HAL open science

Impact of Transient Faults on Timing Behavior and Mitigation with Near-Zero WCET Overhead

Pegdwende Romaric Nikiema, Angeliki Kritikakou, Marcello Traiola, Olivier Sentieys

To cite this version:

Pegdwende Romaric Nikiema, Angeliki Kritikakou, Marcello Traiola, Olivier Sentieys. Impact of Transient Faults on Timing Behavior and Mitigation with Near-Zero WCET Overhead. ECRTS 2023 - 35th Euromicro Conference on Real-Time Systems, Jul 2023, Vienna, Austria. pp.1-22, 10.4230/LIPIcs.ECRTS.2023.15 . hal-04397374

HAL Id: hal-04397374

https://hal.science/hal-04397374v1

Submitted on 16 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Impact of Transient Faults on Timing Behavior and Mitigation with Near-Zero WCET Overhead

Pegdwende Romaric Nikiema
Univ Rennes, Inria, IRISA, CNRS, France

Angeliki Kritikakou
Univ Rennes, Inria, IRISA, CNRS, France

Marcello Traiola
Univ Rennes, Inria, IRISA, CNRS, France

Olivier Sentieys
Univ Rennes, Inria, IRISA, CNRS, France

Abstract

As time-critical systems require timing guarantees, Worst-Case Execution Times (WCET) have to be employed. However, WCET estimation methods usually assume fault-free hardware. If proper actions are not taken, such fault-free WCET approaches become unsafe, when faults impact the hardware during execution. The majority of approaches, dealing with hardware faults, address the impact of faults on the functional behavior of an application, i.e., denial of service and binary correctness. Few approaches address the impact of faults on the application timing behavior, i.e., time to finish the application, and target faults occurring in memories. However, as the transistor size in modern technologies is significantly reduced, faults in cores cannot be considered negligible anymore. This work shows that faults not only affect the functional behavior, but they can have a significant impact on the timing behavior of applications. To expose the overall impact of faults, we enhance vulnerability analysis to include not only functional, but also timing correctness, and show that faults impact WCET estimations. As common techniques to deal with faults, such as watchdog timers and re-execution, have large timing overhead for error detection and correction, we propose a mechanism with near-zero and bounded timing overhead. A RISC-V core is used as a case study. The obtained results show that faults can lead up to almost 700% increase in the maximum observed execution time between fault-free and faulty execution without protection, affecting the WCET estimations. On the contrary, the proposed mechanism is able to restore fault-free WCET estimations with a bounded overhead of 2 execution cycles.

2012 ACM Subject Classification General and reference → Reliability; General and reference → Measurement; Hardware → Error detection and error correction; Hardware → Transient errors and upsets; Hardware → Safety critical systems; Computer systems organization → Real-time system architecture

Keywords and phrases Transient faults, Timing impact, Near-zero WCET error detection and correction, Vulnerability analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.15

Supplementary Material *Software (Source Code)*: `https://gitlab.inria.fr/srokicki/Comet/-/tree/FSR_comet?ref_type=heads`

Funding This work has been funded by the French National Research Agency (ANR) through the FASY research project (ANR-21-CE25-0008).

1 Introduction

1.1 Context

Time-critical systems, such as safety-critical and mixed-criticality systems, consist of hard real-time applications. For such applications, timing guarantees must be provided, i.e., their worst-case response time must be less than their respective deadlines and/or the total

© Pegdwende Romaric Nikiema, Angeliki Kritikakou, Marcello Traiola, and Olivier Sentieys;
licensed under Creative Commons License CC-BY 4.0
35th Euromicro Conference on Real-Time Systems (ECRTS 2023).
Editor: Alessandro V. Papadopoulos; Article No. 15; pp. 15:1–15:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

execution does not exceed a given latency requirement [47]. To rigorously provide such guarantees, a safe estimation of the Worst-Case Execution Time (WCET) [20] has to be employed during the system design. The estimation of WCET is performed through (i) measurement-based approaches, where the task is executed on the platform under study, and (ii) static analysis approaches, where the software source code and the platform under study are automatically examined, before the application runs [16]. The majority of WCET estimation approaches assumes that the underlying hardware is fault-free, i.e., during WCET estimation no faults occur in the hardware of the target platform [24, 48].

However, in reality, a system is threatened by phenomena that can lead to several permanent or temporary faults, occurring during execution. Especially due to the reduced transistor sizes and lower supply voltages of modern technologies [26, 17], systems are becoming more and more sensitive to environmental sources [39], such as ionization, radiation, and high-energy electromagnetic interference, leading to temporary reliability violations, called transient faults. Transient faults can affect the system behavior by corrupting the system information. Therefore, as systems become more and more prone to faults during execution [21], fault-free WCET bounds cannot be considered safe anymore [48].

1.2 Motivation

To deal with hardware faults, existing approaches apply fault-tolerant techniques to the system. The majority of these works focuses on the impact of faults on the *functional behavior* of the applications. Functional behavior refers to *denial of service*, i.e., no outcome is generated because the application is hanged or crashed, and to *binary correctness*, i.e., the application outcome is different than expected [40].

Fault mitigation considering real-time aspects is usually achieved through scheduling techniques applied at the task-level, such as replication of tasks [27, 5] and task check-pointing/re-execution [12, 51, 50, 27]. When fault tolerance techniques are inserted into the system, their timing impact on WCET has to be taken into account, in order to still provide timing guarantees. To do so, the fault-free WCET is extended with the timing overhead of the applied fault tolerance techniques. However, faults impact not only the functional behaviour, but also the *timing behaviour*, i.e., the application finishes within a given time, but its execution time is different compared to the fault-free execution. This fault impact is bound by the denial of service, i.e., when the execution time exceeds a threshold, it is considered as not responsive. Such application hangs are detected by a watchdog-timer and they are remedied by resetting the system and restarting execution. However, such approaches have significant time overhead, since the transient fault is detected much later than occurred, e.g., when the application finishes execution or the watchdog timer expires. To deal with this limitation, low-level fault-tolerant techniques can circumvent the fault impact, at the time instance when the fault occurs, leading to remedies with significantly lower and bounded time overhead than watchdog timers and less area overhead than replicating the complete processor.

Few approaches address the impact of hardware faults on the timing behaviour of applications. Existing work addresses hardware faults occurring in cache memories, while the rest of the architecture is assumed fault-free. Approaches focus on estimating the timing impact by accounting for the hardware degradation of the cache memory due to the presence of faults, e.g., additional misses due to faulty cache lines [23]. Some approaches have been extended to incorporate the timing impact of inserted fault tolerant techniques to detect, correct or mitigate faults in memories, e.g, when a parity bit is used for error detection [11]. Other works focus on mitigating the hardware degradation in caches, due to occurring faults,

using redundant hardware, e.g., through a shared reliable buffer [24]. As a result, the timing impact of faults on the execution time, and thus the WCET, is mitigated and the timing characteristics of the memory hardware are maintained, leading to a timing behavior close to the fault-free one, despite the presence of faults. However, existing works mainly focus on permanent faults occurring to memories. Nevertheless, with technology size reduction, faults occurring inside the cores cannot be considered negligible anymore [32, 43]. Such faults can significantly affect the execution time of an application.

1.3 Contributions

The contribution of this work is to expose the following key aspect: transient faults affecting the cores impacts not only the functional behavior of an application, but it also has a significant impact on its timing behavior, affecting WCET estimations. To achieve that, we leverage typical fault-free WCET estimations to be fault-aware, by taking into account the impact of transient faults occurring on cores. More precisely, we firstly perform a vulnerability analysis on a target system through extensive fault injection. The analysis verifies not only functional correctness, but also timing correctness of applications, when executed on a core. Then, we apply a typical measurement-based WCET estimation method to verify the impact of faults on WCET estimation. A RISC-V core, named Comet, is used as a case study [41]. Comet is an on open-source High Level Synthesis (HLS) implementation of RV32I base ISA[1].

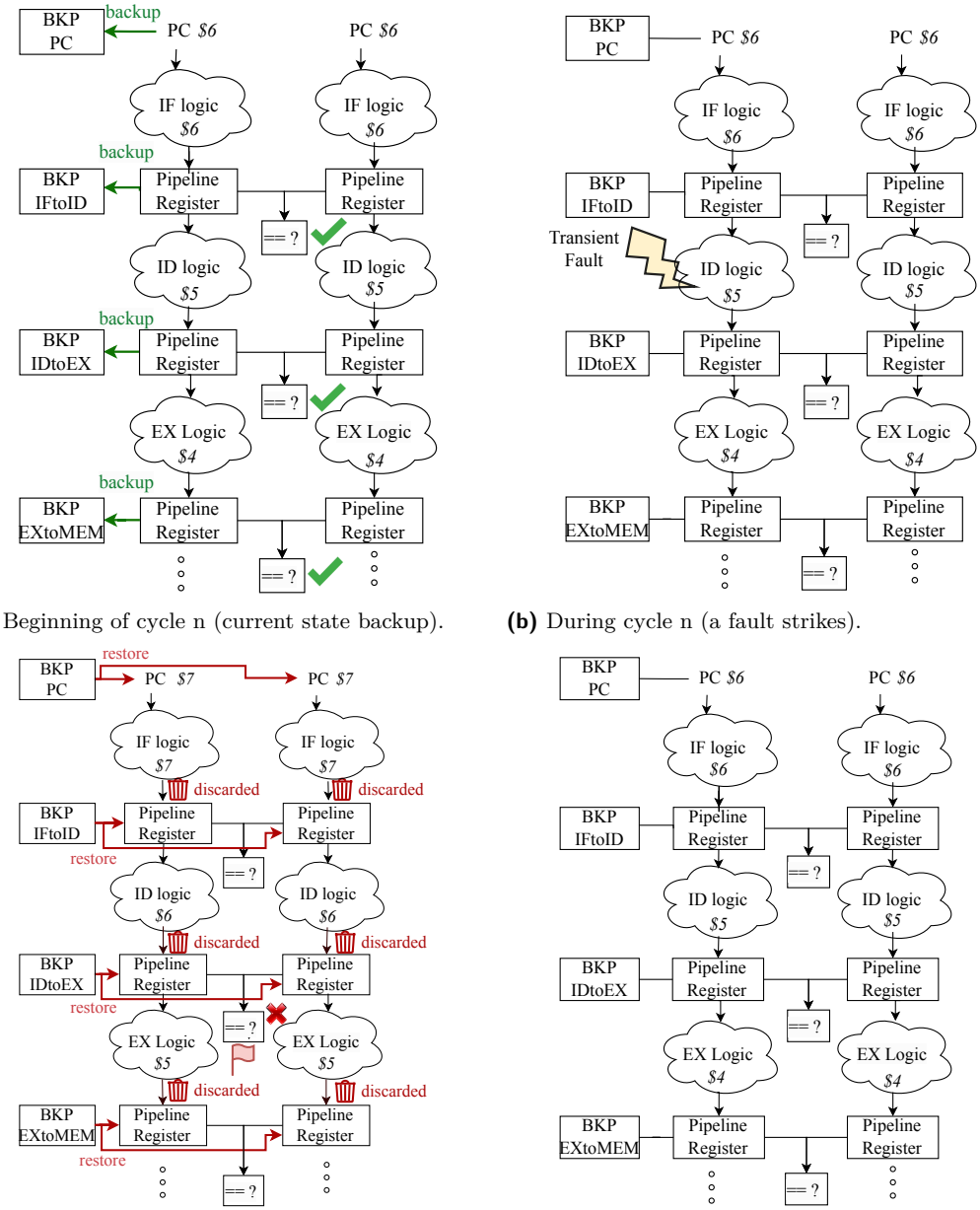
From the obtained results, we observe that the application execution time can be significantly increased under the presence of transient faults, up to 700%, compared to the application execution time without faults. Furthermore, the distribution of execution time traces is significantly modified, compared to the fault-free distribution. The above observations have direct consequences; the time required to finish execution under faults can be significantly higher than the fault-free WCET. Thus, existing approaches should use watchdog timers, in order to bound the impact of transient faults on the application execution time, and keep safe the overall schedule. When the timer expires or an error is detected, the application requires to be re-executed, fully or partially, depending on the approach, leading to high error detection and correction timing overhead. To deal with this limitation, we propose a mechanism with near-zero and bounded overhead (two clock cycles) that circumvents the faults as soon as they occur – before being propagated and affecting the execution time – and thus restores WCET estimations close to the faulty-free one.

The paper is organized as follows. Section 2 describes the methodology followed to obtain fault-aware WCET estimations, based on functional and timing vulnerability analysis combined with a measurement-based WCET approach. Section 3 describes the proposed fault-tolerant mechanism and bounds its timing overhead. Section 4 presents and analyzes the experimental results. Section 5 discusses the related work. Finally, conclusion is presented in Section 6.

2 Fault-aware WCET estimation methodology

This section describes the methodology followed to obtain WCET estimations under transient faults occurring on cores. To obtain realistic fault analysis, hardware fault injection is needed. Thanks to this, faults can be injected in the actual hardware structures, and not only in application variables as done by software fault injection [37]. Hence, a measurement-based WCET estimation method is required in order to be able to analyze the timing impact, when faults are injected in the hardware, compared to a static analysis. Therefore, firstly we perform a vulnerability analysis through hardware fault injection. Then, we apply a

[1] https://gitlab.inria.fr/srokicki/Comet/-/tree/master

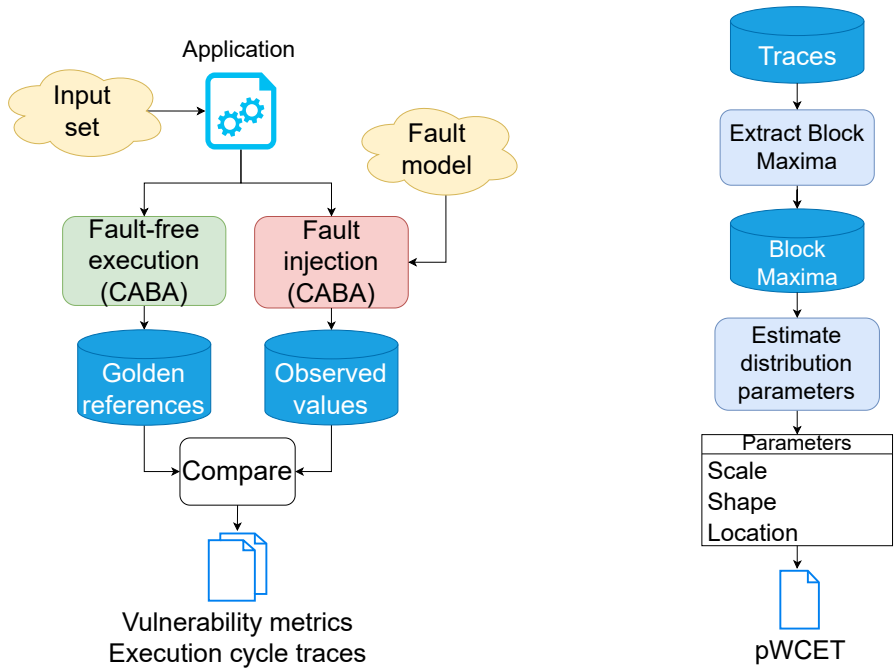


(a) Beginning of cycle n (current state backup). **(b)** During cycle n (a fault strikes).

(c) Beginning of cycle n+1 (error detection & restore). **(d)** Cycle n+2 (error corrected).

Figure 1 Illustration of LESR mechanism.

typical Measurement Based Probabilistic Timing Analysis (MBPTA) to analyze the impact of faults on WCET. The MBPTA is a mathematical method for estimating the extreme values probability of rare events [18, 13]. This method allows us to see the tail behaviour and determine the probabilistic WCET (pWCET) for a set of execution time traces. Note that, the goal of the fault-aware WCET estimation methodology is not to propose a new method to obtain tighter bounds, but to study typical measurement-based WCET estimation approaches in presence of faults. The next paragraphs describe the steps of the fault-aware WCET estimation methodology.



(a) Data collection through vulnerability analysis. **(b)** pWCET estimation flow.

Figure 2 Overview of fault-aware WCET estimation methodology.

2.1 Data collection through vulnerability analysis

During the data collection step, we need to obtain the execution cycles describing the timing behaviour of the application, under transient faults occurring on the core.

To achieve that, we design a *functional and timing vulnerability analysis* and study the impact of transient faults to the functional and timing correctness of an application executed on the core. This is performed through a Cycle-Accurate Bit-Accurate (CABA) simulator, where transient faults are injected based on a given fault model at the pipeline registers of the core. In order to expose the timing impact of faults, we need to monitor any difference between the execution cycles, required for the fault-free execution, and the observed execution cycles under the presence of faults. Therefore, we remove any other source that may lead to variation of the application execution cycles [16], i.e., the application is executed in isolation, with the caches disabled and the initial state of the processor are forced to be the same among executions. Figure 2a illustrates the data collection step. Prior to any fault injection, we execute the application under study with a given set of input data without faults, in order to obtain a set of golden references: i) the application output, ii) the system state (core registers), and iii) the number of cycles required for the execution of the application with the given set of input data. Then, the core simulator is extended with fault injection capabilities in order to execute the application and to inject faults, based on the considered fault model, to the registers, while the application runs. The cycle to inject the faults is chosen randomly between the first cycle and the total number of cycles needed for the fault-free execution for the given set of data. The location, where the faults are injected, is driven by the size of the logic of each pipeline stage. The larger the area, the higher its probability to be selected. After the fault injection and upon application termination, the observed results are compared to the golden references to categorize the impact of faults as:

- *Execution Cycles Mismatch (ECM):* The execution cycles of the application are different than those of the golden reference.
- *Hang (H):* The execution time of the application has exceeded a waiting threshold, and thus, it is assumed that it has entered an infinite loop. A cycle counter is used to stop the current execution, when the counted cycles exceed the given threshold.
- *Crash (C):* The execution of the application has terminated unexpectedly and an exception has been thrown (out of bound memory access, misaligned PC, hardware trap, etc.)
- *Application Output Mismatch (AOM):* The application output is different than the golden reference.
- *Internal State Mismatch (ISM):* The system state (registers) are different than the golden reference.
- *Functionally Masked (FM):* The application has finished execution, with no AOM and no ISM.

By using the aforementioned vulnerability analysis with several random inputs, we obtain the required set of execution cycle traces under faults to be used for the WCET estimation.

2.2 Data grouping, distribution fitting and pWCET estimation

After the collection of the execution cycle traces under faults, the next step is to group the data, so as to select the tail values, perform distribution fitting and estimate the pWCET, as illustrated in Fig 2b.

To select the tail values, we use the Block Maxima (BM) approach, one of the two common methods used, along with Peak-Over-Threshold. Following the BM approach, the data collected from the vulnerability analysis are grouped into blocks of equal size. Note that, grouping of data is performed in the order the values have been collected, without applying any shuffling or sorting. Then, the maximum value is picked from each data block to obtain the BM block, to be used for the distribution fitting. The most commonly used distributions for pWCET estimation are Weibul, Gumbel and Frechet [13], and our approach currently uses the Gumbel distribution, as it is one of the most representative ones [45].

Note that, the way the data is grouped affects the distribution fitting, which affects the pWCET estimation. Selecting a big block size may result into having very few values in the BM block, while selecting a small block size may result into taking into account all the values, some of those may not be representative values as tail values. The proposed approach performs block size exploration in order to select the best representative size considering the Gumbel distribution. In order to qualify the fitting of the distribution, we use the Kolmogorov-Smirnov (KS) test to get the p-value and the ks-statistic value of BM block. The KS test compares the Cumulative Distribution Function (CDF) of the empirical data with the CDF of the theoretical distribution. The p-value tests the null hypothesis H_0 that the data came from the fitted distribution. With a significance level of $\alpha = 0.05$, the H_0 can be rejected, meaning that the data does not come from the fitter distribution, if the p-value is bellow α. However, if the p-value is higher than the significance level, the H_0 cannot be rejected. The ks-statistic value is the maximum absolute difference between the two CDFs, the smaller the value the better the fit. Thus, we select the configuration with the smallest ks-statistic value that does not reject the hypothesis of having a Gumbel distribution, as the most fitting configuration.

The selected configuration gives the distribution parameters, such as the scale (σ), the location (μ) and the shape (ξ). These values are used, along with a given threshold value, to derive the maximum value that we can observe using the Percent Point Function (PPF) (inverse of cdf – percentiles).

3 Fault-tolerant mechanism with near-zero WCET overhead

This section describes the proposed fault-tolerant mechanism with near-zero WCET overhead based on *Lock-step Execution and Shadow Register (LESR)* and reports the upper bound of the error detection and correction time.

Overview. Figure 1 illustrates the proposed LESR mechanism. Two identical cores are working in lock-step, executing the same instruction at each clock cycle. Each pipeline stage stores the result of its logic computation in a pipeline register. The error detection and correction logic is the following: in each clock cycle, we compare the pipeline registers of the two cores, containing the results of the computation of the previous cycle. If no error is detected, all the pipeline registers are copied to a BacKuP copy (BKP) and the execution continues normally (Figure 1a). Otherwise, if a fault impacted the logic during the cycle (Figure 1b) or the pipeline register itself, a wrong result is stored in the register. In this case, a flag is raised, the results of the current computation are discarded and the pipeline registers of both cores are restored with the values in BKP (Figure 1c). In this way, in the next cycle, the pipeline re-executes the cycle that was impacted by a fault (Figure 1d).

To illustrate the proposed mechanism with a simple example, let us consider the C code of listing 1. Listing 2 depicts the assembly code snippet that corresponds to the subtraction ($1 – $4), multiplication ($5 – $6) and the addition ($7 – $9) instructions, considering a RISC-V core with 5 pipeline stages, i.e., Fetch (F), Decode (D), Execute (EX), Memory (MEM) and WriteBack (WB), as the one used in our case study in Section 4.

Listing 1 C program.

```
#include <stdio.h>
int a = 10; int b = 20; int c = 0; int d = 0;
int main() {
    d = a-b;
    c = a+b*4;
    return 0;
}
```

Listing 2 Assembly code of illustration example program.

```
0001018c <main>:
                        ---
    $1 101a8:   lw   a4,a(r0)          ;load word (variable a)
    $2 101ac:   lw   a5,b(r0)          ;load word (variable b)
    $3 101b0:   sub  a5,a4,a5          ;subtraction operation
    $4 101b4:   sw   a5,d(r0)          ;store word (variable d)
    $5 101b8:   lw   a5,b(r0)          ;load word (variable b)
    $6 101bc:   slli a5,a5,0x2         ;logical left shift by 2
    $7 101c0:   lw   a4,a(r0)          ;load word (variable a)
    $8 101c4:   add  a5,a4,a5          ;addition
    $9 101c8:   sw   a5,c(r0)          ;store word (variable c)
                        ---
```

Table 1 (left part) shows a snapshot of the processor pipeline stages during a fault-free execution of this program. Let us suppose that, at the end of cycle $n-1$, the computation had no errors. As highlighted in the right part of Table 1, at the beginning of cycle n the pipeline registers are compared and no error is detected; hence, the content of the pipeline is

Table 1 Pipeline status for Listing 2 example.

	Fault-free execution					Execution under faults with LESR				
Pipeline stage	**n-1**	**n**	**n+1**	**n+2**	**n+3**	**n-1**	**n**	**n+1**	**n+2**	**n+3**
F	$5	$6	$7	$8	$9	$5	$6	$7	*$6*	$7
D	$4	$5	$6	$7	$8	$4	$5	$6	*$5*	$6
EX	$3	$4	$5	$6	$7	$3	$4	*$5* ⚑	*$4*	$5
MEM	$2	$3	$4	$5	$6	$2	$3	$4	*$3*	$4
WB	$1	$2	$3	$4	$5	$1	$2	$3	*$2*	$3

copied to the BKP registers. Let us now suppose that a transient fault impacts the D stage logic during cycle n. In cycle $n+1$, the pipeline registers of the two cores are compared and an error is detected, due to the fault in cycle n. In detail, the error is detected by comparing input registers of stage EX, which are also output registers of stage D. Thus, the results of the computations are discarded and the content of BKP is copied back. Finally – in cycle $n+2$ – the cycle impacted by the fault can be re-executed and the computations goes back to normal.

Bound WCET overhead. The LESR approach entails a constant overhead of two clock cycles, namely the cycle where the fault occurred, and the cycle where the fault is detected and the values of the core registers are restored from the BKP registers, for processors with hardware function units that require one cycle to execute the instruction. Further discussion is provided in Section 4.3.

4 Evaluation for RISC-V case study

4.1 Experimental setup

Our case study is Comet, an open-source HLS 32-bit RISC-V processor [41], which supports the RV32I base ISA[2]. Note that, by using HLS, a unique high-level synthesis and simulation C++ model is used to design the processor. The model is used to generate both the hardware target design through High-Level Synthesis, as well as a Cycle-Accurate Bit-Accurate (CABA) simulator through software compilation. The processor consists of a standard 5-stage pipeline, including a forwarding mechanism and a register file with 32 registers in the write-back stage, as illustrated in Figure 3. Table 2 depicts the area of each pipeline stage of the core.

Table 2 Area of RISC-V pipeline stages.

Pipeline stage	Fetch	Decode	Execute	Memory	WriteBack
Area	6.01%	11.02%	35.47%	5.10%	42.41%

The LESR approach has been implemented in the RISC-V CABA simulator, and it is available in `FSR_comet` branch from the Comet repository. We have enhanced both the unprotected and the protected version of the RISC-V core with hardware fault injection capabilities. The used fault model is a bit-flip. A framework based on pythons scripts has been designed in order to perform the data collection with and without fault injection, obtain

[2] `https://gitlab.inria.fr/srokicki/Comet/-/tree/master`

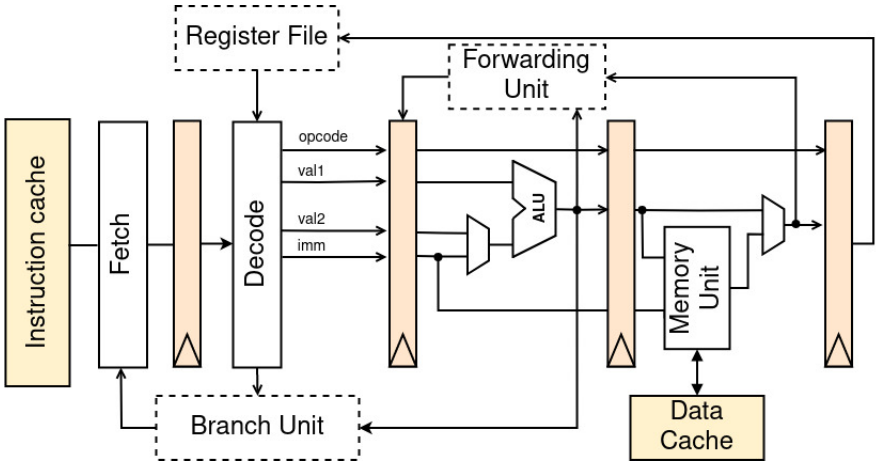


Figure 3 RISC core with 5-stage pipeline, forward mechanism, and data and instruction caches [41].

the vulnerability metrics and execution cycle traces, perform the data grouping, distribution fitting and pWCET estimation. Note that, the threshold for considering that an application is not responsive is set to eight times the execution cycles without faults.

In this first step towards the exploration of the impact on the execution time and WCET estimation of transient faults occurring inside the processor, we used as benchmarks typical kernels, applied in many application domains, such as multimedia, automotive, image processing etc. The goal is to first explore the fault impact on the kernels, before dealing with more complex applications. Five benchmarks with different complexities and execution cycles have been analyzed. More precisely, `Binary Search (BS)` searches an index in a sorted array of a size equal to 15 and `Prime` checks whether two input integers are prime or not. Both benchmarks are taken from the TACLeBench suite. `Qsort` sorts the elements of an array of size 10 and its implementation is inspired from MiBench. `Moving Average (MA)` makes the average of nearby pixels of an 8x8 matrix and is inspired from AxBench. `Matmult` multiplies two 4x4 matrices and it is taken from Polybench. The app, kernel, sequential and test benchmarks from TACLeBench, except those with floating point operations, have been successfully compiled and executed on the proposed lockstep version and fault injection campaigns will be performed in the future. The source code of the benchmarks is available in the `FSR_comet`[3] branch of the Comet repository. For the data collection step, based on [13], we use 650 different inputs for each benchmark, in order to obtain the data for the benchmark timing behavior, leading to 650 fault-free executions per benchmark. The inputs are generated by selecting each integer randomly between the integer range $[INT_{MIN}, INT_{MAX}]$, except for `Prime`, where we used positive numbers.

For the estimation under faults, note that, exhaustive fault injection is not computationally possible, due to the prohibitive number of fault injection points during the execution of an application. The different fault injection points are given by the number of different register bits of the processor and the number of cycles required for the fault-free benchmark execution. Thus, the vulnerability analysis is based on statistical fault injection, as in the the state-of-the-art approaches. The number of faults N to be injected in order to have statistically confident results is defined based on the required confidence level of the statistical analysis

[3] `https://gitlab.inria.fr/srokicki/Comet/-/tree/FSR_comet/tests/basic_tests`

as $N = \frac{t^2 \times p \times (1-p)}{e^2}$, 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 [29, 53]. With $p = 0.5$, we obtain the maximum number of faults to be injected in order to have statistically confident results, considering infinite number of fault injection points. Based on the above formula, we have injected 250,250 faults per benchmark, which lead to results with a 99.8% confidence interval and a 0.3% error margin. More precisely, we have injected 385 faults per different input, providing 5% confidence interval and a 5% error margin [53] for each input, considering 650 different inputs. Note that, to keep the collected data independent and identically distributed, we keep the maximum clock cycle observed out of the 385 injections on every input generated to be used for the pWCET estimation.

4.2 Experimental Results

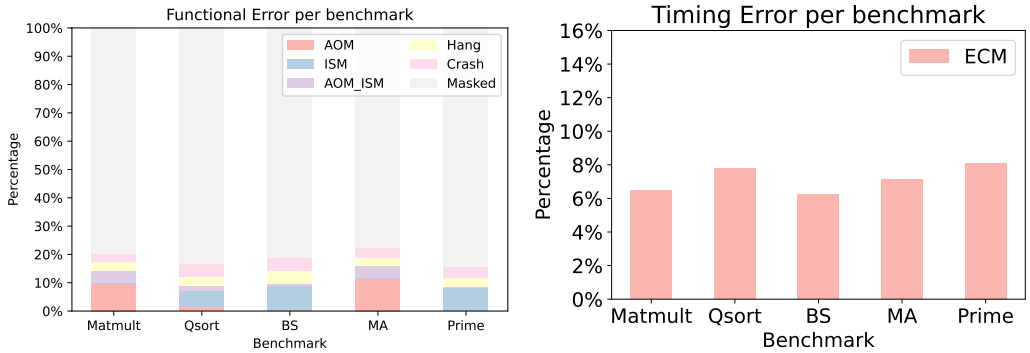
This section presents the execution cycle traces, the best selected configuration for the BM and the WCET estimation for: i) the unprotected version without faults, as currently done in the State-Of-the-Art, ii) the unprotected version under faults, and iii) the protected version using the LESR mechanism under faults. Furthermore, we provide the functional and timing vulnerability metrics, as discussed in Section 2.1, for the last two set-ups.

Table 3 shows the each vulnerability metric for the unprotected version under faults in absolute values and Figure 4 schematically illustrates the corresponding percentages. For instance, for the `Matmult` benchmark, 2.5% of the fault injections has led to application hangs, 2.72% to application crashes, 8.31% to wrong output, 0.32% to wrong internal state, 3.85% to both wrong application output and wrong internal state, and 82.28% were masked. Similar are the results for the rest of the benchmarks. On average, 3.02% of the fault injections has led to application hangs, 3.45% to application crashes, 3.95% to wrong output, 4.16% to wrong internal state, 1.99% to both wrong application output and wrong internal state, and 83.43% were masked. Regarding timing correctness, all benchmarks experienced mismatches in their number of execution cycles. More precisely, the benchmark affected the least is `Binary search`, where 6.25% of the total benchmark executions, under the presence of faults, lead to a different number of execution cycles compared to the fault-free execution. The most affected benchmark is `Prime`, where 8.10% of the benchmark executions under faults lead to ECM. On average, 7.14% of the executions under faults lead to ECM among all benchmarks. For the protected version with LESR, mechanism, all faults have been corrected.

Figures 5, 6, 7, 8 and 9 show the distribution of execution cycles for the five benchmarks. In each figure, the subfigures (a), (b) and (c) correspond to unprotected version without faults, the unprotected version with faults and the protected version with faults, respectively. For the experimental set-up with faults, the distribution shows the execution cycles for $250,250$

Table 3 Functional and timing vulnerability metrics (absolute value).

Benchmark	AOM	ISM	AOM & ISM	Hang	Crash	Masked	ECM
`Qsort`	3,284	12,573	4,216	7,424	10,501	212,252	19,429
`Prime`	107	18,868	262	7,582	8,497	214,934	20,276
`BS`	883	18,975	1,548	10,387	9,719	208,738	15,646
`Matmult`	20,800	816	9,637	6,257	6,829	205,911	16,232
`MA`	24,351	894	9,102	6,105	7,671	202,127	17,832



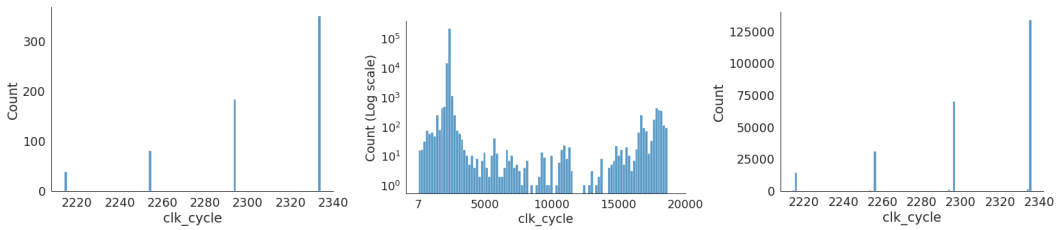
(a) Functional correctness. **(b)** Timing correctness.

Figure 4 Functional and timing Errors for the five benchmarks under study.

executions (excluding the Crash and Hangs cases for the unprotected version). Note that, for the unprotected version as the value variations are high, the histogram is presented in logarithmic scale. The overall observation among all benchmarks is that, when faults impact the unprotected core, the distribution is modified significantly, both in shape and location, as shown by Figures 5b, 6b, 7b, 8b, and 9b. Note that, the x-axis for the unprotected version under faults is significantly larger than the unprotected version without faults and the protected version with faults. Furthermore, the high peak observed in the unprotected version under faults corresponds to the execution cycles obtained for the executions where the faults have been masked.

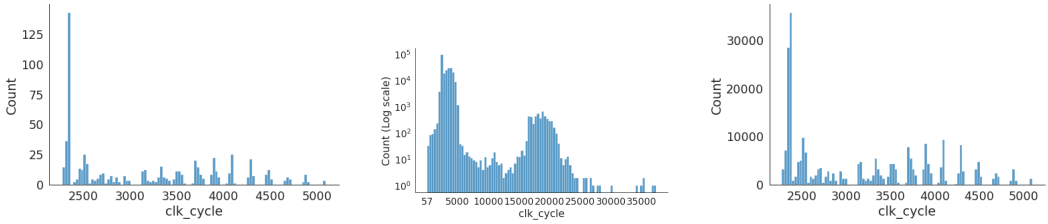
Let's further analyze this impact using the `Binary search`, which is the simplest benchmark. The execution time of binary search depends on the position of the index of the sorted array and it is upper bounded by $log_2(M)$, with M the size of the array. This statement is in line with the observations during the experiments, as `Binary search` searches in an array of 15 elements, and thus, 4 different values are observed during the 650 executions, as depicted in Fig. 5a. However, when faults are injected in the unprotected version, the distribution of collected execution traces is significantly modified. On the contrary, the protected version, using the proposed LESR mechanism, it is able to maintain a distribution very close to the original one under faults, i.e., the number of execution cycles is increased by two cycles.

To illustrate the applied methodology, Figure 10a depicts the histogram of the BM block, along with the Gumbel distribution, and Figure 11a the Quantile-Quantile plot for `Matmult` benchmark, which is one of the benchmarks with higher complexity. We observe a rather good resemblance to the line $x = y$, which means that the collected data follows the Gumbel



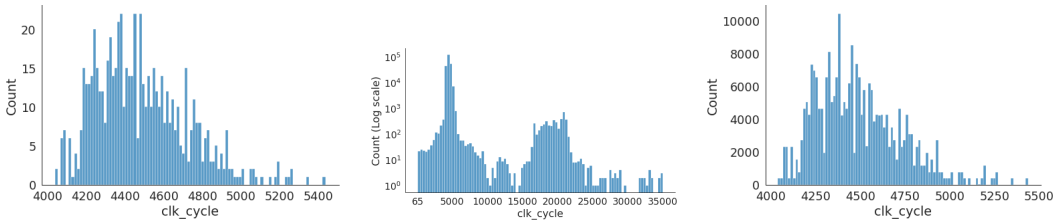
(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

Figure 5 `Binary search`: Collected data regarding execution cycles for all processor versions.



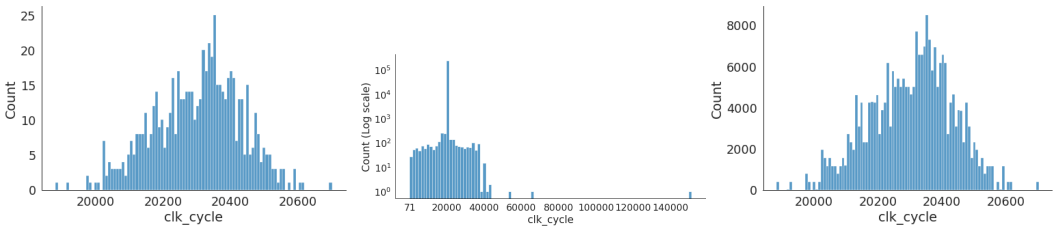
(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

Figure 6 `Prime`: Collected data regarding execution cycles for all processor versions.



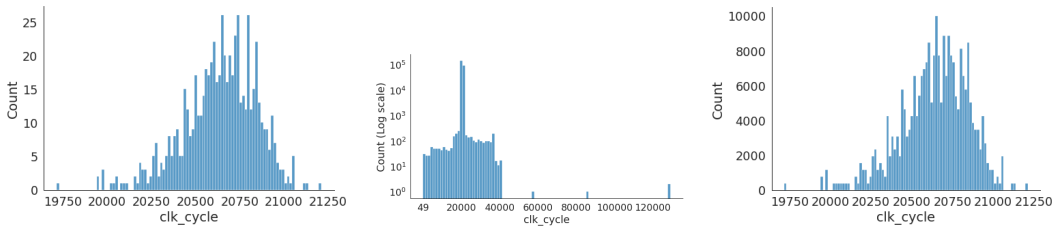
(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

Figure 7 `Qsort`: Collected data regarding execution cycles for all processor versions.



(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

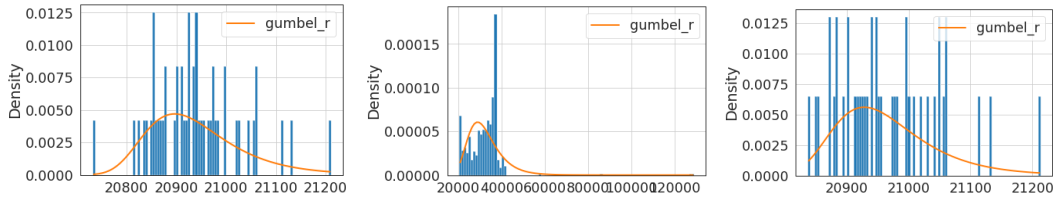
Figure 8 `Moving Average`: Collected data regarding execution cycles for all processor versions.



(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

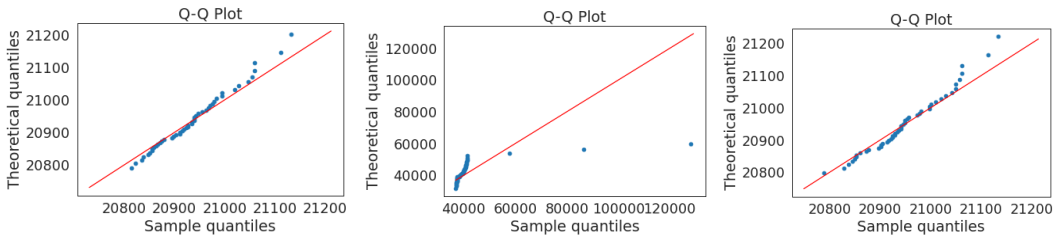
Figure 9 `Matmult`: Collected data regarding execution cycles for all processor versions.

distribution. However, when faults are injected in the unprotected version, the shape of the BM histogram is modified (Figure 10b), as shows Figure 11b. On the contrary, the protected version with LESR is able to keep the shape of the distribution similar to the fault-free distribution (Figure 10c) and obtain a similar fitting (Figure 11c). Table 4 shows the best configuration obtained during experiments.



(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

Figure 10 `Matmult:` Block Maxima and Gumbel distribution for all processor versions for the best configuration.



(a) Unprotected without faults. **(b)** Unprotected under faults. **(c)** Protected under faults.

Figure 11 `Matmul:` Q-Q plot of the distribution for all processor versions for the best configuration.

Table 4 Best BM configuration per version and benchmark.

Benchmark	Binary search	Prime	Qsort	Moving Average	Matmult
Unprotected without faults					
Number of blocks	3	108	650	44	50
Block size	217	6	1	15	13
Unprotected under faults					
Number of blocks	24	18	39	5	3
Block size	28	36	17	130	217
Protected under faults					
Number of blocks	3	81	217	50	41
Block size	217	8	3	13	16

Table 5 illustrates the pWCET estimation, using the best configuration shown in Table 4, and the maximum observed value during experiments, for all versions and benchmarks. The red (green) color highlights pWCET estimations that have a lower (higher) value than the maximum observed one. As long as the pWCET is lower than the maximum observed value, we increase the threshold until we are able to obtain an estimation higher than the maximum observed during experiments. From Table 5, we observe that typical WCET estimation approaches are able to tightly bound the unprotected version without faults.

However, when faults impact the processors, such methods provide less tight bounds with respect to the maximum observed value. This is due to the impact of the faults in the execution cycle distribution, which modifies the shape and the location. Overall, we observe that the difference between the pWCET and the maximum observed value is higher for the majority of the benchmarks. Furthermore, this difference is more significant as the complexity of the benchmark increases, e.g., as shown by the difference that progressively increases with the benchmark complexity, from `Binary Search` to `Matmult`. On the contrary, the proposed LESR protection mechanism is able to restore the execution cycle distribution close to the fault-free distribution, with an impact in the number of execution cycles equal to two. As a result, we are able to obtain pWCET estimations similar to the fault-free execution. Furthermore, the two cycles difference can be observed in the difference between the maximum observed value of the unprotected version without faults and the maximum observed value of protected version under faults.

4.3 Discussion

For the lock-step processor core that we implemented based on Comet [41], the proposed fault-tolerant mechanism entails a number of additional cycles bounded to two (one cycle to detect the fault and one to restore the correct pipeline register values), as confirmed by the experimental results. For other processor versions, the bound of two cycles will hold for similar cores, where the function units require one cycle for the instruction execution. To support a processor with the different extensions, capable of executing more complicated instructions in hardware, two approaches exist, i.e., insert a hardware function unit or implement multi-cycle operations sharing existing function units. In the first case, the proposed mitigation approach will be applied without modifications. Note that, different execution cycles will still be observed in the fault-free execution for applications that have different execution paths, which are selected based on data values. In the second case, the multi-cycle instruction is broken down into small control steps and is expressed as Finite State Machines (FSM). Each state of the FSM corresponds to a computation cycle. For instance, in the case of the multiplication, there is a state for each bit (or group of bits) in the operand. Note that, when a multi-cycle opcode enters the execution stage, the pipeline will be stalled until the FSM has reached its final state and the result is produced. We expect that this behavior will not jeopardize the fact that the proposed approach is bounded. To implement the proposed approach on a processor with a multi-cycle operation, a shadow register is required to be added in the internal register that accumulates the partial results. If the proposed approach is applied as it is, the bound is expected to increase from two cycles to the number of cycles required for the instruction, in the worst case. Therefore, there is a trade-off in the processor design between the overhead of inserting an additional hardware function unit and the overhead of the fault recovery approach.

As future work, we will leverage the proposed approach for different extensions of the RISC-V core and perform extensive fault injection campaigns for more complex applications. We expect that the results will be of similar nature, in the sense that, the more complex the application is, the more execution paths we expect to have, and thus, more execution cycle traces are expected.

Table 5 pWCET estimation (cycles) based on the best fitting configuration for different threshold value, the maximum observed cycle and their difference (%).

Benchmark		threshold			Max observed
		0.9	0.99	0.999	
Unprotected without faults					
Binary Search	Cycles	2,334	2,334	2,334	2,334
	Difference (%)	0	0	0	
Prime	Cycles	4,972	5,976	6,962	5,093
	Difference (%)	-2.37	17.33	36.39	
Qsort	Cycles	4,811	5,267	5,715	5,436
	Difference (%)	-11.50	-3.11	5.13	
Moving Average	Cycles	20,592	20,702	20,810	20,700
	Difference (%)	-0.52	≈ 0	0.53	
Matmult	Cycles	21,073	21,257	21,438	21,211
	Difference (%)	-0.65	0.22	1.07	
Unprotected with faults					
Binary Search	Cycles	18,696	18,826	18,953	18,671
	Difference (%)	0.13	0.83	1.51	
Prime	Cycles	34,873	45,176	55,291	37,381
	Difference (%)	-6.70	20.85	47.91	
Qsort	Cycles	34,444	43,480	52,351	35,085
	Difference (%)	-1.82	23.93	49.21	
Moving Average	Cycles	107,264	163,318	218,353	151,384
	Difference (%)	-29.14	7.88	44.24	
Matmult	Cycles	170,521	267,413	362,545	129,179
	Difference (%)	31.79	107	180	
Protected under faults					
Binary Search	Cycles	2,336	2,336	2,336	2,336
	Difference (%)	0	0	0	
Prime	Cycles	5,047	5,959	6,855	5,095
	Difference (%)	-0.94	16.95	34.54	
Qsort	Cycles	4,984	5,387	5,783	5,438
	Difference (%)	-8.35	-0.94	6.34	
Moving Average	Cycles	20,582	20,686	20,788	20,702
	Difference (%)	-0.58	-0.08	0.41	
Matmult	Cycles	21,077	21,231	21,383	21,213
	Difference (%)	-0.64	0.08	0.80	

5 Related Work

The state-of-the-art, relevant to our work, concerns i) real-time approaches for WCET estimation and fault-tolerant techniques under the presence of faults, ii) lock-step techniques, with focus on RISC-V related implementations, and iii) vulnerability analysis approaches. Table 6 summarizes the related work using the following criteria:

1. Hardware faults under study are Permanent Faults (PF) or Transient Faults (SE).
2. Hardware faults under study impact the Memory (M) or Core (C) of the target platform.

3. The Functional Behaviour (FB) or Timing Behaviour (TB) of the applications is checked. Functional behaviour refers to Denial of Service (DS), i.e., no outcome is generated because the application hanged or crashed, and to Binary Correctness (BC), i.e., the application's outcome is different than expected [40]. Timing behaviour refers to an application execution time that is different than the fault-free execution, due to a hardware fault.
4. Vulnerability analysis is performed through SoftWare (SW) or HardWare (HW) fault injection or placing the platform under Radiation Beam (RB).
5. WCET estimation assumes that hardware faults do not have a timing impact on execution, i.e., Fault-Free (FF), or not, i.e., Fault-Aware (FA).

Table 6 Summary of related work and positioning.

	Fault model		**Fault location**		**Fault detection**			**Vulnerability analysis**			**WCET estimation**	
Ref.	**PF**	**TF**	**M**	**C**	**DS**	**BC**	**TB**	**SW**	**HW**	**RB**	**FF**	**FA**
[52, 22, 1, 16, 57, 44]											√	
[14, 3, 36, 25, 12, 54]		√		√		√					√	
[5, 27, 50]		√		√	√						√	
[19]	√			√	√						√	
[35]	√	√		√	√	√					√	
[23]	√		√									√
[48]	√		√			√						√
[24, 9, 2]	√		√			√						√
[11, 10, 49]	√	√	√			√						√
[56]		√		√	√	√		√				
[33, 34, 30]		√		√		√	√	√				
[38, 55, 8, 37, 4]		√		√	√	√			√			
[28]		√		√	√	√	√		√			
[6]		√		√	√	√						
[46, 31]		√		√	√	√			√			
[59]		√	√	√	√	√				√		
[15, 58]		√	√	√	√	√			√	√		
This work		√		√	√	√	√		√			√

Regarding WCET estimation approaches, the estimation is performed through safe measurements, based on application execution, or static analysis of the programs [16]. For instance, a number of static analysis methods have been proposed, such as [52, 22], focusing on caches, and measurement-based approaches, such as [16, 44, 1]. A more detailed description of WCET estimation methods and tools is available in surveys, such as [57]. The majority of existing approaches does not consider faults, since the hardware of the target platform is assumed to be fault-free, during WCET estimation [24, 48].

To protect the system from faults, real-time approaches apply fault tolerant techniques. The faults under study usually lead to application execution failure or to erroneous outputs. To deal with these issues, the majority of real-time approaches focus on fault mitigation, through scheduling techniques applied at the task-level, such as replication of tasks [14] and task check-pointing/re-execution [19], while the fault detection is assumed to be performed by the hardware. When fault techniques are inserted to the system, their timing impact on WCET has to be taken into account, in order to still provide the timing guarantees.

To do so, existing approaches extend the fault-free WCET with the time overhead of the applied fault tolerant techniques. Works analyse this overhead by exploring how the applied fault tolerant technique impacts schedulability and providing schedulability analysis, e.g., for task replication techniques [5, 3, 36, 25] and task re-execution/check-pointing techniques [12, 27, 50]. Probabilistic worst-case schedulability analysis are also presented, e.g., for active and passive replicas [35]. Last, other works consider faults are rare events, and thus, the WCET should not consider the time overhead for recovery to avoid over-dimensioning the system, and fault recovery is modeled as an overshoot [54]. The above works can have significant time overhead, since the transient fault is detected very late, potentially after fault-free WCET bound is exceeded.

Few approaches address the impact of hardware faults on the timing behaviour of applications. Existing works focus on hardware faults in cache memories, while the rest of the architecture is assumed fault-free [7]. Approaches focus on estimating the timing impact, by accounting for the hardware degradation due to the presence of faults. For instance, static analysis probabilistically quantifies the WCET impact of permanent faults at instruction caches. The probability of an SRAM cell to be faulty is used to evaluate the additional cache misses that may occur [23]. A measurement-based approach for permanent faults occurring to caches provides the WCET impact, when cache lines are disabled due to faults [48]. Such approaches have been extended to incorporate the timing impact of inserted fault tolerant techniques to detect, correct or mitigate faults. For instance, the computation of the worst-case additional misses, due to defected cache lines, and the use of a parity bit for error detection [11]. Static probabilistic timing analysis is performed with fault detection mechanisms that periodically checks caches for faults and disable faulty cache blocks, under permanent faults [9] and also soft errors [10]. The maximum delay, introduced by error detection and correction codes, is computed in [49]. Other approaches focus on mitigating the hardware degradation, due to occurring faults, using redundant hardware. As a result, the timing impact of faults on WCET is mitigated and the timing characteristics of hardware are maintained, leading to WCET estimations close to fault-free WCET ones, despite the presence of faults. For instance, timing analysis considers a reliable victim cache to replace faulty entries [2], an extra reliable cache way per set and a shared reliable buffer [24]. Existing works mainly focus on permanent faults occurring to memories. Nonetheless, with technology size reduction, faults inside the processors cannot be considered negligible anymore [32].

Regarding vulnerability estimation approaches, existing approaches mainly focus only on estimating the functional correctness of the system under study. To achieve that, they apply fault injection at the software level and at the hardware level or put the device under radiation. Software fault injection is hardware agnostic. It is capable of flipping bits only in the data structures of the application [33, 34, 56, 30]. To improve accuracy, vulnerability analysis approaches have to consider the hardware details and perform bit-flips [38, 55, 8, 4, 37, 15, 46, 31]. Other approaches place the platform under radiation to analyze its behavior [15, 59]. However,the majority of existing vulnerability approaches focus on functional behaviour, i.e., checking for functional interruptions and erroneous values of the system under study [56, 38, 55, 8, 4, 37, 15]. However, not only the functional behaviour, but also the timing behaviour must be taken into account during vulnerability analysis for safety-critical systems. Few recent studies explore the impact of soft errors on the timing behaviour. They use software fault injection and their application domain is limited to iterative methods, e.g., the performance impact is given by the number of iterations required for iterative solvers to converge [34, 33] and their execution time [30], and hardware fault injection [28] using a single input. However, such approaches focus on the average behavior, neglecting WCET aspects.

Other fault tolerant approaches exist, which, however, do not focus on WCET aspects. Regarding lock-step execution, it can be based on *non-intrusive* and *intrusive* approaches. *Non-intrusive* approaches do not modify the processor architecture, and are typically used when the internal architecture details are hidden or difficult to modify, e.g., Commercial Off-The-Shelf (COTS) processors. For instance, lockstep approach uses ARM A9 as hard core and RISC-V as soft core [15]. Lockstep execution is achieved by inserting checkpoints in the application, where a synchronisation module is activated to check for mismatch between the status of the cores and apply roll-back. However, to perform lockstep with hard cores, processors should have specific architecture support. However, this functionality is not present on all processors [15]. *Intrusive approaches* modify internally the processor architecture. Hence, when rollback mechanisms are applied, they do not require to insert checkpoints at the application level. For instance, RISC ISA SH-2 processors and rollback are used to achieve error correction [59]. Interleaved multithreated execution is used to implement a dual lockstep approach using two virtual RISC-V cores [46]. Other approaches extend the pipeline registers with error detection and correction codes, e.g., a RISC-V core with Single Error Correction Double Error Detection (SECDED) [31]. Last, approaches triplicate components inside the RISC-V core to enhance its reliability. For instance, Control and Status Registers, Program Counter and the register file [6], FFs, LUTs, BRAMS, and DSPs [58], and the arithmetic and logic unit (ALU) are triplicated [42]. However, existing approaches do not focus on providing simple mechanisms with low WCET bounds regarding the error detection and correction time.

Compared to the state of the art, this work leverages vulnerability analysis approaches with timing correctness for transient faults occurring in processors. Through extended set of experiments, it exposes the fault impact to both functional and timing behavior of the application. Such vulnerability analysis is combined with measurement-based WCET estimation, leading to fault-aware WCET estimations. Last, a mechanism is proposed to remedy the impact of transient faults, with a bounded and near-zero timing overhead, compared to existing approaches, without the need of triplicating the complete processor.

6 Conclusion

This work leverages architectural vulnerability analysis to include not only functional correctness, but also timing correctness, under the presence of transient faults on cores. Using this analysis, we show that the number of execution cycles of an application, under the presence of transient faults, may increase significantly, compared to the fault-free execution. Through a measurement-based WCET estimation approach, we show that impact on the WCET estimation. Compared to common approaches, based on watchdog timers and re-execution with long error detection and correction time, we propose a fault tolerant technique with near-zero WCET overhead that circumvent the fault, as soon as it occurs, before being propagated and affects the execution time.

References

1 J. Abella, M. Padilla, J. Castillo, and F. Cazorla. Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM Trans. Des. Autom. Electron. Syst.*, 22(4), June 2017.

2 J. Abella, E. Quiñones, F. J. Cazorla, M. Valero, and Y. Sazeides. Rvc-based time-predictable faulty caches for safety-critical systems. In *IEEE Int. On-Line Testing Symp. (IOLTS)*, pages 25–30, July 2011.

3 Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In *IEEE/ACM Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 97–102, March 2016.

4 D. Asciolla, L. Dilillo, D. Santos, D. Melo, A. Menicucci, and M. Ottavi. Characterization of a risc-v microcontroller through fault injection. In *Applications in Electronics Pervading Industry, Environment and Society (APPLEPIES)*, Lecture Notes in Electrical Engineering, pages 91–101. Springer Open, 2019.

5 A. Bhat, S. Samii, and R. Rajkumar. Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems. In *IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 87–98, April 2017.

6 L. Blasi, F. Vigli, A. Cheikh, A. Mastrandrea, F. Menichelli, and M. Olivieri. A RISC-V fault-tolerant microcontroller core architecture based on a hardware thread full/partial protection and a thread-controlled watch-dog timer. In *Applications in Electronics Pervading Industry, Environment and Society (APPLEPIES)*, pages 505–511, 2019.

7 F. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Comput. Surv.*, 52(1), February 2019.

8 C. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez. Hamartia: A fast and accurate error injection framework. In *IEEE/IFIP Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, 2018.

9 C. Chen, J. Panerati, I. Hafnaoui, and G. Beltrame. Static probabilistic timing analysis with a permanent fault detection mechanism. In *IEEE Int. Symp. on Industrial Embedded Systems (SIES)*, pages 1–10, June 2017.

10 C. Chen, J. Panerati, M. Li, and G. Beltrame. Probabilistic timing analysis of time-randomised caches with fault detection mechanisms. *IET Computers & Digital Techniques*, 13(3):129–139, 2019.

11 C. Chen, L. Santinelli, J. Hugues, and G. Beltrame. Static probabilistic timing analysis in presence of faults. In *IEEE Int. Symp. Industrial Embedded Systems (SIES)*, pages 1–10, Krakow, PL, July 2016.

12 G. Chen, N. Guan, K. Huang, and W. Yi. Fault-tolerant real-time tasks scheduling with dynamic fault handling. *Journal of Systems Architecture*, 102:101688, 2020.

13 L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F.J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 91–101, 2012. doi:10.1109/ECRTS.2012.31.

14 M. Cui, A. Kritikakou, L. Mo, and E. Casseau. Fault-tolerant mapping of real-time parallel applications under multiple dvfs schemes. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

15 Á.B. de Oliveira, G.S. Rodrigues, F.L. Kastensmidt, N. Added, E.L.A. Macchione, V.A. P. Aguiar, N.H. Medina, and M.A.G. Silveira. Lockstep dual-core arm a9: Implementation and resilience analysis under heavy ion-induced soft errors. *IEEE Trans. Nuclear Science*, 65(8):1783–1790, 2018. doi:10.1109/TNS.2018.2852606.

16 J.F. Deverge and I. Puaut. Safe measurement-based WCET estimation. In *Int. Workshop on Worst-Case Execution Time Analysis (WCET)*, 2007.

17 A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Int. Reliability Physics Symp. (IRPS)*, pages 5B.4.1–5B.4.7, April 2011.

18 S. Edgar and A. Burns. Statistical analysis of wcet for scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 215–224, 2001. doi:10.1109/REAL.2001.990614.

19 G. Fohler, G. Gala, D. Gracia Pérez, and C. Pagetti. Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator, January 2018.

20 Marc Gatti. Development and certification of avionics platforms on multi-core processors. In *Tutorial Mixed-Criticality Systems: Design and Certification Challenges, ESWeek*, 2013.

21 S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot. Reliability challenges of real-time systems in forthcoming technology nodes. In *IEEE/ACM Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 129–134, March 2013.

22 D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *Real-Time Systems Symp. (RTSS)*, pages 456–466, November 2008.

23 D. Hardy and I. Puaut. Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. *Real-Time Systems*, 51:128–152, March 2015.

24 D. Hardy, I. Puaut, and Y. Sazeides. Probabilistic wcet estimation in presence of hardware for mitigating the impact of permanent faults. In *IEEE/ACM Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 91–96, March 2016.

25 P. Huang, H. Yang, and L. Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *ACM/EDAC/IEEE Design Automation Conf. (DAC)*, pages 1–6, June 2014.

26 E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba. Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule. *IEEE Trans. on Electron Devices*, 57(7):1527–1538, July 2010.

27 J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim. Safer: System-level architecture for failure evasion in real-time applications. In *IEEE Real-Time Systems Symp. (RTSS)*, pages 227–236, December 2012.

28 A. Kritikakou, P. Nikolaou, I. Rodriguez-Ferrandez, J. Paturel, L. Kosmidis, M.K. Michael, O. Sentieys, and D. Steenari. Functional and timing implications of transient faults in critical systems. In *IEEE Int. Symp. On-Line Testing and Robust System Design (IOLTS)*, pages 1–10, 2022.

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

30 D. Li, J. S. Vetter, and W. Yu. Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool. In *Int. Conf. on High Performance Computing, Networking, Storage & Analysis (SC)*, pages 1–11, November 2012.

31 J. Li, S. Zhang, and C. Bao. Duckcore: A fault-tolerant processor core architecture based on the risc-v isa. *Electronics*, 11(1), 2022. `doi:10.3390/electronics11010122`.

32 N. N. Mahatme, S. Jagannathan, T. D. Loveless, L. W. Massengill, B. L. Bhuva, S. Wen, and R. Wong. Comparison of Combinational and Sequential Error Rates for a Deep Submicron Process. *IEEE Trans. on Nuclear Science*, 58:2719–2725, December 2011.

33 B.O. Mutlu, G. Kestor, A. Cristal, O. Unsal, and S. Krishnamoorthy. Ground-truth prediction to accelerate soft-error impact analysis for iterative methods. In *Int. Conf. High Performance Computing, Data, and Analytics (HiPC)*, pages 333–344, 2019.

34 B. Ozcelik Mutlu, G. Kestor, J. Manzano, O. Unsal, S. Chatterjee, and S. Krishnamoorthy. Characterization of the Impact of Soft Errors on Iterative Methods. In *IEEE Int. Conf. on High Performance Computing (HiPC)*, pages 203–214, December 2018.

35 Risat Pathan. Real-time scheduling algorithm for safety-critical systems on faulty multicore environments. *Real-Time Systems*, September 2016.

36 R.M. Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 50(4), July 2014.

37 J. Paturel, A. Kritikakou, and O. Sentieys. Fast Cross-Layer Vulnerability Analysis of Complex Hardware Designs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 328–333, Limassol, Cyprus, July 2020. IEEE.

38 A. Ramos, J.A. Antonio Maestro, and P. Reviriego. Characterizing a RISC-V SRAM-based FPGA implementation against Single Event Upsets using fault injection. *Microelectronics Reliability*, 78, November 2017.

39 S. Rehman, M. Shafique, and J. Henkel. *Reliable Software for Unreliable Hardware: A Cross Layer Perspective*. Springer Publishing, 2016.

40 D. Rodopoulos, G. Psychou, M.M. Sabry, F. Catthoor, A. Papanikolaou, D. Soudris, T.G. Noll, and D. Atienza. Classification framework for analysis and modeling of physically induced reliability violations. *ACM Comput. Surv.*, 47(3), February 2015.

41 S. Rokicki, D. Pala, J. Paturel, and O. Sentieys. What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications. In *IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*. IEEE, November 2019.

42 D.A. Santos, L.M. Luza, C.A. Zeferino, L. Dilillo, and D.R Melo. A low-cost fault-tolerant risc-v processor for space systems. In *Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–5, 2020. doi:10.1109/DTIS48698.2020.9081185.

43 N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Q. Shi, R. Allmon, and A. Bramnik. Soft Error Susceptibilities of 22 nm Tri-Gate Devices. *IEEE Trans. Nuclear Science*, 59, 2012.

44 Hardik Shah, Andrew Coombes, Andreas Raabe, Kai Huang, and Alois Knoll. Measurement based wcet analysis for multi-core architectures. In *Int. Conf. on Real-Time Networks and Systems (RTNS)*, pages 257–266, New York, NY, USA, 2014. Association for Computing Machinery.

45 K.P. Silva, L.F. Arcaro, and R. Silva De Oliveira. On using gev or gumbel models when applying evt for probabilistic wcet estimation. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 220–230, 2017. doi:10.1109/RTSS.2017.00028.

46 M.T. Sim and Y. Zhuang. A dual lockstep processor system-on-a-chip for fast error recovery in safety-critical applications. In *Conf. IEEE Industrial Electronics Society (IECON)*, pages 2231–2238, 2020.

47 S. Skalistis and A. Kritikakou. Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules. In *IEEE Real-Time Systems Symp. (RTSS)*. IEEE, 2019.

48 M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Dtm: Degraded test mode for fault-aware probabilistic timing analysis. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, pages 237–248, July 2013.

49 M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Timing verification of fault-tolerant chips for safety-critical applications in harsh environments. *IEEE Micro*, 34(6):8–19, November 2014.

50 J. Song and G. Parmer. C'mon: a predictable monitoring infrastructure for system-level latent fault detection and recovery. In *IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 247–258, April 2015.

51 J. Song, J. Wittrock, and G. Parmer. Predictable, efficient system-level fault tolerance in c^3. In *IEEE Real-Time Systems Symp. (RTSS)*, pages 21–32, December 2013.

52 H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache andpath analyses. *Real-Time Syst.*, 18(2/3):157–179, May 2000.

53 I. Tuzov, D. de Andrés, and J. Ruiz. Accurate Robustness Assessment of HDL Models Through Iterative Statistical Fault Injection. In *European Dependable Computing Conf. (EDCC)*, pages 1–8, September 2018.

54 G. von der Brüggen, K.H. Chen, W.H. Huang, and J.J Chen. Systems with dynamic real-time guarantees in uncertain and faulty execution environments. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 303–314, 2016.

55 N.J. Wang, A. Mahesri, and S.J. Patel. Examining ace analysis reliability estimates using fault-injection. In *Int. Symp. Computer Architecture (ISCA)*, pages 460–469, New York, NY, USA, 2007. Association for Computing Machinery.

56 J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN)*, pages 375–382, 2014.

57 R. Wilhelm, J. Engblom, A. Ermedahl, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.

58 A.E. Wilson, M. Wirthlin, and N.G. Baker. Neutron radiation testing of risc-v tmr soft processors on sram-based fpgas. *IEEE Transactions on Nuclear Science*, pages 1–1, 2023. doi:10.1109/TNS.2023.3235582.

59 J. Yao, S. Okada, M. Masuda, K. Kobayashi, and Y. Nakashima. Dara: A low-cost reliable architecture based on unhardened devices and its case study of radiation stress test. *IEEE Trans. Nuclear Science*, 59(6):2852–2858, 2012. doi:10.1109/TNS.2012.2223715.