Advanced Master in Space Systems Engineering

Master's Thesis

Performance implications of software fault mitigation techniques

Author :

Mr. Maxence Goubaud

Supervisors :

Dr. Arnaud Dion

Mrs. Fiona Zoe Lund

October 20, 2025

Acknowledgements

I would especially like to thank my parents for everything, for their unwavering support. I owe all my success to them, and it is theirs above all else.

I would like to thank my supervisors Fiona Lund and Arnaud Dion for their advice and suggestions throughout my thesis. I would also like to express my gratitude again to Fiona Lund, with Jan Sommer and Daniel Lüdtke, for giving me the opportunity to work on this project at the DLR. Of course, I would also like to thank all my colleagues who were always there when I needed help.

Special thanks to Mathilde, who gave me all the emotional support I needed during the last six months.

Thanks to Stéphanie Lizy-Destrez for giving me the opportunity to study in her programme at ISAE SUPAERO.

Thanks to Kevin Kollenda for answering all my countless questions, I would not have had all these results without this help.

Thanks to everyone I have not mentioned but who has also been part of my journey and believe in my capabilities: my family and my friends.

Contents

List of Figures

List of Tables

Acronyms

ABFT *Algorithm-Based Fault Tolerance*. 13, 14

API *Application Programming Interface*. 19

CFC *Control Flow Checking*. 13

CFG *Control Flow Graph*. 14

CME *Coronal Mass Ejection*. 5

COTS *Commercial Off-The-Shelf*. 1, 2, 13–15, 52

CPU *Central Processing Unit*. 18, 30, 32

DD *Displacement Damage*. 8

DLR *Deutsches Zentrum für Luft- und Raumfahrt*. 13

DMT *Duplex Multiplexed in Time*. 14

DUP *Duplication and Checksum*. 26, 34, 37–39, 41, 52

DWC *Duplication with Comparison*. iv, 22, 23, 34, 39, 41

EDAC *Error Detection and Correction*. vi, 2, 12, 14, 21, 23–25, 34, 38, 39, 41, 42

FI *Fault-Injection*. v, 28, 30, 31

FPGA *Field Programmable Gate Array*. 13, 18

LEO *Low Earth Orbit*. 10

MBU *Multiple Bit Upset*. 10

MOSFET *Metal-Oxide-Semiconductor Field-Effect Transistor*. 9

RESIL *Resilient Evaluation Framework for Tolerance*. iv, 19, 42, 52

RTOS *Real-Time Operating System*. 43

SAA *South Atlantic Anomaly*. 6, 10

SBU *Single Bit Upset*. 10

ScOSA *Scalable On-board Computing for Space Avionics*. 13, 18

SDC *Silent Data Corruption*. 30, 32, 37–41, 52

SDSC *Selective Duplication and Selective Comparison*. 14

SEB *Single Event Burnout*. 9

SEC-DED *Single Error Correction - Double Error Detection*. vi, 24, 26

SEE *Single Event Effects*. 8, 9, 12, 52

SEFI *Single Event Functional Interrupt*. 10

SEGR *Single Event Gate Rupture*. 9

SEL *Single Event Latchup*. 9

SER *Soft Error Rate*. 10, 13

SET *Single Event Transient*. 10

SETA *Software-only Error-detection Technique using Assertions*. 14

SEU *Single Event Upset*. iv, 9–12

SoC *System-on-Chip*. 18

SOI *Silicon-on-Insulator*. 12

SOS *Silicon-on-Sapphire*. 12

TID *Total Ionizing Dose*. 8

TMR *Triple Modular Redundancy*. iv, 2, 12, 14, 22, 26, 34, 37–42, 52

UML *Unified Modeling Language*. iv, 20, 21

Chapter 1

Introduction

1.1 Motivation

With the arrival of the New Space – a movement characterized by the privatization and commercialization of space sector, driven by lower costs and higher accessibility – has transformed the industry and become increasingly active, as shown in Figure 1.1. With this shift, the development of space equipment systems using *Commercial Off-The-Shelf* (COTS) components has increased. Modern space missions, requiring a lot of resources, are now driven by the need for cost efficiency, compact size and high processing performance. Consequently, COTS components are used due to their ability to meet these needs.

However, this trend has also introduced new challenges in ensuring the reliability of these systems. That is especially important when COTS devices instead of radiation-hard devices are used, as the latter are specifically designed to withstand the harsh conditions of space. One of the primary concerns is the vulnerability of COTS components to space environment. Radiation is one possible contributing factor, among others, and it can cause *Single Event Effects* (SEE). A typical example is a *bit-flip*, where a bit unintentionally changes its value, potentially causing system failure.

To mitigate these effects, software-based fault tolerance techniques have emerged as a

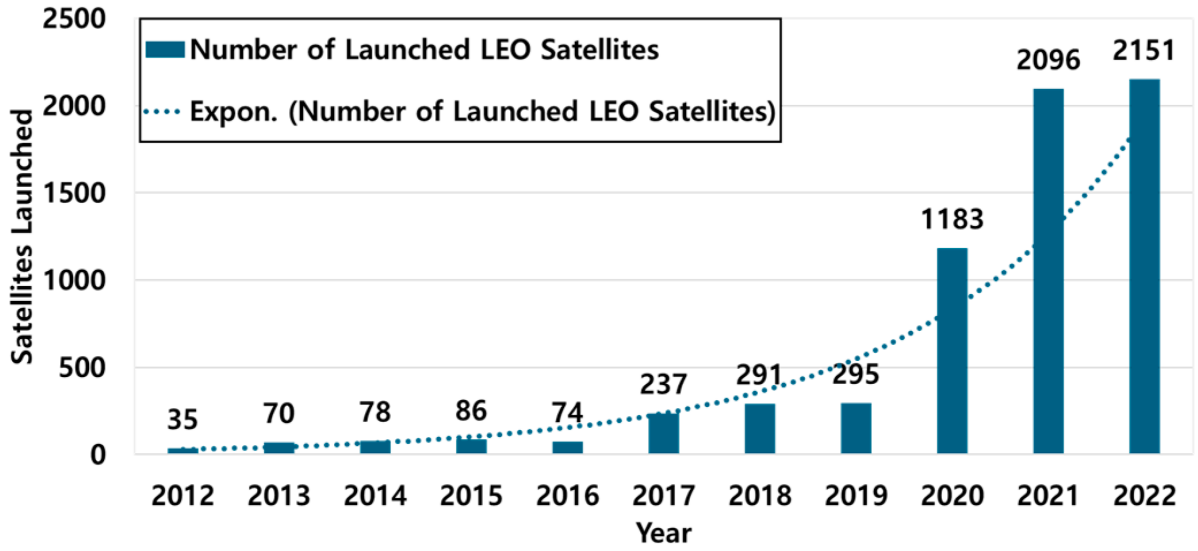


Figure 1.1: Number of launched LEO satellites. [10]

key strategy for improving the reliability of embedded systems. Techniques such as *Error Detection and Correction* (EDAC), *Triple Modular Redundancy* (TMR) and checkpointing are well-known fault tolerance methods. However, the implementation of these techniques comes with performance and resource overheads, which can impact the overall efficiency of the system.

The growing demands of space exploration and the increasing reliance on COTS components in space equipment systems necessitate a deeper understanding of the performance implications of software fault tolerance techniques. The development of high-performance computing systems for space applications requires a delicate balance between reliability and performance. The implementation of fault tolerance techniques must be evaluated to ensure that they do not compromise the overall efficiency of the system.

Thus, this thesis addresses the following research question:

"How do software-based fault tolerance techniques balance performance overhead and robustness against radiation-induced faults"

1.2 Contribution

The present work aims to develop a framework designed to evaluate and compare software-based fault tolerance techniques for embedded systems.

The specific objectives are to implement and benchmark fault mitigation strategies to measure the performance overhead, and then conduct a fault-injection campaign to simulate radiation-induced errors to evaluate the robustness of each technique under identical workload. This study should propose a final trade-off between implemented strategies, for embedded applications.

The remainder of this thesis is structured as follows. Chapter Two reviews the sources of radiation-induced faults in space environments, their effects on electronic systems, and existing hardware/software mitigation techniques. Chapter Three details the design of the framework, the selected fault tolerance mechanisms, and the experimental setup for performance and robustness evaluation. Chapter Four presents benchmarking results, fault-injection outcomes, and a comparative analysis of techniques. Chapter Five summarizes key findings, discusses limitations, and proposes future research directions.

Chapter 2

Background and Related Work

2.1 Faults in Computing Systems

2.1.1 Sources of errors

When we hear about radiation, we immediately think of the harmful effects on the human body. Similarly, space objects operate in a very hostile environment. After the difficult launch phase, they are exposed to the vacuum of space, to sharp variations in temperature, and also to flows of particles and harmful radiation. According to [18], 45% of spacecraft anomalies due to the space environment come from radiation. It is therefore essential to study this radiative environment to prevent risks and understand what surrounds us.

The components onboard space missions are subjected to a higher level of radiation than on Earth. The Earth's surface is protected from radiation by its magnetosphere, which traps charged particles. Radiation affects the efficiency and lifespan of instruments, as well as the health of astronauts in space. This radiation can be grouped into three distinct categories: the Sun, the Van Allen belts and cosmic rays.

The main source of radiation to which we are constantly exposed, even on Earth, is the Sun. Located 150×10^6 km from Earth and being a yellow dwarf star, it is composed of

75% hydrogen and 25% helium. Its core reaches 15×10^6 K, which triggers nuclear fusion reactions, releasing so-called solar energy that is transmitted by solar radiation [32]. By observing the intensity of the Sun's magnetic field and the spots on its surface, we can see that solar activity varies by reproducing identical phenomena over a period equivalent to 11 Earth years [42]. This is known as the solar cycle, and was first observed by the German astronomer Heinrich Schwabe around 1843. However, these cycles are not of equal intensity, as showed with the cycles since 1985 in Figure 2.1. Solar minimum and maximum periods are defined by the observation of various phenomena resulting from solar activity. These active zones, which include solar flares, solar winds and *Coronal Mass Ejection* (CME), have a direct impact on the Earth's environment, as well as on the equipment used on space missions [33].

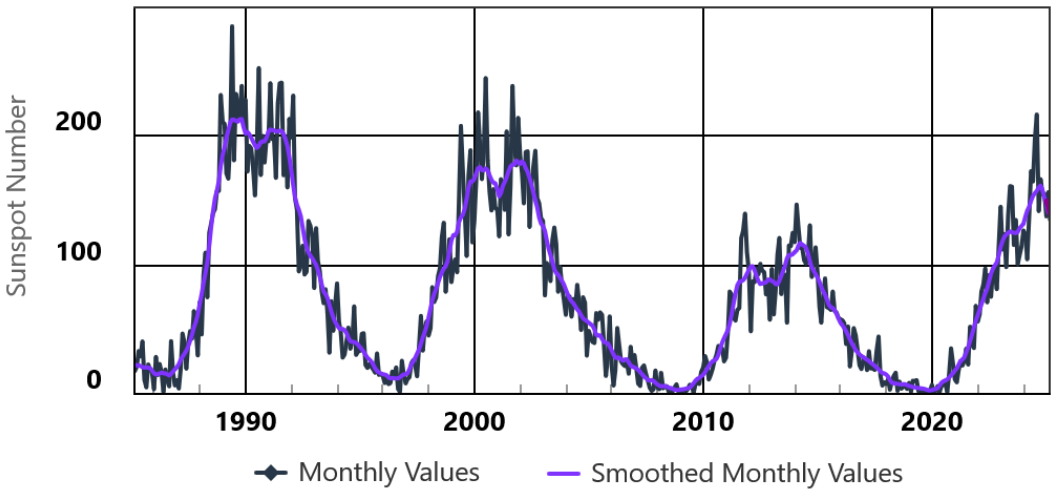


Figure 2.1: Solar cycle sunspot number progression. [10]

The Van Allen belts, first describe in 1958 by the American physicist James Van Allen, are a ring zone of the Earth's magnetosphere that surrounds the magnetic equator and contains a high density of charged particles (protons, electrons) [52]. These belts are formed by the interaction of solar radiation from solar activity with the Earth's magnetosphere, which acts like a radiative shield trapping the charged particles, thus creating the Van Allen belts. It is the meeting of these particles with the Earth's upper atmosphere that

is responsible for the polar lights in regions close to the Earth's magnetic poles. These particles travel along the lines of the Earth's magnetic field between the two magnetic poles, making it possible to distinguish two main belts as shown in Figure 2.2. The inner belt is made up of high-energy protons, located less than two Earth radii away, and the outer belt is made up of high-energy electrons, located between 3.5 and 8 Earth radii away. Being symmetrical to the Earth's magnetic axis, which is itself inclined at around 11° to the Earth's axis of rotation, the result is a gap between the inner belt and the Earth's surface, located above the *South Atlantic Anomaly* (SAA) [40].

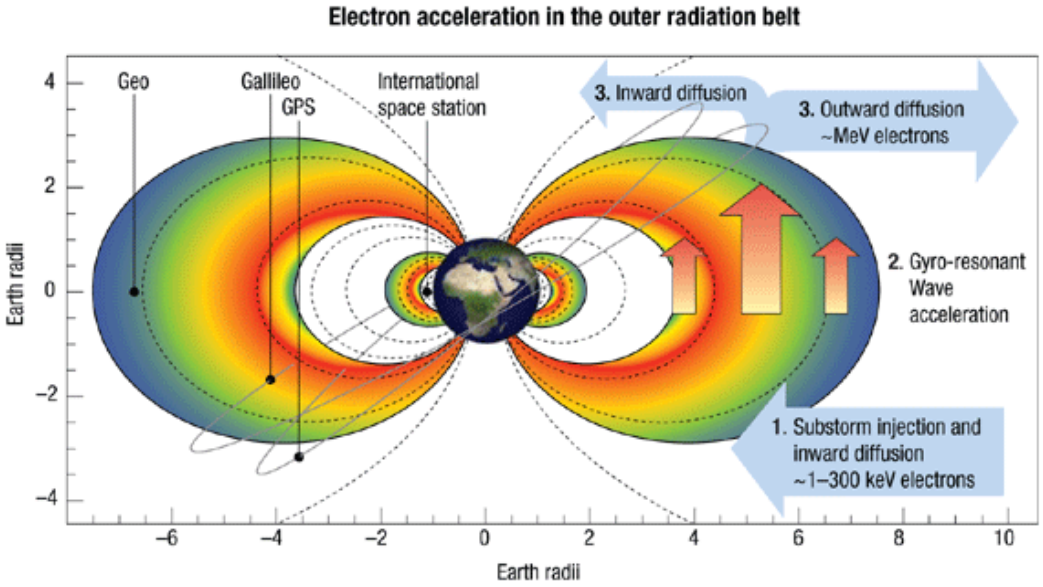


Figure 2.2: Illustration of the Van Allen radiation belts. [22]

Another source of radiation that can impact space equipment is cosmic rays from the far reaches of the universe. These cosmic rays are very high-energy particle streams that can reach up to 10 GeV. They are made up of 87% high-energy protons, 9% alpha particles (helium nuclei) and 1% charged particles (electrons) and neutral particles (gamma rays, neutrons) [9]. It is likely that this cosmic radiation comes from the remnants of supernovas, powerful explosions that occur during the final stages of the life of massive stars, which then become black holes or are destroyed [8]. This radiation is extremely fast and almost

impossible to block with current shielding. Nevertheless, the magnetic fields of the Sun and the Earth contribute to our protection by deflecting the majority of particles from these cosmic rays [34].

2.1.2 Effects on electronic components

The various types of radiation and particle flows can have harmful effects on electronic equipment. Studying the effects of radiation is therefore essential for the future of space exploration, as they can lead to both immediate and long-term damage in critical systems.

Radiation damage in electronic materials primarily arises from two broad categories of interactions: ionizing effects and transient failures (see Figure 2.3). These interactions occur when high-energy particles collide with the molecules and atoms that make up the material, causing damage [3].

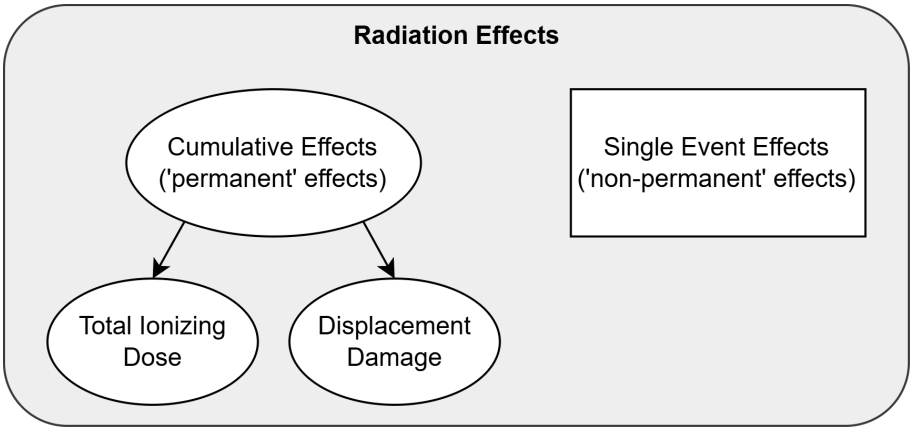


Figure 2.3: Radiation Effects.

Energetic charged particles deposit energy in materials through two main processes. First, they can interact directly with the electrons of target atoms, displacing them from their orbits and releasing them into the surrounding material. This process, known as ionization, generates electron-hole pairs in solid-state materials. The subsequent behavior of these pairs depends on the electrical properties of the affected device. Second, ionization can also occur indirectly when particles rapidly decelerate upon entering dense matter.

This sudden braking converts some of their kinetic energy into high-energy photons, a phenomenon called *Bremsstrahlung*, or "braking radiation". These photons can then travel deeply into materials, ionizing even well-shielded targets along their path [18][30].

Depending on the type of particles encountered, these interactions contribute to cumulative effects, including both *Total Ionizing Dose* (TID) and *Displacement Damage* (DD) due to long-term exposure to radiation, as well as *Single Event Effects* (SEE)s, transient effects that occur randomly, leading the electronic devices to fail [3].

Unlike cumulative effects, which causes parameters to drift, SEEs have a direct impact on the component. They are malfunctions of electronic components caused mainly by interactions with heavy ions and protons [48]. SEEs fall into two categories: *non-destructive* SEEs that cause temporary failure and *destructive* SEEs that result in permanent damage and persistent loss of the device. The exact type of SEE depends on the location, the process technology and the amount of charge injected, but it mainly occurs when ionising particles strike a semiconductor node with sufficient energy to trigger erroneous behaviour in the p-n junction [37]. The two categories of SEE are discussed in more detail below and presented in Figure 2.4.

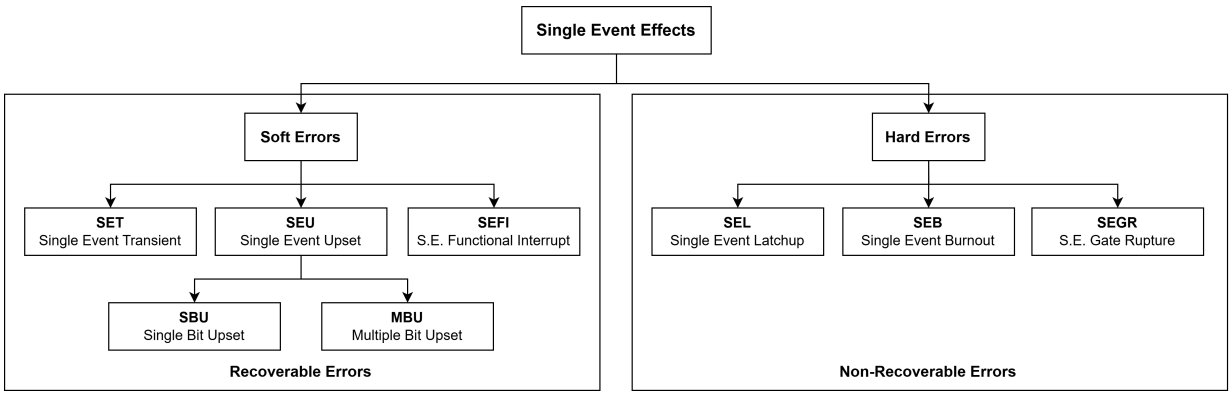


Figure 2.4: Single Event Effects classification. [37]

Destructive SEEs are a critical concern because they result in irreversible damage, potentially leading to loss of important data and complete system failure [4]. The most common hard error types are presented in Table 2.1.

Table 2.1: Hard error types.

Type	Comment
SEB	A rapid increase in current is generated when the excited particle triggers a pair of parasitic transistors, resulting in a short circuit
SEGR	A damaging ionisation column between the gate oxide and the drain of a MOSFET is generated, resulting in the disability to control the current flow
SEL	High current states and Joule effects may be generated when the particle strike enables a parasitic transistor, resulting in device's destruction

Susceptibility to hard errors is directly influenced by silicon level device design [24], hence it is beyond the scope of this work.

On the other hand, non-destructive SEEs are transient and lead to a temporary, random failure that does not damage the hardware and can be corrected [4]. However, soft errors – though non-destructive – pose their own insidious risks. Unlike destructive SEEs, soft errors often go unnoticed, silently altering data values without immediate or obvious consequences. For example, a bit flip in a pixel value of an image might be harmless, but an undetected change in the sign bit of a navigation coordinate could have catastrophic results (e.g., misrouting a spacecraft or corrupting mission-critical calculations). What makes soft errors particularly dangerous is their stealthy nature; they may evade detection until their effects manifest in unexpected system behavior. The most common soft error types are presented in Table 2.2

A soft error happens when a charged particle hits a sensitive part of an electronic chip, creating a brief electrical disturbance that can flip stored data. Figure 2.5 illustrates the *Single Event Upset* (SEU) process in a silicon junction: (2.5a) an ion strike initiates the event, followed by (2.5b) drift collection and (2.5c) diffusion collection of the generated charge. The resulting transient current in the junction is shown in (2.5d) as a function of time. SEUs are a major cause of reduced reliability and stability in electronic devices operating in space environments. For example, the communication subsystem of a satellite

Table 2.2: Soft error types.

Type	Comment
SEFI	A bit fault in the configuration memory leading to a disruption in functionality of the logic circuit
SEU	A bit fault in a logic cell (register, memory devices) causing corruption to the instruction or data of the computation payload. Depending on the number of the affected cells, they are called SBU or MBU
SET	Momentary voltage spike with sufficient duration to propagate towards other analog or digital circuits

may experience data transmission problems due to the effects of an SEU. In the most severe cases, a malfunction of the attitude control subsystem – leading to failure in orbit correction – may result in mission failure [29]. This risk is especially pronounced over the SAA, particularly during periods of high solar activity. For example, UOSAT-2, one of the early *Low Earth Orbit* (LEO) microsatellites developed by the University of Surrey in the 1980s, experienced a high concentration of memory upsets in this region, as illustrated in Figure 2.6 [1]. UOSAT-2 demonstrated how vulnerable systems can be to radiation-induced soft errors.

These soft errors have since have become one of the most important and challenging failure mechanisms in modern electronic devices. The *Soft Error Rate* (SER) of commercial chips can be predicted through simulation or analytical model during the mission analysis phase [55]. Thus, it is essential to address the challenges associated with SEU, which is why this thesis focuses on it.

Moreover, most of the microprocessor performance gain over the last decades is due to smaller dimensions and low voltage transistors (i.e., Moore's Law). However, the same technology that made possible all this progress also lowered the transistor reliability by reducing threshold voltage and tightening the noise margins [6], hence making them more susceptible to faults caused by radiation effects. These faults could cause the processor

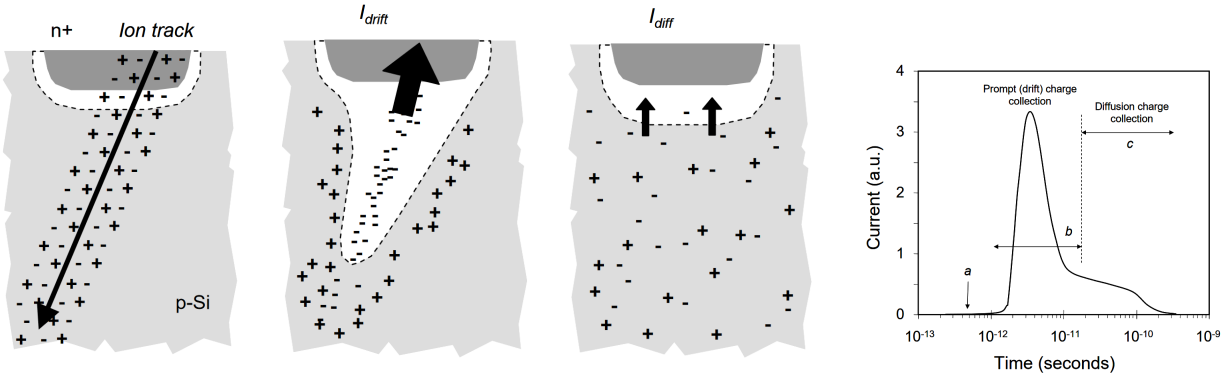


Figure 2.5: Process of a SEU occurrence. [7]

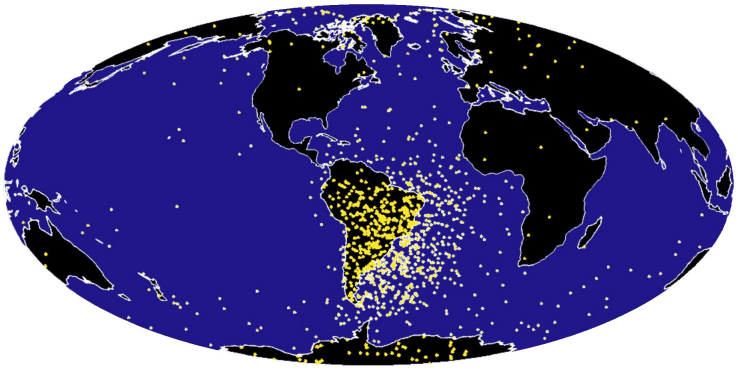


Figure 2.6: Locations of Single-Event Memory Upsets suffered by the UOSAT-2. [36]

to run a program wrongly, produce erroneous data without detection, crash the program entirely, or even to get stuck in a loop and never reach completion. In this context, mitigation fault tolerance techniques can be used [14].

2.2 Mitigation techniques

2.2.1 Hardware-based fault tolerance techniques

Hardware-based fault tolerance techniques rely on device shielding and replicating or adding hardware modules [53].

First, devices can be fabricated in a number of different ways and this can have a large impact on SEE effects. The most basic method for hardening against SEU is to reduce charge collection at sensitive nodes. This can be achieved in memory cells by introducing extra layers to limit substrate charge collection. Even the simple use of an epitaxial substrate instead of a bulk substrate provides some level of reduced charge collection [16].

Another effective technique is using special processing technologies such as *Silicon-on-Sapphire* (SOS) or *Silicon-on-Insulator* (SOI), which cut off the charge collection length associated with the charged-particle track. The former results in two separate p- and n-doped islands on an insulating sapphire substrate, while the latter uses special processing to grow an isolated silicon dioxide insulating layer on a bulk silicon substrate. However, both processes are costly and used only in limited volumes. It is also possible to introduce special circuit techniques, such as decoupling resistors, in order to increase the charge the node must collect to cause an upset. As a penalty, this approach is less effective especially because of the decreased circuit speed [25].

System-level hardening approaches include the use of additional hardware modules. It can involve using EDAC circuitry to continuously monitor and fix memory errors. This method requires storing additional bits alongside the main data, enabling the system to reconstruct accurate information if errors occur [12].

Another widely adopted technique is TMR, which can be applied at different levels. It works by creating three identical copies of the protected module, processing in parallel across all three, and using a voting mechanism to determine the correct output based on majority. This means that even if one module fails, the remaining two can still deliver the correct result, effectively masking the fault.

Other modules, such as watchdog timers or lockstep, are frequently used to detect errors in control systems. Lockstep involves running identical processors in parallel, executing the same application simultaneously. These processors are synchronized from the start and receive identical inputs, so their states should match perfectly at every clock cycle unless an error occurs. The system periodically checks the processors' states at designated

verification points and can resume from the last valid verification point [2].

For *Field Programmable Gate Array* (FPGA)s, scrubbing is another widely used technique to mitigate radiation-induced errors. Scrubbing involves continuously or periodically reading the FPGA configuration memory, detecting and correcting bit-flips, and rewriting the correct configuration to maintain operation [35].

As mentioned in [11], the SER robustness can be achieved from several tens of percentages up to an order level, using design, process and material improvements. Despite the high reliability they provide, they come at the cost of increased circuit area and power consumption, as well as higher design complexity and manufacturing expenses [14]. The necessity for high-performance, low-cost computer systems in space has driven research into novel fault tolerance strategies using COTS components, primarily for missions requiring a lot of resources, such as autonomous rover. The *Scalable On-board Computing for Space Avionics* (ScOSA) Flight Experiment from the *Deutsches Zentrum für Luft- und Raumfahrt* (DLR) is a good example of how COTS multicore processors are now being used in space. This project aims to execute demanding payload tasks with software-managed reliability strategies such as task distribution and reconfigurability [31]. In the case of COTS components, hardening techniques based on software modifications become very attractive.

2.2.2 Software approaches

Software-based fault tolerant techniques are popular and well known to protect systems against bit-flips by including extra code to the original program. These techniques can be used to protect the control flow or the data flow, and they are presented in [19, 47]. For instance, representative examples of these techniques are duplication, checkpointing, *Algorithm-Based Fault Tolerance* (ABFT) or *Control Flow Checking* (CFC) [19]. It is possible to apply software-based fault-tolerant solutions with a minimal increase in development time.

In [17], the soft error vulnerability was studied for several workloads, including their

protected version using software-based built-in TMR. The analysis employed both software-based and hardware-based fault injection. The study demonstrates that the outcomes of the two fault injection campaigns are independent, showing no statistical relationship between their results. Although the protected versions provide an improvement, the execution time workloads are considerably higher. In [39], a lightweight fault-tolerant architecture, *Duplex Multiplexed in Time* (DMT), was developed to complement more conventional solutions such as ABFT, based on two successive executions of the task code on the physical channel, based on the time redundancy principle. They also use hardware support in their proposed solution, to achieve fail-safe performance. In [15], they introduced the *Software-only Error-detection Technique using Assertions* (SETA) control-flow technique using assertions. The code is divided into basic blocks grouped into networks, with checkers to verify the block's signature. This technique aims to keep the same fault coverage compared to other techniques but reducing the overheads in execution time and code size. In [50], they focused on the *Selective Duplication and Selective Comparison* (SDSC) data-flow technique using duplication and comparison, based on *Control Flow Graph* (CFG) to identify vulnerabilities, defining the longest path as the one most susceptible to errors. In [46], they evaluated a software-implemented EDAC to protect the main memory of the COTS board, with a periodic scrubbing on the memory. Their implementation achieve an error correction rate of 5.5 upsets/MB-day. We present a summary of the described techniques in Table 2.3.

As mentioned in [13], there are also hybrid techniques that provide a balanced solution by integrating the strengths of hardware and software methods while considering the system's resource limitations, but they are beyond the scope of this work and will not be discussed in detail.

Although software-based techniques enhance the system's ability to withstand faults, their implementation introduces significant overheads. The execution of additional code requires extra memory and execution time. These techniques rely on software redundancy at various levels of granularity, each offering a distinct degree of protection but also incurring associated performance and resource costs [5].

Table 2.3: Software-based fault tolerance techniques review

Technique	Characteristics	Fault Coverage
TMR	Majority voting on replicates	Masks errors in one erroneous replicate (among three)
EDAC	Fault checking on encoded data	Detects and corrects single- or multi-bit errors
CFC	Inserts assertions at control points	Detects control-flow deviations
ABFT	Fault checking at algorithm level	Detects algorithmic or data operation errors
Checkpointing	Saves program's state and recovery	Recovers from detected faults (e.g., crashes, hangs)

Furthermore, excessive fault mitigation can paradoxically increase vulnerability: adding more protection code expands the memory footprint, thereby enlarging the attack surface for potential faults [51]. This underscores the need for a balanced approach, where mitigation strategies are tailored to minimize both overhead and susceptibility.

However, most existing studies prioritize fault coverage over performance impact, particularly on modern COTS processors. This imbalance leaves a gap in understanding the trade-off between reliability and performance constraints. Without evaluations of these different mechanisms, engineers must make uninformed decisions during the flight software development phase.

Chapter 3

Methodology and Implementation

3.1 Methodology workflow

We have pointed out that it is essential to evaluate and compare mechanisms in a rigorous way. The idea is to design a framework in C++ to measure, analyse and compare different fault tolerance mechanisms, isolating their impact on system behaviour. By doing so, we can objectively assess the trade-off between reliability and performance, and select the most appropriate mechanism for a given application.

The development of the framework implementation focused on fulfilling the following requirements:

- The framework should allow easy integration of new fault tolerance mechanisms without requiring core modifications.
- Common components should be integrated as much as possible to reduce redundancy and increase efficiency.
- Key performance indicators will be captured to accurately quantify the overhead introduced by each selected technique.
- Test scenarios will be established to ensure fair comparison between mechanisms;

- Scripted execution of benchmarks, data collection, and report generation will be enabled.

The next step involves selecting a set of fault mitigation algorithms that are relevant to embedded architectures and have been proven effective in fault tolerance applications. These selected algorithms will serve as the basis for further evaluation.

Subsequent to algorithm selection, performance metrics will be defined to assess the overhead introduced by each technique. Simple benchmarks will be used to provide an assessment of the performance impact for each selected mechanism. A preliminary analysis will enable us to identify potential areas for optimization and refinement.

Finally, fault injection will be performed using a tool that allows automated injection of simulated faults. The resulting data will be analyzed to compare the performance and robustness trade-offs of each selected algorithm. We aim to evaluate and compare the effectiveness of the selected fault tolerance mechanisms.

Through this structured approach outlined in Figure 3.1, we aim to develop a comprehensive framework that enables rigorous evaluation and comparison of different fault tolerance mechanisms, ultimately informing decision-making regarding system design and optimization.

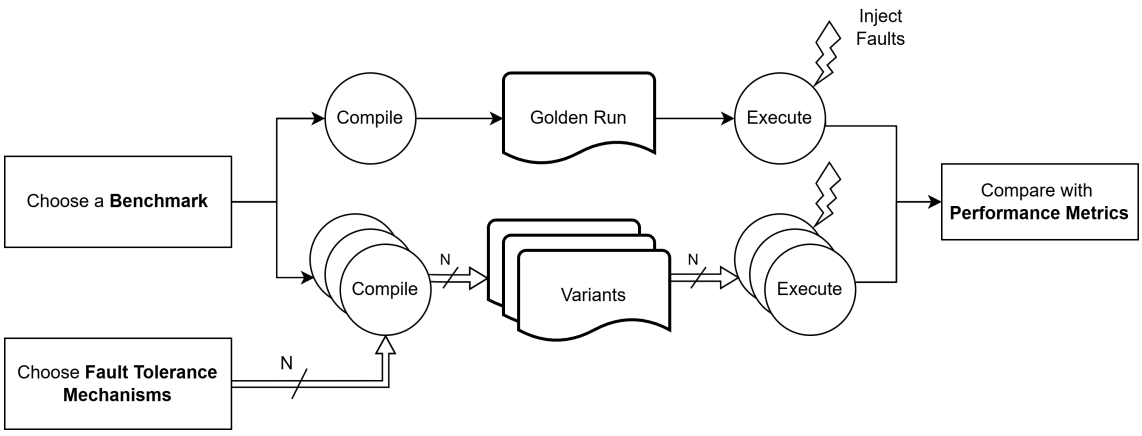


Figure 3.1: Methodology Workflow Diagram.

3.2 Hardware description

System-on-Chip (SoC) technology integrates multiple components, including the *Central Processing Unit* (CPU), memory, and peripherals onto a single chip via an interconnect or bus. This design approach offers several advantages over traditional computing architectures, where separate boards are used to connect individual components. SoCs own significant benefits, such as reduced size, improved reliability, lower power consumption, and increased efficiency, making them an ideal choice for embedded systems.

The Xilinx Zynq7000 is a notable example of this technology, featuring a combination of an ARM Cortex A9 MPCore CPU and a Xilinx FPGA on a single SoC [56], as illustrated in Figure 3.2. This is a flight-equivalent processor that is also used as a high-performance node in the ScOSA project, mentioned before in Section 2.2.1. In our case, implementation will take place both on the machine used to develop the framework (team server with x86-64 architecture) and on the ZedBoard platform, which features the Xilinx Zynq-7020.

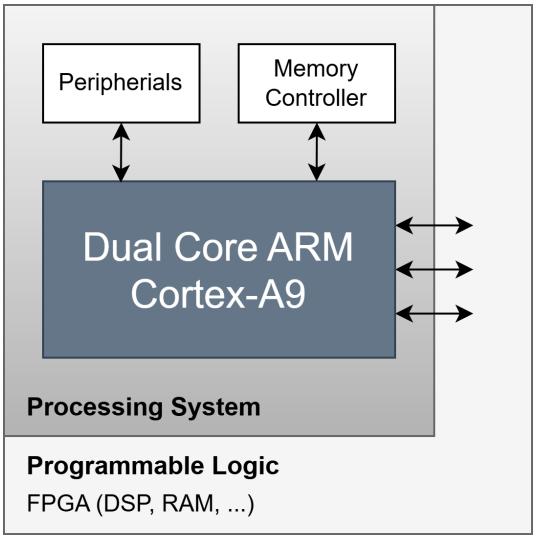


Figure 3.2: Architecture of Xilinx Zynq-7000 SoC.

3.3 Framework design

3.3.1 Overall architecture

The *Resilient Evaluation Framework for Tolerance* (RESIL) framework, developed as part of this work, consists of several components, all encapsulated in the provided *Application Programming Interface* (API) as an entry point for users:

- The *memory* module handles memory-related functionality, including the memory manager and fault protection interface.
- The *mechanisms* module contains various software fault tolerance mechanisms, each with its own implementation details that will be described later.
- The *utils* module contains utility functions for checksum calculation and bit operations, which are used internally.

Figure 3.3 presents the RESIL overall architecture.

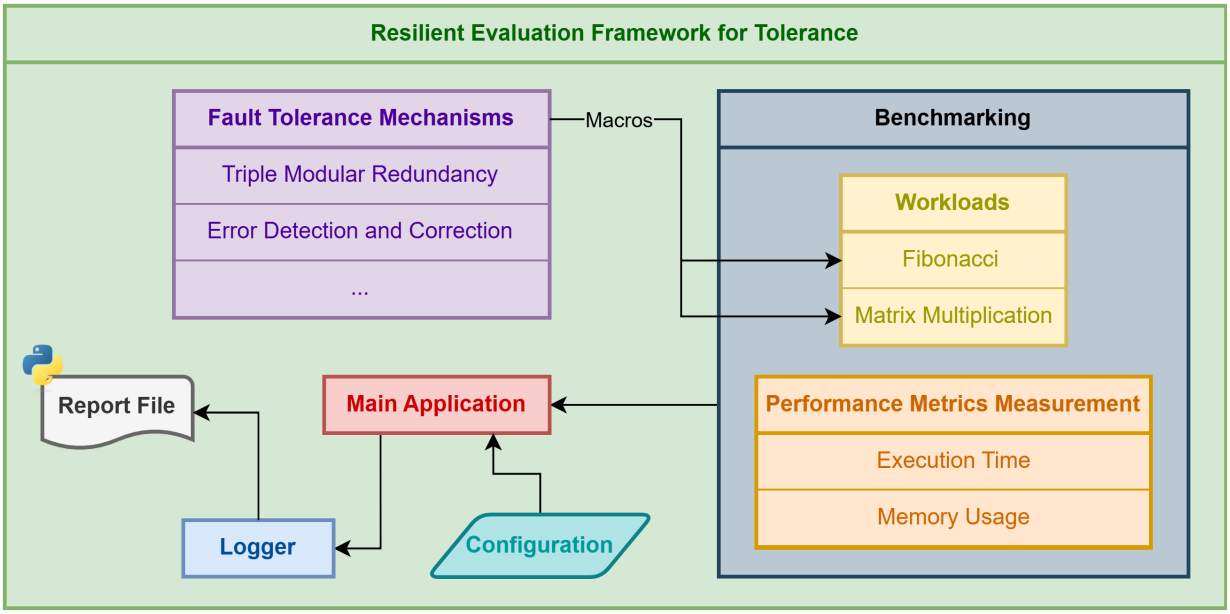


Figure 3.3: RESIL Architecture Overview.

3.3.2 Memory management

The *memory manager* is implemented as a singleton, with the `MemoryManager` class centralizing all memory operations. The module supports multiple *protection levels* to safeguard data integrity. It also allows for unprotected memory allocations when fault tolerance is not required. Each memory allocation is tracked using a `MemoryInfo` structure, which records the memory block's address, size, allocation timestamp, protection status, and the chosen protection level. The module maintains *memory statistics* to monitor usage. These statistics include the total number of allocations and deallocations, current and peak memory usage, and the number of protected allocations.

Memory allocation and deallocation are handled through the `allocate` and `deallocate` methods, respectively. The `allocate` method can optionally initialize memory with a specified value and apply the chosen protection scheme. If protection is enabled, the module adjusts the allocated block size.

The module's design is illustrated in the *Unified Modeling Language* (UML) class diagram shown in Figure 3.4.

The protection interface is implemented through the `FaultVariable` and `FaultProcess` templates, which ensure fault tolerance for both data and computation. The `FaultVariable` template encapsulates a trivially copyable type `T` and leverages the `MemoryManager` to allocate, verify, and update memory with protection flags. It provides the `get()` and `set()` methods to safely access and modify the underlying data. On the other hand, the `FaultProcess` template wraps a user-defined operation (`std::function<T()>`) and executes it under the configured protection level.

3.3.3 Fault-tolerance techniques

In the context of fault-tolerant systems, the granularity at which fault protection is applied plays an important role. Fault protection at the variable level operates with fine granularity,

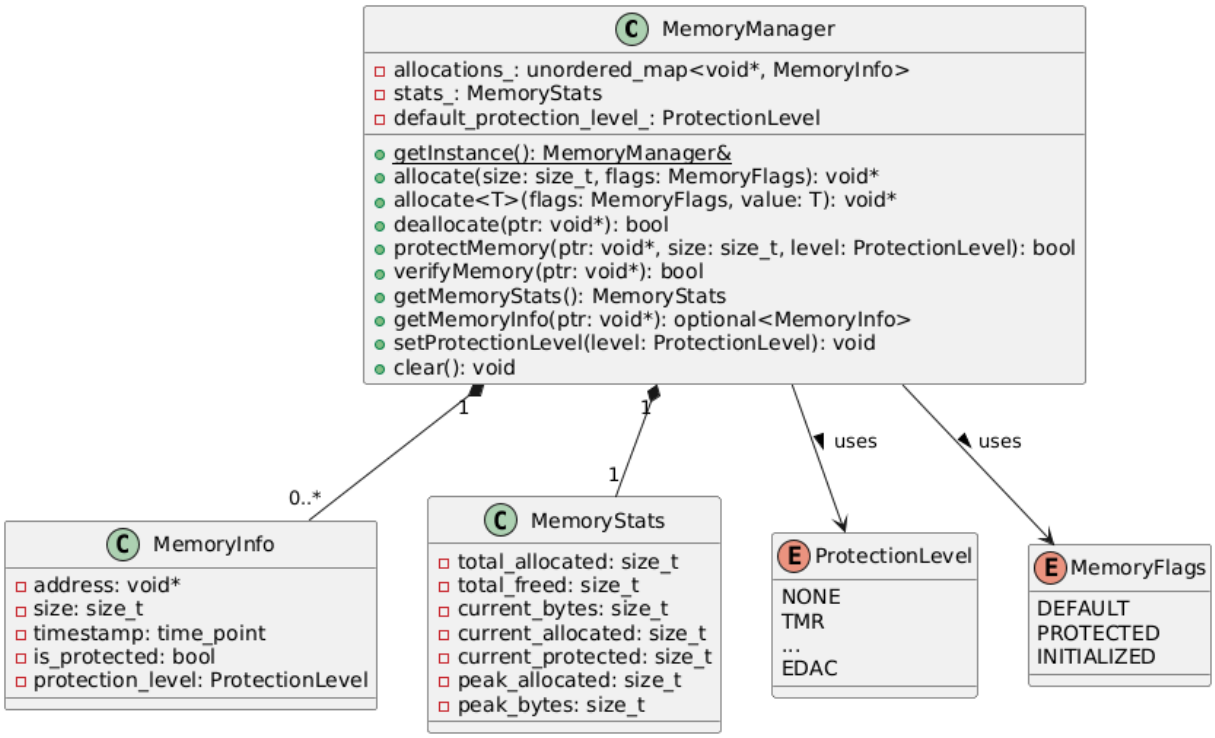


Figure 3.4: UML Class Diagram of the Memory Manager.

targeting individual data elements such as sensor readings, memory values, or configuration parameters. This approach ensures that errors affecting specific variables are detected and mitigated before they propagate through the system [38].

In contrast, fault protection at the process level adopts coarser granularity, focusing on the integrity and correct execution of entire processes or functional units. Here, the emphasis shifts from individual data points to the overall behavior of a process, such as a control loop or software thread [23]. While process-level protection should generally be more resource-efficient and scalable for complex systems, it may not catch faults that affect only a subset of variables within a process, potentially allowing localized errors to persist undetected until they manifest at the process output.

In our case, we will implement both of these to compare the types of granularity. Specific techniques include EDAC and redundancy. We now present the fault-tolerance techniques implemented in the framework.

Time Redundancy

Time redundancy consists of executing a program multiple times and comparing outputs to detect faults. It involves repeated execution, and majority voting can be employed to determine the correct output when discrepancies occur.

For example, the TMR technique consists of running the part of the code to be protected three times, as illustrated in Figure 3.5. Each execution saves the computed data in a separate memory space. The three values are then compared using a majority voting algorithm (see Algorithm 1). If no difference is found among the values, no error is detected. However, if one value differs from the other two, it is identified as erroneous and corrected by selecting the majority result.

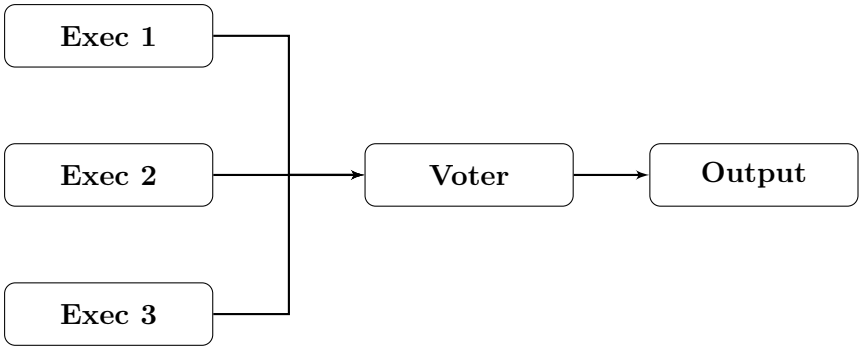


Figure 3.5: TMR Architecture

An alternative method, which aims to achieve similar fault tolerance to TMR but with lower overhead, is *Duplication with Comparison* (DWC) (see Figure 3.6). This approach divides an operation into two identical executions and compares their results, reducing resource usage compared to TMR. The process is repeated until we obtain the same results. It hopes to achieve the same efficiency as TMR without having to execute the same code three times unnecessarily.

Both the TMR and DWC methods present time redundancy and can be applied at various system-levels [26, 45]. We now consider these methods as process-level techniques, as they are applied to operations and not directly to the data itself.

Algorithm 1 Majority Voting

1: $result_1 \leftarrow$ function_call($input$)
2: $result_2 \leftarrow$ function_call($input$)
3: $result_3 \leftarrow$ function_call($input$)
4: **if** $result_1 == result_2$ **or** $result_1 == result_3$ **then**
5: $output \leftarrow result_1$
6: **else if** $result_2 == result_3$ **then**
7: $output \leftarrow result_2$
8: **else**
9: $output \leftarrow$ Error
10: **end if**
11: **return** *output*

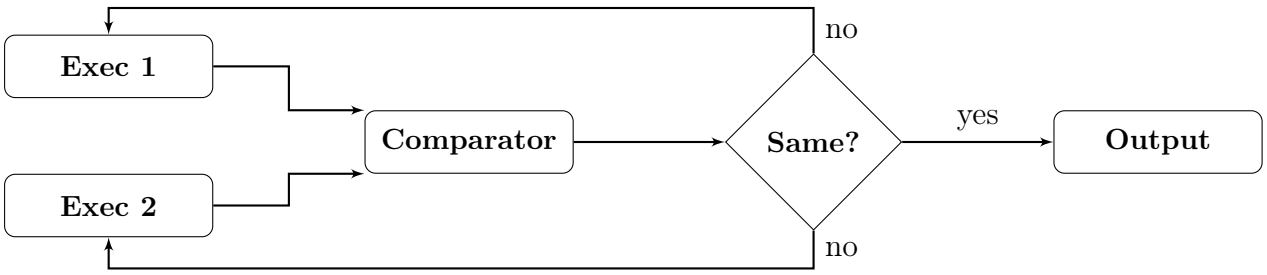


Figure 3.6: DWC Architecture

Information Redundancy

The first approach leverages the capabilities of the EDAC to ensure fault tolerance. By integrating error correcting codes, the method introduces information redundancy into the source code. These transformations, applied automatically, embed additional data and code designed to detect and, when possible, correct errors that may corrupt information stored in memory [19]. This is especially significant because memories feature a very high density of storage cells, making them more vulnerable to ionized particles than logic circuits [54].

A simple way to protect memory is by adding parity bits – a single extra bit that

makes the total number of 1-bits either even or odd – to each memory word. During every write operation, a parity generator calculates the parity bits for the data being written, along with the data already stored. If a particle strike flips a single bit in a memory word, the error can be detected by verifying the parity code during the read operation. The number of parity bits determines whether the scheme can only detect errors or also correct them. However, in most cases, memory error recovery is more complex, making protection schemes based on advanced codes, such as Hamming or Reed-Solomon codes, preferable. Table 3.1 provides an overview of EDAC methods.

Table 3.1: Overview of existing EDAC methods [28].

Method	**Capability**
Parity	Single Bit Error Detect
Hamming Code	Single Bit Error Correct, Double Bit Detect
RS Code	Correct consecutive and multiple bytes in error
Conventional Encoding	Corrects isolated burst noise in a communication stream
Overlying Protocol	Specific to each system implementation

The Hamming *Single Error Correction - Double Error Detection* (SEC-DED) code is a widely adopted EDAC mechanism due to its simplicity. Introduced by R. W. Hamming in 1950, the original Hamming code was designed to correct single-bit errors by using a set of parity bits, where each bit covers specific data bits as determined by the binary representation of the bit position [21]. The syndrome is computed as $s = H \cdot r^T$ with H being the parity check matrix and r the received code word and uniquely identifies the location of a single-bit error [49].

To extend the capability to double error detection, an additional parity bit is introduced, resulting in the SEC-DED variant. This extra bit allows the code to detect, but not correct, double-bit errors, which are less frequent but possible in memory systems due to soft errors. It is used to check overall parity [27].

In Hamming code implementations, parity bits are interleaved with data bits within the same word. However, in this design, all parity bits, including the overall parity bit for double-error detection, are stored in a separate byte immediately following the data. This approach simplifies memory layout but also reduces the risk of soft errors by avoiding data duplication and keeping the original data intact. It allows the EDAC logic to be applied uniformly across different data types without restructuring the original data.

In the encoding process, for each parity bit position p (starting from 0), the code iterates over all bit positions in the combined data and parity space. For each bit position i (1-based), if i is not a power of two (i.e., not a parity bit position), and if the p-th bit of i is set, the corresponding data bit is included in the parity calculation for p. The result is stored in the p-th position of the parity byte. The algorithm is described in Algorithm 2.

Algorithm 2 Hamming SEC-DED Encoding

for $p \leftarrow 0$ **to** $parity\_bits - 1$ **do**
    $data\_idx \leftarrow 0$
    $parity\_check \leftarrow 0$
    for $i \leftarrow 1$ **to** $data\_bits + parity\_bits$ **do**
        $isParityPos \leftarrow$ `isPowerOfTwo(`i`)`
        if not $isParityPos$ **then**
            if i **and** $(1 \ll p)$ **then**
                $parity\_check \leftarrow parity\_check \oplus$ `getBit(`$data$`,` $data\_idx$`)`
            end if
            $data\_idx \leftarrow data\_idx + 1$
        end if
    end for
    `setBit(`$parity$`,` p`,` $parity\_check$`)`
end for
$overall\_parity \leftarrow$ `checksum(`$data$`,` $data\_bits + parity\_bits$`)`
`setBit(`$parity$`,` $parity\_bits$`,` $overall\_parity$`)`

After computing all parity bits, an overall parity bit is calculated as the XOR of all data and parity bits. This bit is stored in the most significant bit of the parity byte (or the next available bit after the Hamming parity bits).

During decoding, the syndrome is calculated by recomputing the parity bits from the received data and comparing them with the stored parity bits. The syndrome is the bit-wise XOR of the expected and stored parity bits. A non-zero syndrome indicates an error.

The overall parity bit is checked to distinguish between single-bit and double-bit errors. The error handling logic follows the rules in Table 3.2.

Table 3.2: Hamming SEC-DED different scenarios

Possible Scenarios			**Conclusion**
Errors	Overall Parity	Syndrome	
0	Even	0	No error
1	Odd	0	Overall parity bit is erroneous
		Not 0	Syndrome indicates erroneous bit
2	Even	Not 0	Double-bit error (not correctable)

We now examine *Duplication and Checksum* (DUP), a technique that combines spatial and information redundancy to protect data integrity during read/write operations.

In this method, each data item is stored twice in separate memory locations. Additionally, a checksum is computed for the data during the write operation and stored alongside the duplicates. During read operations, the checksum of the retrieved data is computed and compared to the stored checksum. If they match, the data is considered valid. If they differ, the system computes the checksum of the second copy and compares it to the stored checksum. If the second copy's checksum matches, the system identifies it as the correct copy and uses it to overwrite the corrupted data.

Finally, the TMR method described in Time Redundancy (see Section 3.3.3) can also be applied here. It extends the principle of redundancy to variables through spatial replication.

In this approach, each variable is stored in three identical copies in separate memory locations. During write operations, all three copies are updated simultaneously with the same value. For read operations, the majority voting mechanism is employed to determine the correct value: the system compares the three copies and selects the value that appears in at least two of them.

Table 3.3 summarizes all strategies implemented in our framework, categorized by their level of operation and including their primary characteristics.

Table 3.3: Summary of Fault Tolerance Strategies Implemented in the Framework

Level	Strategy	Description
Process	TMR	Executes operation three times, uses majority voting on results
	DWC	Executes operation twice, compares results
Variable	TMR	Maintains three copies of each variable, uses majority voting on read
	DUP	Maintains two copies with checksum verification
	EDAC	Uses Hamming SEC-DED codes

3.3.4 Benchmarking

To evaluate the performance of our fault tolerance techniques, we implemented a benchmarking methodology combining computational workloads with a measurement infrastructure. Our approach follows established practices in fault tolerance evaluation by selecting representative computational tasks and employing a structured testing framework [17, 41].

The benchmark suite includes fundamental computational kernels that exercise different aspects of system performance. We selected the Fibonacci sequence calculation in both iterative and recursive forms to evaluate function call overhead and stack usage patterns. Matrix multiplication serves as a representative of memory-intensive operations with com-

plex access patterns.

Our benchmarking infrastructure employs a macro-based implementation similar to established testing frameworks (e.g., Google Benchmark [20]), providing an interface for defining and executing performance tests. The system automatically registers benchmarks at compile time.

The benchmarking process follows a structured workflow, described in Figure 3.7. It begins with system initialization and setup, followed by the execution of each benchmark with metrics measurements. After collecting and storing the results, the process concludes with proper teardown and cleanup procedures. For each benchmark, we record execution time (in microseconds) and memory usage statistics. This methodology ensures reproducible measurements across all test cases. Results are collected and stored in CSV format for subsequent analysis.

3.4 Fault-Injection Approach

3.4.1 FAIL*: A Fault-Injection Framework

The evaluation of our implemented techniques employs the FAIL* framework, an advanced architecture-level *Fault-Injection* (FI) tool designed for assessing fault tolerance. FAIL* emerged as a versatile tool set that was already involved in at least 18 publications by 20 researchers from four different research groups [44], making it suitable for evaluating our diverse set of approaches.

At its highest level, FAIL* adopts a client/server architecture (see Figure 3.8), designed to enable the parallel execution of FI experiments. The Campaign Controller serves as the central coordinator, distributing predefined job parameters from a centralized database to multiple FAIL* instances, such as those deployed across a computing cluster. Conversely, the results of these experiments are aggregated and stored back in the database, facilitating subsequent analysis. To ensure consistency across parallel executions, both the experiment

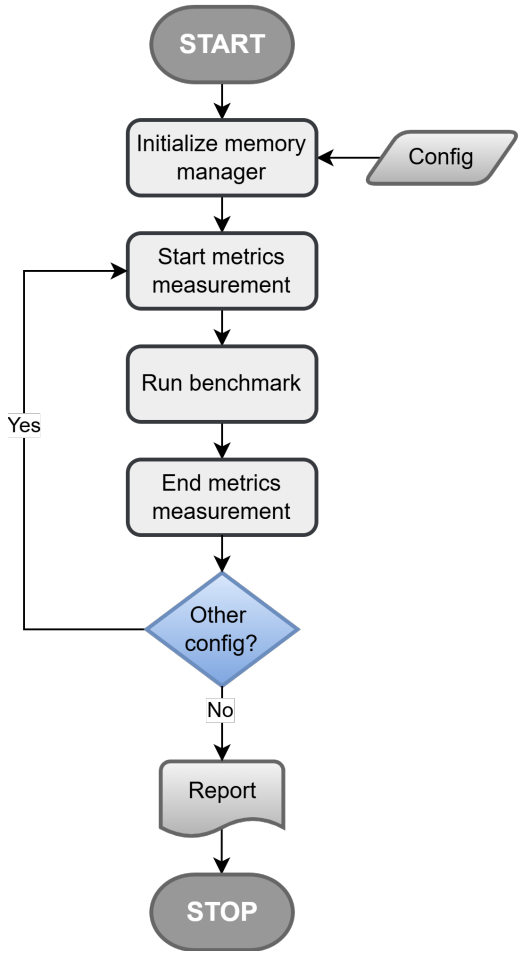


Figure 3.7: Benchmark Flowchart.

and its associated FAIL* instance must exhibit deterministic behavior, irrespective of the host computing node.

A FAIL* instance operates as an instrumented extension of an existing simulator or debugger. By integrating callback hooks at critical points within the back-end code, FAIL* intercepts and controls the execution flow, granting access to the (simulated) system state. The Execution-Environment Abstraction interface enables access to both meta-information about the target system and its real-time state, while also supporting the registration of event listeners for various system events.

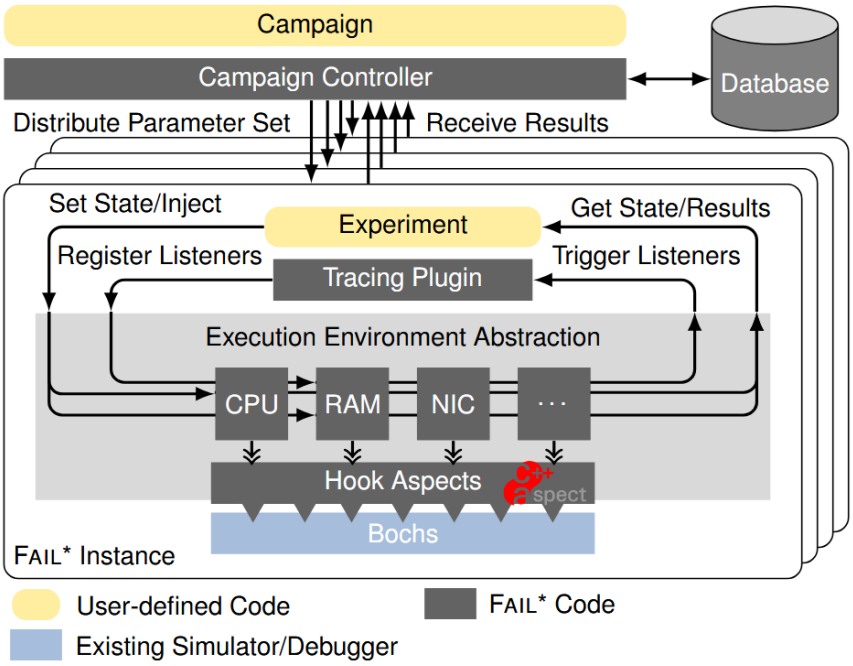


Figure 3.8: FAIL* Architecture Overview. [44]

3.4.2 Fault-Injection Campaign

A FI campaign comprises multiple individual FI experiments, each executing the analyzed target workload, in a consistent manner (see Figure 3.9). The workload is initially processed by a FAIL* instance using the specified experiment setup. During this *golden run*, no faults are introduced. The system's standard behavior and the benchmark's execution time are recorded. Furthermore, an instruction and memory-access trace is captured, which serves as a critical input for the subsequent steps: static analysis and fault-space pruning.

After collecting the pruning data , each experiment proceeds by injecting a single fault at a specific moment after the workload begins execution (e.g., after a defined number of CPU cycles) and at a precise location (e.g., by flipping a specific bit in main memory). Following the fault injection, the workload continues to run while the FI tool monitors its behavior and response to the injected fault. Based on these observations, the FI tool classifies the experiment outcome according to a predefined *failure model* (OK, *Silent Data Corruption* (SDC), TIMEOUT and TRAP) presented in Table 3.4 [43].

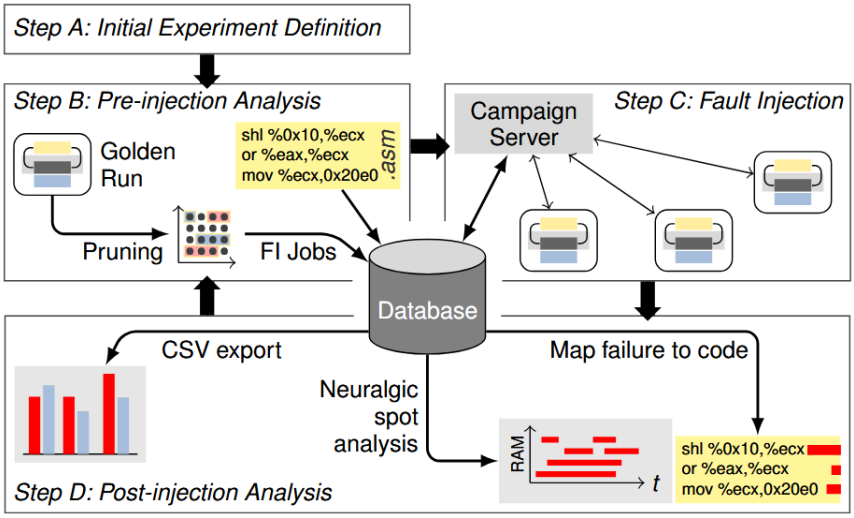


Figure 3.9: Fault-Tolerance Assessment Cycle. [44]

The diagram in Figure 3.10 depicts the setup of the FI campaign using Podman, which consists of interconnected containers operating within a shared environment. This was chosen to address the challenges posed by the numerous dependencies required for FAIL*, in an environment where root access is unavailable.

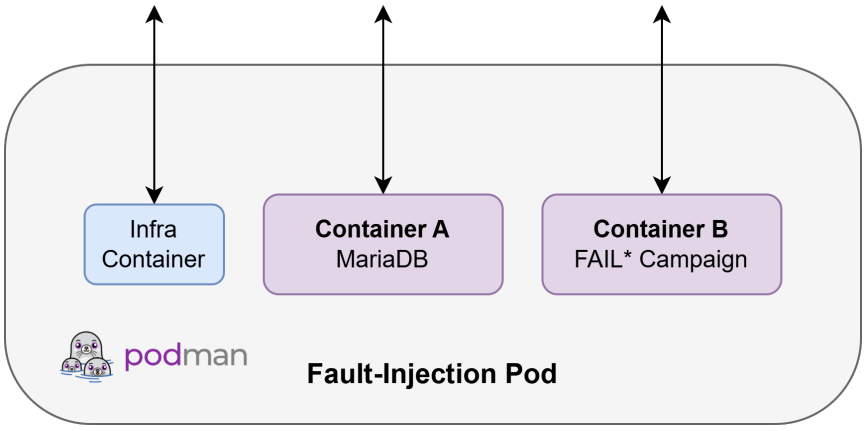


Figure 3.10: FI Campaign Setup.

Container A hosts the MariaDB database, which acts as a centralized data store for experiment parameters, configurations, and results. *Container B* is dedicated to the FAIL* campaign, where FI experiments are conducted. This container queries the MariaDB

Type	Description
OK	Injected fault caused no error.
SDC	The execution output deviates from the correct one obtained during the golden phase.
TIMEOUT	Fails to complete within a predefined time due to entering an infinite loop.
TRAP	A CPU trap or exception occurs, such as division by zero, segmentation fault, etc.

Table 3.4: Fault Type Classifications

database to retrieve experiment configurations and parameters at the start of each campaign, and writes the results back to the database upon completion of the experiments. The use of containerization allows for the encapsulation of all dependencies within *Container B*, mitigating compatibility issues and eliminating the need for root access on the host system. The *Infra Container* provides the foundational infrastructure to support the other containers.

Chapter 4

Results and Discussion

4.1 Performance Overhead

Table 4.1 summarizes the performance overhead of fault-tolerance techniques relative to a baseline execution, measured over 1000 iterations across the two architectures. Overhead is expressed as a slowdown factor (higher values indicate greater performance cost). The outputs have been displayed to be verified.

Table 4.1: Performance overhead of fault-tolerance techniques compared to baseline

Protection Level		Variable			Process	
Application	Platform	TMR	EDAC	DUP	TMR	DWC
Fibonacci Iterative	x86-64	2.275	17.711	1.378	1.758	1.379
N=30	ARM	2.262	26.336	1.503	1.372	1.200
Fibonacci Recursive	x86-64	1.210	1.354	1.293	1.227	1.651
N=30	ARM	1.001	1.004	1.001	2.993	1.996
Matrix Multiplication	x86-64	1.776	6.101	1.160	1.468	1.279
10x10	ARM	1.484	10.032	1.166	1.607	1.385

The bar plots for each benchmark on ARM (see Figure 4.1, 4.2 and 4.3) highlight the relative performance overhead of each technique. We only show the graphs for ARM because the error rates are higher for x86_64 platform. This behavior may be attributed to its use as a shared team server, where concurrent background tasks introduce additional interference, whereas the ZedBoard, being a dedicated circuit board, provides a more isolated execution environment, despite architectural differences.

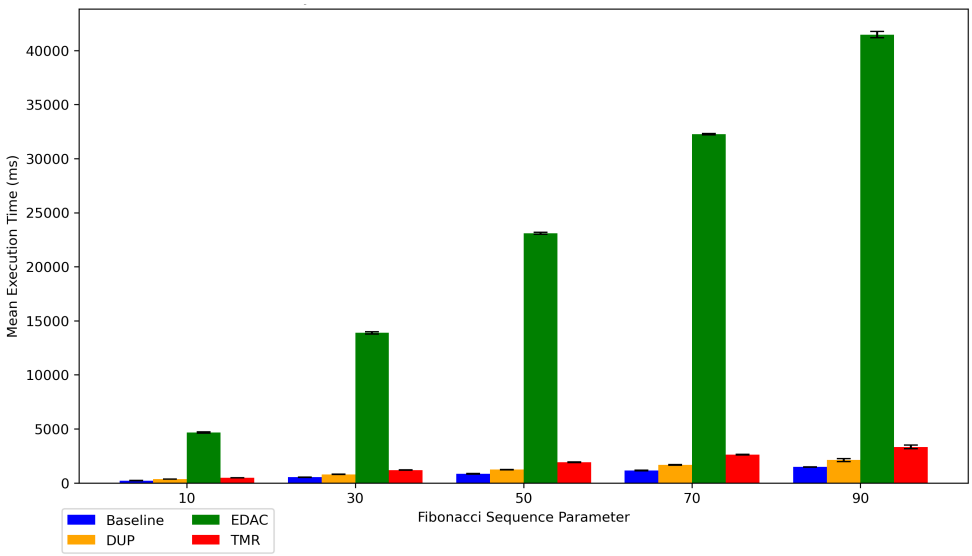
For the Fibonacci Iterative benchmark, EDAC exhibits a significant slowdown factor of 26.3×, while DUP, TMR (Process), and DWC remain close to the baseline performance. For the Fibonacci Recursive benchmark, TMR (Process) introduces the highest slowdown at 3×, whereas the variable-level techniques – DUP, EDAC, and TMR (Variable) – do not introduce a clear overhead. The negligible performance overhead of variable-level fault tolerance techniques is due to the algorithm's exponential time complexity, which is dominated by recursive calls rather than variable operations. In the Matrix Multiplication benchmark, EDAC again dominates the overhead with a 10× slowdown, while DUP and DWC demonstrate greater efficiency.

Table 4.2 details the memory overhead of each technique, expressed as a multiplication factor.

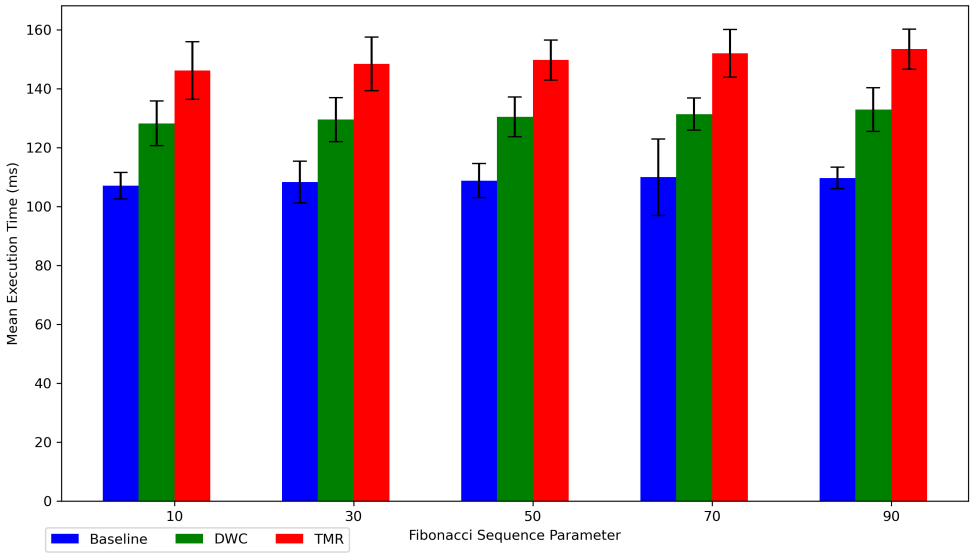
Table 4.2: Memory overhead of fault-tolerance techniques compared to baseline

Protection Level		Variable		Process	
Data Type	TMR	EDAC	DUP	TMR	DWC
uint32_t	3×	1.25×	2.25×	3×	2×
uint64_t		1.125×	2.125×		

TMR triples memory usage due to data redundancy, which is critical for memory-constrained systems. DUP and DWC introduce lower but still significant overheads. EDAC has the lowest overhead.

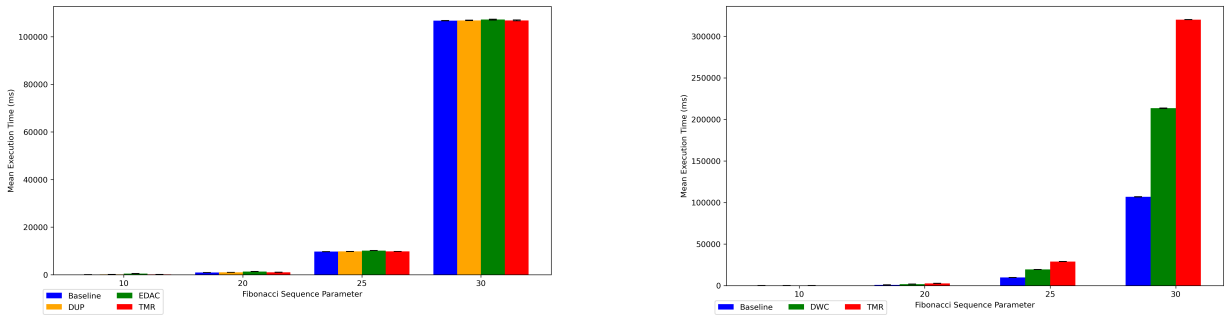


(a) Variable level



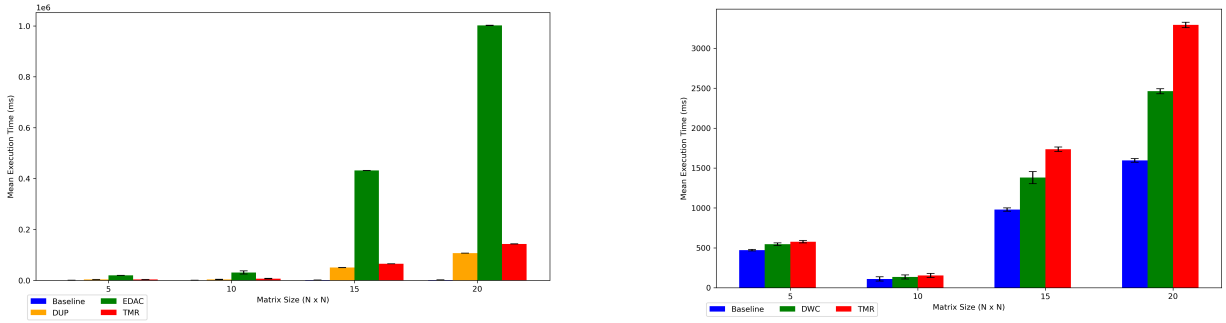
(b) Process level

Figure 4.1: Execution Time by Fault Tolerance Mechanism, for Fibonacci Iterative Benchmark on ARM platform.



(a) Variable level (b) Process level

Figure 4.2: Execution Time by Fault Tolerance Mechanism, for Fibonacci Recursive Benchmark on ARM platform.



(a) Variable level (b) Process level

Figure 4.3: Execution Time by Fault Tolerance Mechanism, for Matrix Multiplication Benchmark on ARM platform.

4.2 Robustness Assessment against Memory Faults

Table 4.3 presents the fault injection results, categorized by error type.

Table 4.3: Fault Injection Results in Percentage of Errors

		Errors [%]			
Application	Version	OK	SDC	TIMEOUT	TRAP
Fibonacci Iterative	Baseline	2.0	0.6	11.3	86.2
	DUP	73.5	3.0	1.6	21.9
	DWC	2.0	9.0	14.1	74.9
	EDAC	64.7	5.2	2.5	27.6
	TMR (Process)	7.3	7.4	15.5	69.8
	TMR (Variable)	75.6	0.1	2.2	22.1
Fibonacci Recursive	Baseline	51.6	6.3	4.4	37.7
	DUP	77.1	4.0	1.4	17.5
	DWC	57.5	0.2	4.8	37.5
	EDAC	62.4	4.8	2.7	30.1
	TMR (Process)	57.3	0.2	4.4	38.1
	TMR (Variable)	77.8	2.6	1.1	18.5
Matrix Multiplication	Baseline	3.0	90.0	0.4	6.6
	DUP	74.4	14.4	3.2	8.0
	DWC	7.9	26.4	62.4	3.3
	EDAC	27.0	3.6	40.7	28.7
	TMR (Process)	59.7	36.5	0.2	3.6
	TMR (Variable)	86.4	7.1	1.1	5.4

With the Fibonacci Iterative, there are only few SDC errors in the baseline and mainly TRAP ones. TMR (Variable) and DUP reduce TRAP errors to <25%, compared to 86% for the baseline. It is important to note that only TMR (Variable) reduces the number of SDC

errors. With the Fibonacci Recursive, all techniques reduce SDC errors, with the process-level techniques achieving the lowest rate (0.2%). On the other hand, the TMR (Variable) and DUP techniques reduce TRAP errors again. With the Matrix Multiplication, SDC are the main source of errors with 90%. EDAC is the most effective against them for this benchmark, but comes with a high rate of TIMEOUT. In general, the TMR (Variable) and DUP techniques achieve the best fault coverage, always higher than 70%.

Figure 4.4 shows the error distribution for each benchmark (ARM architecture).

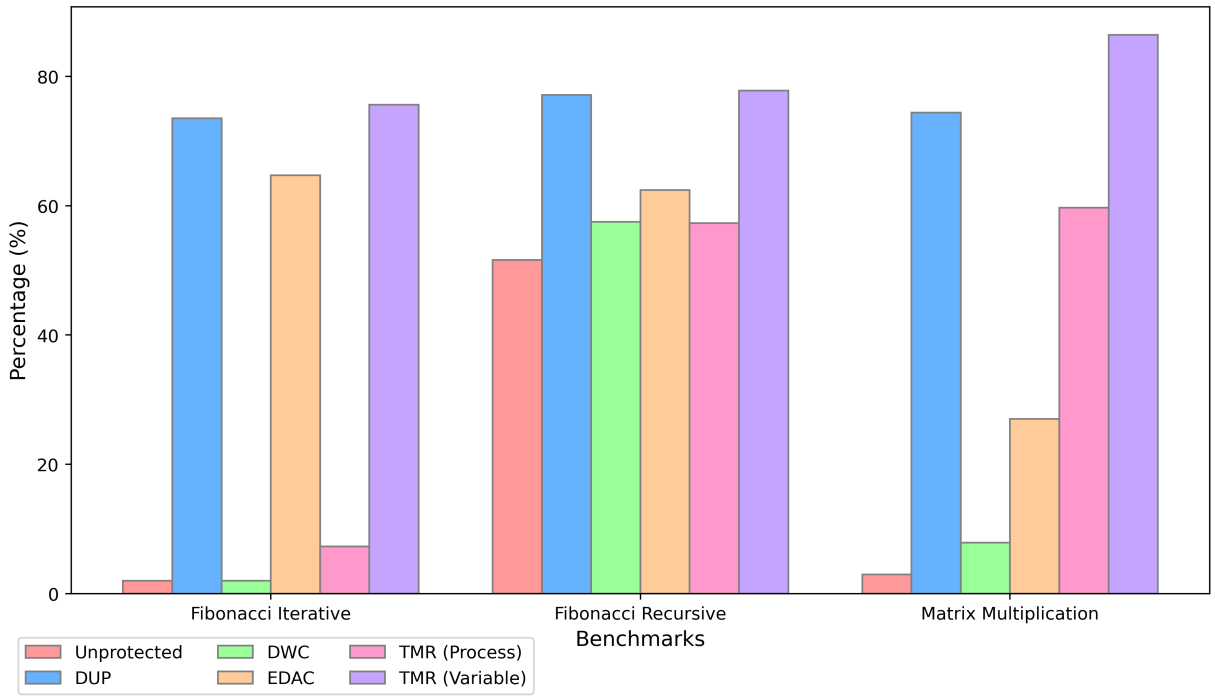


Figure 4.4: Percentage of "No Errors" by Fault Tolerance Mechanisms.

4.3 Discussion

First, as discussed in Section 2, fault tolerance mechanisms improve robustness but at a high performance cost. The scatter plot in Figure 4.5 illustrates the relationship between error rate and performance overhead factor for our techniques applied to the benchmarks.

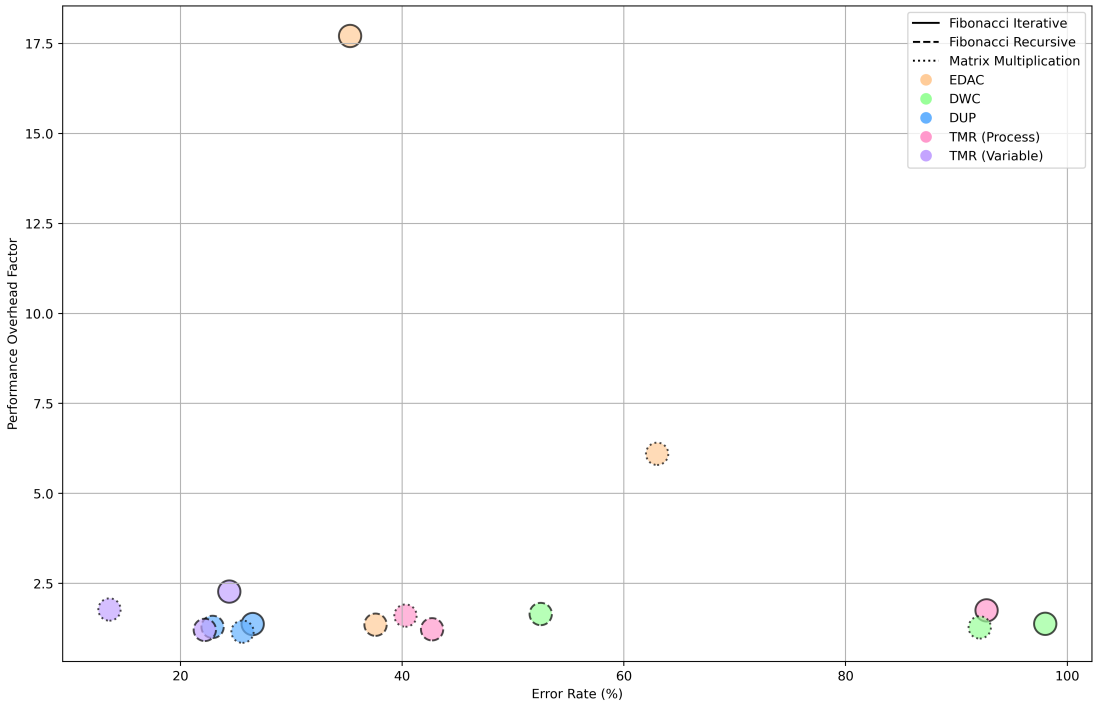


Figure 4.5: Performance Overhead vs. Error Rate.

The performance overhead results reveal that the impact of fault-tolerance techniques varies significantly depending on the application and the specific mechanism used. EDAC introduces a significant slowdown when the application requires high computational and memory access patterns, while DUP and DWC techniques maintain overheads much closer to the baseline. This suggests that EDAC may be less suitable for performance-critical applications.

The fault injection results highlight the strengths and weaknesses of each technique in mitigating different types of errors. They reveal that TMR (Variable) and DUP consistently achieve high fault coverage, reducing both SDC and TRAP errors across all benchmarks. This effectiveness, combined with their lower performance overhead, positions them as versatile solutions for a wide range of applications. However, the Matrix Multiplication benchmark presents a nuanced scenario: while EDAC excels at reducing SDC errors, it does so at the expense of increased TIMEOUT and TRAP errors rates, indicating a trade-off between error correction and system responsiveness. The process-level techniques never

really succeed to demonstrate a strong ability to mitigate the injected fault, except the SDC errors for Fibonacci Recursive.

The results for process-level techniques contrast with some literature expectations when analyzed alongside Table 4.4. While process-level methods exhibit significantly fewer fault injection points (e.g., 18,752 for TMR (Process) vs. 489,184 for TMR (Variable) in Fibonacci Iterative), our findings show that variable-level techniques achieve higher robustness – reducing both SDC and TRAP errors more effectively – despite their finer granularity and higher injection counts. This suggests that the increased overhead of variable-level techniques (as seen in performance metrics) is justified by their superior fault coverage.

Table 4.4: Number of fault injection points by mechanisms

Protection Level		Variable			Process	
Application	Baseline	TMR	EDAC	DUP	TMR	DWC
Fib. It.	6,336	489,184	7,660,288	24,474,328	18,752	12,384
Fib. Rec.	2,089,536	25,330,048	104,177,536	615,158,720	6,382,912	4,236,064
Mat. Mul.	2,786,688	64,282,816	8,416,680,208	850,319,800	11,081,216	7,747,328

Chapter 5

Conclusion

5.1 Summary

This thesis evaluated the performance overhead and robustness of software-based fault-tolerance techniques, applied to benchmark applications. The objective was to assess their suitability for embedded systems, against hardware-based fault-tolerance solutions.

The results highlight distinct trade-offs among the evaluated techniques. TMR (Variable) and DUP provide a balanced compromise between performance and robustness, significantly reducing critical errors such as SDC and TRAP errors, with only moderate overhead. In contrast, EDAC delivers strong error correction capabilities but at a substantial performance cost. Meanwhile, DWC and TMR (Process) techniques introduce higher latency due to their process-level redundancy.

For embedded systems, where memory and performance constraints are critical, DUP emerge as the most viable option. This technique offers lower memory overhead and a moderate performance impact. Although TMR (Variable) provides robust error mitigation, its triple memory usage may be prohibitive for embedded devices. EDAC is effective in error correction but less practical due to its high computational cost, unless strict error correction is mandatory.

In terms of cost comparison, software-based fault-tolerance techniques present a significant advantage over hardware-based solutions, as they incur no additional expenses for specialized components. However, they introduce additional development time for engineers to integrate and test these mechanisms. This includes the effort required for code instrumentation, validation, and debugging, which can extend project timelines and increase labor costs. Hardware solutions, such as EDAC memory or redundant processors, require specialized components which increase production costs. For budget-constrained projects, software-based techniques may offer an economical alternative to expensive hardware redundancy.

In conclusion, this work demonstrates that software-based fault-tolerance techniques can achieve a good balance between robustness and performance for embedded systems, especially with techniques like Duplication and Checksum or TMR.

5.2 Limitations

While this study provides valuable insights into the performance and robustness of software-based fault-tolerance techniques, several limitations must be acknowledged regarding both the RESIL framework and the fault injection campaign.

The current implementation of the RESIL framework presents certain constraints that may affect the generality of the results. First, the framework only focuses on variable-level and process-level protections, without addressing control-flow errors for example. Additionally, the framework does not support dynamic adaptation of protection levels based on run-time conditions. Furthermore, the framework does not currently support hybrid techniques which could offer additional optimizations for performance-critical applications. The framework's compatibility is also limited to specific architectures and may require significant modifications to support other platforms.

The robustness evaluation presented here is subject to several constraints. First, the campaign focuses on bit-flip faults in memory and registers but does not account for tran-

sient faults in the control path. Such faults may require additional protection mechanisms not evaluated in this study. Second, faults are injected at the software level via simulation, which may not fully capture the behavior of faults in real hardware. Additionally, the fault injection campaign does not consider multi-bit faults or burst errors and may challenge the effectiveness of the techniques. Finally, the study is confined to a specific set of benchmarks, which may not fully represent the diversity of embedded applications.

5.3 Future Work

Future work should explore fine-grained protection mechanisms, such as selective protection of code sections, and investigate hybrid techniques. A radiation campaign on real hardware would also provide a more comprehensive evaluation of robustness. Additionally, evaluating the framework on a wider range of benchmarks, including real-world space embedded applications (e.g., OBPMark) would improve the generality of the results.

Finally, integrating multi-threading into the framework would leverage multi-core architectures to enhance the efficiency of duplication techniques. A key approach would involve to make redundant computations parallel by executing duplicated or triplicated threads simultaneously on separate cores. Integration with *Real-Time Operating System* (RTOS) would also allow the framework to use priority-based scheduling.

Bibliography

[1] European Space Agency (ESA). *Spacecraft faults: more common over the South Atlantic*. https://www.esa.int/ESA_Multimedia/Images/2005/01/Spacecraft_faults_more_common_over_the_South_Atlantic, Accessed on 22/09/2025.

[2] Francesco Abate et al. "New Techniques for Improving the Performance of the Lockstep Architecture for SEEs Mitigation in FPGA Embedded Processors". In: *Nuclear Science, IEEE Transactions on* 56 (Sept. 2009), pp. 1992–2000. DOI: 10.1109/TNS.2009.2013237.

[3] Vitor A. P. Aguiar, Saulo G. Alberton, and Matheus S. Pereira. "Radiation-Induced Effects on Semiconductor Devices: A Brief Review on Single-Event Effects, Their Dynamics, and Reliability Impacts". In: *Chips* 4.1 (2025). ISSN: 2674-0729. DOI: 10.3390/chips4010012. URL: https://www.mdpi.com/2674-0729/4/1/12.

[4] Ygor Aguiar et al. "Introduction to Single-Event Effects". In: Aug. 2024, pp. 29–47. ISBN: 978-3-031-71722-2. DOI: 10.1007/978-3-031-71723-9_2.

[5] Alexander Aponte-Moreno, Felipe Restrepo-Calle, and Cesar Pedraza. "Using Approximate Computing and Selective Hardening for the Reduction of Overheads in the Design of Radiation-Induced Fault-Tolerant Systems". In: *Electronics* 8.12 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8121539. URL: https://www.mdpi.com/2079-9292/8/12/1539.

[6] R. Baumann. "Soft errors in advanced computer systems". In: *IEEE Design Test of Computers* 22.3 (May 2005), pp. 258–266. ISSN: 1558-1918. DOI: 10.1109/MDT.2005.69.

[7] R. Baumann. "SOFT ERRORS IN COMMERCIAL INTEGRATED CIRCUITS". In: *International Journal of High Speed Electronics and Systems* 14.02 (2004), pp. 299–309. DOI: 10.1142/S0129156404002363. eprint: https://doi.org/10.1142/S0129156404002363. URL: https://doi.org/10.1142/S0129156404002363.

[8] Peter L. Biermann et al. "Supernova explosions of massive stars and cosmic rays". In: *Advances in Space Research* 62.10 (Nov. 2018), pp. 2773–2816. ISSN: 0273-1177. DOI: 10.1016/j.asr.2018.03.028. URL: http://dx.doi.org/10.1016/j.asr.2018.03.028.

[9] Sébastien Bourdarie and Michael Xapsos. "The Near-Earth Space Radiation Environment". In: *IEEE Transactions on Nuclear Science* 55.4 (Aug. 2008), pp. 1810–1832. ISSN: 1558-1578. DOI: 10.1109/TNS.2008.2001409.

[10] NOAA / NWS Space Weather Prediction Center. *Solar Cycle Progression*. https://www.swpc.noaa.gov/products/solar-cycle-progression, Accessed on 10/09/2025.

[11] Krishna Mohan Chavali. "Mitigation of Soft Error Rate using Design, Process and Material Improvements". In: *2019 IEEE International Integrated Reliability Workshop (IIRW)*. Oct. 2019, pp. 1–5. DOI: 10.1109/IIRW47491.2019.8989915.

[12] C. L. Chen and M. Y. Hsiao. "Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review". In: *IBM Journal of Research and Development* 28.2 (Mar. 1984), pp. 124–134. ISSN: 0018-8646. DOI: 10.1147/rd.282.0124.

[13] Eduardo Chielle et al. "Hybrid soft error mitigation techniques for COTS processor-based systems". In: *2016 17th Latin-American Test Symposium (LATS)*. Apr. 2016, pp. 99–104. DOI: 10.1109/LATW.2016.7483347.

[14] Eduardo Chielle et al. "Reliability on ARM Processors Against Soft Errors Through SIHFT Techniques". In: *IEEE Transactions on Nuclear Science* 63.4 (Aug. 2016), pp. 2208–2216. ISSN: 1558-1578. DOI: 10.1109/TNS.2016.2525735.

[15] Eduardo Chielle et al. "S-SETA: Selective Software-Only Error-Detection Technique Using Assertions". In: *IEEE Transactions on Nuclear Science* 62.6 (Dec. 2015), pp. 3088–3095. ISSN: 1558-1578. DOI: 10.1109/TNS.2015.2484842.

[16] P.E. Dodd and L.W. Massengill. "Basic mechanisms and modeling of single-event upset in digital microelectronics". In: *IEEE Transactions on Nuclear Science* 50.3 (May 2003), pp. 583–602. ISSN: 1558-1578. DOI: 10.1109/TNS.2003.813129.

[17] Mojtaba Ebrahimi et al. "Revisiting software-based soft error mitigation techniques via accurate error generation and propagation models". In: *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. July 2016, pp. 66–71. DOI: 10.1109/IOLTS.2016.7604674.

[18] R. Ecoffet. "Overview of In-Orbit Radiation Induced Spacecraft Anomalies". In: *IEEE Transactions on Nuclear Science* 60.3 (2013), pp. 1791–1815. DOI: 10.1109/TNS.2013.2262002.

[19] Olga Goloubeva et al. *Software-Implemented Hardware Fault Tolerance*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387260609.

[20] Google. *Google Benchmark: A microbenchmark support library*. https://github.com/google/benchmark. 2024.

[21] R. W. Hamming. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (Apr. 1950), pp. 147–160. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[22] Richard Horne. "Plasma astrophysics: Acceleration of killer electrons". In: *Nature Physics* 3 (Sept. 2007), pp. 590–591. DOI: 10.1038/nphys703.

[23] Farnoosh Hosseinzadeh, Petr Pfeifer, and Heinrich Theodor Vierhaus. "The coarse and fine granular fault tolerance techniques in FPGA-based processors". In: *2017 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*. Sept. 2017, pp. 116–120. DOI: 10.23919/SPA.2017.8166849.

[24] A.H. Johnston, G.M. Swift, and D.C. Shaw. "Impact of CMOS scaling on single-event hard errors in space systems". In: *1995 IEEE Symposium on Low Power Electronics. Digest of Technical Papers*. Oct. 1995, pp. 88–89. DOI: 10.1109/LPE.1995.482476.

[25] NASA JPL. *Design for Radiation Tolerance*. {https://parts.jpl.nasa.gov/asic/Sect.3.4.html},Accessedon23/09/2025.

[26] Israel Koren and C. Mani Krishna. "CHAPTER 1 - Preliminaries". In: *Fault-Tolerant Systems*. Ed. by Israel Koren and C. Mani Krishna. Burlington: Morgan Kaufmann, 2007, pp. 1–10. ISBN: 978-0-12-088525-1. DOI: https://doi.org/10.1016/B978-012088525-1/50004-3. URL: https://www.sciencedirect.com/science/article/pii/B9780120885251500043.

[27] Dave Kythe and Prem Kythe. *Algebraic and Stochastic Coding Theory*. Jan. 2012, pp. 1–481. ISBN: 9781315216560. DOI: 10.1201/b11707.

[28] K.A. LaBel et al. "Commercial microelectronics technologies for applications in the satellite radiation environment". In: *1996 IEEE Aerospace Applications Conference. Proceedings*. Vol. 1. Feb. 1996, 375–390 vol.1. DOI: 10.1109/AERO.1996.495897.

[29] Richard Leach. "Spacecraft system failures and anomalies attributed to the natural space environment". In: *Space Programs and Technologies Conference*. Sept. 1995. DOI: 10.2514/6.1995-3564. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.1995-3564. URL: https://arc.aiaa.org/doi/abs/10.2514/6.1995-3564.

[30] A. Lechner. "Particle interactions with matter". In: *CERN Yellow Reports: School Proceedings* 5 (2018), pp. 47–68. DOI: 10.23730/CYRSP-2018-005.47.

[31] Andreas Lund et al. "ScOSA system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture". In: *CEAS Space Journal* 14 (May 2021). DOI: 10.1007/s12567-021-00371-7.

[32] NASA Marshall. *The Solar Interior*. {https://solarscience.msfc.nasa.gov/interior.shtml},Accessedon16/10/2025.

[33] Rositsa Miteva, Susan W. Samwel, and Stela Tkatchova. "Space Weather Effects on Satellites". In: *Astronomy* 2.3 (2023), pp. 165–179. ISSN: 2674-0346. DOI: 10.3390/astronomy2030012. URL: https://www.mdpi.com/2674-0346/2/3/12.

[34] "Monitoring the daily variation of Sun-Earth magnetic fields using galactic cosmic rays". In: *The Innovation* 5.6 (2024), p. 100695. ISSN: 2666-6758. DOI: https://doi.org/10.1016/j.xinn.2024.100695. URL: https://www.sciencedirect.com/science/article/pii/S2666675824001334.

[35] Mahsa Mousavi et al. "MTTR reduction of FPGA scrubbing: Exploring SEU sensitivity". In: *Microprocessors and Microsystems* 101 (2023), p. 104841. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2023.104841. URL: https://www.sciencedirect.com/science/article/pii/S014193312300087X.

[36] National Academies of Sciences, Engineering, and Medicine. *Radiation and the International Space Station: Recommendations to Reduce Risk*. Washington, DC: The National Academies Press, 2000. DOI: 10.17226/9725. URL: https://doi.org/10.17226/9725.

[37] Kalle Ngo, Tage Mohammadat, and Johnny Öberg. "Towards a single event upset detector based on COTS FPGA". In: *2017 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. Oct. 2017, pp. 1–6. DOI: 10.1109/NORCHIP.2017.8124960.

[38] Mahtab Niknahad, Oliver Sander, and Juergen Becker. "A study on fine granular fault tolerance methodologies for FPGAs". In: *6th International Workshop on Recon-*

figurable Communication-Centric Systems-on-Chip (ReCoSoC). June 2011, pp. 1–5. DOI: 10.1109/ReCoSoC.2011.5981537.

[39] M. Pignol. "DMT and DT2: two fault-tolerant architectures developed by CNES for COTS-based spacecraft supercomputers". In: *12th IEEE International On-Line Testing Symposium (IOLTS'06)*. July 2006, 10 pp.-. DOI: 10.1109/IOLTS.2006.24.

[40] Christian Poivey. "RADIATION EFFECTS IN SPACE ELECTRONICS". In: *6th EIRO forum School on Instrumentation*. ESA. May 2019.

[41] Gennaro S. Rodrigues et al. "Analyzing the Impact of Fault-Tolerance Methods in ARM Processors Under Soft Errors Running Linux and Parallelization APIs". In: *IEEE Transactions on Nuclear Science* 64.8 (Aug. 2017), pp. 2196–2203. ISSN: 1558-1578. DOI: 10.1109/TNS.2017.2706519.

[42] K.H. Schatten. "Solar activity and the solar cycle". In: *Advances in Space Research* 32.4 (2003). Heliosphere at Solar Maximum, pp. 451–460. ISSN: 0273-1177. DOI: https://doi.org/10.1016/S0273-1177(03)00328-4. URL: https://www.sciencedirect.com/science/article/pii/S0273117703003284.

[43] Horst Schirmeier. "Efficient fault-injection-based assessment of software-implemented hardware fault tolerance". PhD thesis. Jan. 2016. DOI: 10.17877/DE290R-17222.

[44] Horst Schirmeier et al. "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance". In: *2015 11th European Dependable Computing Conference (EDCC)*. Sept. 2015, pp. 245–255. DOI: 10.1109/EDCC.2015.28.

[45] Zihai Shi et al. "2 - Reviews of resilience theories and mathematical generalization". In: *Structural Resilience in Sewer Reconstruction*. Ed. by Zihai Shi et al. Butterworth-Heinemann, 2018, pp. 17–78. ISBN: 978-0-12-811552-7. DOI: https://doi.org/10.1016/B978-0-12-811552-7.00002-X. URL: https://www.sciencedirect.com/science/article/pii/B978012811552700002X.

[46] Philip Shirvani and E. McCluskey. "Software-Implemented Hardware Fault Tolerance Experiments COTS in Space". In: (Jan. 2000).

[47] Mohammadreza Amel Solouki, Shaahin Angizi, and Massimo Violante. "Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques". In: *IEEE Access* 12 (2024), pp. 180939–180967. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3509633.

[48] Yi Sun et al. "Heavy ion-and Proton-induced SEU Simulation and Error Rates Calculation in 0.15um SRAM-based FPGA". In: *2019 3rd International Conference on Circuits, System and Simulation (ICCSS)*. June 2019, pp. 84–88. DOI: 10.1109/CIRSYSSIM.2019.8935625.

[49] Massachusetts Institute of Technology. *Introduction to Digital Communicatiom Systems*. https://web.mit.edu/6.02/www/currentsemester/handouts/L05_slides.pdf. Accessed: 10/06/2025. 2014.

[50] Venu Babu Thati et al. "Selective Duplication and Selective Comparison for Data Flow Error Detection". In: *2019 4th International Conference on System Reliability and Safety (ICSRS)*. Nov. 2019, pp. 10–15. DOI: 10.1109/ICSRS48664.2019.8987731.

[51] Robin Thunig et al. "DECO: Optimizing Software-based Soft-Error Detector Configurations". In: *2022 18th European Dependable Computing Conference (EDCC)*. Sept. 2022, pp. 73–80. DOI: 10.1109/EDCC57035.2022.00022.

[52] J. A. VAN ALLEN et al. "Observation of High Intensity Radiation by Satellites 1958 Alpha and Gamma". In: *Journal of Jet Propulsion* 28.9 (1958), pp. 588–592. DOI: 10.2514/8.7396. URL: https://doi.org/10.2514/8.7396.

[53] Imran Wali. "Circuit and System Fault Tolerance Techniques". PhD thesis. Université de Montpellier, 2016. URL: https://theses.hal.science/tel-01807927/file/2016_WALI_archivage.pdf.

[54] Fan Wang and Vishwani D. Agrawal. "Single Event Upset: An Embedded Tutorial". In: *21st International Conference on VLSI Design (VLSID 2008)*. Jan. 2008, pp. 429–434. DOI: 10.1109/VLSI.2008.28.

[55] G. I. Zebrev et al. *Soft Error Rate in Space: A Unified Analytical Approach*. 2025. arXiv: 2501.06260 [physics.app-ph]. URL: https://arxiv.org/abs/2501.06260.

[56] *Zynq-7000 SoC Data Sheet: Overview*. DS190. v1.11.1. XILINX. 2018.

Abstract — The increasing use of *Commercial Off-The-Shelf* (COTS) components in space systems introduces significant reliability challenges due to their vulnerability to radiation-induced faults, particularly *Single Event Effects* (SEE). This thesis presents the development and evaluation of a modular framework – *Resilient Evaluation Framework for Tolerance* (RESIL) – designed to assess the performance overhead of software-based fault tolerance techniques for embedded systems in space applications. The study implements and compares several fault tolerance mechanisms with different types of granularity. These techniques are evaluated using representative computational benchmarks and a fault-injection campaign to simulate bit-flip errors. Results show that *Triple Modular Redundancy* (TMR) (Variable) and *Duplication and Checksum* (DUP) provide a balanced trade-off between performance and robustness, significantly reducing critical errors (e.g., *Silent Data Corruption* (SDC) and TRAP errors) with moderate overhead. This work demonstrates that software-based fault tolerance techniques can achieve a favorable balance between reliability and performance for embedded systems. The findings provide valuable insights for engineers designing fault-tolerant systems for space applications.

Keywords: fault tolerance, performance evaluation, space applications, radiation, embedded systems.

ISAE
10, avenue Édouard Belin
BP 54032
31055 Toulouse CEDEX 4