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
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.

### 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

<table>
<thead>
<tr>
<th>Reference</th>
<th>Masking type</th>
<th>Fault model</th>
</tr>
</thead>
<tbody>
<tr>
<td>[6]</td>
<td>-</td>
<td>SEU, MBU</td>
</tr>
<tr>
<td>[8]-[11]</td>
<td>-</td>
<td>SEU, MBU</td>
</tr>
<tr>
<td>[12]</td>
<td>-</td>
<td>SEU, MBU</td>
</tr>
<tr>
<td>[13]</td>
<td>✔</td>
<td>SEU, MBU</td>
</tr>
<tr>
<td>Proposed</td>
<td>✔</td>
<td>SEU, MBU</td>
</tr>
</tbody>
</table>

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

![Fig. 1: Overview of proposed cross-layer vulnerability analysis](image)

**Algorithm 1: Gate-level fault injection campaign**

```plaintext
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
L15.     // The target cell is a flipflop, the error is latched
L16.     waitForClockRisingEdge();
L17.     error = compareDUTOuput();
L18.     log(error);
```
TABLE II: Number of faults required to be injected.

<table>
<thead>
<tr>
<th>Confidence</th>
<th>95% (t=1.96)</th>
<th>99% (t=2.5758)</th>
<th>99.8% (t=3.0902)</th>
</tr>
</thead>
<tbody>
<tr>
<td>e = 5%</td>
<td>385</td>
<td>664</td>
<td>955</td>
</tr>
<tr>
<td>e = 1%</td>
<td>9604</td>
<td>16587</td>
<td>23874</td>
</tr>
<tr>
<td>e = 0.1%</td>
<td>960400</td>
<td>1658687</td>
<td>2387335</td>
</tr>
</tbody>
</table>

\[ N = \frac{t^2 \times p \times (1-p)}{e^2} \] (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 campaigns 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**

```plaintext
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 Catapult HLS 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.

The write-back stage has the largest area, as it includes the register file, followed by the execution stage, which occupies the largest combinational logic. Table III depicts the area that each pipeline stage occupies. Due to page limitations, this section presents the results obtained for the execution stage.

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

<table>
<thead>
<tr>
<th>Pipeline stage</th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>WriteBack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Area</td>
<td>6.01%</td>
<td>11.02%</td>
<td>35.47%</td>
<td>5.10%</td>
<td>42.41%</td>
</tr>
<tr>
<td>nb. Seq. cells</td>
<td>66</td>
<td>198</td>
<td>126</td>
<td>168</td>
<td>1,024</td>
</tr>
<tr>
<td>nb. Comb. cells</td>
<td>292</td>
<td>578</td>
<td>4,872</td>
<td>734</td>
<td>1,196</td>
</tr>
<tr>
<td>nb. Seq. cells</td>
<td>226</td>
<td>5,850</td>
<td>4,746</td>
<td>566</td>
<td>172</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Pipeline stage</th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>WriteBack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Est. Time (s)</td>
<td>37</td>
<td>392</td>
<td>496</td>
<td>82</td>
<td>32</td>
</tr>
</tbody>
</table>

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 8th 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 an 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

<table>
<thead>
<tr>
<th>Application</th>
<th>matmul</th>
<th>qsort</th>
<th>blowfish</th>
<th>stringsearch</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error class</td>
<td>Masked</td>
<td>Crash</td>
<td>AOM</td>
<td>ISM</td>
<td>ISM &amp; AOM</td>
</tr>
<tr>
<td>MA(S)</td>
<td>72.7%</td>
<td>8.29%</td>
<td>1.58</td>
<td>14.07%</td>
<td>3.62%</td>
</tr>
<tr>
<td>MA(S,A)</td>
<td>73.28%</td>
<td>5.53%</td>
<td>1.94%</td>
<td>14.17%</td>
<td>3.64%</td>
</tr>
<tr>
<td>CL(S,M,A)</td>
<td>67.4%</td>
<td>9.11%</td>
<td>2.1%</td>
<td>14.33%</td>
<td>4.1%</td>
</tr>
<tr>
<td>MA(S)</td>
<td>67.29%</td>
<td>10.92%</td>
<td>2.8%</td>
<td>14.16%</td>
<td>4.3%</td>
</tr>
<tr>
<td>MA(S,A)</td>
<td>61.48%</td>
<td>11.16%</td>
<td>2.36%</td>
<td>14.06%</td>
<td>4.2%</td>
</tr>
<tr>
<td>CL(S,M,A)</td>
<td>54.75%</td>
<td>8.24%</td>
<td>2.36%</td>
<td>14.05%</td>
<td>4.2%</td>
</tr>
<tr>
<td>MA(S)</td>
<td>61.48%</td>
<td>11.16%</td>
<td>2.36%</td>
<td>14.06%</td>
<td>4.2%</td>
</tr>
<tr>
<td>MA(S,A)</td>
<td>70.46%</td>
<td>11.24%</td>
<td>2.36%</td>
<td>14.05%</td>
<td>4.2%</td>
</tr>
<tr>
<td>CL(S,M,A)</td>
<td>63.38%</td>
<td>11.24%</td>
<td>2.36%</td>
<td>14.05%</td>
<td>4.2%</td>
</tr>
<tr>
<td>MA(S)</td>
<td>69.07%</td>
<td>6.14%</td>
<td>2.36%</td>
<td>14.05%</td>
<td>4.2%</td>
</tr>
<tr>
<td>MA(S,A)</td>
<td>69.07%</td>
<td>6.14%</td>
<td>2.36%</td>
<td>14.05%</td>
<td>4.2%</td>
</tr>
<tr>
<td>CL(S,M,A)</td>
<td>63.38%</td>
<td>11.24%</td>
<td>2.36%</td>
<td>14.05%</td>
<td>4.2%</td>
</tr>
</tbody>
</table>

TABLE VI: Comparison of estimation times

<table>
<thead>
<tr>
<th>Application</th>
<th>matmul</th>
<th>qsort</th>
<th>blowfish</th>
<th>stringsearch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Est. Time (s)</td>
<td>3,058,991</td>
<td>19,812,722</td>
<td>1,184,838</td>
<td>163,669,3</td>
</tr>
<tr>
<td>Full gate-level</td>
<td>483 min 19 sec</td>
<td>303 min 9 sec</td>
<td>405 min 59 sec</td>
<td>1,039 seconds</td>
</tr>
<tr>
<td>Micro-level</td>
<td>242 min 20 sec</td>
<td>1177314</td>
<td>41.57%</td>
<td></td>
</tr>
<tr>
<td>Gate-level</td>
<td>406 min 32 sec</td>
<td>67.29%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As a conclusion, the experimental results show that the **Full gate-level analysis** 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
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 by 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 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.

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