpectedly and an exception has been raised (out of bound memory access, misaligned PC, hardware trap, etc.)
- Application Output Mismatch (AOM): Only the output of the application differs from the golden reference.
- Internal State Mismatch (ISM): Only the system state (memory and registers) differs from the golden reference.
- ISM & AOM: Both the application output and the system

Algorithm 2: Microarchitecture-level fault injection

```
L1  output, state, cycles = collectGoldenReference();
L2  for fault in [0...$N_{ml}$-1] do
L3      waitUntilInjectionCycle();
L4      logicType = selectLogicTypeWithArea();
L5      if logicType == sequential then
L6          injectSEUInRandomSeqStructure();
L7      else
L8          combSection = selectCombSectionWithArea();
L9          injectAnErrorPattern(combSection);
L10     waitForSimulationEnd(timeout);
L11     classifySimulationOutcome(output, state);
```

state differ from the golden references. Usually, an AOM will also present a mismatch in the memory of the design.

Note that, the AOM and ISM classes do not completely overlap. For instance, assume that the DUT is a processor executing a `for` loop. A fault affects the register that stores the iteration counter. As result, the execution skips an iteration. The application output will not be correct, but if the `for` loop has not written to the memory, the final state of the processor will be the same as the golden reference.

III. Experimental Results

To evaluate the proposed approach, a processor is used as DUT, i.e., the Comet processor [17] that implements the 32-bit RISC-V instruction set using a standard 5-stage pipeline. The processor is designed using a high-level C++ model, enabling a fast CABA simulator to be compiled from its description. The processor is synthesized to the gate-level using Mentor Graphics CatapultHLS and Synopsys Design Compiler with a target frequency of 650 MHz. The target technology is the 28 nm FDSOI design kit from ST-Microelectronics using the nominal corner with a supply voltage of 1.0 V. The critical path of is 1.53 ns, having a maximum operating frequency of 653.6 MHz. The technology node has been selected due to its low power characteristics, making this implementation of Comet akin to a low power microcontroller. The sequential logic represents 45.85% of the total processor area (including the register file), leaving 54.15% to the combinational logic. Table III depicts the area that each pipeline stage occupies. The write-back stage has the largest area, as it includes the register file, followed by the execution stage, which has the largest combinational logic.

Pipeline stage	**Fetch**	**Decode**	**Execute**	**Memory**	**WriteBack**
Total Area	6.01%	11.02%	35.47%	5.10%	42.41%
nb. cells	292	3,778	4,872	734	1,196
nb. Comb. cells	226	3,580	4,746	566	172
nb. Seq. cells	66	198	126	168	1,024

TABLE III: Area of each pipeline stage of the Comet processor

To obtain realistic fault models, the different cells of the design kit are analyzed using MUSCA [16]. Considering neutron injections and an LET set to 58MeV/cm, the peak of the SET pulse width distribution is 400 ps. Section III-A presents gate-level analysis and Section III-B the microarchitecture-level analysis for an SET pulse width equal to 400 ps and an operating frequency of 500 MHz. Section III-C analyses the impact of the processor frequency and the SET width.

A. Single-Cycle Gate-Level Analysis

To apply the gate-level analysis, the processor has been divided into sections, each being a pipeline stage. Due to page limitations, this section presents the results obtained for the execution stage. The results for the other stages are similar.

1) Gate-Level Error Patterns: The output of the gate-level analysis is a set of error patterns. Figure 2.a) shows the probability of each bit of the output register, i.e., the pipeline register between the execution and memory stages, to be erroneous.

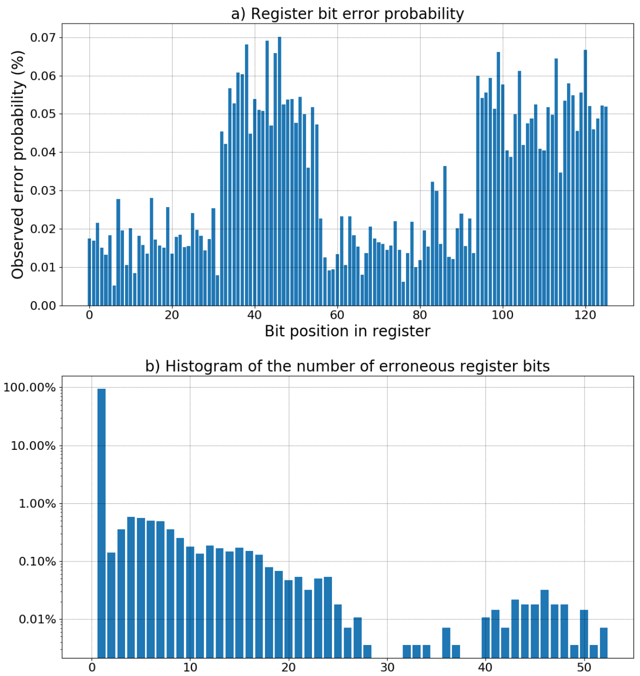

Fig. 2: Gate-level analysis results of the execution stage

The average probability for a bit to be erroneous is 0.033%. Some regions of the register exhibit higher error probabilities than others. Such a region is between bit 32 and bit 63, with an average error probability of 0.043% (30.9% above average). This region corresponds to the *result* field of the register, i.e., the ALU output. It is more susceptible to errors because: i) the ALU combinational cells have large forward cones, and thus, the SETs are propagated through multiple paths to the register, and ii) the ALU logic masking depends on the operation – a multiplication operation has lower masking factor compared to a logical AND operation. The second most susceptible region is between bit 94 and bit 125, which corresponds to data resulting from logical operations on values from Control and Status Registers (CSRs). The remaining register regions correspond to data forwarded from the previous pipeline stage. Fig. 2.b) shows the amount of patterns that exhibit a specific amount of corrupted bits. We observe that 5.12% of the error patterns have more than one bit concurrently erroneous, i.e., they describe MBUs. The higher number of erroneous bits observed in an error pattern is 52 faulty bits, i.e, 41.3% of the register bits is corrupted. Although these error patterns are latched in the register, they do not necessarily lead to an error on the application execution. To assess their impact, the microarchitecture-level analysis of the DUT is required.

2) Estimation time: Based on Eq. (1), considering a 99.8% confidence interval with 5% error margin for each input set values, $N_{input} = 10^3$ different set of values are used as inputs and $N_{gl} = 10^3$ faults are injected per input, leading to $N = 10^6$ injections in total. Questa Advanced Simulator 10.7b running on a 2nd generation Intel Xeon CPU is used for gate-level simulation. Table V shows the CPU time required to perform the gate-level fault injection for all pipeline stages.

Pipeline stage	Fetch	Decode	Execute	Memory	WriteBack
Est. Time (s)	37	392	496	82	32

TABLE V: Gate-level analysis time of each pipeline stage

B. Microarchitecture-Level Analysis

The gate-level analysis of each pipeline stage of the RISC-V processor provides the error patterns for each stage. To simulate the core, a CABA simulator was compiled from the C++ description of Comet, allowing the fault injection in any core register at any time. The simulator performance is 18.2 million instructions per second on average, using an 8^{th} generation Intel i7 CPU running at 2.40 GHz. We built a script around the CABA simulator to drive the microarchitecture-level campaign. Initially, the number of faults required to be injected is computed based on Eq. (1). Using a 99.8% confidence interval and a 1% error margin, the number of faults to inject is 23,874. Then, faults are injected in the microarchitecture and the simulation outcomes are classified, as described in Section II-B. Four applications are analyzed: `stringsearch`, `qsort` and `blowfish` from MiBench suite [18] and an 4x4 integer matrix multiplication.

1) Reliability results: The microarchitecture-level output is the distribution of observed errors to the error classes. Table IV compares the following vulnerability analysis methods:

- *MA(S)*: SEUs are injected at microarchitecture-level, with a uniform occurring probability for all registers, as in [8]–[10].
- *MA(S,A)*: SEUs are injected at microarchitecture-level, with a occurring probability proportional to pipeline stage area.
- *CL(S,M,A)*: the proposed cross-layer vulnerability analysis method, considering SEUs, MBUs and pipeline stage area.

Considering that the Comet processor is microarchitecturally close to the Rocket chip [19], the *MA(S)* results are coherent with those of [20], strengthening the statement that processors, sharing the same ISA, also share similar vulnerability analysis. The obtained results show, on average, that the proposed *CL(S,M,A)* approach leads to 8.32% less masked faults, which results in a significant increase of other classes: 57.81% more crashes, 24.76% more hangs and 5.78% more data corruptions, compared to the standard *MA(S)* approach. Contrary to the proposed approach, the *MA(S)* analysis cannot explore the combinational logic and gate-level masking. More precisely, when evaluating the `blowfish` application, the proposed methodology provides a more than 10% decrease in the number of masked faults (equivalent to 1,457 faults) compared to the *MA(S)* approach. This significantly increases by 63.4% the number of observed crashes, which can be explained by the control-dominated nature of `blowfish` application. For the `matmul` application, we observe the maximum increase in data corruptions (AOM and ISM&AOM classes), which is 19.23% compared to *MA(S)* approach. This behavior is due to the high computation and data-dominated nature of the application. Since the execution stage has the largest area, it has a high injection probability, and, thus, a high probability in corrupting the applications operations.To further evaluate our cross-layer vulnerability analysis approach, we compare it with the *MA(S,A)* approach. The goal is to explore the origin of the observed results, i.e., whether it is the use of error patterns to model MBUs or the use of area of the pipeline stage. Since the percentage of *MA(S,A)* masked faults is even higher compared to *MA(S)*, this implies that the origin is the MBUs modelled by our gate-level error patterns.

Application	Nb. cycles	Full gate-level	Proposed		
			Micro/al-level	Gate-level	Total
matmul	7096	188.23 days	388 min 40 sec	1039 sec	405 min 59 sec
qsort	6478	171.84 days	327 min 20 sec		365 min 59 sec
blowfish	3058991	81144.83 days	466 min		483 min 19 sec
stringsearch	1636693	43416 days	427 min 20 sec		444 min 39 sec
average	1177314	31230.22 days	406 min 32 sec		423 min 51 sec

TABLE VI: Comparison of estimation times

2) Estimation time: Table VI shows the estimation times to obtain the processor vulnerability for the: i) proposed cross-layer approach, and ii) gate-level fault injection campaign applied on the complete processor *(Full gate-level)* running the target application. The simulator (Section III-A) simulates a gate-level model of the full Comet processor at only 10.2 cycles per second. The smallest application (`qsort`) requires 6,478 cycles and the largest application (`blowfish)` 3,058,991 cycles, which correspond to simulation times of 171.84 and 81,144.83 days, respectively. The gate-level time of the proposed approach is 1,039 seconds of CPU time, as depicted in Table V. Since the same error patterns can be used to characterize several applications, the gate-level analysis is performed only once. Therefore, the overall estimation time of the proposed approach ranges between 366 minutes for `qsort` and 483 minutes for `blowfish`.

As a conclusion, the experimental results show that the *Full gate-level* approach requires prohibitive estimation time. On the contrary, the proposed cross-layer method requires significantly lower estimation time, while maintaining the accuracy of the analysis. This is achieved thanks to the gate-level analysis is applied only once per DUT, since the obtained error patterns are reusable at the microarchitecture-level, and because of the use of a fast CABA microarchitecture simulator.

C. Target operating frequency and SET pulse width impacts

In this section, we carry out a set of campaigns in order to explore the impact of the target operating frequency and

Application	matmul			qsort			blowfish			stringsearch			average		
Analysis method / Error class	MA(S)	MA(S,A)	CL(S,M,A)	MA(S)	MA(S,A)	CL(S,M,A)	MA(S)	MA(S,A)	CL(S,M,A)	MA(S)	MA(S,A)	CL(S,M,A)	MA(S)	MA(S,A)	CL(S,M,A)
Masked	72.97%	73.28%	67.4%	72.71%	73.27%	67.29%	60.85%	61.48%	54.75%	69.76%	70.46%	63.86%	69.07%	69.62%	63.32%
Crash	6.29%	5.55%	9.71%	7.05%	6.22%	10.92%	6.83%	6.16%	11.16%	7.24%	6.56%	11.45%	6.85%	6.12%	10.81%
Hang	1.54%	1.42%	2.36%	1.07%	1.16%	1.6%	2.14%	2.36%	2.46%	3.65%	3.51%	4.08%	2.1%	2.11%	2.62%
AOM	1.58	1.94%	2.1%	2.47%	2.47%	2.8%	0.06%	0.06%	0.04%	1.67%	1.82%	1.81%	1.45%	41.57%	1.69%
ISM	14	14.17%	14.33%	13.44%	13.72%	14.16%	3.48%	3.32%	3.34%	14.05%	14.06%	14.6%	11.24%	11.33%	11.61%
ISM & AOM	3.62	3.64%	4.1%	3.26%	3.16%	3.23%	26.64%	26.62%	28.25%	3.63%	3.59%	4.2%	9.29%	9.25%	9.95%

TABLE IV: Comparison of vulnerability analysis results

SET pulse width on MBUs population for the execution stage. Table VII and Table VIII show both the absolute number and the corresponding percentage of observed SEUs and MBUs, when varying the target operating frequency and the SET width, respectively. When frequency is varied, a fixed SET width of 400 ps is used. We observe that the percentage of faults leading to MBUs have small fluctuations, on average 5.24% of faults are MBUs. When SET pulse width is varied, the frequency is set to 500 MHz. We observe a similar behavior, i.e., on average 4.01% of faults are MBUs. Note that, when the pulse width is 100 ps and below the SETs are not often propagated within the latching window of the register. With a pulse width below 50 ps, no SETs are latched. These results show that MBUs represent a significant part of the error population, regardless of the operating frequency or SET pulse width. This observation, coupled to the impact of MBUs at the microarchitecture-level, imply that MBUs generated bu combinational logic, should not be neglected to obtain an accurate vulnerability analysis.

IV. Discussions

It is noteworthy that a full gate-level simulation of the design, running the application, could yield a better accuracy, but it would require prohibitive exploration time. Also note that, once an SET has been propagated to the flip-flops of its forward cone, the gate-level simulator effectively serves as a CABA simulator. Therefore, from this point and on, simulation can continue at a higher abstraction level, such as the microarchitecture level, making it faster than full gate-level simulators. The difference between the proposed approach and a complete gate-level injection campaign is that we replace the analysis of each application specific instructions, with a reusable statistically significant distribution of error patterns.

V. Conclusion and Future Work

The proposed cross-layer vulnerability analysis is based on two fault injection campaigns, carried out at different hardware abstraction levels. The gate-level analysis considers realistic soft-errors, occurring both at the combinational cells and the flip-flops cells. It is performed only once to reduce the overall analysis time. The obtained error patterns drive the fault injection at the microarchitecture level. The experimental results show less masked errors, compared to standard microarchitecture-level analysis, and significant reduction in estimation time, compared to full gate-level analysis methods. On top, although the number of MBUs required to be injected is small, they have significant impact on the error populations.

Freq.	200 MHz	300 MHz	400 MHz	500 MHz	600 MHz
SEU	9,308 (93%)	15,592 (96.3%)	23,613 (94.1%)	26,489 (94.9%)	30,919 (95.5%)
MBU	699 (7%)	599 (3.7%)	1,473 (5.9%)	1,429 (5.1%)	1,447 (4.5%)

TABLE VII: Occurred SEUs and MBUs w.r.t. frequency

SET	100 ps	200 ps	400 ps	500 ps
SEU	5,144 (97.6%)	10,529 (95.3%)	26,489 (94.9%)	33,449 (95.9%)
MBU	127 (2.4%)	755 (4.7%)	1,429 (5.1%)	1,432 (4.1%)

TABLE VIII: Occurred SEUs and MBUs w.r.t SET pulse width

As future work, we will focus on improving further the accuracy and the speed of the proposed methodology. Regarding accuracy, the precision of the gate-level analysis can be improved by considering that a single high-energy particle can impact to several combinational cells (creating multiple simultaneous SETs) on scaled technology nodes [3].

References

[1] E. Ibe *et al.*, "Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule," *IEEE Trans. on Elect. Dev.*, vol. 57, no. 7, pp. 1527–1538, 2010.

[2] S. Rehman *et al.*, *Reliable Software for Unreliable Hardware: A Cross Layer Perspective*. Springer, 1st ed., 2016.

[3] M. Ebrahimi *et al.*, "A layout-based approach for multiple event transient analysis," in *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, May 2013.

[4] N. N. Mahatme *et al.*, "Comparison of Combinational and Sequential Error Rates for a Deep Submicron Process," *IEEE Trans. on Nuclear Science*, vol. 58, pp. 2719–2725, Dec. 2011.

[5] R. Psiakis *et al.*, "NEDA: NOP Exploitation with Dependency Awareness for Reliable VLIW Processors," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 391–396, May 2017.

[6] B. Mutlu *et al.*, "Characterization of the Impact of Soft Errors on Iterative Methods," in *IEEE Int. Conf. on High Performance Computing (HiPC)*, pp. 203–214, Dec. 2018.

[7] J. Laurent *et al.*, "Fault injection on hidden registers in a risc-v rocket processor and software countermeasures," in *IEEE/ACM Design, Automation Test in Europe Conference (DATE)*, pp. 252–255, 2019.

[8] C. Mao *et al.*, "An Automated Fault Injection Platform for Fault Tolerant FFT Implemented in SRAM-Based FPGA," in *IEEE Int. System-on-Chip Conf. (SOCC)*, pp. 192–196, Sept. 2018.

[9] Y. Xie *et al.*, "An Automated FPGA-Based Fault Injection Platform for Granularly-Pipelined Fault Tolerant CORDIC," in *Int. Conf. on Field-Programmable Technology (FPT)*, pp. 370–373, Dec. 2018.

[10] J. Azambuja *et al.*, "A Fault Tolerant Approach to Detect Transient Faults in Microprocessors Based on a Non-Intrusive Reconfigurable Hardware," *IEEE Trans. on Nuclear Science*, vol. 59, pp. 1117–1124, Aug. 2012.

[11] N. J. Wang *et al.*, "Examining ace analysis reliability estimates using fault-injection," *SIGARCH Comput. Archit. News*, vol. 35, p. 460–469, June 2007.

[12] M. Wilkening *et al.*, "Calculating Architectural Vulnerability Factors for Spatial Multi-Bit Transient Faults," in *IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, pp. 293–305, Dec. 2014.

[13] C. Chang *et al.*, "Hamartia: A fast and accurate error injection framework," in *IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, pp. 101–108, 2018.

[14] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *IEEE/ACM Design, Automation Test in Europe Conference (DATE)*, pp. 502–506, April 2009.

[15] I. Tuzov *et al.*, "Accurate Robustness Assessment of HDL Models Through Iterative Statistical Fault Injection," in *European Dependable Computing Conf.*, pp. 1–8, Sept. 2018.

[16] G. Hubert *et al.*, "A generic platform for remote accelerated tests and high altitude SEU experiments on advanced ICs: Correlation with MUSCA SEP3 calculations," in *IEEE Int. On-Line Testing Symp.*, pp. 180–180, June 2009.

[17] S. Rokicki *et al.*, "What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications," in *38th IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, Nov. 2019.

[18] M. R. Guthaus *et al.*, "MiBench: A free, commercially representative embedded benchmark suite," in *Int. Workshop on Workload Characterization*, pp. 3–14, Dec. 2001.

[19] K. Asanović *et al.*, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, EECS Dept., Univ. of California, Berkeley, Apr 2016.

[20] H. Cho, "Impact of Microarchitectural Differences of RISC-V Processor Cores on Soft Error Effects," *IEEE Access*, vol. 6, pp. 41:302–313, 2018.