RAT: A Lightweight Architecture Independent System-Level Soft Error Mitigation Technique

Jonas Gava, Ricardo Reis, Luciano Ost

RAT: A Lightweight Architecture Independent System-level Soft Error Mitigation Technique

Jonas Gava[1], Ricardo Reis[1], and Luciano Ost[2]

[1] Instituto de Informática, PGMicro, Universidade Federal do Rio Grande do Sul - UFRGS, Porto Alegre, Brazil
jfgava@inf.ufrgs.br, reis@inf.ufrgs.br
[2] Loughborough University, United Kingdom
l.ost@lboro.ac.uk

Abstract. To achieve a substantial reliability and safety level, it is imperative to provide electronic computing systems with appropriate mechanisms to tackle soft errors. This paper proposes a low-cost system-level soft error mitigation technique, which allocates the critical application function to a pool of specific general-purpose processor registers. Both the critical function and the register pool are automatically selected by a developed profiling tool. The proposed technique was validated through more than 400K fault injections considering a Linux kernel, different benchmarks, and two multicore Arm processor architectures (ARMv7-A and ARMv8-A). Results show that our technique significantly reduces the code size and performance overheads while providing soft error reliability improvement compared with the Triple Modular Redundancy (TMR) technique.

Keywords: Multicore, soft error reliability, mitigation technique, fault tolerance

1 Introduction

Multicore architectures are being adopted in many industrial segments such as automotive, medical, consumer electronics, and high-performance computing (HPC). Applications running on such architectures differ in terms of security, reliability, performance, and power requirements. To achieve a substantial reliability and safety level, it is imperative to provide electronic computing systems with appropriate mechanisms to tackle systematic or transient faults, also known as soft errors or Single Event Upset (SEU). While the former originates from hardware and software design defects, soft errors are those caused by alpha particles or atmospheric neutrons [24]. The occurrence of soft errors can either corrupt the memory data, the output of a program, or even crash the entire system, which depending on its criticality level can lead to life-threatening failures.

The soft error mitigation problem can be tackled both in hardware and software [23]. While hardware approaches lead to the area and power overhead, software techniques are generally implemented on a per-application basis that

usually incurs in performance penalties due to the redundant computation. Such additional overhead might restrict the use of costly mitigation techniques under resource-constrained devices. Furthermore, the adoption of soft error mitigation techniques also adds development complexity, which has a direct impact on the time-to-market. Examples of soft error mitigation techniques include, among others, Error Detection and Correction Code (EDAC) and Triple Module Redundancy (TMR).

This paper addresses the above challenges by proposing a novel lightweight system-level soft error mitigation technique, called Register Allocation Technique (RAT) [13]. The proposed technique along with the developed profilling toolset enables software engineers to isolate and allocate the most critical application function to a pool of least used general-purpose processor registers. RAT was compared against a selective TMR technique [11], considering a Linux kernel, 13 applications, a dual-core and a quad-core ARM processor. Results demonstrated that RAT reduces the code size and performance overheads while providing reliability improvement.

The rest of this paper is organized as follows. Section 2 presents basic concepts and related works in software soft error mitigation techniques. Section 3 describes the proposed mitigation technique. In Section 4, the experimental setup and adopted evaluation metrics are presented. In Section 5, the efficiency of RAT is evaluated, and a specific case study analyzing the registers criticality is presented. Section 6 evaluates the impact of instruction set architectures (ISAs) on the RAT efficiency (i.e., ARMv7-A 32 and ARMv8-A 64 bits). Finally, Section 7 presents final remarks and future works.

2 Fundamental Concepts and Related Works

2.1 Fault Tolerance Taxonomy

The soft error assessment and mitigation literature is abundant, requiring a taxonomy to classify the different approaches. This work considers the definitions from [3, 17] for fault, error, and failure. A fault is an event that may cause the internal state of the system to change, e.g., a radiation particle strike. When a fault affects the system's internal state, it becomes an error. If the error causes a deviation of at least one of the system's external states, then it is considered as a failure.

The most commonplace classification for soft error assessment considers three classes: Silent Data Corruption (SDC) occurs when the system does not detect a fault and the outcome of the application is affected; In Detected Unrecoverable Error (DUE) on the other hand, the fault is detected and it is not possible to continue the execution (e.g., segmentation fault); and Masked, when the application outcome and the system state are the same as a faultless execution.

As mentioned before, soft error mitigation techniques can be implemented in hardware, software, or a combination of both. The next Section reviews only the software-based approaches.

2.2 Software-based Soft Error Mitigation Techniques

A processor-based system can be affected by two main types of soft errors: control-flow and data-flow. A control-flow error occurs when the error causes deviation from the correct program flow (e.g., incorrect branch). The data-flow error refers to the soft error caused by a bit-flip in a storage component, such as a register or memory element. They can, for instance, affect the output of a program generating an SDC, or leading to a DUE when computing a wrong memory address.

Aiming to mitigate both types of soft error effects, the following works have promoted some software system-level techniques, i.e., techniques that can be applied at the software architectural level (e.g., application, operating system (OS)). In [6] and [18] tools that apply fault tolerance techniques in C/C++ applications are proposed. Supported transformations are architecture-independent, but the language is fixed, and the compiler may remove redundant code during the compiler optimization phases. The focus of [18] is on low-cost safety-critical applications, where the high memory and speed overheads (about 3-4 times) are not important metrics. Another similar tool is the REliable Code COmpiler (RECCO) [6], which relies on code reordering and selective variable duplication. In [22], authors use genetic algorithms to find a combination of optimization parameters (i.e., compilation flags) that increase the reliability of the final binary and present a reasonable trade-off in terms of performance, and memory size. The proposed technique was evaluated considering an FPGA implementation that was exposed to a proton irradiation test. In [21], the authors implemented in C code two mitigation techniques: the TMR and the Conditional Modular Redundancy (CMR). Their results have shown that both techniques do not provide a reasonable protection to a complex system executing Linux kernel. According to the authors, the OS itself is an enormous source of errors and need to be protected if employed on safety-critical systems.

The downside of aforementioned approaches is the fact that during the compiler optimization phase, parts of the protected code (e.g., redundant functions) may be wrongly removed. One solution to overcome such restriction relies on modifying the assembly code after the compilation. A popular instruction-level mitigation technique introduced by [19] is the Swift-R, which implements TMR to recover from soft errors in the register file. Instead of duplicating instructions, it triplicates, and changes the checking points to a voter mechanism. In [16], they apply the SWIFT-R to protect specific registers and find the best trade-off. They developed a generic intermediate language and their own compilation infrastructure. Although the idea is interesting, a considerable effort is necessary to support new processor architecture, limiting its usability. Authors in [10] proposed the Configurable Fault Tolerance Tool (CFT-tool) that modifies the assembly code by applying different data-flow and control-flow protection techniques. Although this approach does not suffer from compiler optimization, it is architecture-dependent. The CFT-tool uses a configuration file to minimize this limitation. However, this file needs to be hand-made for each new ISA. Shirvani et al. [23] propose a software implementation of EDAC, i.e., an independent task

that is executed periodically. Results show their approach provides protection for code segments and can enhance the system reliability with a lower check-bit overhead with relation to other techniques (e.g., Hamming, Parity).

Different from the reviewed works, RAT does not involve code redundancy, and it is an architecture-independent approach. Furthermore, RAT is a fully automated approach that is developed on the basis of LLVM backend, enabling its extension and combination with other soft error mitigation techniques, as shown in Section 5.

3 Register Allocation Technique (RAT)

Rather than implementing a toolset from scratch, we have adopted a flexible virtual platform (VP) that provides us with the necessary means (i.e., simulator with processor and component models, full software behavior observability) to implement the proposed technique. RAT was implemented on the basis of OVPsim framework [14] to enable a fast design space exploration, but other VPs with similar support could also be used (e.g., gem5 [7]). The main steps of the RAT (Fig. 1) are as follows:

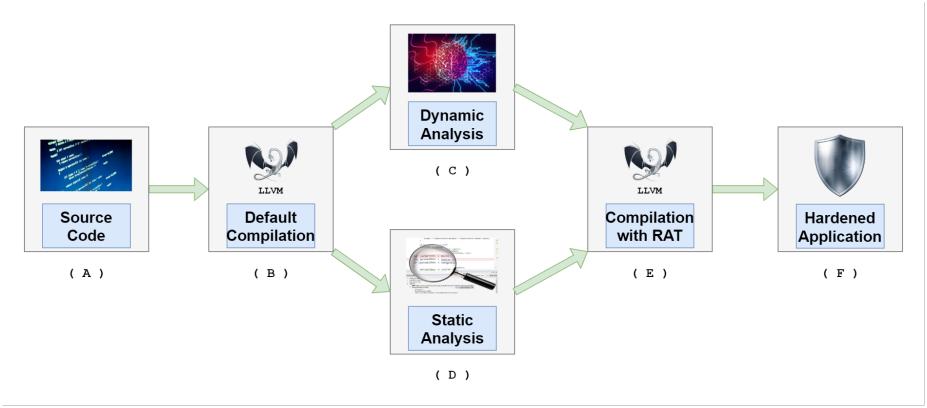


Fig. 1: Main steps of proposed register allocation technique (RAT).

A. Software stack (i.e., application, operating system, drivers) source code selection.
B. Target processor architecture selection and source code compilation using Clang/LLVM 6.0.1.
C. In this step, the application is executed, and essential information are extracted (i.e., processor register file utilization and critical function). Note that the software engineer can either determine the most critical application function or use the default option of our toolset, which selects the most executed one.

D. Here, our tool extracts, from the object code, the type (i.e., 32 or 64-bit) and the number of registers needed to be reserved to the function defined as critical in the previous step. In this stage, the register pool is set following the strategy of allocating least-used general-purpose registers for the critical function.
E. In this step, a new compilation is performed, taking into account the critical function and the register pool previously set. The underlying compilation uses a modified version of the LLVM Fast Register Allocator, which considers arguments (i.e., restrictions) that are passed to LLVM Static Compiler (LLC) through a command line (Fig. 2). Note that we do not control the use of the registers available in the pool, the compiler decides which ones to prioritize.
F. Finally, the resulting hardened binary is generated by the LLVM linker (LLD).

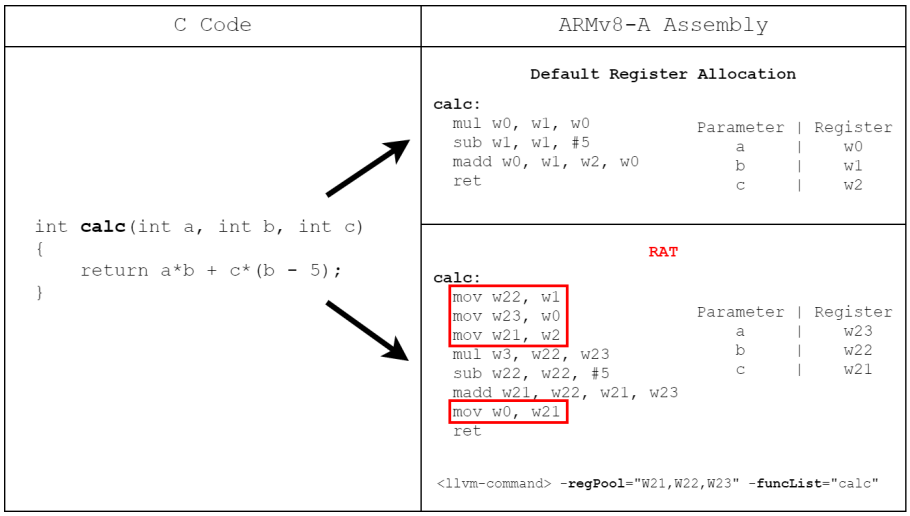


Fig. 2: Example of C code conversion to ARMv8-A assembly without and with RAT flags compilation.

The left-side of Fig. 2 shows an example of a C language function that takes three integer parameters as input, performs arithmetic operations, and returns an integer value. The resulting 64-bit ARM (Aarch64) assembly code is shown in the right-side of Fig. 2, where at the top the default register allocation is shown. In turn, at the bottom right of Fig. 2 the RAT technique is applied, limiting the function register pool to "W21, W22, W23". By the calling convention, the ARMv8-A general-purpose registers with indexes from 0 to 7 are used for inputs and result. When restricting registers outside this range, the compiler only needs to insert some MOV instructions at the beginning and end of the function. As

mentioned before, RAT is a compiler-based mitigating technique, thus it can be associate with other techniques as well. Such capacity is explored in Section 5.

4 Experimental Setup and Evaluation Metrics

In order to demonstrate the effectiveness of RAT, we adopted the fault injection simulator proposed in [5], which is also implemented on the basis of OVPsim. Fault analyzes are obtained by injecting faults (i.e., bit-flips) in the general-purpose registers (i.e., X0-X30) of a dual-core and a quad-core ARM Cortex-A72, in a random order. Conducted experiments include 320K fault injections in a realistic software stack including unmodified Linux kernel, a standard parallelization library (OpenMP), and considering 13 applications taken from the Rodinia Benchmark Suite [9] as shown in Table 1. One of the main concerns when assessing the reliability of a system is to develop a precise, well-covered and realistic approach. In this sense, this work sought to ensure that the number of fault injections has a statistical significance by applying the equations developed by [15]. This work injects 3100 faults per campaign, thus generating a 1.75% error margin with 95% confidence level.

Table 1: Rodinia Benchmarks

#	Benchmark	Domain
`A`	Backprop	Pattern Recognition
`B`	BFS	Graph Algorithms
`C`	HeartWall	Medical Imaging
`D`	HotSpot	Physics Simulation
`E`	HotSpot3D	Physics Simulation
`F`	Kmeans	Data Mining
`G`	LUD	Linear Algebra
`H`	Myocyte	Biological Simulation
`I`	NN	Data Mining
`J`	particle-filter	Medical Imaging
`K`	PathFinder	Grid Traversal
`L`	SradV1	Image Processing
`M`	SradV2	Image Processing

Depending on the application's nature, the three categories classification described in Section 2. A may be inadequate to express all the possible misbehaviors. With this in mind, the results are classified according to Cho [12], which defines five possible behaviors for a system in the presence of a fault: **Vanish**: no fault traces are left in both memory and architectural state; **Output not Affected (ONA)**: the resulting memory is not modified, however, one or more remaining bits of the architectural state is incorrect; **Output Memory Mismatch (OMM)**: the application terminates without any error indication, and

the resulting memory is affected; **Unexpected Termination (UT)**: the application terminates abnormally with an error indication; **Hang**: the application does not finish requiring a preemptive removal.

Software engineers might categorize the criticality of application functions entirely differently depending on their criteria and/or system domains. For the sake of simplicity, this work assumes that most executed functions are the critical ones. Although not ideal, such an approach is adequate to evaluate the benefits and drawback of the proposed mitigation technique. RAT reliability, code and performance overheads are compared against the selective TMR implementation (i.e., VAR3+) [4].

4.1 Reference Mitigation Technique - Selective TMR

In [11], the authors describe a set of rules for data-flow techniques that aim to detect faults affecting values stored in registers bank and memory devices. In this work, we use a triplication instead of duplication since the target is to mitigate the occurrence of soft error. The selective TMR technique implementation was based on [8] inside the Clang/LLVM 6.0.1. The VAR3+ technique was chosen due to its capability of increasing reliability while maintaining a low code and performance overhead compared with previous TMR-based techniques. In this technique, each register has a replica (rule G1), and all instructions, except for branches and stores, are replicated (D2). The replicas are checked before every load, store, or branch instruction (C3, C4, C5, C6). Some acronyms used in the following sections are `RAT`: reference application + register allocation technique, `TMR`: selective TMR technique (VAR3+), and `TMR+RAT`: TMR + register allocation technique.

4.2 Evaluation Metrics

To adequately assess the soft error mitigation technique reliability, [20] introduced a metric called Mean Work To Failure (MWTF), which is calculated by the average amount of work that an application can perform for each error. A unit of work is a general concept whose specific definition depends on the application. The unit work is defined here as a correct program execution (i.e., Vanished fault), while the number of errors is defined as the sum of ONA, OMM, UT, and Hang results as shown in (1).

$$MWTF = \frac{Vanished}{ONA + OMM + UT + Hang} \tag{1}$$

This work also employs the *Fault Coverage* metric, which describes the percentage of faults that are either detected or masked. It is represented as the ratio of detected and masked faults (i.e., Vanished) to the total number of faults that occurred, as shown in (2).

$$F_{coverage} = \frac{UT + Vanished}{ONA + OMM + Hang} \tag{2}$$

Finally, we use the Fault Coverage Increase (FCI) to describe the gain in the percentage of fault coverage when comparing the mitigation techniques.

5 RAT Efficiency Analysis

5.1 RAT Code and Performance Overhead

To provide relevant overhead measures, the **code size** information was extracted from the application object files, while the **performance** figures were obtained from the gem5 full system simulator [7].

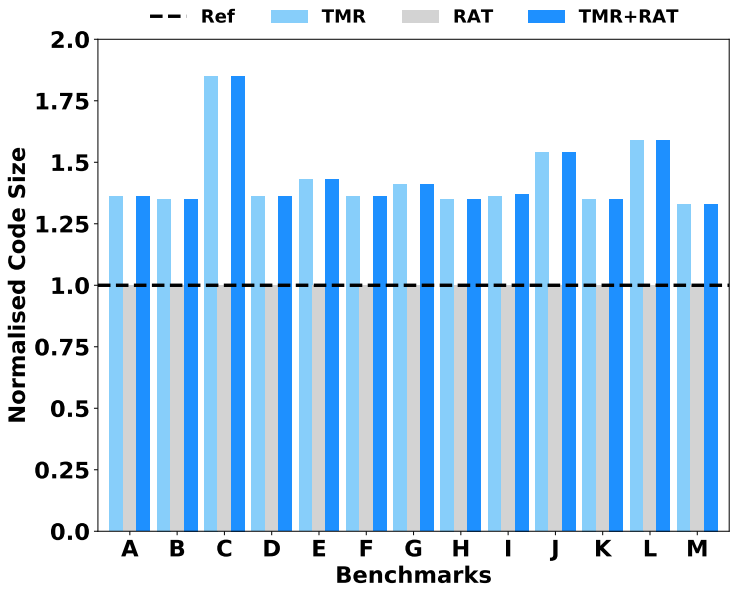
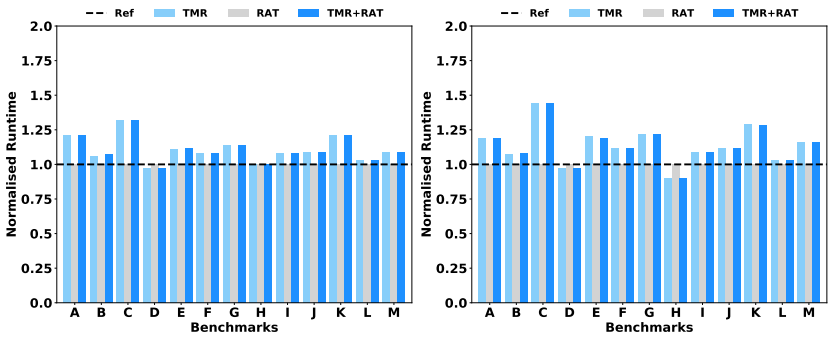


Fig. 3: Code size overhead for ARM Cortex-A72 processor when comparing the impact of the mitigation techniques with the original reference benchmark (Ref).

Fig. 3 shows a substantial code size overhead (e.g., up to 84.86% - benchmark C) when the `TMR` technique is used. In turn, the cost of the proposed technique is negligible, 0.15% in the worst case (benchmark K). Such low overhead is due to the `RAT` approach, which only adds MOV instructions at the beginning and end of the critical functions. As a consequence, the performance of applications is not jeopardized when `RAT` is used (i.e., less than 1% for all scenarios).

Results in Fig. 4a and 4b show that the use of the `TMR` can lead to up to 38.5% and 50% of performance penalty (benchmark C) when running on dual and quad-core ARM Cortex A72 processors. The reason why there is an increase in the execution time in the quad-core when compared to the dual-core is due to

the increasing execution of OS thread synchronization routines that is not linear with the number of cores. Note that the additional execution time of `TMR` is small for a technique that triples instructions and inserts voters into the code. This behavior is justified by the fact that only instructions inside the application's scope are replicated, and the majority of Rodinia applications rely on external library calls. One possible solution to this problem implies replicating function calls; however, there are possible collateral damages inherent to this approach (e.g., modifying the same data structure multiple times).



(a) Dual-core performance overhead (b) Quad-core performance overhead

Fig. 4: Performance overhead for dual (b) and quad-core (c) ARM Cortex-A72 processor when comparing the impact of the mitigation techniques with the original reference benchmark (Ref).

5.2 RAT Soft Error Reliability Evaluation

Techniques Comparison Fig. 5 and 6 show the reliability comparison between the three mitigation techniques. In terms of MWTF on Fig. 5, the `TMR` implementation provides higher reliability in 5 out of 13 cases (C, D, F, I, K), while the `RAT` in 4 cases (A, E, J, L), and the `TMR+RAT` in the other 4 cases (B, G, H, M). Results show that RAT can also provide reliability improvements of up to 40% in some cases compared to `TMR`. Results also show that, depending on the application nature, `TMR+RAT` is an appropriated combination to improve system reliability. For instance, taking the benchmarks B and K as examples, it is possible to identify a considerable difference in the MWTF gain when comparing the two TMR implementations. While benchmark B showed a reliability improvement of 40% for `TMR+RAT`, the use of `TMR` provides an improvement of 51% for K.

Fig. 6 shows a significant increase in the FCI average compared to the results in the dual-core processor, 5.47% versus 1.48%. Note that all reliability metrics

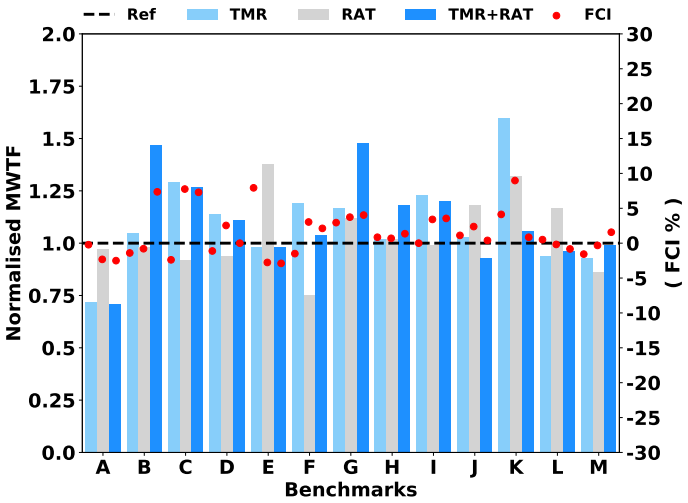


Fig. 5: Normalized reliability comparison between each technique considering the original benchmark code as reference (Ref) for a dual-core ARM Cortex-A72.

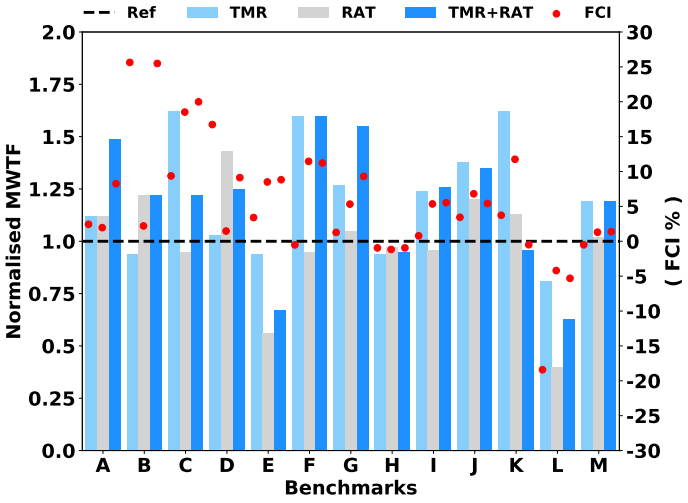


Fig. 6: Normalized reliability comparison between each technique considering the original benchmark code as reference (Ref) for a quad-core ARM Cortex-A72.

have been reduced from dual-core to quad-core, and the increase was only about the reference benchmark. This behavior occurs due to the rise in the execution of thread management tasks, which have a higher susceptibility to soft errors, as mentioned earlier. The `TMR` technique obtained better reliability results in 6 of the 13 benchmarks (C, E, F, J, K, L), `RAT` was better in 2 cases (D, H), and

`TMR+RAT` was better in 5 cases (A, B, G, I, M). Note that the applications' reliability varies from one mitigation technique to another. For that reason, we claim that engineers can use our toolset to analyze the impact of different mitigation techniques at the system-level, so they might be able to identify the most suitable one considering their application's/system's constraints. Further, a more in-depth analysis is carried out, verifying the results of the fault injections in each register for a specific case study.

Registers Criticality Analysis Fig. 7 shows how the 64-bit ARM (AArch64) calling convention works. The X0-X7 registers are used for input parameters and return functions; the X8 is used to hold an indirect return location address; the X9-X15 are used to hold local variables (caller saved); the X16 and the X17 are the Intra-Procedure-call scratch registers; the X18 can be used for some OS-specific purpose; the X19-X28 are callee-saved registers; the X29 is the frame pointer; while the X30 is the link register, used to return from subroutines. To better explain the `RAT` benefits, we chose the particle-filter benchmark (**J**) as a case study.

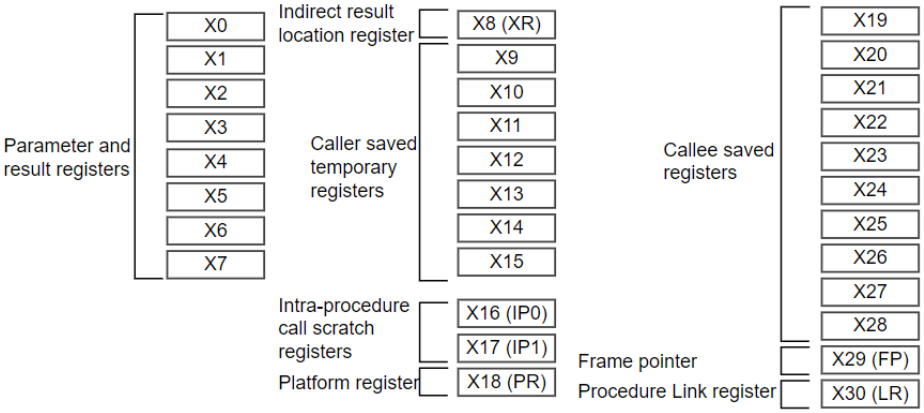


Fig. 7: Allocation of the general-purpose registers following the AArch64 calling convention. [2]

The results show that half of the registers (X0-X16) do not suffer significantly from soft errors (Fig. 8), when the particle-filter benchmark (**J**) is executed on a dual-core ARM Cortex-A72 processor. In contrast, the rest of the registers suffers strongly from the injected faults. Especially the callee-saved category that is used to hold long-lived values that must be preserved across calls and are used by the Linux kernel. Theoretically, there are registers that take a longer time to get written, but they are continuously read. However, as shown in Fig. 9, the fault masking increases when we apply the `RAT` technique and limit the number of registers that will be used to execute the most performed function. In general,

this effect occurs because when entering the critical function, the callee-saved registers are saved in memory and return to their original values at the end of the execution. In practice, this behavior ends up reducing the lifetime of these registers, making them more resilient to soft errors. The best examples are from the X17 and X19 registers. For the X17 register, we have a fault-masking rate of 70% in the reference application, and 98% when using the `RAT` mitigation technique. For the X19 register, we have a fault-masking rate of 37% in the reference application, and 58% when using the `RAT` technique.

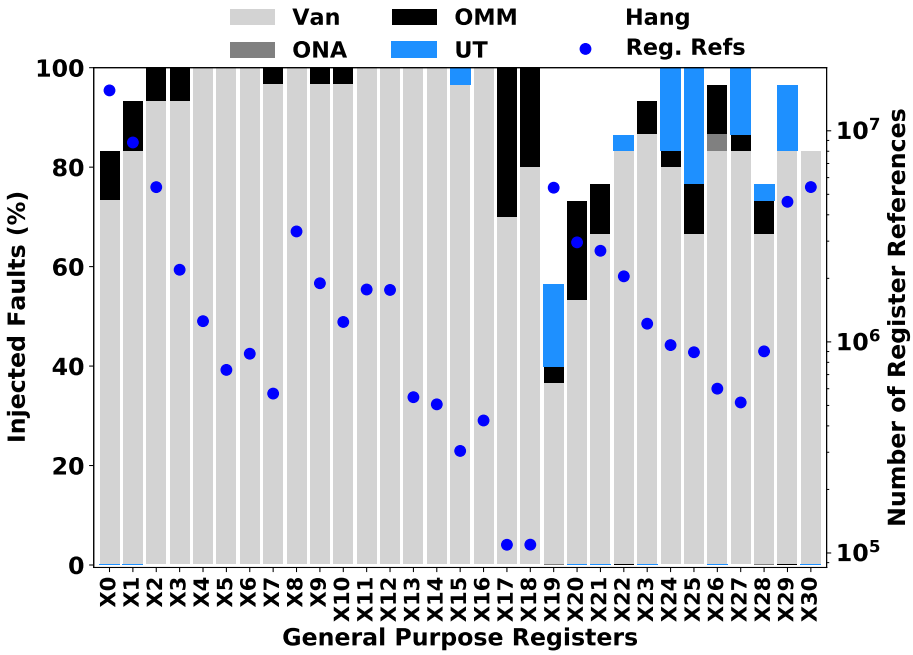


Fig. 8: Registers criticality for the Reference particle-filter benchmark running on a dual-core ARM Cortex-A72.

Results demonstrated that RAT reduces the code size and performance overheads while providing reliability improvement when considering a state-of-the-art 64-bits processor, which has a large register pool (i.e., 32 general-purpose registers). Researchers and industrial leaders are also developing optimized machine-learning algorithms [1], aiming to enable their execution in resource-constrained devices. The resulting scenario calls for lightweight soft error mitigation techniques such as the one proposed here. The next Section investigates the RAT efficiency when applied to a more resource-constrained architecture.

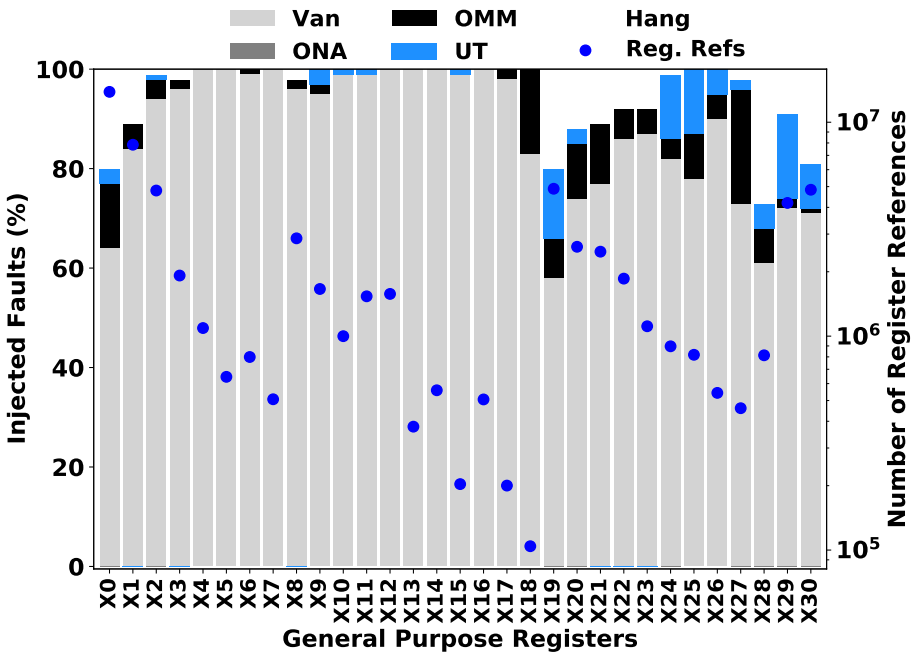


Fig. 9: Registers criticality for the RAT version of particle-filter benchmark running on a dual-core ARM Cortex-A72.

6 RAT Efficiency in Distinct Processor Architectures

To assess the impact of the processor architecture on RAT efficiency, this Section considers the ARMv7-A 32-bit and the ARMv8-A 64-bit instruction set architectures.

6.1 ARMv7-A General-purpose Registers

The ARMv7-A has 16 registers (R0-R15) with 32 data bits each. Removing the special use registers (IP, SP, LR, PC), there are only 12 extra registers that RAT can use to allocate the application critical function. As explained in the Section 5.2, there is also a particular ARMv7-A calling convention. As shown in Figure 10, the initial registers (R0-R3) are used to pass input and function return parameters, the R4-R11 are used for local variables, and the R12-R15 are special registers responsible for managing stack, function return address, and jumps during the application execution.

For example, if a routine has more than four arguments, besides using R0-R3, the stack will need to be used to store the extra parameters. Moreover, if R4-R11 are not sufficient, R0-R3 and R12 can be used, and even LR when there are no other subroutine calls.

Registers	Function	Value preserved during call
R0-R3	Arguments / Return values	No
R4-R11	Local variables	Yes
R12 (IP)	Intra-procedure-call scratch reg.	No
R13 (SP)	Stack Pointer	Yes
R14 (LR)	Link register	No
R15 (PC)	Program Counter	No

Fig. 10: Register usage for ARMv7 architecture.

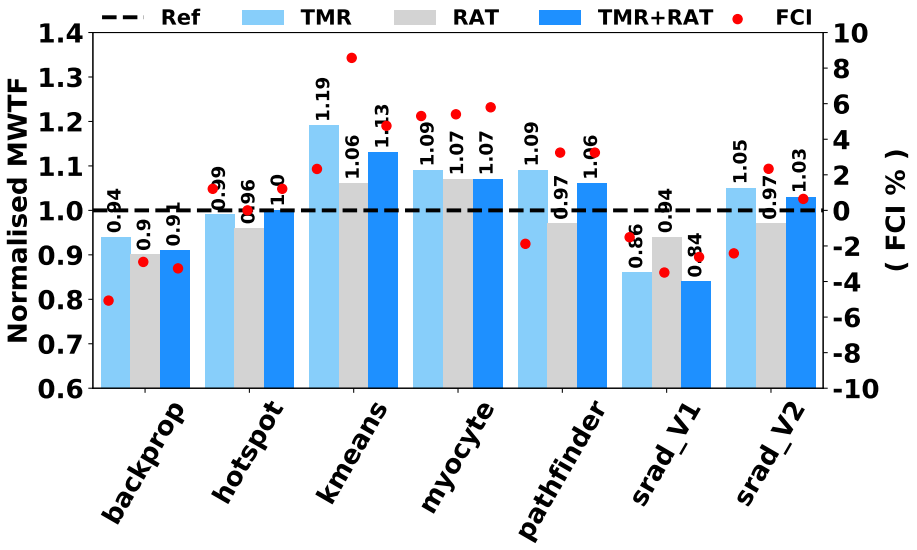
6.2 Soft Error Reliability Assessment for the ARMv7-A considering different Mitigation Techniques

In order to understand how limiting the number of available registers affects the soft error reliability results, the experiments consider a subset of seven applications of the Rodinia Benchmark Suite executing in dual-core and quad-core Arm Cortex-A9 processors. For each scenario, 1600 SEU fault injections were performed targeting the 16 general-purpose registers. Based on the equation defined in [15], our results have a margin of error of 2.45% with a 95% confidence level.

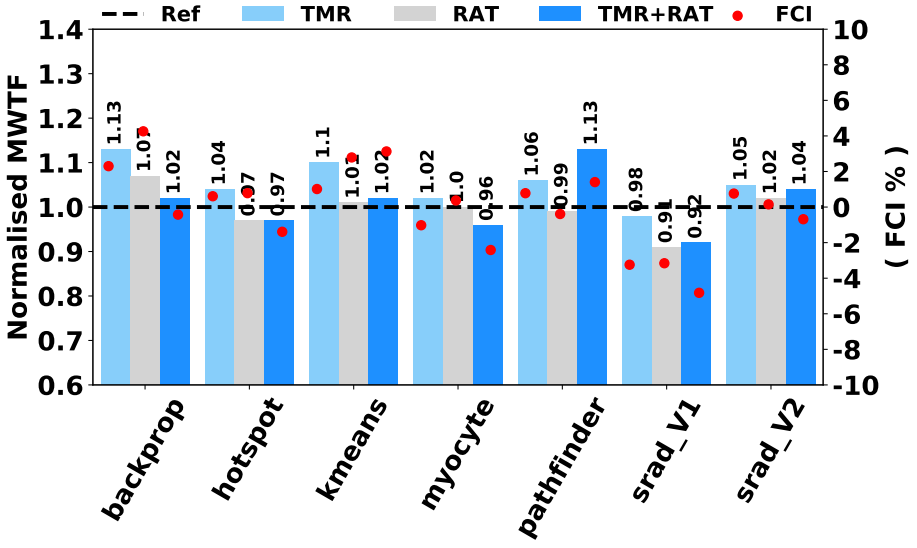
Figure 11 shows the MWTF results normalized by the reference application version indicated on the left y-axis. Each bar in the graph indicates a mitigation technique, and each group of three bars refers to a different application. The right y-axis shows the increase/decrease in the fault coverage for each case, which are indicated by the red dots. Following the same format adopted in the previous Section, results consider dual-core and quad-core processor architectures and the three soft error mitigating techniques (i.e., TMR, TMR-RAT, and RAT). For the dual-core results (Figure 11a), it is possible to observe a low soft error reliability improvement when applying the mitigation techniques (see MWTF and FC values). While TMR presents the higher MWTF factor for the kmeans application (19%), RAT shows the best FCI factor for the same application (9%).

The application of `RAT` leads to a low reliability improvement (MWTF factor equal to 7% - best case) at a low extra code overhead. The low reliability improvement is expected; since the number of available registers is low, the registers' allocation can be precisely the same as the reference version if the function defined as critical already uses all possible registers.

Quad-core soft error reliability results (Figure 11b) provide a lower MWTF and FCI average compared with the dual-core configuration. The more cores the higher is the probability of a fault happening during the operating system execution. In this case, the operating system puts more pressure on the registers, leading to more spilling to temporary values stored in memory, thus requiring an



(a) Dual-core reliability results



(b) Quad-core reliability results

Fig. 11: Reliability improvement for dual (a) and quad-core (b) Arm Cortex-A9 processor when comparing the impact of the mitigation techniques with the original reference benchmark (Ref).

increase in the proportional time slice of the application's total execution. This effect reduces the chance of a fault being masked within one of the hardened functions. For instance, the best achieved FCI factor is only 4% when RAT is applied to the backprop application. In turn, the higher MWTF factor of 13% is achieved when TMR is applied for the same application.

6.3 RAT Soft Error Efficiency Comparison: ARMv7-A *vs* ARMv8-A

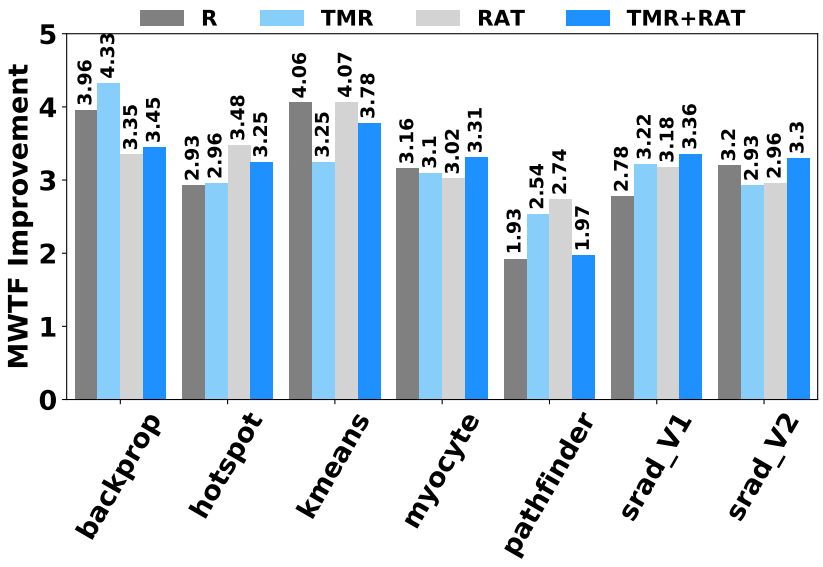
The purpose of this section is to make a more detailed comparison of the reliability results when applying TMR, TMR-RAT, and RAT techniques to seven Rodinia applications running on different processor architectures.

Figure 12 shows the normalized MWTF of each application (i.e., unprotected and protected versions) obtained from the fault injection campaigns considering the Arm Cortex-A72 and the Arm Cortex-A9. Each bar in the 4-bar structure of the graph indicates a different version of each application.

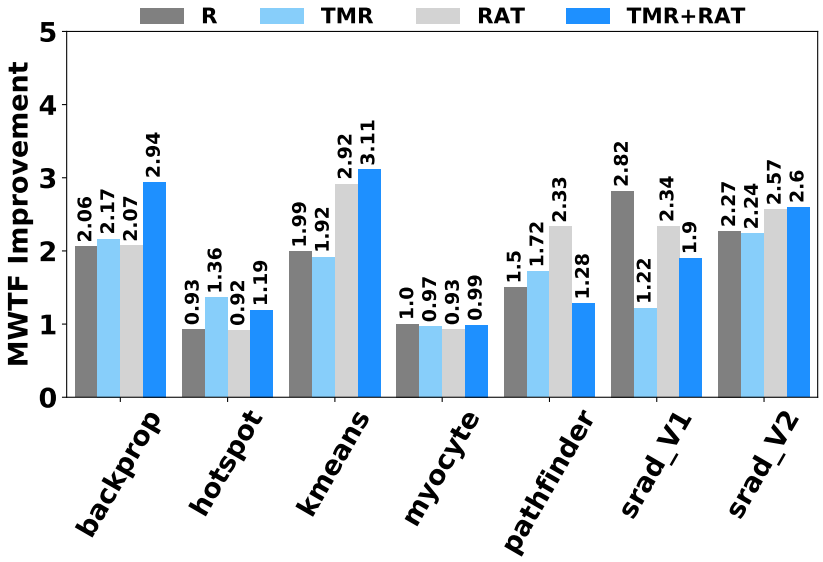
Analyzing Figure 12a, we see that the ARM Cortex-A72 dual-core provides a significant MWTF improvement in all applications. The minimum increase is 1.93× (pathfinder - R), and the maximum 4.33× (backprop - TMR). Results show RAT can benefit from processors with a larger number of registers. Results obtained from the quad-core processor scenarios (Figure 12b), show a reasonable reduction in the MWTF improvement. The minimum reliability improvement of 0.92× is achieved when applying RAT to the hotspot application. In turn, the use of TMR+RAT incurs an improvement of 3.11× for the kmeans application. Therefore, the increase of system resource utilization leads to a decrease of more than 70% in the normalized MWTF in some cases (i.e., hotspot and myocyte).

7 Conclusion and Future Works

The importance of using selective and lightweight soft error mitigation techniques is increasing every year. The results show that redundancy does not always ensure reliability, and the other factors such as code size and performance overheads must be considered. In this regard, the proposed RAT offers a good compromise in terms of reliability improvement, code size overhead, and performance penalty when compared to TMR. Hardened applications, resulting from adopted mitigation techniques, present a lower soft error reliability improvement when executed in the Cortex-A9 (i.e., ARMv7-A ISA). An improvement in the MWTF factor of up to 4.33x is achieved for the same configuration (i.e., mitigation technique and application) when executed in Arm Cortex-A72. Future works include further investigation of RAT considering other processor architectures and more complex benchmarks that do not depends on external libraries. It may also be interesting to analyze the RAT's impact when dealing with floating point registers.



(a) Dual-core reliability mismatch results



(b) Quad-core reliability mismatch results

Fig. 12: Reliability mismatch for dual (a) and quad-core (b) Arm Cortex-A72 processor when comparing with Arm Cortex-A9.

References

1. Abich, G., Gava, J., Reis, R., Ost, L.: Soft error reliability assessment of neural networks on resource-constrained iot devices. In: 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS). pp. 1–4 (2020). https://doi.org/10.1109/ICECS49266.2020.9294951
2. Arm: ARMv8-A parameters in general-purpose registers (2020), https://developer.arm.com/docs/den0024/latest/the-abi-for-arm-64-bit-architecture/register-use-in-the-aarch64-procedure-call-standard/parameters-in-general-purpose-registers
3. Avižienis, A., Laprie, J.C., Randell, B.: Dependability and its threats: A taxonomy. In: Jacquart, R. (ed.) Building the Information Society. pp. 91–120. Springer US, Boston, MA (2004)
4. Azambuja, J.R., Lapolli, A., Altieri, M., Kastensmidt, F.L.: Evaluating the efficiency of data-flow software-based techniques to detect sees in microprocessors. In: 2011 12th Latin American Test Workshop (LATW). pp. 1–6 (2011). https://doi.org/10.1109/LATW.2011.5985914
5. Bandeira, V., Rosa, F., Reis, R., Ost, L.: Non-intrusive fault injection techniques for efficient soft error vulnerability analysis. In: 2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC). pp. 123–128 (2019). https://doi.org/10.1109/VLSI-SoC.2019.8920378
6. Benso, A., Chiusano, S., Prinetto, P., Tagliaferri, L.: A C/C++ source-to-source compiler for dependable applications. In: Proceeding International Conference on Dependable Systems and Networks. DSN 2000. pp. 71–78 (2000). https://doi.org/10.1109/ICDSN.2000.857517
7. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (Aug 2011). https://doi.org/10.1145/2024716.2024718
8. Bohman, M., James, B., Wirthlin, M.J., Quinn, H., Goeders, J.: Microcontroller Compiler-Assisted Software Fault Tolerance. IEEE Transactions on Nuclear Science **66**(1), 223–232 (2019). https://doi.org/10.1109/TNS.2018.2886094
9. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC). pp. 44–54 (2009). https://doi.org/10.1109/IISWC.2009.5306797
10. Chielle, E., Barth, R.S., Lapolli, A.C., Kastensmidt, F.L.: Configurable tool to protect processors against SEE by software-based detection techniques. In: 2012 13th Latin American Test Workshop (LATW). pp. 1–6 (2012). https://doi.org/10.1109/LATW.2012.6261259
11. Chielle, E., Kastensmidt, F.L., Cuenca-Asensi, S.: Overhead Reduction in Data-Flow Software-Based Fault Tolerance Techniques, pp. 279–291. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-14352-1_18
12. Cho, H., Mirkhani, S., Cher, C.Y., Abraham, J.A., Mitra, S.: Quantitative evaluation of soft error injection techniques for robust system design. In: Proceedings of the 50th Annual Design Automation Conference. DAC '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2463209.2488859, https://doi.org/10.1145/2463209.2488859

13. Gava, J., Reis, R., Ost, L.: Rat: A lightweight system-level soft error mitigation technique. In: 2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC). pp. 165–170 (2020). https://doi.org/10.1109/VLSI-SOC46417.2020.9344080
14. Imperas: OVPsim Simulator (2020), http://www.ovpworld.org
15. Leveugle, R., Calvez, A., Maistri, P., Vanhauwaert, P.: Statistical fault injection: Quantified error and confidence. In: 2009 Design, Automation Test in Europe Conference Exhibition. pp. 502–506 (2009). https://doi.org/10.1109/DATE.2009.5090716
16. Martinez-Alvarez, A., Cuenca-Asensi, S., Restrepo-Calle, F., Palomo Pinto, F.R., Guzman-Miranda, H., Aguirre, M.A.: Compiler-directed soft error mitigation for embedded systems. IEEE Transactions on Dependable and Secure Computing **9**(2), 159–172 (2012). https://doi.org/10.1109/TDSC.2011.54
17. Mukherjee, S.S., Emer, J., Reinhardt, S.K.: The soft error problem: an architectural perspective. In: 11th International Symposium on High-Performance Computer Architecture. pp. 243–247 (2005). https://doi.org/10.1109/HPCA.2005.37
18. Nicolescu, B., Velazco, R.: Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results, pp. 39–51. Springer US, Boston, MA (2003). https://doi.org/10.1007/0-306-48709-8_4
19. Reis, G.A., Chang, J., August, D.I.: Automatic instruction-level software-only recovery. IEEE Micro **27**(1), 36–47 (2007). https://doi.org/10.1109/MM.2007.4
20. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., Mukherjee, S.S.: Software-controlled fault tolerance. ACM Trans. Archit. Code Optim. **2**(4), 366–396 (Dec 2005). https://doi.org/10.1145/1113841.1113843
21. Rodrigues, G.S., Kastensmidt, F.L., Reis, R., Rosa, F., Ost, L.: Analyzing the impact of using pthreads versus openmp under fault injection in arm cortex-a9 dual-core pp. 1–6 (2016). https://doi.org/10.1109/RADECS.2016.8093180
22. Serrano-Cases, A., Morilla, Y., Martín-Holgado, P., Cuenca-Asensi, S., Martínez-Álvarez, A.: Nonintrusive automatic compiler-guided reliability improvement of embedded applications under proton irradiation. IEEE Transactions on Nuclear Science **66**(7), 1500–1509 (2019). https://doi.org/10.1109/TNS.2019.2912323
23. Shirvani, P.P., Saxena, N.R., McCluskey, E.J.: Software-implemented edac protection against seus. IEEE Transactions on Reliability **49**(3), 273–284 (2000). https://doi.org/10.1109/24.914544
24. Snir, M., Wisniewski, R.W., Abraham, J.A., Adve, S.V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., et al.: Addressing failures in exascale computing. The International Journal of High Performance Computing Applications **28**(2), 129–173 (2014). https://doi.org/10.1177/1094342014522573