An ALU Protection Methodology for Soft Processors on SRAM-Based FPGAs

Alexis Ramos, Ricardo G. Toral, Pedro Reviriego, and Juan Antonio Maestro

Abstract—The use of microprocessors in space missions implies that they should be protected against the effects of cosmic radiation. Commonly this objective has been achieved by applying modular redundancy techniques which provide good results in terms of reliability but increase significantly the number of used resources. Because of that, new protection techniques have appeared, trying to establish a trade-off between reliability and resource utilization. In this paper, we propose an application-based methodology, to protect a soft processor implemented in an SRAM-based FPGA, against the effect of soft errors. This is done creating a library of adaptive protection configurations, based on the profiling of the application. This hardware configuration library, combined with the reprogramming capabilities of the FPGA, helps to create an adaptive protection for each application. We propose two partial TMR configurations for the Arithmetic Logic Unit (ALU) as an example of this methodology. The proposed scheme has been tested in a RISC-V soft processor. A fault injection campaign has been performed to test its reliability.

Index Terms—ALU, fault tolerance, FPGA, RISC-V, soft core, soft error

1 Introduction

Space is a harsh environment for any electronic circuit, like microprocessors. Radiation can affect its correct behaviour, causing fatal consequences in a satellite like Anik E-1 and Anik E-2, which started to spin without control [1]. It is because of this that electronics systems like microprocessors, and therefore *soft processors*, should be protected in order to be used in space. However, the protection could imply a significant increase in the amount of resources or the power consumption. This paper proposes a methodology to reduce these costs, creating a catalog of selective TMR configurations that could easily be loaded in an FPGA thanks to its reconfiguration capability. To test this approach, we have protected a functional unit against Single Event Upsets (SEU), in particular, the Arithmetic Logic Unit (ALU). We have chosen to protect this unit because it is one of the largest units of the pipeline and, consequently, the amount of impacts it will receive would be higher. Due to the fact that the soft processor is implemented in an SRAM-based FPGA, SEUs will be the most common type of error, because the configuration memory is the predominant structure in this platform [2].

The FPGA is composed by two types of memories: the configuration memory and the user memory. The first one, implements the logical behavior of the design while the second one holds the user memory bits used by the design during the execution time. Depending on which one of these types of memories is the one affected by an SEU, the effect will be different. On one hand, an SEU will create a persistent error in the configuration memory, until it is restored or reprogrammed. On the other hand, an SEU in user memory will create a transient error until the corrupted data is overwritten. Protection against SEU can be achieved through the use of alternative fabrication processes, following *Radiation Hardening by Process (RHBP)* methodologies, or by altering the design to mitigate or prevent errors, as in the case of *Radiation Hardening by Design (RHBD)* approaches. Since RHBP implies modifying the CMOS manufacturing process and therefore increasing the cost of the platform [3], this work is focused on RHBD. This is the approach used by most projects which utilize *Commercial Off The Shelf (COTS)* based technology, typically chosen for academic or similar institutions with a tight budget [4]. RHBD techniques are based on altering the cell design or introducing redundancy in the circuit, which can be modular, temporal or informational. Finally, the configuration memory of an FPGA (in particular the SRAM-based ones) can be protected by memory scrubbing, a technique that periodically checks the memory to correct the errors [5].

Regarding information redundancy, the usual protection approaches are based on single parity checks [6] or Error Detection and/or Correction Codes like Hamming [7]. These codes require less additional storage units than modular redundancy, but they increase the latency of the system [8]. For this reason, numerous alternative code designs focused on minimizing the impact on performance have been proposed [9], [10].

In the mentioned modular redundancy, the most common techniques are *Double Modular Redundancy (DMR)* and *Triple Modular Redundancy (TMR)* [11], which duplicate or triplicate the module or the circuit to be protected, and use a comparator or a majority voter to detect or mitigate the error, respectively. Some works like [12] rely on *NMR* to protect this functional unit. The problem with this approach is that it significantly increases the area. Furthermore, because a DMR is used, no correction is done. This means that in order to avoid a faulty result, the operation must be recalculated, introducing a delay in the pipeline execution of at least one clock cycle.

Because of that, some authors have proposed ways to establish a trade-off between the reliability and the resources utilization [13], [14], [15], [16]. Authors in [17] present a technique called *Reduced Precision Redundancy (RPR)*, which replicates part of the design to protect only a subset of bits in the outputs. This creates a redundant system that requires less resources, but provides lower protection capabilities. Other authors like [18], focus on protecting the ALU saving all the resources that they can. In order to do this, they create an additional path, which comprises an auxiliary ALU, that calculates the parity of the current operation. This way the normal path as well as the parity path work in parallel. After the execution stage, the parity is checked beetwen the default ALU result and the one from the parity path. Using this approach in combination with other protections, single errors can be corrected. The problem is that if there are more than one errors, the result cannot be corrected. Apart from that, the parity calculation of the result plus the comparison and correction, could imply a delay. In an effort to make an optimal protection technique that saves resources as much as possible, some authors exploit the architecture of the module. This is the case of Hasan et al. [19], where they managed to protect this functional unit against SEUs, by using the inherent redundancy of the ALU implementation plus data encoding. All these works show a tendency in finding a solution that makes a trade off between resources utilization and reliability. The way to achieve this is to focus on creating an *ad-hoc* design technique that exploits the features of the ALU or the application. The authors in [20] propose a design hardening which is performed by doing an analysis of the circuit. In addition, they also suggest a hybrid technique which duplicates the module to be

- *A. Ramos and J.A. Maestro are with the ARIES Research Center, Escuela Politécnica Superior, Universidad Antonio de Nebrija, Pirineos 55, Madrid 28015, Spain. E-mail: {aramosam, jmaestro}@nebrija.es.*
- *R.G. Toral is with Xilinx, Edinburgh EH2 2PQ, Scotland. E-mail: rgonzale@xilinx.com.*
- *P. Reviriego is with the Department of Telematics Engineering, Universidad Carlos III de Madrid, Avda. Universidad 30, Leganes 28911, Spain. E-mail: revirieg@it.uc3m.es.*

Manuscript received 18 Apr. 2018; revised 18 Mar. 2019; accepted 20 Mar. 2019. Date of publication 24 Mar. 2019; date of current version 15 Aug. 2019.
(Corresponding author: Alexis Ramos.)
Recommended for acceptance by J. D. Bruguera.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2019.2907238

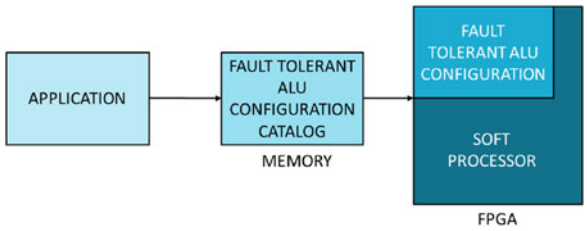

Fig. 1. Proposed system architecture.

protected, and takes a few outputs for comparison and error detection. In this way, the error can be detected and in a next stage, corrected. Regarding embedded microprocessors, there are some common techniques to deal with SEUs and SEEs like control flow checking [21] and data path protection [22]. However, in most cases, these techniques suffer from a big overhead in terms of resources or time.

Our proposed scheme tries to shed new light in ALU protection, particularly for microprocessors implemented in FPGAs. In order to do this, we customize the protection of this functional unit according to a dynamic profiling of the application. The profiling is used to identify the most executed assembler instructions of the program, that are computed in the execute stage of the pipeline. This work assumes that the microprocessor is going to be embedded in a system that will execute a reduced set of programs. Because of this, the profiling operation will be needed to be done only once. This is a common situation in satellites.

With this idea in mind, the architecture of the ALU can be modified to replicate only the paths used by those instructions. This will allow to create a library of protected paths that could be loaded easily in the FPGA on demand.

The rest of the paper is organized as follows. Section 2 presents a protection technique with two approaches based on redundancy, while Section 3 describes our case study. Section 4 presents the evaluation process and shows the results, while Section 5 concludes this work.

2 Protection Methodology

The proposed scheme presents a methodology to protect the ALU of a soft processor against the effects of SEUs, using a catalog of fault tolerant application-based configurations. The goal of this approach is to reduce the number of resources (compared with other techniques like TMR) by establishing a trade-off with fault tolerance. This protection only applies to the configuration memory. Furthermore because it is larger in size than the user memory, it will be affected by SEUs more frequently [23], [24]. Thanks to the reconfiguration capabilities of the SRAM-based FPGAs, a set of fault tolerant ALU designs can be stored in the system. Each of them is designed to protect this module of the microprocessor according to the program that is going to be executed. When the program changes, another ALU design adapted to that program will be loaded. Fig. 1 presents the architecture of the system.

In order to test this approach we have created an application-adapted partial TMR. The particularity of this TMR with respect to the conventional one is that two of the copies only include the hardware related to the instructions most used by the program. This strategy can be applied at different abstraction levels. In our case, we have considered two different implementations. The first one creates two partial copies of the ALU. The second one targets the resources devoted to each instruction inside the ALU, which yields a single circuit with a subset of triplicated logic paths. These two approaches achieve lower protection levels than TMR but with less resources utilization and no timing penalty in terms of pipeline synchronization.

Fig. 2. Methodology workflow.

TABLE 1
Most Used Instructions per Benchmark

Benchmark	Most used instruction	Second most used instruction
Matrix mul.	Multiplication	Addition
Quicksort	Shift	Comparison
Fibonacci	Addition	Comparison
Dijkstra	Shift	Addition

The proposals have been tested in a RISC-V [25] soft processor, in particular the lowRISC implementation [26]. In this case study, four well known programs have been used as applications to create the configuration catalog. We have chosen the two most used instructions of each one. This increases the fault tolerance with respect to the unprotected design significantly while requiring a lower resource occupation than a full TMR. If more instructions were to be protected, both the number of resources and the error resilience would increase. A fault injection campaign has been performed to evaluate the reliability of these configurations.

Fig. 2 summarizes the methodology workflow. First of all, the application that is going to be run in the microprocessor is profiled. That analysis identifies the most used instructions of the algorithm. With that information, an adaptive protection is created.

2.1 Dynamic Profiling

As stated before, the design of the fault tolerant ALU configuration will depend on the application that is going to be run. These variations will be associated to the differences in use of each instruction determined by the operations performed by the program. Because of this, a dynamic profiling is necessary in order to estimate which the most used instructions are. This process should be repeated for each application that is going to be executed in the microprocessor.

In this particular case, the algorithms that were tested are the matrix multiplication, Quicksort, the iterative Fibonacci sequence and Dijkstra's algorithm. Each of them exercises the ALU in different ways, which means that different instructions will be used. For each one, a dynamic analysis was performed in order to determine the most used instructions. Table 1 summarizes the results of this analysis. In the case of matrix multiplication the most used instruction is the multiplication, followed by the addition. Regarding Quicksort, the most used instruction is the shift operation (due to the movement of the pivot) followed by the comparison. However the difference between them in terms of repetitions is small. In the Fibonacci algorithm, the most used instruction is the addition followed by the comparison. This is because it was coded in an iterative way. Finally, the most used instructions in Dijkstra's algorithm are shift and addition.

All the benchmarks have been executed 10,000 times each one. The execution time (per run) is the following:

1) Dijkstra: 13 ms.
2) Fibonacci: 20 ms.
3) Matrix multiplication: 17 ms.
4) Quicksort: 148 ms.

It must be noted that the ALU will be protected by partial TMR only for the instructions above. If another instruction is not taken up in this set of selected instructions (for each program), then it will be steered to the default path, without using the TMR voting. The design of the fault tolerant module will be explained in the next section.

Finally, Fig. 3 shows the most used instructions per benchmark. For the sake of clarity, only the five most used instructions are represented. For simplicity, we have decided to protect the two most

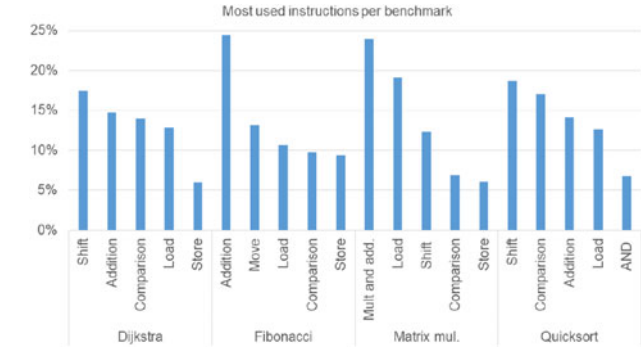

Fig. 3. Five most used instructions per benchmark.

used instructions but more can be protected, if the designer decides so, using the presented methodology.

2.2 ALU Protection

As has been introduced, two different approaches were followed in order to implement the redundant copies. Each of them are described in the following lines.

2.2.1 ALU Replication

The first approach is based on a partial TMR [27], as seen in Fig. 4. The node tagged as *ALU 1* implements the full logic. The nodes tagged as *ALU 2* and *ALU 3*, implement only the logic necessary to perform the most used instructions. In this example, only one operation is represented in order to make the figure clearer. Since *ALU 2* and *ALU 3* will always execute the *add* instruction regardless of the input *Operation code*, it is necessary to exclude these results from the vote in the *TMR* system whenever another operation is performed. To this end, the voter also receives the same *Operation code* as the main ALU. Therefore, the result is only voted when the instruction is an addition. This type of TMR will hereafter be called *TMR A*.

2.2.2 Selective Replication Inside the ALU

Fig. 5 shows the other type of TMR, hereafter called *TMR B*. This approach creates a selective replication in the path affected by the instruction to be protected (add instruction). In this case it is not necessary to use the *operation code* in the voter system because all the logic related with steering the results of the adders is managed by the default path of the ALU.

In summary, TMR A triplicates the ALU but only using the amount of resources needed to protect the selected instructions. TMR B only use one ALU, in which the specific paths affected by the instructions that are going to be protected are triplicated. Because both types of TMR could be implemented in a FPGA, they could be swapped as needed thanks to the reconfiguration capability of the platform.

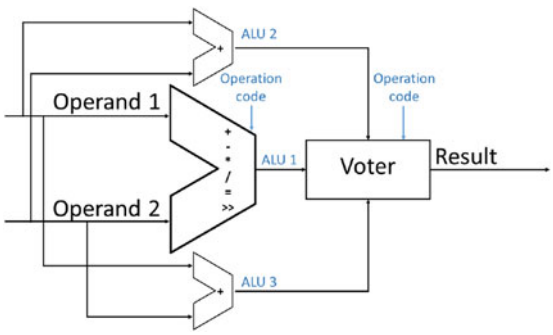

Fig. 4. Selective TMR A, on which two partial ALU modules are used to protect the most used instruction (add instruction).

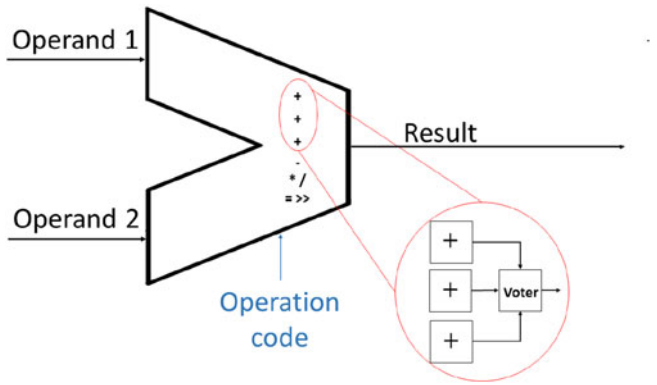

Fig. 5. Selective TMR B at module level, there is only one ALU which has the adders triplicated.

3 Case Study

As stated before, the case study will consider two instructions to protect for each of the four benchmarks. Using the proposed methodology represented in Fig. 2, those instructions will determine the design of the protected ALU implementations. Based on the information in Table 1, six alternative designs were created for each benchmark, three of them per selective TMR alternative. These three designs were created to protect the most used instruction, the second most used instruction, and both of them. The soft processor used for this case study was lowRISC loaded in a Nexys 4 DDR board [28], which contains an Artix-7 FPGA [29]. The synthesis and implementation of the design were done using the Xilinx Vivado synthesis tools [30].

The lowRISC microprocessor is coded using Chisel [31]. Chisel is a high level language similar to the Scala language, which generates RTL code and it is used on most RISC-V implementations [32]. Therefore, to keep the consistency between lowRISC and our design, the protection techniques were coded using this language. Regarding the resources utilization, since most of the ALU uses combinational logic, the most used memory is the configuration one. Because this reason, the appropriate resource to take into account is the *Look Up Table (LUT)*. The number of used flip-flops barely change in comparison with the default case.

Tables 2, 3, 4, and 5 summarize the resource utilization of each protected design depending on the benchmark program. As can be seen, TMR A requires a higher number of resources than TMR B due to the increased complexity of the voting strategy.

These circuits represent a catalog of protected designs for the ALU, which allows to implement a fault tolerance strategy tailored to the characteristics of the program to be run in the microprocessor. To evaluate the error resilience of each alternative design, they were implemented in an FPGA and subjected to a fault injection campaign, as discussed in Section 4.

TABLE 2
Resources Used by the Matrix Multiplication Algorithm

TMR type	Protected instruction	LUTs as logic	Resources overhead
-	Without protection	1891	-
Full TMR	All	5688	200,79%
TMR A	Multiplication	5073	168,27%
	Addition	2127	12,48%
	Multiplication and addition	5563	194,18%
TMR B	Multiplication	2314	22,37%
	Addition	2021	6,87%
	Multiplication and addition	2645	39,87%

TABLE 3
Resources Used by the Quicksort Algorithm

TMR type	Protected instruction	LUTs as logic	Resources overhead
-	Without protection	1891	-
Full TMR	All	5688	200,79%
TMR A	Shift	2519	33,21%
	Comparison	2112	16,98%
	Shift and comparison	2534	34,00%
TMR B	Shift	2454	29,77%
	Comparison	2095	10,79%
	Shift and comparison	2473	30,78%

TABLE 4
Resources Used by the Fibonacci Algorithm

TMR type	Protected instruction	LUTs as logic	Resources overhead
-	Without protection	1891	-
Full TMR	All	5688	200,79%
TMR A	Addition	2127	12,48%
	Comparison	2112	16,98%
	Addition and comparison	2213	17,03%
TMR B	Addition	2021	6,87%
	Comparison	2095	10,79%
	Addition and comparison	2161	14,28%

TABLE 5
Resources Used by the Dijkstra Algorithm

TMR type	Protected instruction	LUTs as logic	Resources overhead
-	Without protection	1891	-
Full TMR	All	5688	200.79%
TMR A	Shift	2519	33.21%
	Addition	2127	12.48%
	Shift and addition	2672	41.30%
TMR B	Shift	2454	29.77%
	Addition	2021	6.87%
	Shift and addition	2586	36.75%

4 Evaluation

An FPGA fault emulation based campaign was performed in order to test the correct behavior of the fault tolerant designs. We chose to use this type of SEU emulation test because it provides enough precision and the cost is lower than other techniques [33]. The fault injector emulator used to perform the campaign was SEM IP [34].

Prior to performing the campaign, the fault model must be defined. This fault model will be isolated errors (SEU) in the configuration memory of the FPGA. Depending on the output, the result of the SEU can be classified into five different categories. These categories have been defined using other works as reference, like the one presented by Cho [35] and other authors. The output of the benchmark is the result of the algorithm alongside with the context of the microprocessor. The reason to retrieve the context is that even if the result is correct, a failure in the context could create a further problem. The error categories are the following:

- *Correct Result:* The output of the algorithm as well as the context are correct.
- *Hard Fault:* There is an exception in execution time.
- *Hang:* The execution does not finish.
- *Architecture Internal Failure (AIF):* The output is correct, but the context is erroneous and a failure could occur later. This type of error could be considered as an *Unnecessary for Architectural Correct Execution) (unACE)* fault [36], due to the fact that is not critical for the correct execution of the architecture *ACE*.
- *Application Output Mismatch (AOM):* An error causes an incorrect result in the result of the algorithm. These type of errors are also called *Silent Data Corruption (SDC)* [36].

Since we are testing the isolated failure model, we only inject one error per execution. The results are retrieved after each benchmark execution and saved in a computer. At the end of the campaign, all the results are compared with the *golden* outcome. Because we are protecting the ALU, we have only injected errors in this module.

The fault injection results are presented in Tables 6, 7, 8 and 9.

TABLE 6
Results for Matrix Multiplication Benchmark

TMR type	Protected instruction	Correct	Hard Fault	Hang	AOMs	AIFs
-	Without protection	86,19%	2,40%	5,57%	4,96%	0,88%
Full TMR	All	99,53%	0,06%	0,19%	0,20%	0,02%
TMR A	Multiplication	99,06%	0,09%	0,36%	0,45%	0,04%
	Addition	93,80%	0,21%	1,24%	4,66%	0,09%
	Mult. and addition	99,21%	0,04%	0,35%	0,38%	0,02%
TMR B	Multiplication	97,43%	0,02%	0,58%	1,96%	0,01%
	Addition	92,01%	1,20%	3,51%	2,69%	0,59%
	Mult. and addition	98,38%	0,18%	0,48%	0,77%	0,19%

AOMs: Application Output Mismatches
AIFs : Architecture Internal Failures

TABLE 7
Results for Quicksort Benchmark

TMR type	Protected instruction	Correct	Hard Fault	Hang	AOMs	AIFs
-	Without protection	92,50%	1,04%	2,74%	2,22%	1,50%
Full TMR	All	99,28%	0,08%	0,37%	0,30%	0,10%
TMR A	Shift	97,16%	0,05%	0,97%	0,94%	0,88%
	Comparison	97,02%	0,07%	0,59%	0,97%	1,35%
	Shift and comparison	97,53%	0,02%	0,79%	0,77%	0,89%
TMR B	Shift	96,77%	0,19%	1,23%	2,01%	1,03%
	Comparison	94,32%	0,28%	2,00%	1,86%	1,54%
	Shift and comparison	97,06%	0,04%	0,76%	0,73%	1,41%

AOMs: Application Output Mismatches
AIFs : Architecture Internal Failures

TABLE 8
Results for Fibonacci Benchmark

TMR type	Protected instruction	Correct	Hard Fault	Hang	AOMs	AIFs
-	Without protection	93,71%	0,57%	2,85%	2,64%	0,23%
Full TMR	All	99,64%	0,01%	0,25%	0,09%	0,03%
TMR A	Addition	97,20%	0,08%	0,82%	1,82%	0,08%
	Comparison	96,76%	0,01%	1,12%	2,01%	0,10%
	Addition and comp.	97,21%	0,07%	0,77%	1,86%	0,09%
TMR B	Addition	95,91%	0,28%	1,71%	2,00%	0,10%
	Comparison	94,73%	0,51%	2,53%	2,06%	0,17%
	Addition and comp.	97,43%	0,02%	0,66%	1,87%	0,02%

AOMs: Application Output Mismatches
AIFs : Architecture Internal Failures

TABLE 9
Results for Dijkstra Benchmark

TMR type	Protected instruction	Correct	Hard Fault	Hang	AOMs	AIFs
-	Without protection	95.10%	0.28%	1.51%	0.31%	2.80%
Full TMR	All	99.48%	0.16%	0.22%	0.04%	0.10%
TMR A	Shift	97.84%	0.01%	0.63%	0.09%	1.43%
	Addition	97.15%	0.01%	0.70%	0.01%	2.13%
	Shift and addition	98.85%	0.22%	0.56%	0.08%	0.29%
TMR B	Shift	96.50%	0.32%	1.46%	0.32%	1.40%
	Addition	96.27%	0.32%	1.53%	0.32%	1.56%
	Shift and addition	98.17%	0.02%	0.68%	0.02%	1.11%

AOMs: Application Output Mismatches
AIFs : Architecture Internal Failures

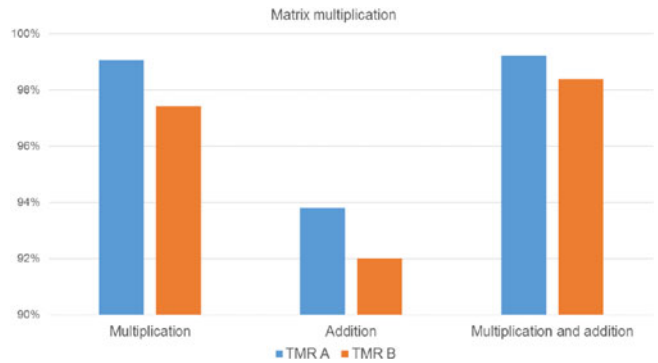

Fig. 6. Protection for the two most used instructions of the matrix multiplication benchmark, achieved by each type of TMR.

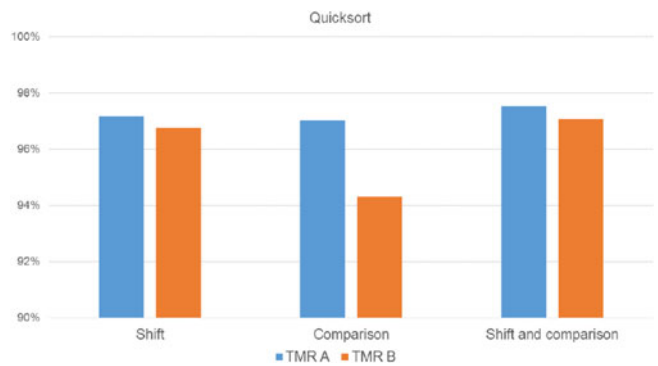

Fig. 7. Protection for the two most used instructions of the Quicksort benchmark, achieved by each type of TMR.

The results for matrix multiplication throw some conclusions. First of all, and as was expected, full TMR is the technique that achieves higher protection. Despite of that, it does not provide 100 percent protection due to errors affecting the routing and the majority voter. Apart from that, TMR A has a closer fault tolerance to full TMR than TMR B, but it must be borne in mind that it requires more resources, as seen in Table 2. Furthermore, the protection of just the most used instruction achieves a significant fault tolerance. This resilience can be increased by protecting the second most used instruction as well, albeit to a lower extent. This can be easily recognized in Fig. 6, where the bars of the multiplication operation are very close to the bars of the multiplication and addition. Apart from that, the most common types of error are the *AOM* as well as *hangs*. It is important to select the protection threshold. If only the second most used instruction (the addition) is protected, the system would achieve about a 93 percent of correct results, but the amount of resources used to protect the module

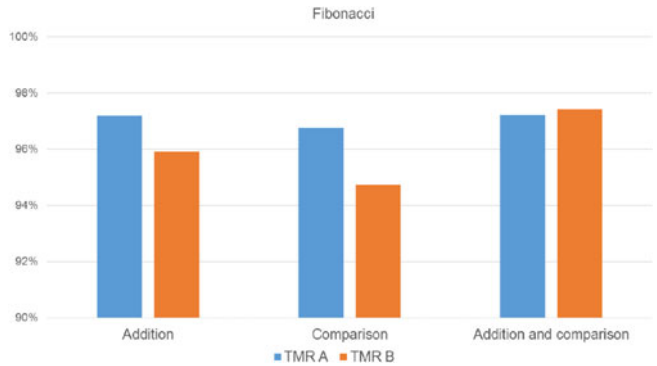

Fig. 8. Protection for the two most used instructions of the Fibonacci benchmark, achieved by each type of TMR.

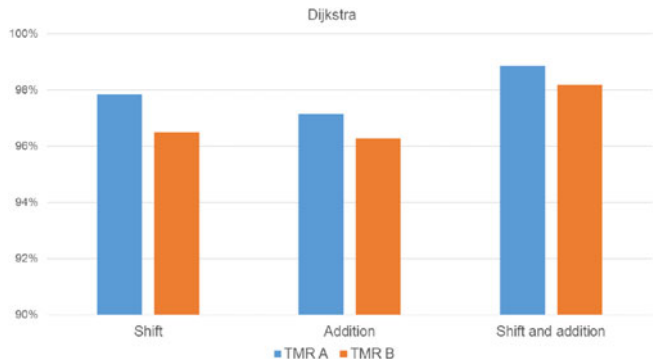

Fig. 9. Protection for the two most used instructions of the Dijkstra's algorithm benchmark, achieved by each type of TMR.

will be significantly less. It is the task of the designer to choose which and how many instructions to protect.

Likewise in the matrix multiplication benchmark, in general terms, Table 7 shows that the most reliable selective TMR implementation is TMR A. This is corroborated in Fig. 7, where it can be seen that the bar corresponding to TMR A is higher than that of TMR B.

The results in Table 8 follow the tendency shown in previous benchmarks. As it can be easily seen in Fig. 8, the most reliable TMR is the type A. However in the last bar, TMR B is slightly better than A.

Finally, the results in Table 9 comply with the expected behavior, where the most effective implementation in terms of fault tolerance is TMR A. Again, the Fig. 9 shows that TMR A usually achieves a higher protection than TMR B. As can be seen in Fig. 13, the loses in terms of resilience incurred by the use of TMR B are still within acceptable limits. Comparing the results with the full TMR, both selective TMR achieves a protection slightly lower than the full version.

To sum up, TMR A is slightly better than TMR B in terms of reliability. For TMR A, the protection of only the most used instruction is good enough, since it is close to the protection offered by the combination of the two most used instructions. Nevertheless, TMR B achieves similar fault tolerance capabilities in the combination of the two most used instructions. Besides this, the most common types of errors are the AOM and hangs. This is coherent with previous works related with the lowRISC characterization [37].

However, it is necessary to compare the fault tolerance achieved with the amount of resources used. This analysis is carried on Figs. 10, 11, 12, and 13. The figures show the resource savings achieved with respect to TMR and the fault tolerance reduction associated to this. In both cases, the metric used was the reduction with respect to the changes introduced in the original design, as

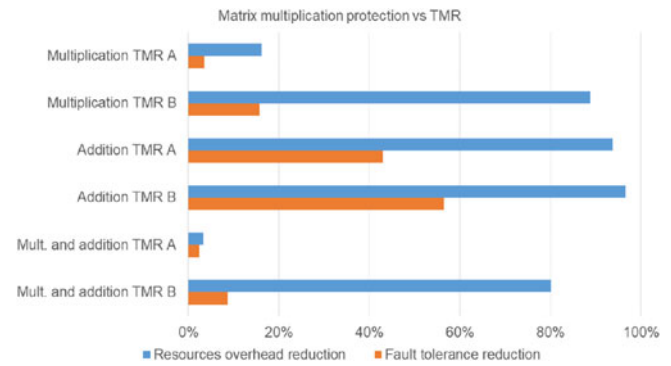

Fig. 10. Resources overhead reduction versus fault tolerance loss for matrix multiplication.

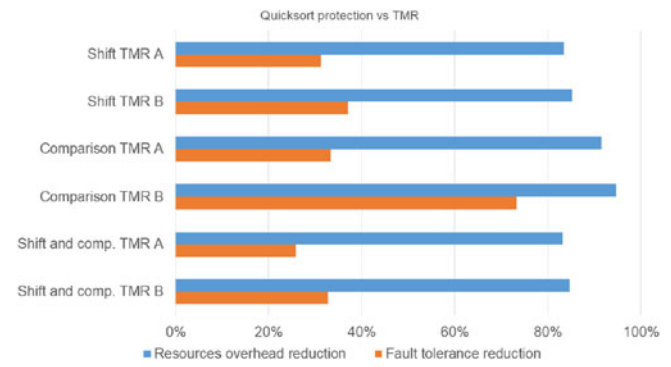

Fig. 11. Resources overhead reduction versus fault tolerance loss for Quicksort.

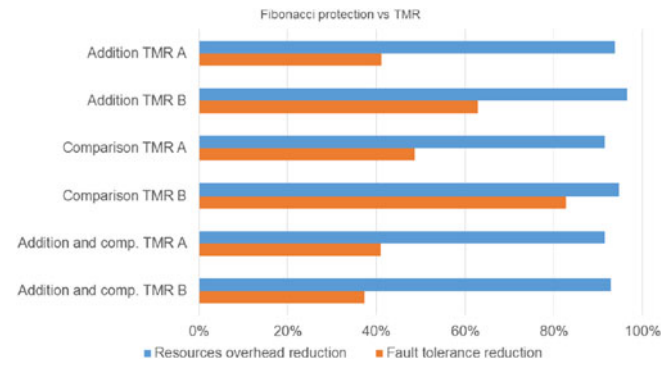

Fig. 12. Resources overhead reduction versus fault tolerance loss for Fibonacci.

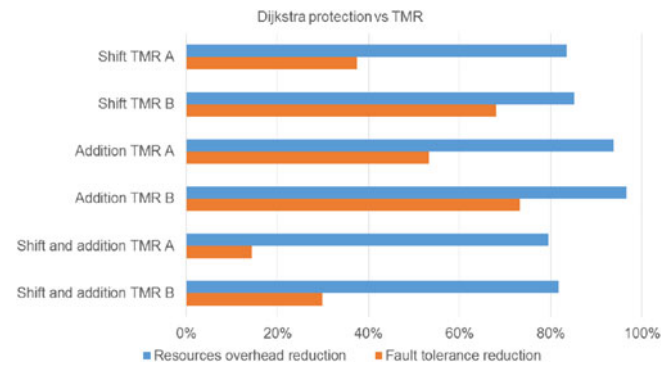

Fig. 13. Resources overhead reduction versus fault tolerance loss for Dijkstra's algorithm.

seen in Equation (1), where δ represents the correspondent parameter for each analysis. By analyzing the data this way, the benefits and drawbacks of using the proposed approach instead of using a complete TMR can easily be seen

$$\Delta Parameter = \frac{\delta_{TMR} - \delta_{Proposed}}{\delta_{TMR} - \delta_{Unprotected}}. \quad (1)$$

In Fig. 10 it is distinguishable that the TMR A achieves better protection (orange bar) than TMR B, but this comes at the cost of higher resources overhead. In all cases, the difference between TMR A and TMR B resources overhead is higher than the difference between TMR A and TMR B fault tolerance. It will depend on the hardware designer to decide if the profit on tolerance compensates the overhead. Nevertheless, both selective TMRs reduce the area overhead compared with the full TMR, with a low penalty on the fault tolerance (in particular in the case of the combination of the two most used instructions).

On the contrary, for the Quicksort algorithm (Fig. 11), the resources overhead difference is minimum, but TMR A fault tolerance is better, specially in the case of the second most used instruction (comparison). Again, comparing them with TMR, the best trade-off between resources overhead reduction and fault tolerance reduction is achieved combining the most used instructions.

The Fibonacci's algorithm benchmark (Fig. 12), throws that although the trend in terms of resource occupation is the same as in the previous cases, TMR B achieves better results for error resilience. The variation with respect to TMR A, however, is negligible. The trade-off between the area and fault tolerance reduction is not as good as the other benchmarks, still it could be interesting depending on the necessities of the design.

Regarding the results obtained for Dijkstra at Fig. 13, it can be observed that, TMR A it the preferred type to be used in terms of fault tolerance. If the results are compared with the full TMR, the ratio between resources overhead and fault tolerant reduction is really interesting, in particular in the combination of the most used instructions using TMR A.

The methodology presented here exploits the trade-off between resource overhead and fault tolerance to design a catalog of selective TMR protections for an ALU based on protecting the hardware devoted to the most used instructions. Thanks to the inherent reconfiguration capabilities of an SRAM-based FPGA, the ALU design can be changed in an affordable time [38], measured in miliseconds (this time could be reduced using an approach like the one presented in [39]). This catalog can be used to alter the protection of this module according to the application to be run in order to reduce its area occupation.

5 Conclusions

In this paper, a methodology to protect the ALU of a soft processor against the effect of SEUs in the configuration memory has been presented. The methodology is based on the construction of a catalog composed of fault tolerant designs of the ALU. Each of these designs is focused on a particular application that is going to be executed in the microprocessor. As a case study, various selective TMR implementations have been created for the ALU, which has been configured depending on the two most used instructions of the benchmark programs. Results show that the protected circuits achieve significant fault tolerance levels while reducing the required resource overhead by tailoring the protection scheme to the application, specially compared with the full TMR. Since a microprocessor can run multiple programs, the creation of a catalog with multiple designs for each application is perfect for a programmable device. The reconfiguration capabilities of SRAM-based FPGAs, as well as the short time required to perform this operation, make them the perfect platform for our methodology. Compared to TMR, the overhead in area is reduced at the cost of slightly decreasing its fault tolerance, which makes it interesting in order to reduce the number of resources and power consumption.

As a future work, we intend to apply this methodology to other modules of a soft processor like the FPU, as it offers a generic solution of reliability improvement. Finally, another line of work could be to determine the optimal number of operations to be protected.

REFERENCES

[1] K. Bedingfield, R. Leach, M. B. Alexander, et al., "Spacecraft system failures and anomalies attributed to the natural space environment," NASA Reference Publication 1390, NASA, MSFC, Alabama, 1996.

[2] H. Quinn, D. Roussel-Dupre, M. Caffrey, P. Graham, M. Wirthlin, K. Morgan, A. Salazar, et al., "The cibola flight experiment," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 1, pp. 3:1–3:22, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2629556

[3] J. D. Cressler and H. A. Mantooth, *Extreme Environment Electronics*. Boca Raton, FL, USA: CRC Press, 2012.

[4] G. Hunyadi, J. Ganley, A. Peffer, and M. Kumashiro, "The university nanosat program: An adaptable, responsive and realistic capability demonstration vehicle," in *Proc. IEEE Aerosp. Conf. Proc.*, Mar. 2004, Art. no. 2858.

[5] A. M. Saleh, J. J. Serrano, and J. H. Patel, "Reliability of scrubbing recovery-techniques for memory systems," *IEEE Trans. Rel.*, vol. 39, no. 1, pp. 114–122, Apr. 1990.

[6] S. B. Akers, "A parity bit signature for exhaustive testing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 7, no. 3, pp. 333–338, Mar. 1988.

[7] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, 1950. [Online]. Available: http://dx.doi.org/10.1002/j.1538-7305.1950.tb00463.x

[8] R. Hentschke, F. Marques, F. Lima, L. Carro, A. Susin, and R. Reis, "Analyzing area and performance penalty of protecting different digital modules with hamming code and triple modular redundancy," in *Proc. 15th Symp. Integr. Circuits Syst. Des.*, 2002, pp. 95–100.

[9] M. Y. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM J. Res. Develop.*, vol. 14, no. 4, pp. 395–401, Jul. 1970.

[10] P. Reviriego, S. Pontarelli, J. A. Maestro, and M. Ottavi, "A method to construct low delay single error correction codes for protecting data bits only," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 32, no. 3, pp. 479–483, Mar. 2013.

[11] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM J. Res. Develop.*, vol. 6, no. 2, pp. 200–209, Apr. 1962.

[12] S. Gupta, N. Gala, G. Madhusudan, and V. Kamakoti, "SHAKTI-F: A fault tolerant microprocessor architecture," in *Proc. IEEE 24th Asian Test Symp.*, 2015, pp. 163–168.

[13] P. Reviriego, C. J. Bleakley, and J. A. Maestro, "Diverse double modular redundancy: A new direction for soft-error detection and correction," *IEEE Des. Test*, vol. 30, no. 2, pp. 87–95, Apr. 2013.

[14] K. Kyriakoulakos and D. Pnevmatikatos, "A novel SRAM-based FPGA architecture for efficient TMR fault tolerance support," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Aug. 2009, pp. 193–198.

[15] S. Nakagawa, S. Fukumoto, and N. Ishii, "Optimal checkpointing intervals of three error detection schemes by a double modular redundancy," *Math. Comput. Modelling*, vol. 38, no. 11, pp. 1357–1363, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0895717703901385

[16] G. Rui, C. Wei, L. Fang, D. Kui, and W. Zhiying, "Modified triple modular redundancy structure based on asynchronous circuit technique," in *Proc. 21st IEEE Int. Symp. Defect Fault Tolerance VLSI Syst.*, Oct. 2006, pp. 184–196.

[17] B. Shim, S. R. Sridhara, and N. R. Shanbhag, "Reliable low-power digital signal processing via reduced precision redundancy," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, no. 5, pp. 497–510, May 2004.

[18] I. D. Elliott and I. L. Sayers, "Implementation of 32-bit RISC processor incorporating hardware concurrent error detection and correction," *IEE Proc. E-Comput. Digit. Techn.*, vol. 137, no. 1, pp. 88–102, Jan. 1990.

[19] S. R. Hasan and P. Tangellapalli, "Area efficient soft error tolerant RISC pipeline: Leveraging data encoding and inherent ALU redundancy," in *Proc. IEEE 60th Int. Midwest Symp. Circuits Syst.*, Aug. 2017, pp. 699–702.

[20] I. Wali, B. Deveautour, A. Virazel, A. Bosio, P. Girard, and M. S. Reorda, "A low-cost reliability versus cost trade-off methodology to selectively harden logic circuits," *J. Electron. Testing*, vol. 33, no. 1, pp. 25–36, Feb. 2017. [Online]. Available: https://doi.org/10.1007/s10836-017-5640-6

[21] T. Michel, R. Leveugle, and G. Saucier, "A new approach to control flow checking without program modification," in *Proc. 21st Int. Symp. Fault-Tolerant Comput. Dig. Papers*, 1991, pp. 334–341. [Online]. Available: doi.ieeecomputersociety.org/10.1109/FTCS.1991.146682

[22] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proc.. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2003, pp. 29–40.

[23] M. Bellato, P. Bernardi, D. Bortolato, A. Candelori, M. Ceschia, A. Paccagnella, M. Rebaudengo, M. S. Reorda, M. Violante, and P. Zambolin, "Evaluating the effects of SEUs affecting the configuration memory of an SRAM-based FPGA," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, Feb. 2004, pp. 584–589.

[24] M. Ceschia, M. Violante, M. S. Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori, "Identification and classification of single-event upsets in the configuration memory of SRAM-based FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 50, no. 6, pp. 2088–2094, Dec. 2003.

[25] A. Waterman and K. Asanovi, "The RISC-V instruction set manual volume I: User-level ISA, document version 2.2," RISC-V Foundation, 2017. [Online]. Available: https://riscv.org/specifications/

[26] LowRISC. "LowRISC project," Jun. 2017. [Online]. Available: http://www.lowrisc.org/docs/

[27] Y. Ichinomiya, S. Tanoue, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi, "Improving the robustness of a softcore processor against SEUs by using TMR and partial reconfiguration," in *Proc. 18th IEEE Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, May 2010, pp. 47–54.

[28] Digilent Inc., "Nexys4 DDR FPGA Board Reference Manual rev.C," Digilent, Inc., Datasheet, Apr. 2016. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual

[29] Xilinx, Inc., "Ds181 - artix-7 FPGAs data sheet: DC and AC switching characteristics v1.23," Xilinx, Inc., Datasheet, Dec. 2017. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds181_Artix_7_Data_Sheet.pdf

[30] Xilinx, Inc., "UG910 - Vivado design suite user guide: Getting started v2017.3," Xilinx, Inc., User Manual, Oct. 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug910-vivado-getting-started.pdf

[31] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, and K. Asanovi, "Chisel: Constructing hardware in a Scala embedded language," in *Proc. 49th Annu. Des. Autom. Conf.*, 2012, pp. 1216–1225. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228584

[32] K. Asanovi, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Dept., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016–17, Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[33] F. Lima, C. Carmichael, J. Fabula, R. Padovani, and R. Reis, "A fault injection analysis of Virtex FPGA TMR design methodology," in *Proc. 6th Eur. Conf. Radiation Effects Compon. Syst.*, Sep. 2001, pp. 275–282.

[34] Xilinx, Inc., "PG036 - soft error mitigation controller v4.1 product guide," Xilinx, Inc., User Manual, Apr. 2017.

[35] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proc. 50th Annu. Des. Autom. Conf.*, 2013, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488859

[36] A. Lindoso, L. Entrena, E. S. Millan, S. Cuenca-Asensi, A. Martinez-Alvarez, and F. Restrepo-Calle, "A co-design approach for set mitigation in embedded systems," *IEEE Trans. Nucl. Sci.*, vol. 59, no. 4, pp. 1034–1039, Aug. 2012.

[37] A. Ramos, J. A. Maestro, and P. Reviriego,"Characterizing a RISC-V SRAM-based FPGA implementation against single event upsets using fault injection," *Microelectron. Rel.*, vol. 78, no. Supplement C, pp. 205–211, 2017.

[38] A. Ullah and L. Sterpone, "Recovery time and fault tolerance improvement for circuits mapped on SRAM-based FPGAs," *J. Electron. Testing*, vol. 30, no. 4, pp. 425–442, Aug. 2014. [Online]. Available: https://doi.org/10.1007/s10836-014-5463-7

[39] L. A. Cardona, A. Ullah, L. Sterpone, and C. Ferrer, "A novel tool-flow for zero-overhead cross-domain error resilient partially reconfigurable X-TMR for SRAM-based FPGAs," *J. Syst. Archit.*, vol. 81, pp. 112–120, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762117300772