Journal of Electronic Testing (2019) 35:9–27
https://doi.org/10.1007/s10836-019-05778-z

Fault Tolerant Soft-Core Processor Architecture Based on Temporal Redundancy

Paulo R. C. Villa[1] · Rodrigo Travessini[2] · Roger C. Goerl[3] · Fabian L. Vargas[3] · Eduardo A. Bezerra[4]

Received: 29 June 2018 / Accepted: 22 January 2019 / Published online: 4 February 2019

Abstract

Embedded soft-core processors are becoming the usual solution to deal with network and data communications inside FPGAs. However, when developing space-based applications, the designer must consider the effects of ionizing radiation such as *Total Ionizing Dose* (TID) and *Single-Event Effect* (SEE). The majority of techniques for mitigation of *Single-Event Upsets* (SEUs) on FPGAs are based on hardware spatial-redundancy. This work presents a fault-tolerance technique, based on the concept of temporal redundancy, with checkpoints and recovery for soft-core processors. The proposed modified architecture is aimed at embedded systems for space applications based on FPGAs. Our experimental results show that the Checkpoint Recovery technique is a valid alternative to traditional spatial-redundancy, especially when considering limited logic area and power budget present on a satellite. The results present levels of reliability comparable to those of the more conventional fault-tolerance techniques. Additionally, the proposed approach does not require modifications of the software source code or compiler.

Keywords Fault-tolerance · Checkpoint recovery · Soft-core processors · FPGAs · Single-event upsets

1 Introduction

Field Programmable Gate Arrays (FPGAs) are not any longer used exclusively for prototyping of *Specific Integrated Circuits* (ASICs) [4, 41]. In fact, they are so versatile that, for one of the most conservative applications — satellites — they have been increasingly taking over the data processing and avionics control [13]. There is a growing trend in the employment of FPGA on space applications [19].

Today's FPGA offer high logic capacity (to implement a circuit), reasonable operating frequencies and a plethora of embedded hard-blocks (such as *Analog-to-Digital Converters* (ADCs) and *Digital Signal Processors* (DSPs)) [17]. Several factors have contributed to reach this stage, mainly the *Integrated Circuit* (IC) transistor density due to manufacturing process scaling down [47]. Nonetheless, ASIC have their place on the market, especially on high-volume and high-performance applications such as SAMPA Chip [8].

When considering space applications, future satellite missions are expected to acquire and process large amounts of data [37]. Additionally, on-board electronics are required to be re-programmable after the mission launch and even further, while still operating. Traditional microprocessors and ASIC cannot fulfill this requirement entirely, leaving FPGA as the primary option.

Apart from custom *Intellectual Property* (IP) blocks inside the FPGA, it is common to have embedded processors [3, 20, 24, 28, 55] to handle data and communications.

Responsible Editor: L. M. Bolzani Pöhls

✉ Paulo R. C. Villa
paulo.villa@veranopolis.ifrs.edu.br

Rodrigo Travessini
rodrigo.travessini@eel.ufsc.br

Roger C. Goerl
roger.goerl@acad.pucrs.br

Fabian L. Vargas
vargas@computer.org

Eduardo A. Bezerra
eduardo.bezerra@ufsc.br

[1] Federal Institute of Rio Grande do Sul, Veranópolis, Brazil

[2] Electrical Engineering Department, Federal University of Santa Catarina, Florianópolis, Brazil

[3] Electrical Engineering Department, Catholic University - PUCRS, Porto Alegre, Brazil

[4] Electrical Engineering Department, UFSC, Brazil and LIRMM, Université de Montpellier, Montpellier, France

All this integration can compromise overall system reliability [12, 42]. Given these circumstances, finding a compromise between the processing capacity and the level of reliability against processor failures is important from the research point of view.

Given the harsh environment satellites are exposed to, external events can cause the system to malfunction. *Electromagnetic Interference* (EMI) and radiation account for effects that the circuits are susceptible. One of the most common problems is known as SEE [11, 46], which can cause temporary or permanent failures in a system, even with the potential to cause invalidation of the entire system, in the form of the premature termination of a space satellite mission, for example.

To attain mission-level reliability, fault-tolerance must be considered throughout the entire design of the system, i.e., from IC layout to software implementation. On the lower level of abstraction, radiation hardened (rad-hard) FPGAs can deal with the effects of radiation on the circuit, assuring minimal conditions to the system to function. But, for some space programs, such as the case in Brazil, the acquisition process of radiation hardened (rad-hard) components is controlled by government agencies as, for instance, the *International Traffic in Arms Regulations* (ITAR) [49] rules[1]. In addition, rad-hard components are significantly more expensive than traditional *Commercial Off-The-Shelf* (COTS) components.

If we assume an unhardened COTS FPGA, the next level of abstraction of the system must mitigate possible errors (i.e., SEEs) from the underlying hardware. Hence, a strong motivation for developing this work, is the possibility to introduce fault tolerance to a system with the use of COTS FPGAs.

On that matter, the LEON3 [2] processor has already been used in some space missions. Considering the Brazil's *National Institute For Space Research* (INPE) interest in migrating from the ERC32 legacy processor[2] without having to redesign the entire code, a soft-LEON3 processor with fault-tolerance is a good substitution for the rad-hard ERC32.

Considering the aforementioned, this work expands a previous work [53], presenting in detail the modified LEON3 processor architecture with fault-tolerance, targeting the used of soft-core processors in space applications.

1.1 Objectives and Contribution

Network and data communications inside FPGAs are often handled with the use of soft-core processors [20, 24, 28, 55]. High-parallel tasks implemented in IP-blocks can be easily integrated with processors during the FPGA development flow. However, when developing space-based applications, the designer of embedded systems must also consider the effects of ionizing radiation, mainly in the form of SEUs [12, 42]. SEUs can affect user flip-flops and memory where the soft-core processor relies upon to function properly.

The majority of techniques for mitigation of SEUs in FPGAs are based on hardware spatial-redundancy. Notably, *Triple Modular Redundancy* (TMR) is the most common. When implemented correctly, TMR can mask single-errors and detect double-errors. But, depending on the level of implementation for a processor, it can be hard to recover the faulty unit.

Therefore, an often neglected fault-tolerance approach in the scope of processors is to use time-redundancy. In the case of SEUs, when rewriting an erroneous value inside a processor register, this action can restore the system correctness [29]. This process is done at the cost of processing time instead of hardware replication.

In general, this work's main contribution is a fault-tolerance technique, based on the concept of temporal redundancy, with checkpoints and recovery aimed at soft-core processors. In our approach, the improved architecture does not require modifications in the software source code or compiler, and is aimed at embedded systems for space applications, based on FPGAs.

The research is intended to demonstrate that the *Checkpoint and Recovery* (CR) technique is a valid alternative to TMR and even *Dual Modular Redundancy* (DMR). This contribution is especially important when dealing with determinant constraints for space applications: limited logic area and power budget. All of these constraints are allied to reach comparable levels of reliability.

1.2 Text Organization

The remaining of this document is organized as follows: Section 2 presents the main concepts regarding fault-tolerance for processors, followed by Section 3 discussing the related works in the area. Section 4 shows the proposed modified architecture in detail, while Section 5 detailing the experiments ran. Lastly, Section 6 concludes this work.

2 Reliability Improvement Strategies for Microprocessors

This Section presents the main problems and definitions associated to the space environment when considering embedded electronic circuits. Also, some of the techniques used in this context are described with focus on the time redundant approach. At the end of the Section, some of the related works are discussed and compared.

[1] Given the two major FPGA companies are based on the USA.

[2] ERC32 is a discontinued radiation-tolerant SPARC V7 processor developed for space applications.

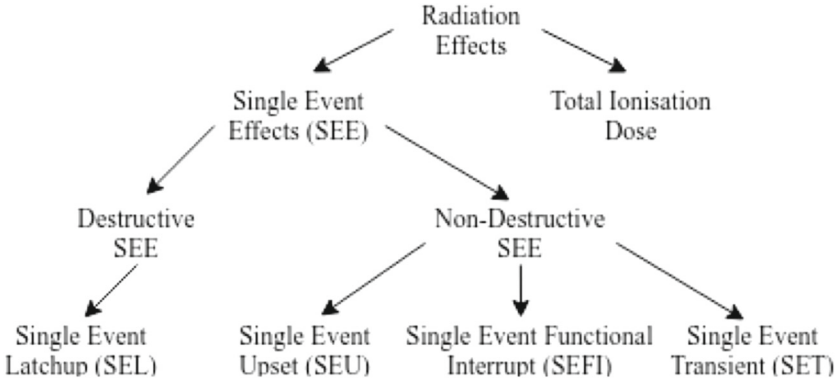

Fig. 1 Radiation effects on integrated circuits

2.1 Radiation Effects on Electronics

Cosmic radiation is generated by particles emitted from various sources that may be originated beyond the solar system. There are three main sources of charged particles responsible for faults in electronic components, namely: Cosmic Rays, Solar Winds and Van Allen's Belt [50]. Cosmic Rays are formed of highly energetic ion nuclei, these heavy ions represent only 1% of the component of cosmic radiation, being the remaining 83% protons, 13% helium nuclei and 3% electrons [10].

These sources of radiation interact with electronics causing different effects on integrated circuits. In reference [43] the common radiation effects, that must be mitigated on FPGA, are presented in the form of a tree, according to Fig. 1.

The SEE is detailed in Section 2.3. The effect called TID changes the voltage which must be applied to turn the device on (i.e. shifts the threshold voltage). If the shift is large enough, the device cannot be turned off, even at zero volts applied, and the device is said to have failed by going depletion mode [9].

Both radiation effects (SEE and TID) need to be taken into consideration when designing systems for space applications. However, each of them has different approaches to be mitigated, and they are also connected with the underlying technology/topology of the system (e.g., Flash memories are more susceptible to TID while *Static Random Access Memory* (SRAM) are more vulnerable to SEEs).

2.2 Fault, Error and Failure

For this section, the concepts are in agreement with [7], the relationship between fault, error, and failure, in the form of a chain of threats, can be seen according to Fig. 2.

For instance, in a processor, one of the outputs of an *Arithmetic Logic Unit* (ALU) may remain stuck at a specific logic level. In this example, the fault is a bit that cannot be not change. The error is a result of the failure (a sum with the wrong value for example). The failure is when another processor unit uses the erroneous result, propagating the problem to the rest of the system.

2.3 Single Event Effects

Errors caused by radiation, known as SEE, can be classified as soft-errors and hard-errors, and subdivided into the following [10]:

- Soft-Errors
  - *Single-Event Transient* (SET)
  - *Single-Event Upset* (SEU)
  - *Multiple Cell Upset* (MCU)
  - *Multiple Bit Upset* (MBU)
  - *Single-Event Function Interrupt* (SEFI)
- Hard-Errors
  - *Single-Event Latch-up* (SEL)
  - *Single-Event Gate Rupture* (SEGR)

The effect called SET occurs when a high energy particle reaches a certain point in the circuit, with the ability to change the output of a transistor. This changes the signal level for a period (in the order of nano/picoseconds), causing a glitch. As the name implies, it is transient, that is, there is a double transition (0 - 1 - 0 or 1 - 0 - 1) within this space of time. The effect of the SET is shown in Fig. 3, where a fault is indicated in the upper left AND gate, the transition of the output can be perceived in the third logical port, where the undesired effect occurs.

SEUs occur on the assumption that the particle reaches an element of memory by changing the stored data. The SEU is not considered permanent because, in the next write operation of the memory element affected, the wrong value

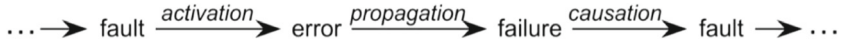

Fig. 2 Error Propagation - Relationship cause/effect between fault, error and failure

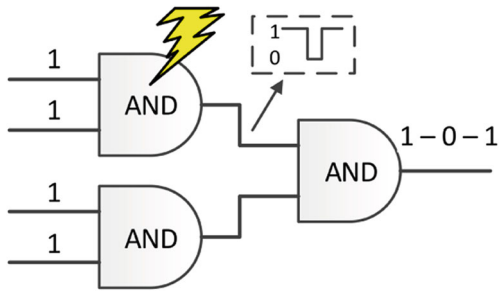

Fig. 3 SET Example

will be replaced. However, if the memory element is read-only from the system, the error can be propagated to the rest of the circuit and thus be considered a permanent error. Another situation of occurrence of SEUs is in the case where the SET propagates until it reaches a memory element, storing the undesired value.

When more than one SEU happens in a circuit, the effect is called the MCU. In case it occurs in elements that make up a larger register, it is then an MBU.

2.3.1 Fault-tolerance Techniques

The techniques for fault tolerance constitute a research topic in the area of systems reliability, and it is a well-established subject in the computing field [21, 29]. It is important to say that there are no 100% fault-tolerant systems [10, 21]; several factors are involved, and therefore there will always be a variable that can not be predicted or controlled.

In the specific case of embedded systems for space applications, fault tolerance is not only a necessity but an indispensable design requirement to increase the chances of success of a mission.

Fault tolerance techniques commonly use the concept of redundancy, which can be defined as the existence (logical or physical) of more than one resource needed to perform the action that must be fault tolerant. Although the word redundant, when used in the context of computational systems, can represent the idea of physical replication of components, there are four basic types of redundancy [29]:

- Hardware: The most natural concept to replicate hardware and use it whenever a fault is identified.
- Software: To be used for software failures, there may be two or more code snippets running to prevent failure.
- Information: When redundant data is added to the basic information, like Hamming codes in registers, for fault tolerance.
- Time: Use redundancy in time to tolerate failures, i.e., perform the same activity two or more times, one after the other, to ensure a correct result.

Examples of hardware redundancy can be simple implementations - such as the addition of circuits' replicas, one of which is used as the primary circuit and the remaining redundant. When the fault is detected, the logic is switched to use one of the redundant circuits. This type of technique is known as static hardware redundancy. The correct output (fault-free) is selected through a majority voter. The TMR technique can be applied at several levels of abstraction, such as architecture (the ALU within a processor) or at lower levels in the system.

In the case of software redundancy, it is possible to have variations at all levels, such as data, program flow control, and hybrid combinations. For example, we perform the same task for two different software versions with the same objective. If there is a divergence of results, an action is taken.

For information redundancy, the most explicit example would be to add data to information of interest, to identify, mask, and tolerate errors. The data coding technique, known as a checksum, calculates the data (as an *xor* operation) and adds the result to its end before transmitting or using it. Once coded, one must make the same calculation and compare with the attached result, in which case, if there is a divergence, the failure can be identified.

Finally, temporal redundancy is the repetition of the computation of the same task over time, with the results of each of the repetitions being compared, to be able to identify the fault. The most common temporal redundancy is called rollback recovery technique, and is done by performing checkpoints during the execution of a program, at specific intervals. Assuring that these points do not contain errors, in the event of a failure, the system can return to the last checkpoint and redo the execution.

The technique of interest in this work is based on the concept of inspection points called Checkpoints. Checkpoint and recovery can be done in computer systems, such as processors, simply by saving the state of interest and, if an error is detected, return to that state to redo the execution.

Considering that there is no single taxonomy for fault tolerance techniques, this section was intended to demonstrate one of the possible approaches to the subject.

2.3.2 Checkpoint Recovery (CR)

The CR technique is a classic fault-tolerance technique, which enables computing systems to execute correctly even when affected by transient faults [25, 44]. The works based on the technique of CR are traditionally classified according to the level of abstraction implemented by the system. This classification is divided into techniques that make changes to software-only or hardware-only [15].

While software solutions are cheaper from the perspective of implementation, purely hardware based have a very low overhead potential in the execution time of the same software. It is also possible to have a combination of both, denominated hybrid (hardware and software).

Although the concept of the technique is simple, several problems arise with the implementation, especially when taking into account the essential details of the development, such as level of abstraction, transparency for the end-user, number of checkpoints, at what point in the program to checkpoint, etc.

2.3.3 Checkpoint Recovery Overhead

Like all redundancy-based techniques, there is an associated overhead, whether temporal or physical. In the case of the CR technique applied to a processor, the overhead is associated with the additional execution time of the program, while there are no errors. In other words, the amount of time when the system is blocked from execution to perform a checkpoint. Figure 4 illustrates the additional execution required on a system with CR. The execution of the program with the CR has points where it is necessary to perform the checkpoint, represented in grey tone in the figure. When execution is interrupted to the checkpoint, the same program suffers an addition at runtime.

3 Related Works

The works developed by [27] and [32] present combined fault-tolerance techniques applies to the LEON3 soft-core processor for FPGAs.

Keller and Wirthlin [27] use five different SEU mitigation variations: no SEU mitigation, TMR alone, TMR with *Block-RAM* (BRAM) scrubbing, TMR with *Configuration-RAM* (CRAM) scrubbing, and TMR with both BRAM scrubbing and CRAM scrubbing. Both fault injection and neutron radiation testing were conducted. Improvement is measured in terms of sensitivity reduction for fault injection and cross section reduction for neutron radiation testing when compared to the unmitigated design. The results from both fault injection and radiation testing demonstrate that each variation of SEU mitigation techniques improve the SEU sensitivity of the LEON3, and that improvement increases as more mitigation techniques are combined. When compared to the unmitigated design, SEU sensitivity is improved from 16 to up 50 times. The full mitigated version comes at a cost of 4.7 times increase in area of the FPGA.

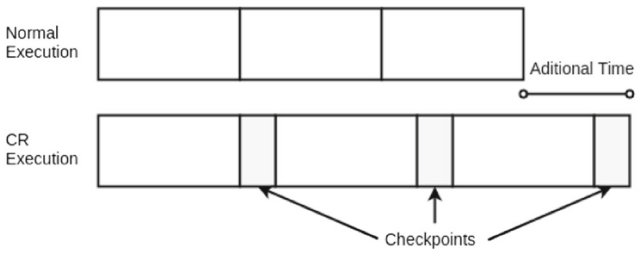

Fig. 4 CR overhead

In reference [32] authors implement a hybrid fault-tolerant LEON3 soft-core processor in a Xilinx Artix-7 FPGA and evaluate its error detection capabilities through neutron irradiation and fault injection. The error mitigation approach combines the use of *Single-Error Correction / Double-Error Detection* (SEC-DED) codes for memories, a hardware monitor to detect control-flow errors, software-based techniques to detect data errors and configuration memory scrubbing with repair to avoid error accumulation. Radiation test results show an improvement of 4.13 times for the hardware-only mitigation techniques. Fault-injection test includes the software hardened approach in combination to hardware and have an average 20 times better improvement. Both results are compared against the unmitigated variant of the processor.

Li et al. [31] propose a transient-fault countermeasure called RELI, which is a fine-grained CR approach for *Application Specific Instruction Processor* (ASIP)-based embedded processors. RELI is supposed to be the first to realize CR at the basic-block level by leveraging custom instruction design. To implement RELI, an ASIP design flow based on one of the existing commercial tool (ASIPmeister), generate the *Register-Transfer Level* (RTL) description of the resultant processors with RELI functionality. The costs concerning execution time, area, and power are reduced significantly compared to existing techniques.

The augmented processor (i.e., RELI processor) allows CR to be executed at a finer granularity than other works, such that the checkpoint data size is reduced. Assembly code from MiBench benchmark suite [22], compiled using SimpleScalar toolset is used to generate the comparisons. The experimental results show that the fault-free execution time overhead is 0.76 percent on average. In the fault injection test, for the worst case, the recovery time is 62 cycles. RELI costs 44.4 percent area and 45.6 percent leakage power overhead on average (for the TMSC65nm technology), and 79.3 and 77.8 percent in the worst case found in SPEC-INT2006 and MiBench suites.

In reference [45] is presented another work, aimed at the embedded processor internal registers. The register data dependency is used to minimize the register file traffic required by the register file CR. The proposed logging CR

scheme, named RECORD, considers various register data dependencies, which can potentially identify and eliminate the redundant executions of register file checkpointing at runtime. This approach is supposed to be the first to realize a hardware-based logging checkpointing mechanism, which strategically utilizes the first processor executions to diminish the additional checkpointing operations at runtime, for embedded processors. RECORD is implemented in an ASIP to evaluate the proposed scheme for embedded processors. The technique presents a lower register file traffic and better dynamic power saving with little hardware and performance overhead when compared to other works.

In reference [16] is proposed a *Dual-Core Lock Step* (DCLS) approach to increase the dependability of hard-core processors embedded in programmable SoC, which combines the programmable logic with the high-performance hard-core processor. The DCLS is a dual-core ARM Cortex-A9 processor embedded into the Zynq-7000 APSoC. It is a novel implementation of lockstep in the dual-core Cortex-A9. ARM provides some processor's versions with built-in lockstep, such as Cortex-R5 processor, which could be configured to application reliability.

Two versions of the technique are compared with the unhardened Cortex-A9 processor. The first uses only the BRAMs to store the checkpoint data, and the second uses the external DDR memory as secondary storage for the checkpoint data. Area results show an increase of 100% for the processor and memories. As for the execution time, three matrix multiply programs are evaluated. Being the longer the execution time, lower is the time overhead. The BRAM version has an increase of 26%, and the DDR version has a 47% increase on the total clock cycles for the 20x20 matrix. The further work by the authors in [16] shows that up to 91% of the bit flips injected in the ARM registers are mitigated by the proposed technique.

The work presented in [54] present a design flow that can be used by designers to mitigate radiation-induced errors affecting processor IP cores embedded in FPGA-based SoCs for systems that have to be deployed in harsh environments. The design flow used the concepts of lockstep, checkpoint with rollback recovery, and on-demand configuration memory scrubbing (in case of SRAM-based FPGAs) to provide a balance between resources overhead and fault tolerance. The flow can be automated, reducing the total development costs, while increasing the quality of the resulting product. The authors provide a prototypical implementation of a design environment, supporting the proposed flow, and applied it to the design of a system using a Leon processor IP core.

The time overhead for this implementation ranges from 17% to 54%, depending on the software executed. In the fault injection campaign, 10,000 random SEEs were injected, 84% of them became latent or detected and corrected; 15% triggered errors in the system (the authors modified the instruction trap of the processor to perform a rollback), and the configuration memory scrubber handled the last 1%.

All strategies focused on processors presented in this section require modification to software and/or compiler in addition to the hardware. We propose a pure hardware-based solution to deal with SEUs. In our approach, there is no need to rewrite — or even recompile — the original software source code. The fault-tolerance technique is performed in the modified architecture.

4 Proposed Checkpoint Recovery Technique

The CR technique works by saving checkpoints considered safe during the execution of a processor [29]. Whenever an error is detected, a rollback to the last known safe state is performed, namely recovery. To better understand the CR technique, Fig. 5 depicts a hypothetical scenario: after a checkpoint (Ck) is performed at $t = 2$, instructions I_{n+1}, I_{n+2} and I_{n+3} are executed. At time $t = 6$ the error is detected, causing the recovery to occur. After recovery, the three instructions are executed in the same fashion and the fault is overwritten with the right result.

If the SEU occurs in an element of the circuit, and, if the element is overwritten with the correct value after the SEU is identified, the error can be corrected. Therefore, the CR technique, which repeats the operation of a point considered safe, is a reasonable solution.

The following subsections present in detail the implementation of the CR technique.

4.1 Constraints and Assumptions

Before we advance into more details about the proposed technique implementation, some of the design decisions made need to be explained. We consider the environment to be the space, more precisely, an embedded FPGA on satellites. Up to *Low Earth Orbit* (LEO), the expected radiation dose is around 0.1 krad/year, meaning a five-year mission can have ∼0.5 krad dose [38]. The *Geosynchronous Earth Orbit* (GEO) can also be considered once it has a dose rate of ∼10 krad/year, but the selected FPGA has to withstand this dose.

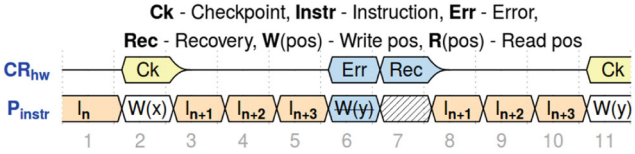

Fig. 5 Checkpoint Recovery Technique Scenario

For that reason, the FPGA hardware is flash-based, in our case the Microsemi ProASIC3e FPGA [34]. In this type of FPGA, the configuration memory is not affected by SEUs [52].

Although the configuration memory on the ProASIC3e is susceptible to TID, a dose of up to 30 krad seems not to affect the FPGA implemented circuit [26]. The use of SRAM-based FPGA, at the present stage, have not been considered, mainly because the configuration memory is highly sensitive to SEEs. Once the configuration memory is affected, the underlying implemented hardware (in our case, the soft-core processor) can behave erroneously. The error mitigation of configuration memory is a vast field of study with specific techniques, that could be integrated into this work.

The assumed fault model that is being mitigated is the SEE, more precisely its subtype SEU. Literature shows that SEU is the predominant failure when considering processors [30, 40]. Also, we assumed in our fault model that only single-faults can occur. Depending on the operation frequency in use, the likelihood of an SET can be considered negligible, given the current technologies [5].

For the CR technique, the granularity of the checkpoints (*i.e.* how often checkpoints are performed) need to be taken into consideration, once it introduces overhead in the processor execution. We perform a checkpoint after every write operation to the main memory, similarly to [54]. This approach assumes that up to that point, if the error-detection mechanism did not identify the error, the state of the system has not been compromised. Another possible approach that could be used, presented by [39], is to save a checkpoint before the occurrence of a jump instruction in the execution of the program.

Once we are dealing with SEUs, the main storage system is vital to keep the system running. Since the program runs on the main memory, if it presents errors the processor can misinterpret the instructions. For that matter, the main memory is assumed to be external and protected by an *Error Detection And Correction* (EDAC) technique. Furthermore, the cache memories are disabled for two reasons: they are additional area susceptible to SEUs, and since we are using writes to the main memory as reference points, the caches can interfere on the processor synchronization.

Also, like any other technique of fault-tolerance, there are two stages to implement fault-tolerant systems: Error-detection and Error-correction. These are two separated phases, which most methods integrate them into one single scheme. *e.g.* the TMR approach works by voting the majority of results and masking the disagreeing information. The voting process can be seen as the error-detection stage, and thus the masking is the error-correction. With this in mind, we propose the use of the CR technique to detect errors, by executing twice every slice of instructions (comprised between two checkpoints) and performing a third execution of the slice to correct a possible detected error. Nonetheless, other error detection schemes are implemented to be compared.

4.2 Test Vehicle

The LEON3 [2] is the processor chosen as the target system of this work, due to the significant acceptance in the scope of space applications. It is a synthesized model, described in VHDL, of a 32-bit processor, 7-stage pipeline, compatible with the SPARC V8 architecture, made available by the company Aeroflex Gaisler, under the GNU GPL license. The source code is free to use for research and educational purposes and is distributed as part of the GRLIB IP library [1].

LEON3 is very configurable, being easily integrated into SoCs, accepting the multiprocessing configuration (up to four CPUs) and a wide variety of peripherals. More specifically, the LEON3 CPU core is based on a seven-stage pipeline, and may include other processing modules, such as a floating-point unit. In addition, a unit called *Debug Support Unit* (DSU) is integrated with the processor, which is also connected to the *Advanced Microcontroller Bus Architecture* (AMBA) bus, to aid in debugging the CPU.

The GRLIB provides several designs, including different FPGA vendors. These designs have a common characteristic of a single VHDL file for the top entity (`leon3mp.vhd`) and another file for the configuration of the processor (`config.vhd`). The top entity contains the instantiation of the `leon3s` that comprises the processor and its internal components. The VHDL code is very modular, with each component within separate file.

Since the same entity responsible for the cache it is also responsible for the AMBA interface, it will always be instantiated inside the LEON3 processor. When the cache memory is disabled, the internal *Finite State Machine* (FSM) bypasses the cache memory access. The main components comprising the `proc3` entity and the

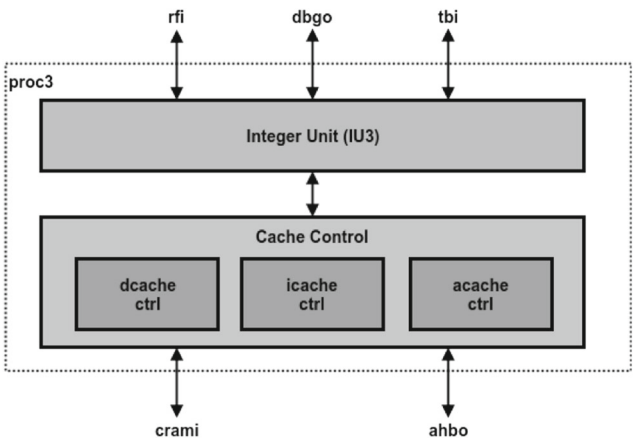

Fig. 6 PROC3 Connections Overview

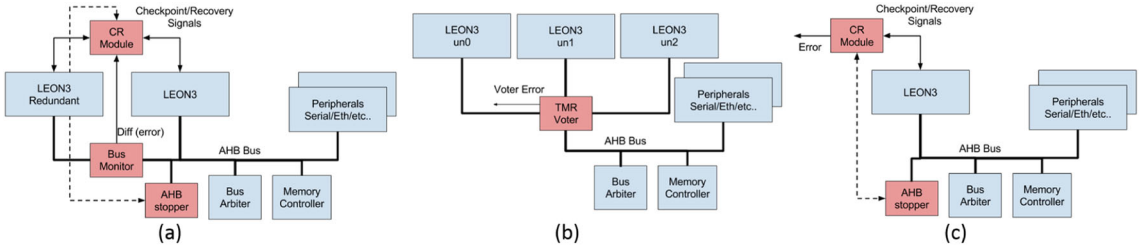

Fig. 7 Different Architectures Used for Error Detection: (**a**) bus-based DMR, (**b**) bus-based TMR, and (**c**) single-processor time-redundant

relationship between the Integer Unit (IU3) and the Cache Controller/AMBA Interface are depicted in Fig. 6.

4.3 Implemented Error-Detection Approaches

To perform the rollback in the processor, the CR hardware needs to be aware of the error, thus an error-detection must be implemented. There are several error-detection techniques in the literature. This study does not primarily aim at the detection of a SEU (i.e., error detection), as it can be considered another field of study by itself. Instead, we used fault tolerance techniques, which have fault-detection as their starting point. Three techniques have been used: the classical TMR [33]; a bus-based DMR approach [18]; and a time-redundant execution. The Fig. 7 presents the three architectures used in the experiment.

Figure 7a uses a bus-based DMR to detect errors and inform to the CR module to perform the rollback on both processors. Figure 7b is a classic TMR where it always detects single errors and masks single-faults using a majority voter. Figure 7c employs the time redundant approach that executes twice every slice of code. In this case, the CR module saves the address and data that is going to be written on the main memory on the first attempt. After, rollback is performed, and the second address and data generated are compared with the ones saved in the first execution. When there is a match, the memory write operation is performed and a new checkpoint is saved, advancing the code execution to the next slice. If the values do not match, the second execution address and data are also stored by the CR hardware and another rollback is done to have a third execution of the code. This way, the CR hardware can use the result of three executions to perform a simple majority vote (similarly to the TMR) and write to the main memory the correct value. In the case of three different executions, an error signal is raised, similar to the voter error on TMRs approach, bringing the processor to a halt.

4.4 Implemented Checkpoint Recovery Approach

During its normal operation, the processor creates checkpoints, which represent consistent states that can be restored. The checkpoints are a copy of the current processor state, more specifically the content of the pipeline registers. Any changes to the register file since the last consistent checkpoint are saved. The granularity of the checkpoints was designed, in such way that one checkpoint is created every time the processor executes an instruction that performs writes in the main memory. Since the main memory is the reference, instruction and data caches were disabled on the processor configuration. Even though the absence of cache in the processor degrades overall performance (regarding execution time), it also introduces another point of failure for SEUs.

To implement the CR technique, the LEON3 hardware was modified. The first step was to find all the registers on

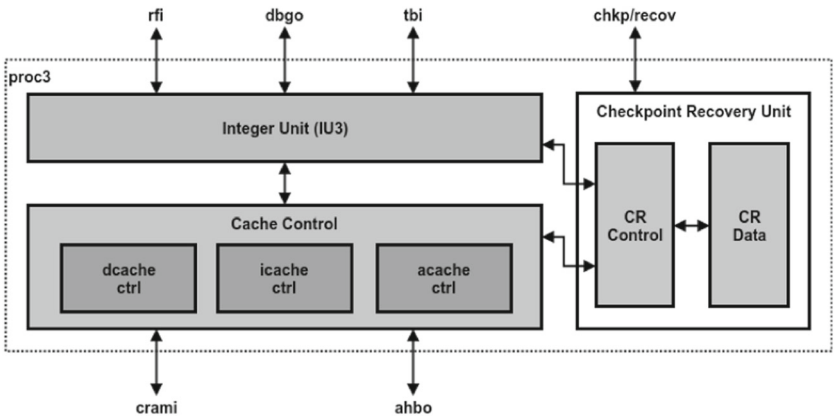

Fig. 8 Modified PROC3 unit with CR Control Unit

the pipeline that holds the current state of the processor. In more detail, the IU3 unit has VHDL processes, comprising the entire pipeline that needed to be saved. Despite the fact that the instruction and data caches were disabled, there are FSMs that control the communication between the *Integer Unit* (IU) and the AMBA bus, and need to be checkpointed as well. A single checkpoint signal is connected to all modules involved. When the main memory write is detected, it causes the checkpoint by copying all the data to redundant registers.

In Fig. 8 the modified `proc3` unit is presented with the internal connections to the IU3 and Cache Control units, the requests to perform the checkpoint or recovery is done through a dedicated set of signals (indicated in the chkp/recov signal on the figure).

The register file, likewise, needs to be taken into consideration when recovering the processor state. In order to do so, a fourth port was added to the register file to perform a read on the register that is currently being written. This way the old value can be saved in a memory stack. On the recover event, the stack is dumped back into the register file, bringing it back to its safe state (last checkpoint). This process was made inside the `leon3x` unit and is presented in Fig. 9. The Register File Checkpoint Unit is responsible for multiplexing the connections between the `proc3` unit and the modified 4-port register file. In the normal operation, the fourth-port address bus is connected to the write port address, meaning that when a write operation is performed, the fourth port data output the register value being overwritten. This data value, along with the address, is then pushed into the stack by the Checkpoint Unit. In the event of a new checkpoint, the stack memory is flushed since all values inside the register file are supposed

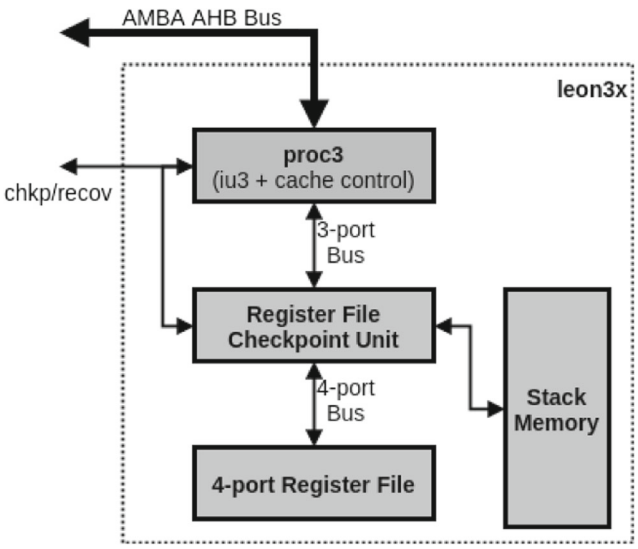

Fig. 9 Modified LEON3X unit with Register-file Checkpoint Unit and Stack Memory

to be correct. If an error is detected, the recovery process is activated and the Checkpoint Unit initiates to perform writes to the Register-file. The address and data are pushed out of the stack and written to the Register-file. When the stack is empty, the recovery process of the Register-file is finished.

To perform a recovery on the aforementioned system, the processor needs to be halted for a period of time. This time is required to write the registers back into the Register-file, and recover the IU3's pipeline. In order to do so, a second AHB-master unit is connected to the AMBA bus. Its function is to request the AMBA-bus, through an write request, forcing the LEON3 processor into a halt state. While the second AHB-master owns the AMBA bus, the recovery process is done. This unit is part of the CR implementation.

Going into detail, the top-level VHDL file (`leon3mp.vhd`) of the design on the GRLIB instantiates the unit `leon3s`. This unit is a wrapper to the aforementioned `leon3x` unit, with a few connections to `gnd` and `vcc`.

4.5 DMR and Time-redundant Implementation

The implementation of the DMR and Time-redundant approaches have different fault-detection schemes, while the former is based on transactions on the AMBA bus, the latter compares the pair address/data being written to the main memory.

For the DMR implementation, there is a module that compares transactions on the AMBA bus. Figure 10 depicts the main connections of the LEON3 in order to achieve the same results presented by [18]. The modification here are the ones presented to get the CR technique running inside of each LEON3 processor (presented in the section above). Note that the controller of the CR technique is in the top-level, along with the instantiation of both processors. Whenever the outputs do not match, a signal error is raised, the controller request the AMBA bus, and when it granted, it sends a recovery signal to both processors. After the recovery, both processors continue to run the program.

The time redundancy is obtained by using the CR mechanism, to run each interval between checkpoints twice. In order to do so, the main connections of the LEON3 Time-redundant are depicted on Fig. 11. Figure 11a presents the top level instantiation of the LEON3 and the AHB unit to the AMBA bus, and Fig. 11b presents the modifications made inside the already modified `proc3` unit (Fig. 8). Note that Fig. 11b is a detailing of the LEON3 unit in Fig. 11a, that includes the modified leon3x unit with the CR control logic and a write mux to the main bus.

On the first run, the processor saves the information of the memory write instruction, but does not allow it to proceed, bypassing the memory write enable signal (Write Mux on Fig. 11b). Then, a rollback is performed, and the

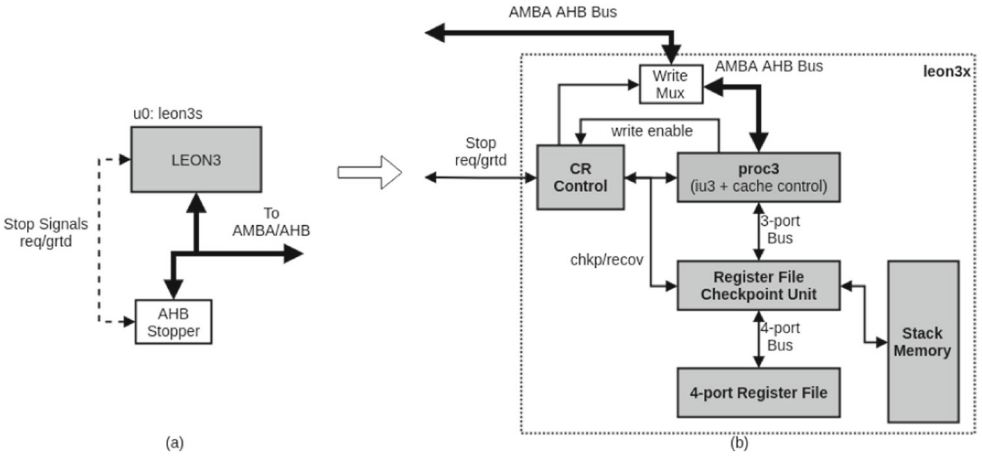

Fig. 11 Detailing of the LEON3 time redundant connections

processor executes all instructions from the last checkpoint. When the second run reaches the memory write instruction, the CR mechanism compares address and data to the ones stored from the first run, if they are equal, the main memory is written and the process repeats, otherwise, the fault is detected and a mismatch is signalised.

Since the checkpoints are based on memory write, both techniques presented here (DMR and Time-redundant) monitor the AMBA write signal coming from the processor, whenever it raises to high, the checkpoint is performed overwriting the old one.

4.6 Checkpoint Recovery Hardware Considerations

Each checkpoint is an image of the state of the system considered safe. Such checkpoint is a form of data redundancy. In our case, the data redundancy is comprised of the processor pipeline and register file modifications. As aforementioned, not only the `iu3` unit and the register file that contain information but also, the units `icache`, `dcache`, and `acache`. Table 1 presents the amount of data in bits for each unit and the size of the stack for the register file address and data.

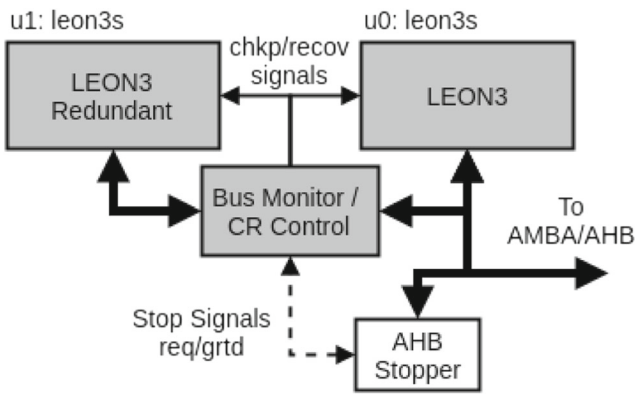

Fig. 10 Detailing of the LEON3 DMR connections

The `iu3` unit individually has the major quantity of data (2502b) since it is a copy of the entire LEON3 processor pipeline. The stacks summed account for 2560 bits since they are 64 positions of 8 bits for the address and 64 positions of 32 bits for the data. Both stacks can be easily protected against errors using an *Error Correction Code* (ECC) based on the requirement, ranging from parity to extended-hamming or *Cyclic Redundancy Check* (CRC).

The other components' checkpoint data can be protected in a similar form, but preferentially with the use of signatures (such as checksums) since there are different registers widths. It would be possible to read the entire checkpoint as a string of bits and calculate a signature to confirm integrity.

Another weak point is the checkpoint hardware control and its components. This hardware is also susceptible to SEUs that could cause the system to malfunction. The checkpoint hardware is mostly comprised of combinational logic and the amount of data stored is relatively low when compared to the entire SoC. Since combinational logic is not affected by SEUs and the stored data can be further mitigated, at this stage, we consider that it would not be affected.

Lastly, it is important to mention that there are no modifications outside the LEON3 RTL code. This means the same code, compiled to the original LEON3, can be run

Table 1 Checkpoint storage data size

Component	Bits
iu3	2502
icache	323
dcache	830
acache	34
stack data	2048
stack addr	512

seamlessly on our architecture. The only difference is how the code is going to be executed and recovered (in case of an error).

5 Experimental Results

In this section, we describe the adopted simulation method, test setup, and benchmarks used in our tests to obtain simulation results. We use the fault definition according to [7]. All results here described, have been based on premises from Section 4.1.

5.1 Simulation Method

[56] presents an extensive survey that compare the different techniques for fault injection, and summarizes their advantages and limitations.According to our objectives, the fault injection technique chosen had to meet a set of characteristics such as: full access to the entire processor design without being intrusive, a good time resolution and high observability. To run our tests, the LEON3 processor was simulated using the Modelsim tool. 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 enough confidence in the results, imposed an upper bound in the size of the workload running in the processor during the experiments.

The fault injection was performed according to the pseudo-algorithm presented in Fig. 12. The fault injection script reads all LEON3's IU registered signals (memory elements). For each signal, a new simulation is run (line 2). In each simulation, a random time is picked (line 3) and ran. After the runtime, the current signal value is read (line 4), and a SEU is simulated by inverting one bit inside the signal value (line 5) and applying it to the current signal using a force command (line 6). Note that this `force` command modifies the signal until it gets overwritten, known as *deposit* on the simulator tool. Finally, the simulation is run until its end. This means that the program comes to its final state, by raising a stop signal, or an error signal (if detected by the simulation script). In line 10 we make sure we have enough samples to fulfill a confidence interval of 95% and a margin of error less than 5% (since it is a simple random sample: $0.98/\sqrt{n}$, or at least 400 runs).

```
1  do{
2    foreach(signal current in L3.IU3){
3      run rand();
4      value = read(current);
5      seu(&value);
6      force(current, value);
7      run all;
8      runs++;
9    }
10 }while(runs < CONFIDENCE);
```

Fig. 12 Simulation Steps Pseudo-algorithm

The simulation results were classified according to Fig. 13. After fault injection, there are three possible results (outcomes): Correct, Detected, or Failure. A correct result is reached when either, no error were detected or the error is latent. A latent error means that the fault in that signal, at a given time did not affect the execution. A failure means that the fault causes a failure in the processor without being possible to detect it. Lastly, the detected fault is the result of an error, which can be further classified in three possible situations according to the fault-tolerant technique used: Recovered, Not-recovered, and Recovered incorrectly. A recovered case is when after detect, the recovery process acts accordingly, and the program finishes its execution with the expected result. A not-recovered error happens when the recovery process fails to complete the program, either without the expected result or a time-out. The last case is when the recovery process is performed, and the program reaches its final state with an incorrect result. This can happen when the error occurs on the variable that controls a loop, for example.

5.2 Experimental Setup

For each architecture of our tests a set of four programs were used to stress the processor instruction set as follows:

1. Basic: a simple arithmetic operation executed 50 times and checked against the correct value.
2. Bubble sort: classic benchmark algorithm that is executed five times on a ten element vector.

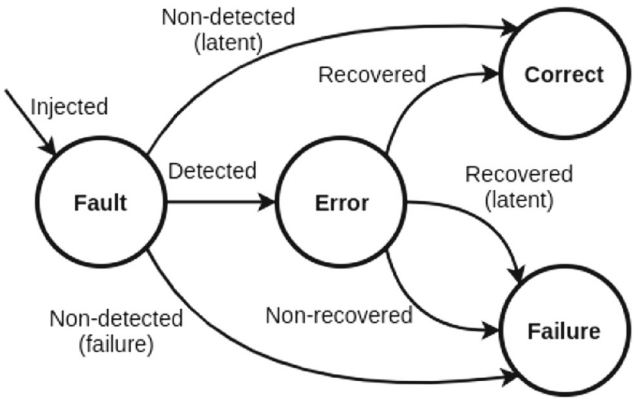

Fig. 13 Fault states diagram

3. NMEA: calculate the checksum (bitwise *xor*) of ASCII codes on a message string five times.
4. Hamming: calculate a hamming encoded message using matrices five times.

It is important to note that we did not use a more classic test program (such as *dhry*, *stanford*, or *whetstone*) since the simulation time was prohibitive, *e.g.* over a day on a high-end computer for a single execution. In order to circumvent this issue, the above programs were written in standard C language trying to comprise some of the classic code flow execution. Nonetheless, the chosen workload applied to the four variations of the LEON3 took over a week of computer simulation. This translates into over 2GiB of raw data logs.

The *General Purpose Input Output* (GPIO) pins are used to signalize the external world when it began and finish. These signals are used to assert the correctness of the execution and/or error states. For instance, if the program `bsort`, on its verify state, find an unordered value, an error signal is raised to communicate the simulation script.

The compiler used is the standard sparc-elf-4.4.2 toolchain. The following flags have been used on compilation and linking:

```
CFLAGS=-msoft-float -Wall -O0
LDFLAGS=-qsvt -qnoambapp -lsmall
```

5.3 Detection and Recovery Capability Analysis

Results from the simulation were analyzed and compiled according to Section 5.1. This section presents a comparative analysis of the four variations of the LEON3 processor using the workload mentioned before.

Figure 14 presents the detection analysis for the three architectures used in the experiment with the inclusion of the LEON3 original (unmodified) configuration. The Y axis on the left shows the total of executions in the simulation ran, and on the right Y axis the percentage of these figures. Note that for the original configuration there is no detection available. Therefore only the Correct/Failure results are presented.

In the original design, it is important to notice also that only around 15% of the injected faults resulted in a failure. An explanation for that is the fact that they have been randomly injected, thus affecting processor resources not involved in the program execution. In the adopted simulation-based strategy, the fault-injection campaign is extremely slow and, at the time this paper was written, it has not been possible to run and to collect the simulation results for all variations of the processor (original, DMR, TMR and CR), considering the proposed workload.

For the architectures of the TMR and the DMR, the correct rates were similar, in the order of 79% on average, which means that the fault is either latent, or not detected. The failure rate of the original is slightly lower than the detected figures in the TMR and DMR approaches. This is due to a detected error not always becoming a failure.

Interestingly, the time redundant approach shows the higher percentage of corrected results (in the order of 95% on average). This is due to the re-execution of the code slice since the injected fault can be overwritten before it manifests itself during the program execution.

The errors classified as failure appears on LEON3 original and time redundant implementations. After a closer look into the simulation results, it is possible to note

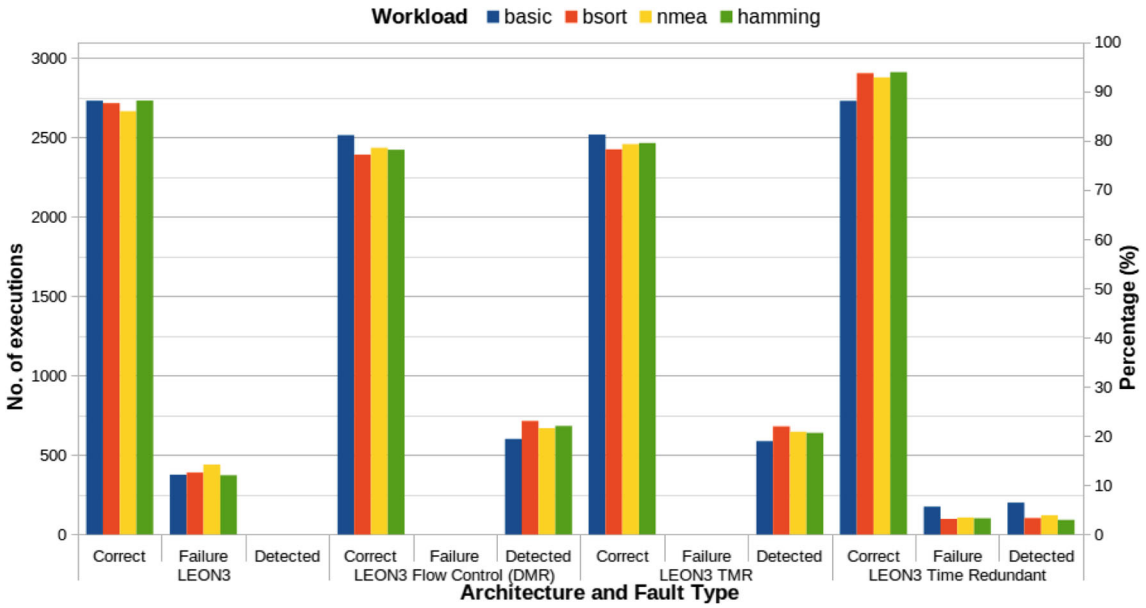

Fig. 14 Detection analysis comparison of different LEON3 architectures

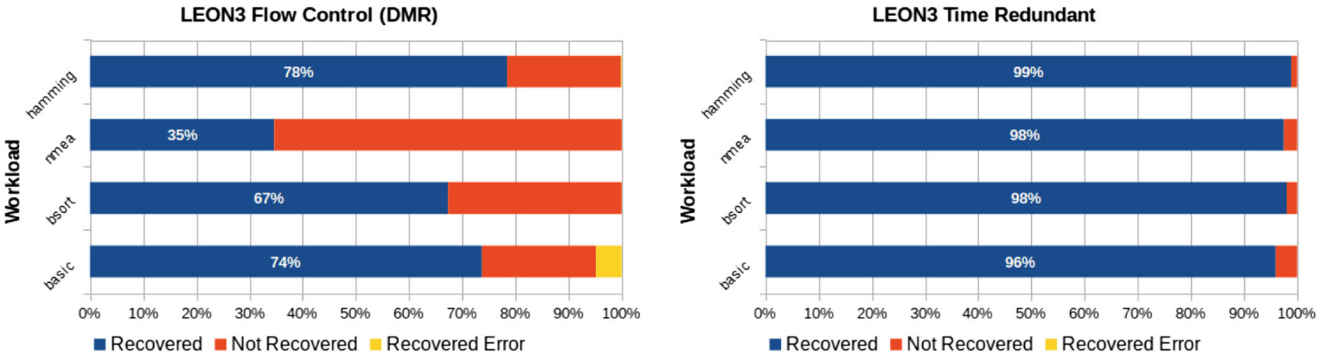

Fig. 15 Recovery analysis for the DMR and Time Redundant approaches

that some signals have an immediate effect on processor execution. For instance, internal signals of the pipeline stage `EX→nerror` and `ALU→Ticc`, are responsible for the general processor error and Trap interruption control, respectively. These signals can cause a failure every time they suffer a simulation SEU. Still, the time redundant presents an improvement over the original implementation for this type of error.

In a previous work [48], we investigated 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. The CR technique could be further improved, in a future work, by performing a rollback whenever the processor is in the process of halting.

Figure 15 shows an analysis for the recovery process on the detected errors for the time redundant and DMR approaches. Note that these charts are based on the absolute number of errors detected, consequently the breakdown of the values are presented on stacked percentages, so it would be possible to compare both techniques. The TMR is not shown since it has 100% correction for single faults. However, TMR would have to, somehow, recover the faulty processor, otherwise, the error gets accumulated on the system. The original configuration is not presented once there are no detection/correction mechanisms.

For the time redundancy approach, the average errors that were corrected, is near the 98% mark, while the average for the DMR is a little over 63%. The main problem with the DMR, for our tests, is to recover both processors correctly.

The overall performance comparison for the DMR and time redundant approaches is depicted in Fig. 16. These charts present the total percentage of executions, for each program, which finished with success, including those detected and corrected.

The averages of correctness are 92% and 95% for DMR and time redundant approaches, respectively. For the time redundant, the average 5% of failures could be further mitigated due to the signal sensibility of the LEON3.

5.4 Execution Overhead Analysis

Although the CR technique presents a competitive recovery capability, it introduces time overhead on the program execution. Whenever a recovery is made, the execution needs to be halted for, at least, one clock cycle, allowing the recovery of the IU pipeline registers and an additional clock cycle for each register in the register file used since the last safe checkpoint. Table 2 presents the increase percentage on the workload execution against the original implementation of the LEON3 processor.

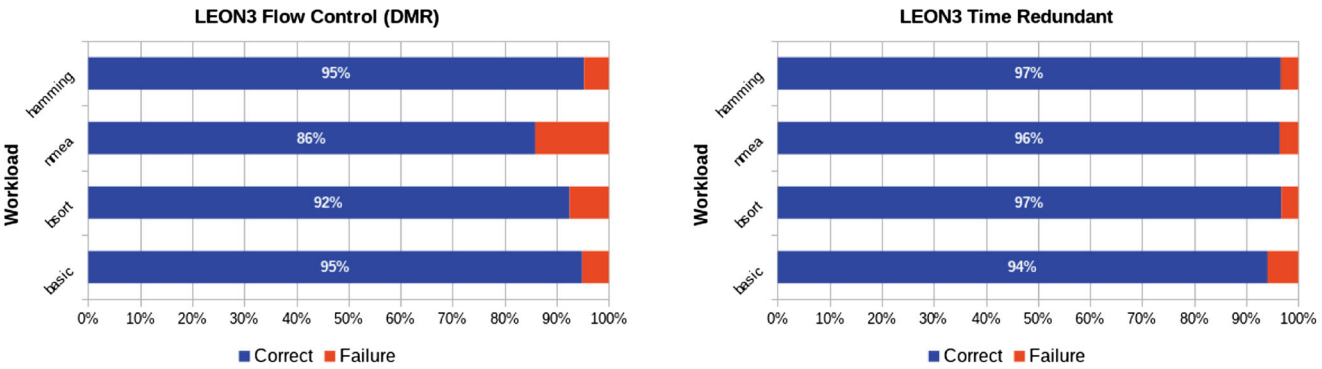

Fig. 16 Overall comparison of LEON3 DMR and time redundant approaches

Table 2 Execution time overhead against baseline

	LEON3 flow control		LEON3 time redundant	
Workload	Correct	Recovered	Correct	Recovered
basic	0.00%	4.39%	112.88%	113.61%
bsort	0.00%	0.25%	104.90%	105.00%
nmea	0.00%	1.09%	104.90%	105.12%
hamming	0.00%	1.07%	106.71%	106.99%
Average	0.00%	1.70%	107.35%	107.68%

Each implementation of the LEON3 has different time overhead on the execution. The LEON3 DMR does not add time to perform the checkpoints since the checkpoint procedure is done in parallel. Therefore, no overhead is noticed when there are no errors detected on the execution. Although, once an error is detected, the recovery procedure takes a few clock cycles to occur, hence the average value of 1.70% of time increase against the baseline execution.

The cost of executing twice each slice of code out-stands on the LEON3 time redundant approach. On average it adds 107.35% for correct execution and 107.68% when the error is detected.

The impact of the time redundant approach can be isolated to analyze how long it takes to perform a rollback (recovery process). Table 3 presents the simulation time that the CR hardware needs to act on the system and its mean occupation of the stack of the register file. In our simulation, the clock cycle was configured to $25ns$ and the stack has 64 positions (Section 4.6). Each rollback process takes, on average, 17 clock cycles to finish and rewrite seven registers on the register file.

5.5 Cache Influence Analysis

Since we have disabled instruction and data caches, it is possible to analyze the time overhead due to this decision. Table 4 presents the simulation time (in ns) and increase ratio. The time increment due to the removal of caches and keeping the unmodified architecture has an average of almost seven times slower than with caches. Also, without cache and adopting the time redundant approach has an average of 14 times the original time.

It is a high price to pay in exchange for reliability. Nonetheless, the primary goal is to have a fault-free execution instead of the fastest possible execution.

5.6 FPGA Area Overhead Analysis

It is important to compare how the different architectures influence on the area occupied. The implementations went through the synthesis tool on the Microsemi design flow. At this stage, preliminary results can be obtained for the target FPGA. The data from Core (VersaTiles) and RAM for a Microsemi ProASIC3E-1500 FPGA are presented in Table 5 along with the increase percentages for each variation.

The TMR could not be implemented on this device. The same goes for the DMR approach, which cannot be fit in the device at the current stage of the design. It is possible to note that both results of area match the order of footprint increase. The lower is the time redundant, followed by the DMR and lastly the TMR. This confirms the consequence of replicating the processor unit inside the SoC.

5.7 FPGA Power Analysis

Another critical figure when designing space applications is power consumption. Microsemi offers a spreadsheet [35] that can estimate power consumption of its devices on very early stages of development. At synthesis, it is possible to use the amount of VersaTiles and RAM occupied, along with the operating frequency of the system, to estimate dynamic and static power.

Table 6 shows the power consumption estimation results for a ProASIC3E-3000 FPGA. The following configurations were used on the power estimation tool:

- Device: A3PE3000
- Range: Commercial
- Condition: Typical
- Mode: Active

The decision to estimate values on a larger device was based on the spreadsheet limitations. The calculator spreadsheet does not allow to enter with a number of Cores/RAMs higher than the available on the chosen device. Since this is an estimation for comparison, the differences on the dynamic power figures[3] are negligible. The major difference is the static power, which is the amount of power that the device consumes independently of the implemented

[3] Experimenting on the spreadsheet, less than 0.5mW difference on the dynamic power was noticed for the 1500 and 3000 device.

Table 3 Recovery impact on time redundant approach

Workload	Simulation time (avg - ns)	Clock cycles	Stack usage (avg)
basic	456	18.24	8
bsort	474	18.96	8
nmea	405	16.2	7
hamming	400	16	7
Average	433.75	17.35	7.5

Table 4 Effect of caches on execution time

Workload	With cache	Without cache	Increase	Without cache + TR	Increase
basic	133825	494000	3.69	1051630	7.86
bsort	601180	5064100	8.42	10376346	17.26
nmea	286770	2057875	7.18	4216682	14.70
hamming	275291	2219775	8.06	4588508	16.67
		Average	6.84	Average	14.12

Table 5 Area overhead comparison for a microsemi ProASIC3E-1500 FPGA

Resource type	Available	Baseline	Time redundant	Increase	DMR	Increase	TMR	Increase
Core	38400	15599	30147	93.26%	41852	168.30%	52243	234.91%
RAM/FIFO	60	52	54	3.85%	60	15.38%	68	30.77%

Table 6 Power consumption comparison for a microsemi ProASIC3E-3000 FPGA

Power source	Original	TR	DMR	TMR
Dynamic Power	39.58	72.94	100.12	124.44
Static Power	37.5	37.5	37.5	37.5
Total	77.08	110.44	137.62	161.94

Table 7 Total cost analysis for Microsemi ProASIC3E-3000 FPGA

Approach	Detection rate	Recovery rate	Runtime	Overhead		Total cost
				Time	Area	
Time Redundant	0.95	0.98	0.70	2.07	1.93	0.16
DMR+CR	1.00	0.92	0.56	1.01	2.68	0.19
TMR	1.00	1.00	0.48	1.00	3.35	0.14

circuit on the FPGA. For the -1500 variant this value is 18mW for the same settings.

The time redundant approach shows the lower increase in power consumption, followed by the DMR and TMR approaches. Since the redundant processors are fed with the main clock source, their dynamic power are proportional to the occupied resources of the FPGA.

On a more practical example, a 1,000mAh/1.5V battery has a 1500mWh capacity. If we consider this capacity as the main power source and ignoring losses, we can calculate the runtime using the Eq. 1.

$$Runtime(h) = Capacity(mWh)/PowerConsumed(mW) \quad (1)$$

In this case, the theoretical runtime are:

- Original: ∼19.4 hours
- Time Redundant: ∼13.5 hours
- DMR: ∼10.9 hours
- TMR: ∼9.2 hours

5.8 Technique Remarks

Our simulation results show that the time redundant based on CR have lower overhead on area and power while sustaining reasonable numbers on the detection and recovery process. The major drawback is time overhead due to its nature of re-execution of code slices.

As a final comparison, in order to obtain a more solid number, we use an adapted formula from [6] and [14], to get a metric on the technique total cost. The total cost formula is presented in Eq. 2. The calculated total cost is dimensionless once it represents a relationship between proportions.

$$TotalCost = \frac{DetectionRate * RecoveryRate * RunTime}{TimeOverhead * AreaOverhead} \quad (2)$$

Table 7 shows the results of the total cost for the three implemented techniques for a Microsemi ProASIC3E-3000 FPGA. Values used in detection/correction rates and overheads columns are percentages (i.e., 1.00 means 100%) compared to the original implementation.

These figures show us that the time redundancy is in between the DMR and TMR technique. Although, the time redundancy approach has a better $Detection * Recovery$ factor, 0.93 against 0.92 for the DMR+CR.

Regarding the TMR technique, its major drawback is $\frac{Runtime}{Area}$ factor. Even though it has 100% detection and recovery rates, without time overhead, the spatial redundancy compromise it's application for low-power applications.

6 Conclusion

This work presented a fault-tolerant architecture using the checkpoint recovery technique for soft-core processors aimed at space-applications using FPGAs. The related work on the area shows that there is room for improvement on time-redundancy *Fault Tolerance* (FT) techniques. From our design premises, we picked the LEON3 soft-core processor as the test vehicle. The LEON3 is already used in space missions with its commercial fault-tolerant version (LEON3FT - [3]).

We named this technique as LEON3 **C**heckpoint **Re**covery **F**ault-**T**olerant (LEON3CReFT). All modifications made to the GRLIB [1] are available at https://github.com/prcvilla/leon3creft as required by the GPL-3.0.

The fault injection campaign was described in detail and the results for three different architectures were compared for a set of programs. From our experimental results, it was shown that the CR technique is a valid alternative to TMR and even DMR. This conclusion is valid also for the limited logic area and power budget, subjects of interest in satellites. The constraints are allied to comparable levels of reliability. In our approach, there is no need to perform modifications to the software source code or compiler.

As stated in Section 4.1, the cache memories have been disabled as they present a large area susceptible to SEUs and may also interfere in the processor synchronization with the checkpoints and recovery. In an actual space application, it is important to implement the proposed strategy using a processor with a cache memory. Nonetheless, considering that there is only one processor in the chosen architecture, it might have been possible to add caches and to test the proposed approach. This means that a few units in the SoC would have needed to be checkpointed as well, implying in more area overhead and testing time to make sure it continued to work. Once the work will be further improved, this is going to be considered in the future development. Additionally, the architecture will be tested in a multi-core fashion which may have new implications on the system design.

Also in a future work, the designed system must be validate with a faster fault-injection mechanism, such as FTUNSHADES [23, 36]. As for our preceding work presented in [51], we aim to perform analysis of SEU-susceptibility for combined effects of EMI and TID.

Nonetheless, we are going to improve the fault-injection campaign by using the FT-UNSHADES, which is a hardware-accelerated fault injection platform. Additionally the SET faults could be analysed in addition to the results.

Acknowledgments This work has been partly funded by the Brazilian National Council for Scientific and Technological Development (CNPq) and Instituto Federal do Rio Grande do Sul (IFRS).

References

1. Aeroflex Gaisler: GRLIB IP Library (2015a) http://www.gaisler.com/index.php/products/ipcores/soclibrary
2. Aeroflex Gaisler: LEON3 Processor (2015b) http://www.gaisler.com/index.php/products/processors/leon3
3. Aeroflex Gaisler: LEON3FT-RTAX Fault-tolerant Processor (2015c) http://www.gaisler.com/index.php/products/components/leon3ft-rtax
4. Alkhafaji FSM, Hasan WZW, Isa MM, Sulaiman N (2018) Robotic controller: ASIC versus FPGA - a review. J Comput Theor Nanosci 15(1):1–25
5. Altera Tech. (2013) White paper: Introduction to single-event upsets
6. Argyrides C, Pradhan DK, Kocak T (2011) Matrix Codes for Reliable and Cost Efficient Memory Chips, vol 19. https://doi.org/10.1109/TVLSI.2009.2036362. http://ieeexplore.ieee.org/document/5352255/
7. Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput 1(1):11–33. https://doi.org/10.1109/TDSC.2004.2. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1335465
8. Barboza SHI, Bregant M, Chambert V, Espagnon B, Herrera HDH, Mahmood SM, Moraes D, Munhoz MG, Noėl G, Pilyar a et al (2016) SAMPA chip: a new ASIC for the ALICE TPC and MCH upgrades. J Instrum 11(02):C02,088
9. Barnaby HJ (2006) Total-Ionizing-Dose Effects in Modern CMOS Technologies, vol 53. https://doi.org/10.1109/TNS.2006.885952. http://ieeexplore.ieee.org/document/4033191/
10. Battezzati N, Sterpone L, Violante M (2010) Reconfigurable field programmable gate arrays for mission-critical applications. Springer, Berlin. https://books.google.com.br/books?hl=en&lr=&id=iVScPZCgp_EC&oi=fnd&pg=PP5&dq=reconfigurable+field+programmable+gate+arrays+for+mission-critical+applications&ots=fjBZtSQupc&sig=jx3fKoLJ61msyfpPwqZJVTtT5lo
11. Baumann R (2003) Impact of Single-Event Upsets in Deep-Submicron Silicon Technology. https://doi.org/10.1557/mrs2003.38. http://journals.cambridge.org/abstract_S0883769400017516
12. Bernardeschi C, Cassano L, Domenici A (2015) SRAM-based FPGA Systems for Safety-Critical Applications: A Survey on Design Standards and Proposed Methodologies. J Comput Sci Technol 30(2):373–390. https://doi.org/10.1007/s11390-015-1530-5
13. Bouhali M, Shamani F, Dahmane ZE, Belaidi A, Nurmi J (2017) FPGA applications in unmanned aerial vehicles - a review. In: Wong S, Beck AC, Bertels K, Carro L (eds) Proceedings of Applied reconfigurable computing. Springer International Publishing, Cham, pp 217–228
14. Castro HdS, da Silveira JAN, Coelho AAP, e Silva FGA, Magalhaes PdS, de Lima OA (2016) A correction code for multiple cells upsets in memory devices for space applications. In: Proceedings of 2016 14th IEEE International New Circuits and Systems Conference (NEWCAS). IEEE, pp 1–4. https://doi.org/10.1109/NEWCAS.2016.7604783. http://ieeexplore.ieee.org/document/7604783/
15. Cetin E, Diessel O, Li T, Ambrose JA, Fisk T, Parameswaran S, Dempster AG (2016) Overview and Investigation of SEU Detection and Recovery Approaches for FPGA-Based Heterogeneous Systems. In: Proceedings of the FPGAs and Parallel Architectures for Aerospace Applications. Springer International Publishing, Cham, pp 33–46. https://doi.org/10.1007/978-3-319-14352-1_3
16. de Oliveira AB, Tambara LA, Kastensmidt FL (2017) Applying lockstep in dual-core ARM Cortex-A9 to mitigate radiation-induced soft errors. In: Proceedings of the 2017 IEEE 8th Latin American Symposium on Circuits & Systems (LASCAS). IEEE, pp 1–4. https://doi.org/10.1109/LASCAS.2017.7948063. http://ieeexplore.ieee.org/document/7948063/
17. EEJournal: The Biggest SoC/FPGAs (2017). https://www.eejournal.com/article/the-biggest-socfpgas/
18. Ferlini F, da Silva FA, Bezerra E, Lettnin DV (2012) Non-intrusive fault tolerance in soft processors through circuit duplication. In: Proceedings of 2012 13th Latin American Test Workshop (LATW). IEEE, pp 1–6. https://doi.org/10.1109/LATW.2012.6261264. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6261264
19. Friend RB, Arroyo C, Hansen J (2016) Big Missions, Small Solutions Advances and Innovation in Architecture and Technology for Small Satellites. In: Proceedings of the AIAA SPACE 2016. American Institute of Aeronautics and Astronautics, Reston, Virginia. https://doi.org/10.2514/6.2016-5229
20. Glein R (2014) BRAM radiation sensor for a Self-Adaptative SEU mitigation. In: Proceedings of the Space FPGA users workshop
21. Goloubeva O, Rebaudengo M, Reorda M, Violante M (2006) Software-implemented hardware fault tolerance. Springer, Berlin. https://books.google.com.br/books?hl=en&lr=&id=qX9GAAAAQBAJ&oi=fnd&pg=PA1&dq=software+implemented+hardware+fault-tolerance&ots=owaXCAdHzD&sig=G5Ql7eRDVfTwvyZRIrP4zYxQsw8
22. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB (2001) Mibench: a free, commercially representative embedded benchmark suite. In: Proceedings of 2001 IEEE international workshop workload characterization, WWC'01. IEEE Computer Society, Washington, pp 3–14. https://doi.org/10.1109/WWC.2001.15
23. Guzman-Miranda H, Aguirre M, Tombs J (2009) Noninvasive Fault Classification, Robustness and Recovery Time Measurement in Microprocessor-Type Architectures Subjected to Radiation-Induced Errors, vol 58. https://doi.org/10.1109/TIM.2009.2014603. http://ieeexplore.ieee.org/document/4787115/
24. Guzmán D, Rowland D, Uribe P, Nieves T (2011) A Low Power Processors for Cubesat Missions. In: Proceedings of the 8th annual cubesat developer's workshop 2011
25. Henkel J, Bauer L, Dutt N, Gupta P, Nassif S, Shafique M, Tahoori M, Wehn N (2013) Reliable On-chip Systems in the Nano-era: Lessons Learnt and Future Trends. In: Proceedings of the 50th annual design automation conference, DAC'13. ACM, New York, pp 99:1–99:10. https://doi.org/10.1145/2463209.2488857
26. Kastensmidt FL, Fonseca ECP, Vaz RG, Goncalez OL, Chipana R, Wirth GI (2011) TID in Flash-Based FPGA: Power Supply-Current Rise and Logic Function Mapping Effects in Propagation-Delay Degradation. IEEE Trans Nuclear Sci 58(4):1927–1934. https://doi.org/10.1109/TNS.2011.2128881. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm? arnumber=5752883
27. Keller AM, Wirthlin MJ (2017) Benefits of Complementary SEU Mitigation for the LEON3 Soft Processor on SRAM-Based FPGAs. IEEE Trans Nuclear Sci 64(1):519–528. https://doi.org/10.1109/TNS.2016.2635028. http://ieeexplore.ieee.org/document/7763831/
28. Kletzing CA, Kurth WS, Acuna M, MacDowall RJ, Torbert RB, Averkamp T, Bodet D, Bounds SR, Chutter M, Connerney J, Crawford D, Dolan JS, Dvorsky R, Hospodarsky GB, Howard J, Jordanova V, Johnson RA, Kirchner DL, Mokrzycki B, Needell G, Odom J, Mark D, Pfaff R, Phillips JR, Piker CW, Remington SL, Rowland D, Santolik O, Schnurr R, Sheppard D, Smith CW, Thorne RM, Tyler J (2013) The electric and magnetic field instrument suite and integrated science (EMFISIS) on RBSP. Space Sci Rev 179(1-4):127–181. https://doi.org/10.1007/s11214-013-9993-6
29. Koren I, Krishna C (2010) Fault-tolerant systems. Morgan Kaufmann, San Mateo. https://books.google.com.br/books?

hl=en&lr=&id=o_Pjbo4Wvp8C&oi=fnd&pg=PR11&dq=fault+tolerant+systems+koren&ots=RYPEQBzbyA&sig=pMKkYxL70ahe4U4U3hTKWLlrR3Y

30. Lesage L, Mejias B, Lobelle M (2011) A software based approach to eliminate all SEU effects from mission critical programs. In: Proceedings of the 2011 12th European Conference on Radiation and Its Effects on Components and Systems. IEEE, pp 467–472. https://doi.org/10.1109/RADECS.2011.6131353. http://ieeexplore.ieee.org/document/6131353/
31. Li T, Shafique M, Ambrose JA, Henkel J, Parameswaran S (2017) Fine-Grained Checkpoint Recovery for Application-Specific Instruction-Set Processors. IEEE Trans Comput 66(4):647–660. https://doi.org/10.1109/TC.2016.2606378. http://ieeexplore.ieee.org/document/7562290/
32. Lindoso A, Entrena L, Garcia-Valderas M, Parra L (2017) A Hybrid Fault-Tolerant LEON3 Soft Core Processor Implemented in Low-End SRAM FPGA. IEEE Trans Nuclear Sci 64(1):374–381. https://doi.org/10.1109/TNS.2016.2636574. http://ieeexplore.ieee.org/document/7776886/
33. Martins VMG, Villa PRC, Neto HCC, Bezerra E (2015) A TMR Strategy with Enhanced Dependability Features Based on a Partial Reconfiguration Flow. In: Proceedings of the 2015 IEEE Computer Society Annual Symposium on VLSI. IEEE, pp 161–166. https://doi.org/10.1109/ISVLSI.2015.84. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7309556http://ieeexplore.ieee.org/document/7309556/
34. Microsemi Inc. (2017) ProASIC3 FPGA. https://www.microsemi.com/products/fpga-soc/fpga/proasic3-overview
35. Microsemi Inc. (2018) Power Estimators and Calculators. https://www.microsemi.com/products/fpga-soc/design-resources/power-calculator
36. Mogollon J, Guzman-Miranda H, Napoles J, Barrientos J, Aguirre M (2011) FTUNSHADES2: A novel platform for early evaluation of robustness against SEE. In: Proceedings of the 2011 12th European Conference on Radiation and Its Effects on Components and Systems. IEEE, pp 169–174. https://doi.org/10.1109/RADECS.2011.6131392. http://ieeexplore.ieee.org/document/6131392/
37. Norton CD, Werne TA, Pingree PJ, Geier S (2009) An evaluation of the Xilinx Virtex-4 FPGA for on-board processing in an advanced imaging system. In: Proceedings of the 2009 IEEE Aerospace conference. IEEE, pp 1–9. https://doi.org/10.1109/AERO.2009.4839460. http://ieeexplore.ieee.org/abstract/document/4839460/
38. Petkov M (2003) The effects of space environments on electronic components. In: JPL Technical Report Server 1992+. https://trs.jpl.nasa.gov/handle/2014/7193
39. Ragel R, Parameswaran S (2012) Reli: Hardware/software Checkpoint and Recovery scheme for embedded processors. In: Proceedings of the 2012 Design, Automation & Test in Europe Conference &Exhibition (DATE). IEEE, pp 875–880. https://doi.org/10.1109/DATE.2012.6176621. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6176621
40. Reorda M, Violante M, Meinhardt C, Reis R (2009) A low-cost SEE mitigation solution for soft-processors embedded in Systems on Programmable Chips. In: 2009 Design, Automation & Test in Europe Conference & Exhibition, pp 352–357. https://doi.org/10.1109/DATE.2009.5090687. http://dl.acm.org/citation.cfm?id=1874620.1874704
41. Rodriguez-Andina JJ, Valdes-Pena MD, Moure MJ (2015) Advanced Features and Industrial Applications of FPGAs—A Review. IEEE Trans Ind Inf 11(4):853–864. https://doi.org/10.1109/TII.2015.2431223. http://ieeexplore.ieee.org/document/7104117/
42. Sabena D, Sterpone L, Scholzel M, Koal T, Vierhaus HT, Wong S, Glein R, Rittner F, Stender C, Porrmann M, Hagemeyer J (2014) Reconfigurable high performance architectures: How much are they ready for safety-critical applications? In: Proceedings of the 2014 19th IEEE European Test Symposium (ETS). IEEE, pp 1–8. https://doi.org/10.1109/ETS.2014.6847820. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6847820
43. Siegle F, Vladimirova T, Ilstad J, Emam O (2015) Mitigation of radiation effects in SRAM-based FPGAs for space applications. ACM Comput Surv 47(2):34. https://doi.org/10.1145/2671181. Article 37
44. Siewiorek D, Swarz R (2017) Reliable computer systems: Design and evaluatuion. Digital Press
45. Li T, Ambrose JA, Parameswaran S (2016) ReCoRD: Reducing Register Traffic for Checkpointing in Embedded Processors. In: Proceedings of the 2016 Conference on Design, Automation & Test in Europe, DATE'16. EDA Consortium, San Jose, pp 582–587. http://dl.acm.org/citation.cfm?id=2971808.2971945, http:// ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7459379
46. Tang HH, Olsson N (2003) Single-Event Upsets in Microelectronics. MRS Bullet 28(02):107–110. http://journals.cambridge.org/abstract_S0883769400017498
47. Torrens G (2017) FPGA-SRAM Soft Error Radiation Hardening. In: Field - Programmable Gate Array. InTech. https://doi.org/10.5772/66195. http://www.intechopen.com/books/field-programmable-gate-array/fpga-sram-soft-error-radiation-hardening
48. Travessini R, Villa PRC, Vargas F, Bezerra E (2018) Processor core profiling for SEU effect analysis. In: Proceedings of the 2018 IEEE 19th Latin-American Test Symposium (LATS). IEEE, pp 1–6. https://doi.org/10.1109/LATW.2018.8347235. https://ieeexplore.ieee.org/document/8347235/
49. U.S. State Department (2018) Directorate of Defense Trade Controls. http://pmddtc.state.gov/index.html
50. Velazco R., Fouillat P., Reis R. (eds) (2007) Radiation Effects on Embedded Systems. Springer, Netherlands. https://doi.org/10.1007/978-1-4020-5646-8
51. Villa P, Bezerra E, Goerl R, Poehls L, Vargas F, Medina N, Added N, De Aguiar V, MacChione E, Aguirre F, Da Silveira M (2017a) Analysis of COTS FPGA SEU-sensitivity to combined effects of conducted-EMI and TID. In: Proceedings of the 2017 11th international workshop on the electromagnetic compatibility of integrated circuits, EMCCompo 2017. https://doi.org/10.1109/EMCCompo.2017.7998076
52. Villa PRC, Goerl RC, Vargas F, Poehls LB, Medina NH, Added N, de Aguiar VAP, Macchione ELA, Aguirre F, da Silveira MAG, Bezerra E Analysis of single-event upsets in a Microsemi ProAsic3E FPGA. In: Proceedings of the 2017 18th IEEE Latin American Test Symposium (LATS). (2017b), pp 1–4. IEEE. https://doi.org/10.1109/LATW.2017.7906772. http://ieeexplore.ieee.org/document/7906772/
53. Villa PRC, Travessini R, Vargas F, Bezerra E (2018) Processor checkpoint recovery for transient faults in critical applications. In: proceedings of the 2018 IEEE 19th Latin-American Test Symposium (LATS). IEEE, pp 1–6. https://doi.org/10.1109/LATW.2018.8349674. https://ieeexplore.ieee.org/document/8349674/
54. Violante M, Meinhardt C, Reis R, Reorda MS (2011) A Low-Cost Solution for Deploying Processor Cores in Harsh Environments, vol 58. https://doi.org/10.1109/TIE.2011.2134054. http://ieeexplore.ieee.org/document/5740344/
55. Wilson DS (2011) Cubesat Flight Software Development. In: Proceedings of the 2011 workshop on spacecraft flight software (FSW11). Baltimore
56. Ziade H, Ayoubi RA, Velazco R et al (2004) A survey on fault injection techniques. Int Arab J Inf Technol 1(2):171–186

Paulo R. C. Villa graduated in Computer Engineering from Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) in 2009. MSc (2013) and PhD (2018) in Electric Engineering from Federal University of Santa Catarina (UFSC). His research is aimed at fault tolerance techniques for embedded processors, he is a member of Embedded System Group at UFSC and Associate Professor at Federal Institute of Rio Grande do Sul.

Rodrigo Travessini is an Electronic Engineer graduated by the Federal University of Santa Catarina. He has a master degree in the same institution. His research is aimed at fault tolerance techniques for embedded processors and is a member of Embedded System Group at UFSC.

Roger C. Goerl is a computer engineer graduated by PUCRS. He has a Master degree on Electric Engineering from PUCRS. Currently he is pursuing computer science PhD in the same institution under advisement of Prof. César Marcon. He is a member of Laboratory of excellence in electronics, automation and embedded systems of high reliability at PUCRS.

Fabian L. Vargas is graduated in Electrical Engineering from the Pontifícia Universidade Católica do Rio Grande do Sul (1988), MSc. in Computer Science from the Universidade Federal do Rio Grande do Sul (1991) and PhD. in Microelectronics from the Institut National Polytechnique de Grenoble (1995). F. Vargas has experience in Computer Science, focusing on Computer Systems Architecture, acting on the following topics: fault-tolerant systems design for critical applications, design of on-chip sensors for reliability insurance, design for electromagnetic/radiation tolerance and on-line testing. Prof. Vargas is an IEEE Senior Member and a Golden Core Member of the IEEE Computer Society since 2003.

Eduardo A. Bezerra is a Researcher and Lecturer of Computer Engineering at Universidade Federal de Santa Catarina (UFSC), where he is with the Department of Electrical Engineering since 2010. He received his Ph.D. in Computer Engineering from the University of Sussex (Space Science Centre), England, UK, in 2002. From 2016 to 2017, he took a sabbatical leave to develop research activities at the Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM), Université de Montpellier, France, where he is now a "long term invited professor" (Invité longue durée Professeur). He is the author and co-author of papers published covering a broad range of scientific topics within the disciplines of Computer Engineering. His research interests are in the areas of embedded systems for space applications, Cubesats, computer architecture, reconfigurable systems (FPGAs), software & hardware testing, fault tolerance and microprocessor applications.