Impact of Compiler Optimizations on the Reliability of a RISC-V-based Core

Romaric Pegdwende Nikiema, Marcello Traiola, Angeliki Kritikakou

(Special Session)

Impact of Compiler Optimizations on the Reliability of a RISC-V-based Core

Pegdwende Romaric Nikiema, Marcello Traiola, Angeliki Kritikakou
Univ Rennes, CNRS, Inria, IRISA - UMR 6074, F-35000 and Institut Universitaire de France (IUF)
{pegdwende.nikiema, marcello.traiola, angeliki.kritikakou}@inria.fr

***Abstract*—The RISC-V Instruction Set Architecture (ISA) has gained popularity among systems designers thanks to its open-source nature. Its high flexibility has allowed it to be preferred in various domains and used to target multiple use cases, from embedded systems as co-processor to high-performance computers. Embedded systems, in general, and safety-critical ones, in particular, have strict requirements in terms of reliability and availability. The hardware is becoming less robust with the adoption of smaller technology nodes. The smaller transistor size, low operating voltage, and high switching frequency make transistors susceptible to Single-Event Upsets (SEU) faults, which can propagate to the application output and possibly cause catastrophic consequences. During the software design phase of the system, compilation optimizations can be made to improve the performance. Compilers have various flags that modify the source code to produce the binary. Although these flags can be crucial in assuring good performance, they can significantly impact the resilience to SEU. This work provides comprehensive insights into the impact of compiler optimizations on the reliability of safety-critical embedded systems. Specifically, a probabilistic fault injection campaign is conducted on various benchmarks running on a RISC-V core to evaluate the effect of several optimizations on reliability. The results are classified into functional and timing errors, offering a detailed understanding of the implications of these optimizations on reliability.**

***Index Terms*—Reliability, Compiler optimizations, fault injection**

I. Introduction

Modern electronic systems have adopted newer technologies that aim to improve efficiency and performance. Nowadays, transistors use a small node. Moreover, combined with their high switching frequencies and lower operating voltages [1], they become more powerful for almost equal power compared to a bigger counterpart. This scale-down has consequences on reliability. For example, the reduced critical charge makes the transistor more vulnerable to single-event upsets (SEU) caused by radiation [2]. In addition, the aging and wear-off of transistors also contribute to transient or permanent faults, which cause functional and timing errors on the system [3]. The radiation threat used to be a matter for space applications where the level of radiation is higher. Nowadays, even at sea level, we experience radiation issues, making regular consumer electronics vulnerable [4]. Different studies exist on vulnerability analysis on various architectures and systems from application level to hardware level, as well as techniques to reduce the impact of the SEU on the systems.

Nowadays, the RISC-V instruction Set Architecture (ISA) has gained popularity and is used in various application domains for various computation demands due to its open source, versatility, and extendability. To perform system deployment, the application is compiled for the specific processor ISA. During compilation, optimizations are applied to modify the application binary in order to improve performance and reduce resource utilization. Optimizations may imply instructions reordering, skipping redundant operations, etc. On one hand, such optimizations can negatively affect the system's vulnerability. On the other hand, it's worth noting that these optimizations can even reduce the exposure to fault due to their execution time reduction. The unpredictability of the impact of compiler optimizations in terms of reliability makes their application, without prior study, dangerous. Existing studies on showing the impact of compiler optimizations either focus on more fine-grained flags and try to find a good flag combination or target different processor architecture and functionalities such as ARM, Out of Order (OoO) processors, GPU [5]–[9]. Moreover, the fault injection campaigns conducted have a lower coverage due to the uniqueness of their input values.

In this paper, we exploit the impact of compiler optimization levels on RISC-V-based processors on reliability, considering both functional and timing effects. The impact of different optimizations on the system is obtained through a vulnerability analysis on a RISC-V-based processor core [10] using a Cycle Accurate Bit Accurate (CABA) simulator [11]. More precisely, we perform a probabilistic architectural-level single-fault injection on the whole processor. To have a high confident level of coverage, we consider different input values for the program, which are randomly selected. We perform experiments with various benchmarks to obtain the vulnerability metrics, to characterize the fault probability, fault propagation and the criticality of several compiler optimizations. The obtained results show that trends exist in the execution speed and the exposure to faults due to different optimizations. Such analysis is helpful during system deployment, driving the selection of an optimization level depending on the application's needs.

The remainder of this paper is structured as follows: Section II discusses compiler optimizations and how they affect reliability, along with the related work on the topic. Section III presents the proposed methodology and fault injection model in depth. Section IV presents the results, and section V concludes this study.

II. Background and Related Work

A. Compiler optimizations

Compiler optimization is a process of improving the compiled code. These improvements are performed by the compiler on demand using compilation flags. Such optimizations are helpful as they take advantage of the hardware depending on the needs, such as increasing execution speed, reducing binary size and memory usage, power consumption, etc. These optimizations are numerous. We can cite some examples, for instance, 1) loop optimizations, which apply techniques, such as unrolling, which increase the instructions in the loop body with the goal of reducing the loop condition tests, 2) loop tiling, which re-orders the iterations to improve cache efficiency, 3) function inlining, which reduces the function calls overhead by copying over the function instructions where the function has been called, 4) dead code elimination, which can reduce the binary size, 5) register renaming to deal with data hazards and dependencies, etc [12].

These optimizations have mainly explored regarding their impact on the binary code and its execution regarding timing. However, few works have been conducted to understand and quantify the impact of compiler optimizations on system reliability.

B. Related Work

The majority of existing works study the impact of compiler optimizations on reliability either target different processor architectures and functionalities, such as ARM, OoO processors etc, or focus on approaches that explore more fine-grained flags in order to find a more reliable flag combination. The main reliability threat discussed is single event upsets (SEU), which modify the control flow or some memory-related data and lead to errors.

Regarding the first category, approaches estimate the architectural vulnerability factor (AVF) of specific x86 processor structures such as Load/Store Queue and the Reorder buffer using zesto [13] simulator, and show the impact of optimization on these processor structures [6]. The AVF of the structures was estimated using equations. A similar study has been conducted for ARM-based OoO processors [5], through an AVF analysis through microarchitectural-level fault injection considering the effects of optimization on eight structures of the processor, such as the Reorder-buffer, Load/Store buffers, processor caches, etc. The vulnerability factor of register files of an ARM cortex-A9 core is studied in [7] considering various optimizations and correlating the register file usage with reliability. The fault injection is carried on the user-accessible registers using interrupts and a heavy-ion radiation. The impact of optimizations on a High-Performance Computing (HPC) AMD Opteron application and the trade-off between performance and reliability are analyzed in [14]. Fault injection is conducted at the software level, showing that more optimization yields poor reliability. Early reliability analysis through fault injection with `-O2` optimization level is performed in [15]. Last, a study on compiler optimizations for GPU and vulnerability assessment through beam experiments has been conducted [16].

Regarding the second category, studies are tailored into finding suitable flags for reliability using LLVM (Low-Level Virtual Machine) in [8], [9]. In [8], meta-heuristic methods are applied to optimize the reliability. In [9], a machine learning-based algorithm is used to derive the best set of flags in the context of real-time where the wcet is evaluated. Though the achieved results are better than the regular `-Ox` levels flags, it's worth noting that we are targeting only the regular optimizations due to increasing complexity with the required amount of fault injections.

This work belongs to the first category. Compared to existing approaches, it presents a vulnerability analysis through intensive microarchitectural-level fault injection on a RISC-V processor to characterize the criticality of optimizations and the fault probability. Furthermore, it studies the application execution profile, showing the impact of compiler optimization on the reliability.

III. Proposed Methodology

Figure 1 describes the overview of the reliability analysis methodology.

The inputs to the methodology are the different compilation flags that lead to different versions of the benchmarks, the benchmarks, and their inputs. The compiler optimizations are based on the processor ISA. Typical compiler optimizations under study consist of a set of individual optimizations applied in a specific order: `-O0, -O1, -O2, -O3, -Ofast, -O, -Og, -Os, -Oz`. Note that the level of optimizations is nested from one flag to another. For example, the `-O2` flag enhances optimizations from `-O1` with other optimization flags, `-O3` enhances `-O2` flag etc. We generate several input values for each selected benchmark to be analyzed for higher statistical confidence in the obtained results.

We consider, as a baseline, the execution of the benchmark binary without any optimization. For each compilation flag we run a set of experiments considering fault-free and faulty executions with different benchmark inputs. Regarding the faulty execution, fault injections are performed during execution using `bit flip` as the fault model. The injection is done on the processor registers, such as the pipeline registers and the register file. The process of generating the fault injection is as follows: we randomly select an area of the processor with respect to its size, as bigger elements are more likely to be targeted by radiation than smaller ones. Then, we randomly select a bit for a given area and then apply a logical xor with a random bit mask to flip it.

The comparison of these results provides insight into the processor's reliability and how it is affected by the compiler optimization flag. The reliability metrics are categorized into functional and timing errors. These errors include Silent Data Corruption (SDC), where the corrupted data is only detected at the end of a run, and Detectable Unrecoverable Errors (DUE), such as Hangs and Crashes. More precisely:

- *Execution Cycles Mismatch (ECM):* The execution cycles of the application are different than those of the golden reference.
- *Hang:* The execution time of the application has exceeded a threshold, and thus, it is assumed that it has entered

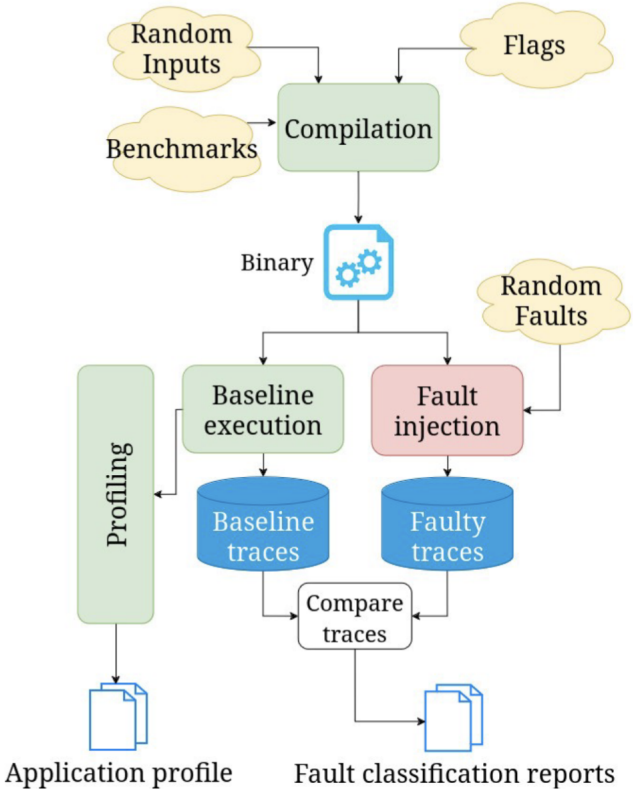


Fig. 1: Methodology

an infinite loop. A cycle counter is used to stop the current execution when the counted cycles exceed a given threshold.

- *Crash:* 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.

In order to characterize the *criticality* of a given optimization on the system, we compute the aforementioned metrics considering the average impact of the compiler optimization on a given program. The bigger a metric, the more vulnerable an application is under a given compiler optimizations on the average case. In this scenario, the execution time duration is not taken into account. Furthermore, to characterize the *fault probability*, i.e., the probability of a fault impacting the system when it executes the optimized code, the execution time is taken into account. In this scenario, the longer the execution, the more the application is exposed to faults. Note that when optimizations reduce the execution time, the fault probability decreases as the application finishes earlier. As a result, resilience may be increased, depending on the optimization flags and the reliability metrics. Last, we perform application profiling in order to obtain the number of execution clock cycles and the number of different instructions.

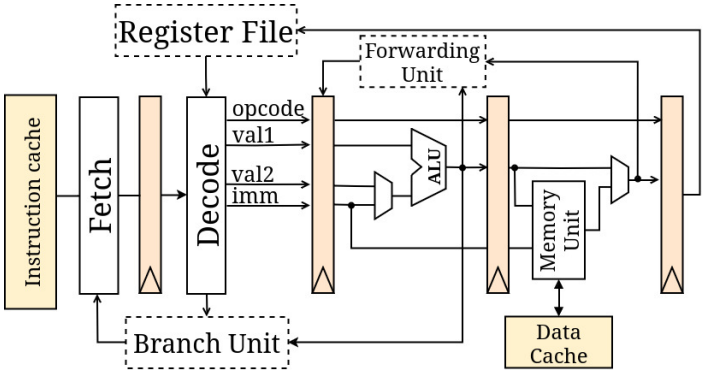


Fig. 2: RISC-V core with 5-stage pipeline [10].

IV. Experimental results

Our case study is a RISC-V processor and the `RISC-V GCC` compiler. The Device Under Test (DUT) is Comet [10], an open-source 32-bit RISC-V processor, which supports the RV32I base ISA[1]. It's written in High-Level Synthesis (HLS), which offers a unique high-level synthesis and simulation. A C++ model is used to design the processor. The model generates the hardware target design through High-Level Synthesis and a Cycle-Accurate Bit-Accurate (CABA) simulator. 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 2.

The studied benchmarks are taken from TacleBench [17]: `Bitonic, Binary search, Bsort, CountNegative, Factorial, InsertSort, Matmul, QuickSort`. For each benchmark, we generate several input values, i.e., 650 different inputs based on [18] for higher statistical confidence in the obtained results. For each of the benchmarks, with one optimization flag, 385 faults are injected per binary, resulting in a margin error `e=5%` and a confidence level of `95%` [19]. We repeat this for the 650 different inputs for higher statistical confidence, totaling `250'250` intensive faults injection per binary. In total, we inject `2,002,000` faults per binary. We first present the benchmarks' profiling regarding the different computations occurring during execution, followed by the criticality and fault probability of the obtained results.

A. Benchmarks profile

During profiling, we used Hardware Performance Counters to obtain the execution clock cycles and the instruction counts during a binary execution. Table III depicts the average clock cycles required to execute each benchmark, considering 650 different inputs. Tables I and II show the computation profile of the benchmarks, i.e., data related to the `instruction count` for each type of instructions for each benchmark. Each instruction count is divided each time by the corresponding execution clock cycles in order to obtain the computation profile. Note that the DUT doesn't support multi-cycle stages, which are used to implement multiplication and division instructions,

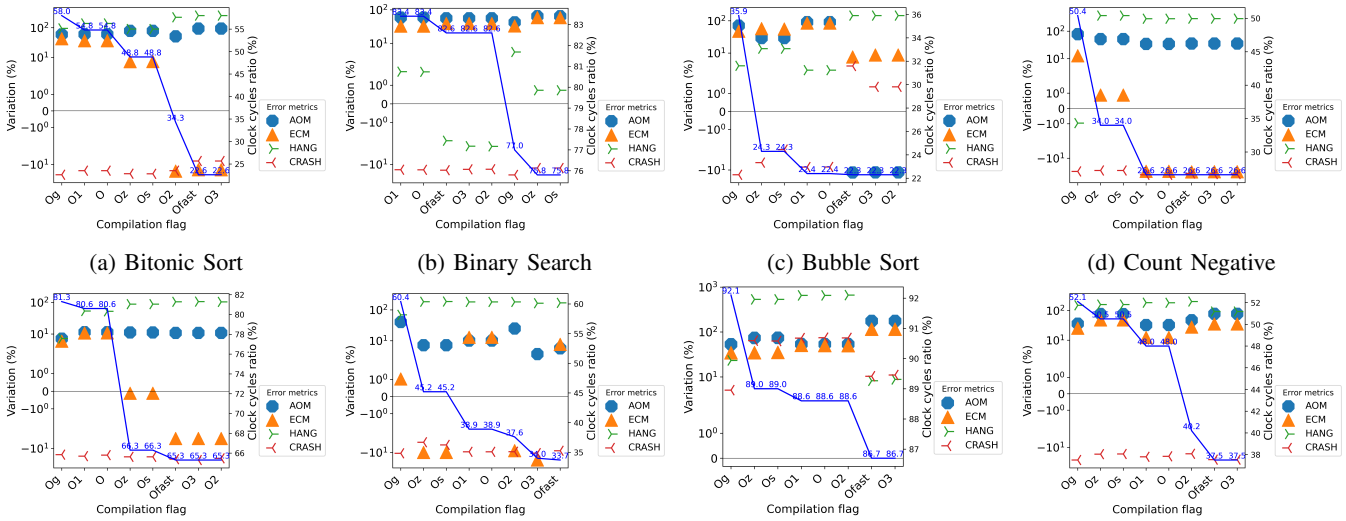
[1] https://gitlab.inria.fr/srokicki/Comet/-/tree/master

Instructions	LUI	LD	ST	OP_SLL	OP_ADD	OP_AND	OPI_SLLI	OPI_ANDI	OPI_ADDI	AUIPC
Bitonic	1.53 (-2.6)	20.6 (-20.7)	16.43 (+2.8)	0.02 (0.0)	8.12 (+0.9)	0.02 (0.0)	3.95 (+0.3)	0.07 (-0.9)	25.37 (+6.4)	0.1 (+0.1)
Binary Search	2.66 (-1.0)	19.05 (-10.9)	14.41 (+1.0)	0.27 (0.0)	3.86 (+0.6)	0.27 (0.0)	2.46 (+0.4)	1.6 (+0.9)	26.0 (+1.9)	1.87 (+0.8)
Bubble Sort	0.02 (0.0)	25.87 (-30.5)	17.26 (+11.5)	0.0 (0.0)	0.01 (-7.6)	0.0 (0.0)	0.01 (-8.6)	0.01 (0.0)	22.0 (+10.2)	0.01 (0.0)
Count Negative	0.27 (-7.7)	21.31 (+2.5)	1.98 (-3.8)	0.02 (0.0)	15.39 (-2.9)	0.02 (0.0)	0.15 (-15.5)	0.15 (+0.1)	32.91 (+7.0)	0.17 (+0.1)
Factorial	1.15 (+0.2)	13.75 (-6.0)	10.59 (-1.7)	0.13 (0.0)	5.15 (+1.9)	0.13 (0.0)	4.32 (+1.7)	4.32 (+1.7)	26.21 (-5.8)	0.89 (+0.4)
Insert Sort	1.21 (+0.7)	22.4 (-21.2)	20.9 (+8.8)	0.11 (+0.1)	3.29 (-4.6)	0.11 (+0.1)	3.07 (-4.8)	0.66 (+0.4)	24.64 (+4.7)	0.77 (+0.5)
Matrix mult.	0.06 (-0.6)	1.14 (-5.5)	1.02 (-0.6)	0.01 (0.0)	11.45 (+0.1)	0.01 (0.0)	11.15 (-0.2)	11.15 (+1.5)	13.25 (-0.8)	0.03 (0.0)
Quick Sort	1.71 (-5.3)	21.18 (-17.8)	12.82 (+0.5)	0.13 (+0.1)	7.19 (+1.7)	0.13 (+0.1)	8.06 (+2.6)	0.75 (+0.5)	25.1 (+4.7)	0.88 (+0.5)

TABLE I: Benchmark's computation profile on `-O3` compared to `-O0`

Instructions	OPI_SRLI	OPI_SRAI	BR_BLT	BR_BNE	BR_BEQ	BR_BLTU	BR_BGE	BR_BGEU	JALR	JAL
Bitonic	2.04 (+1.2)	1.66 (+0.8)	0.43 (-1.1)	6.68 (+5.7)	2.08 (+2.0)	0.08 (+0.1)	3.59 (+2.7)	0.02 (0.0)	1.78 (-0.0)	1.83 (-0.6)
Binary Search	N/A	1.66 (+0.3)	0.53 (+0.1)	2.66 (-0.3)	6.19 (+2.0)	1.33 (0.0)	2.04 (+0.6)	0.53 (+0.3)	5.62 (+1.9)	4.31 (+0.3)
Bubble Sort	N/A	0.01 (0.0)	0.0 (-1.9)	17.35 (+17.3)	8.62 (+8.6)	0.01 (0.0)	8.59 (+4.7)	0.0 (0.0)	0.18 (+0.2)	0.02 (-2.0)
Count Negative	N/A	0.07 (+0.1)	0.05 (-2.6)	10.35 (+10.3)	5.36 (+5.2)	0.15 (+0.1)	10.03 (+7.2)	0.05 (0.0)	0.49 (+0.4)	0.83 (-0.7)
Factorial	5.15 (+1.8)	0.38 (+0.1)	0.38 (+0.2)	8.7 (+3.8)	6.68 (+2.4)	0.64 (0.0)	0.0 (-0.5)	0.25 (+0.1)	6.42 (+1.0)	3.88 (-1.6)
Insert Sort	N/A	0.33 (+0.2)	0.33 (+0.3)	2.74 (+2.2)	2.41 (+1.7)	1.1 (-0.7)	2.52 (+1.2)	5.56 (+5.0)	3.72 (+2.8)	3.06 (+1.9)
Matrix mult.	16.49 (+2.2)	0.02 (0.0)	0.01 (0.0)	11.2 (+1.5)	11.23 (+1.5)	0.05 (0.0)	0.0 (-0.5)	0.01 (0.0)	11.24 (+1.5)	0.44 (-0.0)
Quick Sort	N/A	0.38 (+0.2)	0.81 (-0.7)	1.26 (+0.8)	2.51 (+1.6)	0.63 (+0.4)	8.78 (+6.7)	0.25 (+0.2)	3.17 (+1.4)	3.0 (+1.2)

TABLE II: Benchmark's computation profile on `-O3` compared to `-O0` (cont.)



(a) Bitonic Sort (b) Binary Search (c) Bubble Sort (d) Count Negative

(e) Factorial (f) Insert Sort (g) Matrix multiplication (h) Quick Sort

Fig. 3: Vulnerability results: Fault criticality

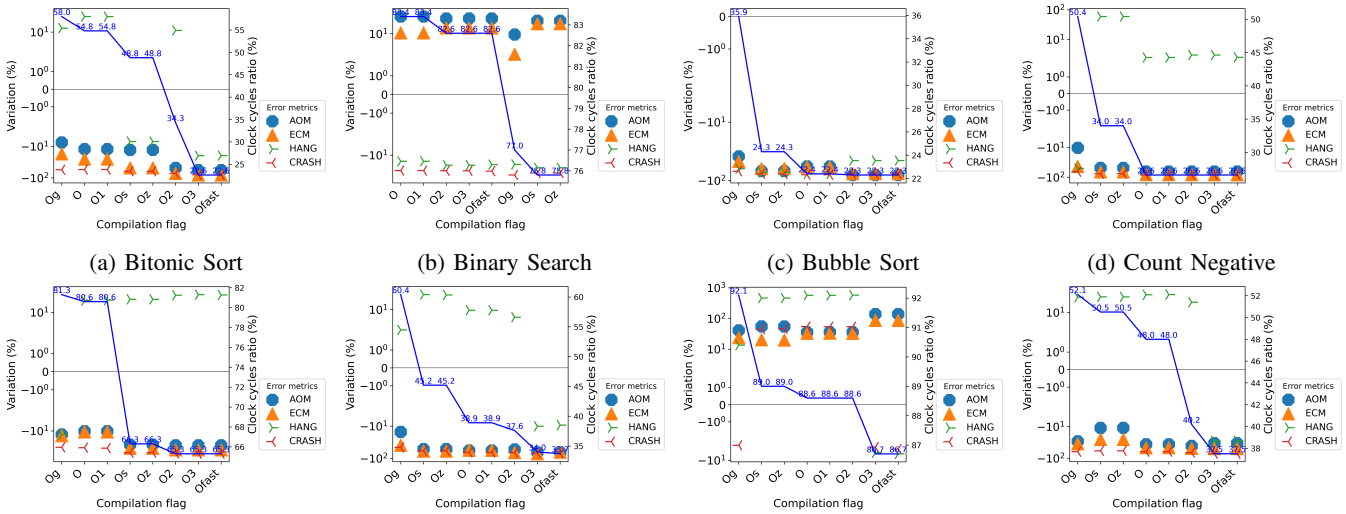
Benchmark	Average Clock Cycles
Bitonic	26,669
Binary Search	455
Bubble Sort	264,048
Count Negative	15,379
Factorial	1,969
Insert Sort	2,686
Matrix mult.	20,573
Quick Sort	2,134

TABLE III: Fault-free average execution cycles (`-O0`)

etc. Considering an implementation with a multiplication (mult) unit, the mult instruction will be split into smaller sub-stages, and the pipeline stalled – fetch and decode stages – until the multi-cycle operation instruction reaches the memory stage. Therefore, the benchmark profiling will be modified in order to take into account the stall caused by the multi-cycle additional cycles.. In order to determine the reliability of the system under different executions and different compiler optimizations, Tables I and II show the increase and decrease in terms of number of different types of instructions. More precisely, the percentage indicates how much is decreased or increased each of the instructions executed with a given optimization, e.g., `-O3`, compared to `-O0` version.

B. Reliability analysis

The average impact of a compiler optimization on the functional and timing behavior of the program is shown in Fig. 3 (fault criticality) and Fig. 4 (fault probability). The x-axis corresponds to the compiler flag. The right-side y-axis is the execution time ratio (%) between the benchmark, compiled with an optimization flag, and its baseline `-O0`. For instance, a value of 100 for a compiler flag means that the optimized version has the same execution time as the baseline, and a value of 50 means that the optimized version finished in half of the time compared to the baseline `-O0`. The left-side y-axis is on a logarithmic scale and shows the effect that a flag has on a reliability metric. It shows the variation in the number



(a) Bitonic Sort (b) Binary Search (c) Bubble Sort (d) Count Negative

(e) Factorial (f) Insert Sort (g) Matrix multiplication (h) Quick Sort

Fig. 4: Vulnerability results: Fault probability

of faults observed between the benchmark, compiled with an optimization flag, and its baseline `-O0`.

1) Instructions and error correlations: Combining the computation profile and the vulnerability metrics information allows us to observe several correlations between the benchmark instructions and the observed errors.

The number of Crashes of a benchmark correlates with the number of load instructions (LD) during its execution. Overall, we observe that the more loads there are, the more crashes are reduced. This is expected since the more frequent loads occur, the more frequently the content of registers is overwritten with new data. For instance, with `Matmul`, the `-O3` flag reduces the number of loads, which leads to an increase in the number of crashes (Fig. 3g).

We observe a correlation between the Branching instructions (BR) and the number of hang occurrences. The more branch instructions exist, the more susceptible the benchmark is to Hangs. The benchmarks, such as `Bitonic` 3a and count negative 3d, have an increased number of branch instructions when compiled with the `-O3`, and result in a higher number of Hangs. On the contrary, `Bsearch` 3b and `Matmul` 3g, benchmarks, have a lower number of branch instructions on average and show a lower number of Hangs.

For the AOM, we notice that the more the `LUI` instructions are reduced, the more the AOM. AOM is also correlated to the number of arithmetic operations (OP) that occurred during the execution. On average, `Matmul` has the highest number of operations, which exposes it to wrong computations and, thus, to AOM. The less operation-intensive benchmarks, such as Bubble Sort and Insert Sort, result in a lower AOM counts under `-O3`.

Last, but not least, the ECM is correlated to the branching and operation instructions.When a fault hits a branch instruction, either conditional or unconditional, aside from functional error, it also produces timing errors. For example, a loop index may be altered, leading to skipping or re-execution of iterations. This behavior can also appear when we alter the condition test in conditional jumps. For instance, `JALR` for subroutines shows increased ECM for `Bsearch`, `Matmul`, etc.

2) Impact of compiler optimizations: The key findings from Fig. 3 and Fig. 4 indicate that applying compiler optimizations to the program consistently impacts the reliability.

In the overall trend regarding fault criticality (Fig. 3), we observe a significant increase in AOMs for most of the benchmarks and optimization flags. However, we observe the opposite trend for the Crash, with an overall average decrease, except the `Matmul` benchmark. The Hangs and the ECM vary depending on the benchmark, but the overall observed trend is an increase.

These observations reveal a trade-off between performance and vulnerability, enabled by the different compilation flags. While some flags have significantly reduced execution time, specific functional errors have worsened. For instance, on one side, with the `Matmul` benchmark, all selected flags increase the percentage of faults, considering the criticality. Conversely, most flags have increased the tolerance to Crashes and decreased the ECMs, especially for `Insertsort, Cntnegative`benchmarks. This suggests that no universal flag can optimize for performance while maintaining robustness to faults. This is particularly evident in the `Matmul` benchmark, where the most optimized version in terms of execution cycles remains high at 86%, and the optimizations have not been achieved to reduce the observed errors.

Looking at a more fine-grained comparison, i.e., the impact between different optimization flags, we observe that specific errors are reduced depending on the flag. For instance, on the Count Negative benchmark, the less optimized version in terms of clock cycles (-Og) gives better results for hangs compared to more optimized versions, such as -O2. It's worth highlighting that, regarding the execution time reduction, `-O3` is not always the best option.

These results suggest that the more the application is optimized, the more likely it is to produce wrong results,

hangs, and timing errors.

To get the intrinsic reliability of an application, we have to analyze its exposure to faults. To achieve that, we present the fault probability, which takes into account the execution time of the application for the different optimization flags (Fig. 4). When applying optimization flags, the result usually yields a faster execution of the benchmark regarding clock cycles. Considering the time required to execute the benchmarks, the number of faults is reduced, on average, when the benchmark runs faster. When faults occur after the benchmark finishes, then they have no impact on the execution of the benchmark.

From the obtained results, we observe that `Bsearch` and `Matmul` give the least improvements in terms of functional and timing errors for the different optimization flags, as only a part of Hangs and Crashes is reduced with the different flags. Note that, this is also a consequence of the nature of these benchmarks, which are less optimized, and thus, more likely to have faults.

V. Conclusion

This work presented a study on compiler optimizations' impact and safety-critical systems' reliability. A vulnerability analysis was done by running probabilistic fault injections on the processor registers file and the pipeline registers. In order to have statistically sound results, we apply techniques in order to take into account the worst-case inputs in the fault injections by using different inputs. Our experiment, run on a RISC-V processor core with no hardware multiplier and disabled caches, shows that the optimization can be beneficial for some use cases. If, on one side, the goal of the design is to reduce the probability of faults, then using optimization that reduces the execution time is beneficial as it will reduce the exposure to fault. This can increase the Mean Work To Fault, for example. On the other hand, to reduce the fault criticality, a prior study is recommended as the impact changes with respect to the application computation profile. Some flags that optimize the application less show better resilience than others. In safety-critical real-time systems, where the system tasks have been scheduled with the unoptimized version, applying optimizations and maintaining the same schedule can lead to better reliability. It would be interesting to further assess the impact on the wcet estimation for real-time systems. In addition, a mean to estimate the reliability of a given benchmark by only studying it's application profile could be beneficial for system development.

Acknowledgments

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

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] A. Dixit *et al.*, "The impact of new technology on soft error rates," in *Int. Reliability Physics Symp.*, Apr. 2011, pp. 5B.4.1–5B.4.7.

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

[3] A. Kritikakou *et al.*, "Functional and timing implications of transient faults in critical systems," in *IEEE Int. Symp. On-Line Testing and Robust System Design (IOLTS)*, 2022, pp. 1–10.

[4] F. Catthoor *et al.*, "Will chips of the future learn how to feel pain and cure themselves?" *IEEE Design & Test*, vol. 34, no. 5, pp. 80–87, 2017.

[5] G. Papadimitriou *et al.*, "Characterizing soft error vulnerability of cpus across compiler optimizations and microarchitectures," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 113–124.

[6] M. Demertzi *et al.*, "Analyzing the effects of compiler optimizations on application reliability," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 184–193.

[7] F. M. Lins *et al.*, "Register file criticality and compiler optimization effects on embedded microprocessor reliability," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2179–2187, 2017.

[8] N. Narayanamurthy *et al.*, "Finding resilience-friendly compiler optimizations using meta-heuristic search techniques," in *2016 12th European Dependable Computing Conference (EDCC)*, 2016, pp. 1–12.

[9] M. Dardaillon *et al.*, "Reconciling compiler optimizations and wcet estimation using iterative compilation," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2019, pp. 133–145. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/RTSS46320.2019.00022

[10] S. Rokicki *et al.*, "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, Nov. 2019.

[11] P. R. Nikiema *et al.*, "Impact of transient faults on timing behavior and mitigation with near-zero wcet overhead," in *ECRTS 2023 - 35th Euromicro Conference on Real-Time Systems*, Vienna, Austria, July 2023.

[12] B. Gough *et al.*, "An introduction to gcc : for the gnu compilers gcc and g++," 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:116776160

[13] G. H. Loh *et al.*, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 53–64.

[14] R. A. Ashraf *et al.*, "Exploring the effect of compiler optimizations on the reliability of hpc applications," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 1274–1283.

[15] N. Lodéa *et al.*, "Early soft error reliability analysis on risc-v," *IEEE Latin America Transactions*, vol. 20, no. 9, pp. 2139–2145, 2022.

[16] F. F. D. Santos *et al.*, "Assessing the impact of compiler optimizations on gpus reliability," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 2, feb 2024. [Online]. Available: https://doi.org/10.1145/3638249

[17] H. Falk *et al.*, "Taclebench: a benchmark collection to support worst-case execution time research," 01 2016.

[18] L. Cucu-Grosjean *et al.*, "Measurement-based probabilistic timing analysis for multi-path programs," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012, pp. 91–101.

[19] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 502–506.