Processor Core Profiling for SEU Effect Analysis

Rodrigo Travessini*, Paulo R. C. Villa*, Fabian L. Vargas‡, Eduardo Augusto Bezerra*†
‡Electrical Engineering Dept., Catholic University - PUCRS, Porto Alegre, Brazil
*Department of Electrical Engineering, Federal University of Santa Catarina (UFSC), Florianopolis, Brazil
†TEST Research Group, LIRMM, Montpellier, France
E-mails: {rodrigo.travessini, paulo.villa, eduardo.bezerra}@eel.ufsc.br, vargas@computer.org

***Abstract*—This paper presents the analysis of a fault injection campaign in the CPU registers of the LEON3 softcore processor. The faults are injected through the use of simulation scripts that force a bit flip while the processor is running a set of three different workloads. The study is restricted to single bit upsets (SBU) and investigates the effects of the injected faults and how they propagate to the CPU core boundaries. The obtained results show that the majority of the failures are due to faults injected in only a small number of the processor registers. Furthermore, in this study, it is proposed a partial triple modular redundancy approach to protect only the CPU's most sensitive registers, achieving a 99.25% SBU tolerance with only a marginal increase in area.**

***Index Terms*—Fault Injection, FPGA Reliability, SEU, Soft Error.**

I. Introduction

Embedded processors are used in a wide number of applications, ranging from aerospace and avionics industry to household automation systems. Many of those applications are safety-critical and thus have very rigid dependability requirements. One of the major concerns when it comes to dependability are the soft errors, which have the potential to induce the highest failure rate of all other reliability mechanisms combined [1].

With the modern processors, the rate of radiation-induced soft errors is increasing dramatically [2]. The continuous technology downscaling and the lower supply voltages, leads to an increase in the susceptibility of memory and logic to radiation coming from atmospheric neutrons or by on-chip radioactive impurities.

The soft error, or single event upset (SEU), occurs when a radiation event causes the data state of a memory element (e.g., register, latch, flip-flop) to be reversed [1]. In soft errors there is no hardware damage, the affected memory element can be overwritten with new data. In the most common scenario the radiation event affects only a single bit, called single bit upset (SBU). Higher energy radiation events may cause multiple bits to be affected, leading to a multiple bit upset (MBU). Furthermore, radiation events that travel through combinational logic are called single event transient (SET), and if they propagate to a memory element, they also lead to a SEU.

There are many works in the literature that propose techniques for SEU mitigation in the memory hierarchy (e.g., register file, caches and main memory). Those techniques are usually based on using error correcting codes (ECC) [3] [4]. When dealing with SEU in the control logic, the most common approach is the use of the triple modular redundancy (TMR) [5]. This approach provides single error masking and double error detection, but when applied to the entire design, TMR implies in very high area overhead. These costs can be mitigated with partial TMR approaches [6] [7], which selectively apply the redundancy in the most vulnerable areas of the design.

Embedded processors provide a good opportunity to explore the use of partial redundancy since they usually are very complex designs that implement hundreds of instructions, with many of them never being executed, or executed very sparsely in common workloads. In order to take advantage of this, deeper knowledge of how SEU-induced faults manifest in the target processor, and how they propagate to its boundaries is fundamental. ARM Research did a very comprehensive study [2] in this regard, conducting an intensive fault injection campaign in the ARM Cortex-R5 CPU core. They concluded that only 10% of the sequential elements in the Cortex-R5, accounts for more than 70% of the errors, suggesting that an important reliability improvement can be obtained protecting just these most sensitive components.

In this paper, the investigated processor is the LEON3 from Cobham Gaisler [8], which was developed targeting critical space applications, supported by the European Space Agency (ESA). Previous works had conducted fault injection experiments in the LEON family. In [9] the authors presented the preliminarily results of a fault injection campaign directed to the LEON3 integer pipeline. The study divided the pipeline registers based on their functionality into five categories, and analyzed each of them regarding their sensitivity to both SBU and MBU. In [10] the authors compare the default LEON2 with a fault tolerant version. This work was also focused in the pipeline unit of the processor, with the results highlighting the different vulnerabilities to faults between the pipeline stages. In [11], a study evaluated the susceptibility of the LEON3 with different fault models, including SBU, MBU, and transient faults. The analysis covered the processor pipeline registers, register file and caches. Although suggested, partial redundancy to protect the most vulnerable registers has never been actually implemented and properly evaluated in any of the references considered in this paper.

The purpose of this paper is to present an extensive analysis of the results of a fault injection campaign targeting the LEON3 processor core, which comprises both the pipeline

978-1-5386-1472-3/18/$31.00

execution unit and the cache controllers. The study investigates the effects of the injected faults, and how they manifest in the processor interfaces with other modules such as the caches, main memory and register file. Based on the information of the LEON3 most vulnerable registers, a partial TMR technique is evaluated regarding its fault tolerance and the area/performance overhead.

The remaining of the paper is organized as follows. Section II describes an overview of the LEON3 Processor. Then, Section III describes the fault injection methodology used for the experiments. Section IV discusses the obtained results, including the evaluation of the partial TMR implementation. Finally, Section V concludes the paper.

II. LEON3 Processor

The LEON3 is a 32-bit processor compliant with the SPARC V8 instruction set architecture. It is designed for embedded applications, combining high performance with low complexity and low power consumption. The processor is described using the VHDL description language, and is available under the GNU GPL license.

The LEON3 core features a seven stage pipeline with separate instruction and data caches. Furthermore, the presence of an AMBA-2.0 bus interface, allows the processor to be integrated with many other IPs from the Gaisler Research IP Library (GRLIB) [12]. Being highly configurable, the LEON3 is particularly suitable for system-on-a-chip (SOC) designs.

In addition to the default LEON3 implementation, Cobham Gaisler also offers a fault tolerant version (LEON3FT), which provides protection against SEU errors. The LEON3FT fault tolerance is focused on the protection of the on-chip RAM blocks, which are used to implement the register file and the cache memories, and therefore does not comprehend the processor control logic.

In Fig. 1 the LEON3 processor core (PROC3) is presented, which is the target of this study. The PROC3 is composed of two submodules. The larger one, the integer unit (IU3) implements the entire processor seven stage pipeline. The other one is the cache controller, which can be further divided in the instruction cache controller (ICACHE), the data cache

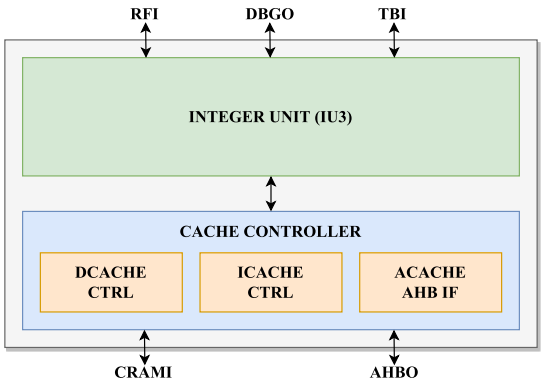


Fig. 1. LEON3 Processor Core (PROC3).

TABLE I
Number of Registers in each PROC3 Submodule

Submodule	Registers	Bits
IU3	232	1193
ICACHE	34	264
DCACHE	83	525
ACACHE	13	17

controller (DCACHE), and the interface between the cache controllers and the AMBA AHB bus (ACACHE). The entire PROC3 has a total of 362 registers, all of them investigated in the fault injection experiments conducted in this work. Table I provides a breakdown of the register distribution in each of the PROC3 submodules.

Also presented in Fig. 1 are the PROC3 main output interfaces (i.e., ports), which are observed during the fault injection campaign to identify the fault propagation paths and obtain relevant statistics such as fault manifestation times. The interfaces connected to the IU3 are the integer register file (RFI), the processor debug support unit (DBGO), and the instruction trace buffer (TBI). Moreover, the ones connected to the cache controller, are the cache memory array (CRAMI), and the AMBA AHB bus (AHBO).

Concerning the implementation of the partial TMR technique proposed in this work, the most vulnerable registers found in the fault injection campaign were triplicated and connected to a majority voter. Since no combinational circuit was also replicated, the suggested approach does not offer protection against transient faults. As the LEON3 operating frequency in FPGAs is under the 125MHz, the likelihood of an SET is considered negligible in current technologies [13].

III. Fault Injection Methodology

This section first describes the fault injection strategy that was adopted. Then, it presents the chosen fault model for the experiments, followed by the fault effect classification and the workload description. Finally, the technique to perform the fault propagation investigation is detailed.

A. Fault Injection Environment

There are several different fault injection strategies proposed in the literature. They can be classified in five main categories: hardware-based fault injection, software-based fault injection, simulation-based fault injection, emulation-based fault injection and hybrid fault injection. In [14] it is presented an extensive survey that compare the different techniques, and summarizes their advantages and limitations.

In order to fully achieve the objectives of this work, the fault injection technique chosen had to meet a set of characteristics such as: full access to the entire processor design without being intrusive (i.e., not requiring the processor to be stalled, or any other interference in the execution flow), a good time resolution and high observability. The method that was found to best fit those requirements is the simulation-based fault injection. The main disadvantage of this technique is that it is time consuming, as simulation time is substantially longer than

real time execution. This limitation combined with the high number of experiments required to obtain great confidence in the results, imposed an upper bound in the size of the workload running in the processor during the experiments.

The simulation-based strategy adopted in this work relied on the use of built-in simulator commands within TCL [15] scripts. This setup enabled interaction with the simulation engine, allowing the manipulation of signals (fault injection), and observation of the fault effects. The HDL simulator used for the experiments was the Modelsim [16] from Mentor Graphics.

The following steps summarize the tasks executed in each fault injection experiment:

1) Choose a random flip-flop where the fault will be injected. The selection is done from a list containing all the registers that are being analyzed in the campaign. It is important to note, that since the injection target is selected randomly at the bit level, by the end of the fault injection campaign, the number of injected faults by register will be proportional to the register size.
2) Choose a random instant when the fault will be injected. In order to avoid injecting faults in the processor warm up phase, the time interval considered for fault injection comprehend the final 80% of the simulation runtime.
3) Simulate until the chosen injection instant.
4) Inject the fault by forcing a bit flip in the target flip-flop. Note that the value is not stuck, and can be normally overwritten during the remaining of the simulation.
5) Simulate until the workload finishes execution or a predefined timeout is reached.
6) Compare the final registers contents to the ones obtained from a golden run (executed previously).
7) Finally, store all the collected statistics in a CSV file, for later analysis.

B. Fault Model

The adopted fault model for the fault injection campaign is the SBU. In each experiment a single fault is injected by inverting the logic value of the target signal. Multiple faults due to a single radiation event (MBU) are not addressed in this paper. In order to accurately model this effect, it is necessary to have information regarding the final design layout and the register neighborhood, which is not available during the HDL simulation.

C. Fault Effects Classification

After the end of each fault injection experiment, the data obtained during the simulation is used to classify the fault effects in five categories. The classification, which is based on the one used in [10] and [17], is the following:

- *No Effect* - The program finishes execution normally, with correct results, and the contents of the processor core registers match with the golden run.
- *Latent* - The program finishes execution normally, with correct results, but the contents of the processor core registers do not match with the golden run.
- *Wrong Result* - The program finishes execution, but with incorrect results.
- *Timed Out* - The program took an abnormal amount of time without finishing execution and the simulation was interrupted. Many conditions may lead to this scenario, such as an incorrect branching due to the injected fault.
- *Exception* - The processor detected an unexpected event, generating a trap and aborting execution.

Note that only the last three categories (i.e., *Wrong Result*, *Timed Out*, and *Exception*) correspond to effects in which the processor exhibited erroneous behavior. For the sake of clarity, the term *harmful effects* will therefore be used in the remaining of the text whenever referring to any of these effects.

D. Workload

Three different workloads were used in the fault injection campaign. These include a proportional integral derivative controller (*PID*), a bubble sort implementation (*BSORT*) and a hamming encoder (*HAMMING*). The number of iterations executed in each workload was adapted so that all three had almost the same execution time, around 35000 clock cycles. Due to space constraints, the workloads are not analyzed individually in this paper, but rather overall performance.

E. Fault Propagation Evaluation Method

In order to properly investigate the fault propagation to the PROC3 interfaces, some adjustments in the fault injection setup were necessary. The main change, was the inclusion of a redundant copy of the PROC3 module in the LEON3 VHDL description. During the fault injection campaign, the faults were injected in only one of the copies, and the other was used for reference. Additionally, the interfaces of both copies were constantly monitored by a TCL script, and the identified mismatches were recorded for posterior analysis. It is relevant to note, that this modifications did not introduce any interference in the processor operation. The redundant PROC3 only received the same inputs as the original PROC3, but its outputs were not connected to the rest of the system.

IV. Experimental Results

This section presents and discusses the results of the fault injection campaign.

A. Overall PROC3 Performance

In Fig. 2 is presented a plot containing the overall results of the fault injection experiments targeting the LEON3 processor core. It is interesting to see, that even without any mechanism

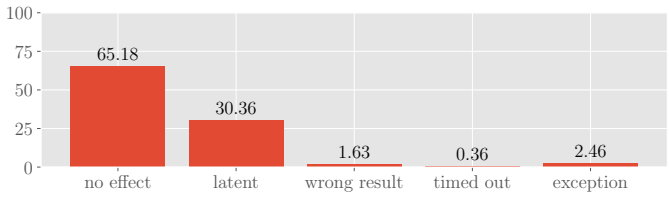


Fig. 2. Overall PROC3 Performance

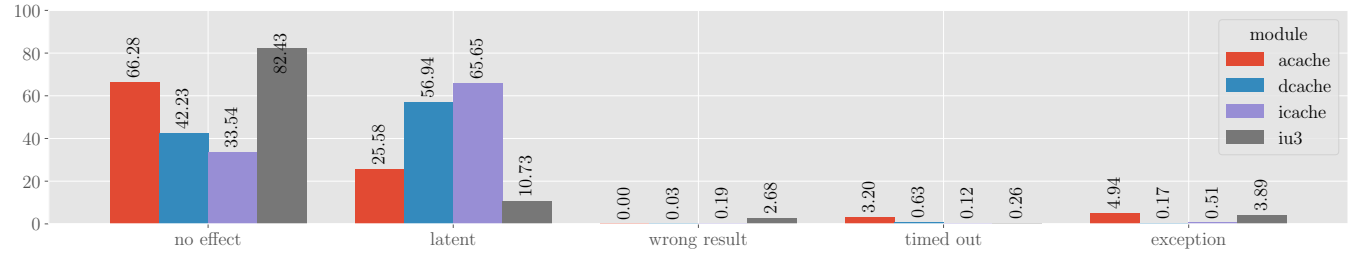


Fig. 3. Overall Performance by Module

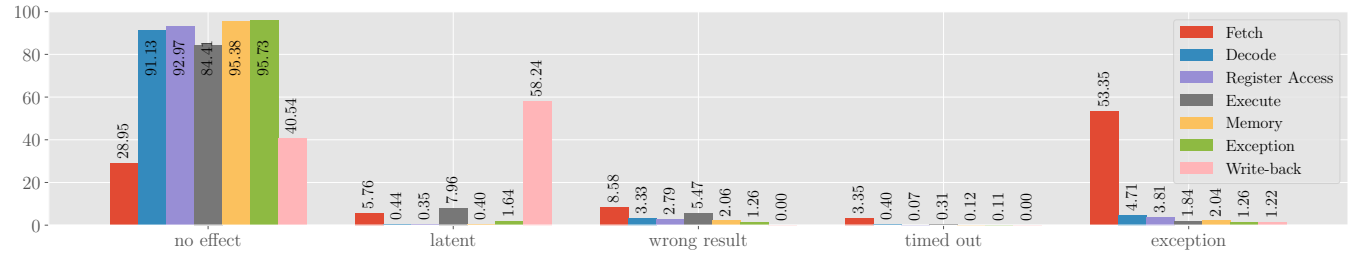


Fig. 4. IU3 Performance Split by Pipeline Stage

for SEU mitigation, most of the injected faults were overwritten without compromising the program execution. This can be explained by the fact that a large part of the processor components are not used in every instruction, and the injected faults in those components are masked when new instructions are fetched into the pipeline. Another relevant result, is that the processor trap/exception mechanism was able to detect only slightly more than half of the faults that led to harmful effects.

With respect to the individual performance of each module that composes the PROC3, the results can be seen in Fig. 3. It is possible to observe that the cache controllers present a much higher ratio of latent faults than the integer pipeline. This is due to some of the registers being used on only some specific cache configurations (e.g., when using the LRU replacement policy), which causes the injected faults in those registers to never be overwritten. Furthermore, the higher exception value observed in the *acache* module is expected, since this module has a small number of registers, and one of them corresponds to critical register used for error warning.

B. Integer Pipeline Performance

Being the integer pipeline the largest module in the LEON3 processor core, it is of great interest to analyze it closely. For this purpose, the IU3 registers were split by pipeline stage, and the SEU vulnerability obtained for each stage was investigated separately. The same analysis was done in [10] for the LEON2 processor, which has a slightly different pipeline structure with only five stages, and the conclusions were very similar. The investigation results are presented in the Fig. 4.

According to the results, the *Fetch* stage presents a much higher rate of harmful effects than the other pipeline stages. This behavior is expected, since the *Fetch* stage contains only two registers, with one of them being the *Program Counter* (PC). Faults injected in the PC may cause the wrong instruction being fetched, or even an attempt to read an invalid memory location. Another remark is that the *Write-Back* stage contains a high rate of latent faults. This is due to this stage being composed mostly of special registers that were barely accessed during the execution of the workloads.

C. Individual Register Performance

Fig. 5 contains the individual performance of the thirty registers which presented the highest number of harmful effects. The registers are sorted with the most vulnerable placed at the top. In this analysis *latent* and *no effect* faults are grouped together, however it is important to remark that should a longer duration have been considered for the workload, some of the *latent* faults could become harmful. Note that

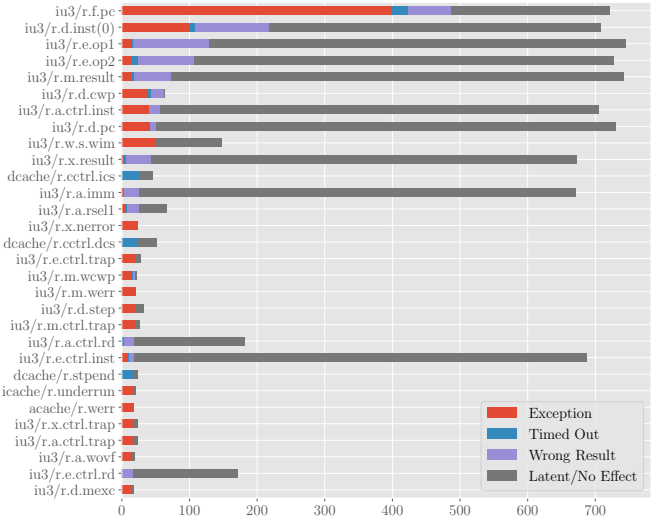
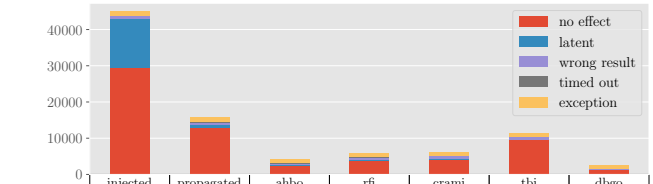


Fig. 5. Individual Register Performance



	injected	propagated	ahbo	rfi	crami	tbi	dbgo
no effect	29331	13029	2448	3861	3944	9520	1170
latent	13663	602	133	158	535	156	133
wrong result	733	733	422	698	592	638	235
timed out	164	164	136	157	156	150	115
exception	1109	1109	949	1058	994	1003	872

Fig. 6. Fault Propagation in PROC3 interfaces

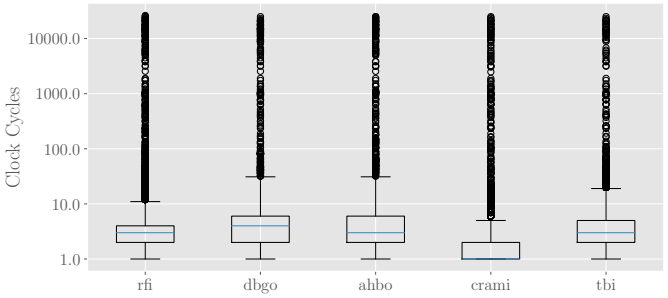


Fig. 7. Boxplot of the Fault Manifestation Time in PROC3 interfaces. The bottom and top of the box corresponds to the first and third quartiles, the line inside the box is the median, the whiskers are at 1.5 IQR, values outside of that range are represented by dots.

as mentioned before, the number of injected faults by register is proportional to its size. Also, since the analysis is based on absolute quantities, even though some one bit registers presented an 100% exception rate, they are considered less vulnerable than larger registers with smaller exception rates, but higher absolute values. As can be seen in the results, the program counter (*r.f.pc*) presented the worst performance by a big margin, followed by other important registers such as the fetched instruction (*r.d.inst*), both the ALU operands (*r.e.op1* and *r.e.op2*) and the ALU operation result (*r.m.result*). Note that the number of harmful effects per register decreases in a fast pace. This characteristic can be exploited to obtain an improved fault tolerance with low overhead. Moreover, it is interesting to observe how the fault effects relate to the register functionality. Faults injected in registers used by the ALU frequently lead to wrong results, while the most common effect of faults injected in error and trap registers is the generation of exceptions.

D. Fault Propagation

Fig. 6 presents the results obtained regarding fault propagation to the PROC3 boundaries. The *injected* bar contains the fault effect distribution of all injected faults, the *propagated* bar only contains the ones that propagated to one or more interfaces, and the rest of the bars correspond to the propagation on each interface individually. As shown in the results, only one third of the injected faults have led to a disturbance in the interfaces. Most *no effect* and *latent* faults remained inside the PROC3. Also, as expected, all the faults that generated harmful effects propagated to at least one interface.

Between the interfaces, the *tbi* had the highest propagation rate, mostly due to the high number of *no effect* faults. This behavior is consistent since the trace buffer track statistics of all the executed instructions (e.g., address, opcode and result). As to faults that led to harmful effects, the register file interface is where they have manifested the most. It is important to note, however, that the results indicate that harmful effects cannot be detected by looking to only a single interface, for a complete coverage, a fault detection strategy would need to look more than one location.

With respect to the fault manifestation times, Fig. 7 contains a box plot with the obtained values in each PROC3 interface. It follows that for all interfaces, most propagated faults take less than ten clock cycles to manifest. Specifically in the cache memory array interface, more than half of the propagated faults manifested in the clock cycle following the injection.

E. Partial TMR Evaluation

As shown in Fig. 5, the concentration of harmful effects in a small number of registers presents a clear indication of the possibility of obtaining a significant improvement in the overall reliability by using a selective protection strategy. The same conclusion can be obtained by looking at Fig. 8, where is derived the expected SBU tolerance when the most sensitive registers are protected (e.g., through spatial redundancy). In order to validate these results, the thirty registers identified as most vulnerable (from Fig. 5) were protected using the partial TMR technique previously introduced, and the fault injection experiments were redone. Fig. 9 contains the updated results. As expected, the number of *latent* and *no effect* faults improved to around 99.25%, which is a 3.71% increase compared to the original version. The improved fault tolerance means having only one incorrect computation every 133 faults, whereas in the original architecture there was one every 22. In order to obtain the performance and area penalty, both versions of the LEON3 were synthesized for the GR-XC3S board. The maximum achieved frequency dropped from 58.973MHz to 51.781MHz. This decrease is mainly due to the inclusion of the majority voter logic in the critical path. Moreover, the FPGA resources utilization can be seen in table II.

V. Conclusion

This paper has presented a detailed analysis of the SEU effects in the LEON3 processor core. The analysis was based on the data obtained from an extensive fault injection

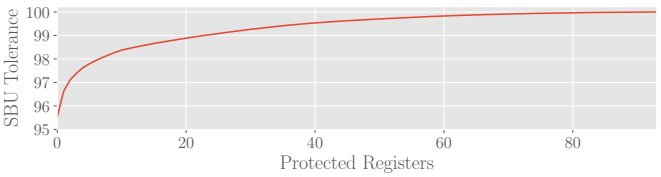


Fig. 8. Expected SBU Tolerance

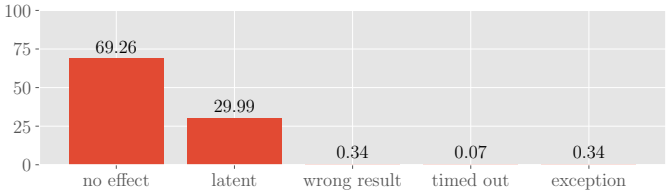


Fig. 9. Protected LEON3 Overall Performance

TABLE II
FPGA RESOURCES UTILIZATION

	Unprotected	Protected	Overhead
Slice Flip Flops	2919 (10%)	3568 (13%)	22.23%
4 input LUTs	10425 (39%)	11060 (41%)	6.09%
Slices	5660 (42%)	6040 (45%)	6.71%

campaign conducted through the use of a simulation-based injection technique. The investigation revealed that most of the faults that led the processor to erroneous behavior were limited to a small group of registers, mainly the program counter and the ALU operands. Moreover, was found that only a third of the injected faults actually propagated to the CPU core interfaces. These results presented a solid indication that a reliability improvement was possible by protecting only the most vulnerable parts of the processor. Based on these knowledge, an efficient fault tolerant strategy was proposed, providing a significant gain in the overall reliability without incurring a noticeable performance penalty.

ACKNOWLEDGMENT

This work has been partly funded by the Brazilian National Council for Scientific and Technological Development (CNPq).

REFERENCES

[1] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and materials reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[2] X. Iturbe, B. Venu, and E. Ozer, "Soft error vulnerability assessment of the real-time safety-related arm cortex-r5 cpu," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 91–96.

[3] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[4] C.-L. Chen and M. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124–134, 1984.

[5] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[6] P. K. Samudrala, J. Ramos, and S. Katkoori, "Selective triple modular redundancy (stmr) based single-event upset (seu) tolerant synthesis for fpgas," *IEEE transactions on Nuclear Science*, vol. 51, no. 5, pp. 2957–2969, 2004.

[7] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin, "Improving fpga design robustness with partial tmr," in *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International*. IEEE, 2006, pp. 226–232.

[8] A. Cobham Gaisler, "Leon3 processor." [Online]. Available: http://www.gaisler.com/index.php/products/processors/leon3

[9] T. Bonnoit, A. Coelho, N.-E. Zergainoh, and R. Velazco, "Seu impact in processor's control-unit: Preliminary results obtained for leon3 soft-core," in *Test Symposium (LATS), 2017 18th IEEE Latin American*. IEEE, 2017, pp. 1–4.

[10] E. Touloupis, J. A. Flint, V. A. Chouliaras, D. D. Ward *et al.*, "Study of the effects of seu-induced faults on a pipeline protected microprocessor," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1585–1596, 2007.

[11] H. Abbasitabar, H. R. Zarandi, and R. Salamat, "Susceptibility analysis of leon3 embedded processor against multiple event transients and upsets," in *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*. IEEE, 2012, pp. 548–553.

[12] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc, "Grlib ip core users manual," *Gaisler research*, 2017.

[13] "White paper: Introduction to single-event upsets," Altera, Tech. Rep. WP-01206-1.0, 2013.

[14] H. Ziade, R. A. Ayoubi, R. Velazco *et al.*, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[15] J. K. Ousterhout *et al.*, *Tcl: An embeddable command language*. University of California, Berkeley, Computer Science Division, 1989.

[16] M. Graphics, *ModelSim*. [Online]. Available: https://www.mentor.com/products/fv/modelsim/

[17] M. Rebaudengo, M. S. Reorda, and M. Violante, "Accurate analysis of single event upsets in a pipelined microprocessor," *Journal of Electronic Testing*, vol. 19, no. 5, pp. 577–584, 2003.