Run-time Redundancy Management of Processor Functional Units for Mixed-Critical Scenarios

Von der Fakultät MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik der Brandenburgischen Technischen Universität Cottbus-Senftenberg genehmigte Dissertation zur Erlangung des akademischen Grades eines

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

vorgelegt von

Master of Science (M.Sc.)

Raphael Segabinazzi Ferreira

geboren am 26.03.1990 in Porto Alegre/RS, Brasilien

Vorsitzender:	Dr.-Ing. Marc Reichenbach
Gutachter:	Prof. Dr.-Ing. Jörg Nolte
Gutachter:	Prof. Dr.-Ing. Heinrich T. Vierhaus
Gutachter:	Prof. Dr. Fabian Luis Vargas

Tag der mündlichen Prüfung: 14.04.2022

DOI: https://doi.org/10.26127/BTUOpen-5888

Run-time Redundancy Management of Processor Functional Units for Mixed-Critical Scenarios

Doctoral thesis approved by the Faculty 1 - Mathematics, Computer Science, Physics, Electrical Engineering and Information Technology of the Brandenburg University of Technology Cottbus-Senftenberg submitted to obtain the academic degree of

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

presented by

Master of Science (M.Sc.)

Raphael Segabinazzi Ferreira

born on 26.03.1990 in Porto Alegre/RS, Brazil

Chairperson:	Dr.-Ing. Marc Reichenbach
Examiner:	Prof. Dr.-Ing. Jörg Nolte
Examiner:	Prof. Dr.-Ing. Heinrich T. Vierhaus
Examiner:	Prof. Dr. Fabian Luis Vargas

Date of the oral exam: 14.04.2022

DOI: https://doi.org/10.26127/BTUOpen-5888

For my parents and my sister, who despite being far away, always encouraged me.

For my wife who supported and accompanied me on this journey, without her, I would not have made it.

For my daughter, for being the joy of my life.

Finally, to demonstrate my gratitude, I let here my fifteen-month-old daughter make her own contribution to this thesis. Here are the words as she freely typed by herself:

> *njbvvvv v bjntngddy˙ngbfhcngcdnsdjndcfcddxmjmdnccb vjdens55fjjntmgtgv bgv ggb hb gvfm kgm nh l; .„4 v nh*

Disclaimer

The work developed in this thesis was partially developed under the funding of the RESCUE European Training Network. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 722325.

Abstract

Since electronics started to scale down, a growing concern about the reliability of these electronic devices has emerged. At the same time, the increased demand for high performance within the safety- and mixed-critical domains, such as the aerospace and automotive industry, motivated a shift from previous consolidated and mature technology to the new cutting edge devices with smaller feature sizes. Therefore, there is a need to improve the fault tolerance of these high-end devices so that minimum failure rates can be obeyed. Although redundancy has been a great solution for these problems, their drawbacks such as power and area overheads must be watched carefully, so that per-unit price does not extrapolate affordable limits, and the redundancy does not add more sources of error than it improves the fault tolerance. This thesis proposes an approach for run-time management of redundancy among the processor internal Functional Units (FUs) within mixed-critical scenarios, tackling the compensation of the trade-offs between fault-tolerance, power consumption, hardware usage (ageing), and hardware area (cost).

With these objectives in mind, this thesis presents a concept for a dynamic processor architecture capable to enable and disable redundancy of FUs on-demand, and a software mechanism for criticality-aware management of these units for mixed-critical processes within an Operating System (OS). For this purpose, a processor design was extended with a few additional instructions that enabled different replication schemes in the processor at run-time. Furthermore, a compatible Real-Time Operating System (RTOS) is also extended to enable the desired criticality-aware management of units.

Evaluating the implemented test platform when the extended processor was running bare-metal code, the latency to shift between different replication schemes was of only *one instructions cycle*. Furthermore, when the processor was running the adapted RTOS, the run-time overhead over the latency to switch between processes remained below 2.5%. Meanwhile, resulting from the processor extensions, the hardware overhead remained smaller than standard full core replication schemes such as core lock-step approaches. Regarding fault tolerance, the expected failure rate of the FUs module decreased by approximately 80% when its FUs were configured with Triple Modular Redundancy (TMR). Furthermore, when considering the whole area of the processor core, its respective failure rate decreased by about 15% when configured these units with the same triplication scheme. Finally, it is also presented that the run-time management of FUs was likewise able to decrease the power consumption and hardware ageing for the proposed mixed-critical scenario. After all, we

can say that the concept can increase fault tolerance on-demand of a processor design with moderately low hardware overhead, while it also minimises the power consumption and hardware usage (ageing) for its intended mixed-critical scenario.

Zusammenfassung

Seit die Elektronik immer kleiner wird, ist eine wachsende Besorgnis über die Zuverlässigkeit dieser elektronischen Geräte entstanden. Gleichzeitig motivierte die gestiegene Nachfrage nach hoher Leistung in sicherheits- und gemischt kritischen Bereichen wie der Luft- und Raumfahrt- und Automobilindustrie zu einer Umstellung von früher konsolidierter und ausgereifter Technologie auf die neuen Spitzengeräte mit kleineren Strukturgrößen. Daher muss die Fehlertoleranz dieser High-End-Geräte verbessert werden, damit minimale Ausfallraten eingehalten werden können. Obwohl Redundanz eine großartige Lösung für diese Probleme ist, müssen ihre Nachteile wie Energie- und Flächen-Overhead sorgfältig beobachtet werden, damit der Preis pro Einheit nicht erschwingliche Grenzen extrapoliert und die Redundanz nicht mehr Fehlerquellen hinzufügt, als sie verbessert Fehlertoleranz. Diese Arbeit schlägt einen Ansatz für das Laufzeitmanagement der Redundanz zwischen den prozessorinternen Funktionseinheiten (“Functional Units”, FUs) in gemischt kritischen Szenarien vor, wobei die Kompensation der Kompromisse zwischen Fehlertoleranz, Stromverbrauch, Hardwarenutzung (Alterung) und Hardwarebereich (Kosten).

Vor diesem Hintergrund stellt diese Arbeit ein Konzept für eine dynamische Prozessorarchitektur vor, die in der Lage ist, die Redundanz von FUs bei Bedarf zu aktivieren und zu deaktivieren, sowie einen Softwaremechanismus zur kritikalitätsbewussten Verwaltung dieser Einheiten für gemischt kritische Prozesse innerhalb eines Betriebssystems (“Operating System”, OS). Dazu wurde ein Prozessordesign um einige zusätzliche Anweisungen erweitert, die zur Laufzeit unterschiedliche Replikationsschemata im Prozessor ermöglichen. Darüber hinaus wird auch ein kompatibles Echtzeit-Betriebssystem (“Real-Time Operating System”, RTOS) erweitert, um die gewünschte kritikalitätsbewusste Verwaltung von Einheiten zu ermöglichen.

Die Rekonfiguration des Prozessors erfordert nur einen Befehlszyklus, der gesamte zusätzliche overhead für den Prozesswechsel beträgt weniger als 2,5%. In der Zwischenzeit blieb der Hardware-Overhead aufgrund der Prozessorerweiterungen kleiner als bei standardmäßigen vollständigen Kernreplikationsschemata, wie z. B. Core-Lock-Step-Ansätzen. In Bezug auf die Fehlertoleranz verringerte sich die erwartete Ausfallrate des FU-Moduls um etwa 80%, wenn es mit Dreifacher Modularer Redundanz (“Triple Modular Redundancy”, TMR) konfiguriert wurde. Darüber hinaus verringerte sich bei Betrachtung der gesamten Fläche des Prozessorkerns die jeweilige Ausfallrate um etwa 15%, wenn die Einheiten mit demselben

Verdreifachungsschema konfiguriert wurden. Abschließend wird auch dargestellt, dass das Laufzeitmanagement von FUs den Stromverbrauch und die Hardwarealterung für das vorgeschlagene gemischt-kritische Szenario ebenfalls verringern konnte. Schließlich können wir sagen, dass das Konzept die Fehlertoleranz eines Prozessordesigns mit mäßig niedrigem Hardware-Overhead bei Bedarf erhöhen kann, während es auch den Stromverbrauch und die Hardwarenutzung (Alterung) für das beabsichtigte gemischt-kritische Szenario minimiert.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Since the beginning of More's law and Dennard scaling, the miniaturisation of devices has been driving the electronics industry, which resulted in today's high-end devices with billions of transistors integrated into a single chip [10]. However, this level of miniaturisation and integration resulted in reduced reliability of these devices (e.g., they may no longer function as they were designed to, and work for a shorter period of time). For example, smaller transistor dimensions and new computing principles such as lower supply voltages, and near transistor threshold operation resulted in less energy necessary to change the electric charge stored in memory cells, logic latches, or flip-flops. Therefore, charged particles (e.g., protons and alpha particles) or neutrons hitting the silicon of an electronic device can easier cause a bit-flip in its logical components [8, 65]. Furthermore, since the beginning of the last decade, as transistor sizes became even smaller and operating voltages lower, the leakage current of the transistors started to be even more significant for the overall power consumption of an electronic device. Such an effect posed a barrier in the scaling down of the operating voltage, and the power density no longer remained constant as transistors' size decreased (no longer following Dennard's scaling predictions) [10]. Therefore, the smaller the transistors and the bigger the integration rate, the bigger the power density and the smaller is the space for power dissipation in these devices. Thus, since cooling technologies did not evolve proportionally, these devices became exposed to higher temperatures, which accelerates natural ageing effects, such as increasing the threshold voltage - the necessary voltage to switch on a transistor - and, as a consequence, the speed at which transistors can be switched on and off [60]. Therefore, the lifespan in which an electronic device can be operated at its maximum designed frequency is reduced.

In the past, safety-critical applications could rely on consolidated and mature technologies [8]. However, since there is an increasing demand for performance in this domain, these applications needed to start looking into the newest hardware technologies of $3nm$ and below [96, 83]. For instance, autonomous driving applications of today demand high performance with real-time processing power, but a failure caused by a possible bit-flip in the hardware elements can lead to harmful consequences [76, 103]. These kinds of applications lead the scientific community and industry to look for solutions that increase the reliability of these high-end devices by overcoming the aforementioned negative effects of miniaturisation and the high rate of integration while keeping the speed-up.

The most common strategy to cope with these reliability problems is the use of redundancy. Multiple ways to apply redundancy can be found in the literature, some approaches use *temporal redundancy* by simply performing multiple computations distributed over time, while others use *spatial redundancy*, adding redundant (e.g., duplicating, or triplicating) hardware modules to perform multiple computations in parallel. Once the redundancy is in place, an error can be detected end/or corrected by comparing the results of these multiple computations. The Triple Modular Redundancy (TMR) is the most common of these approaches, based on triplication and majority voting, it was first envisioned by John Von Neumann in 1960, and it is still used to provide error correction in safety-critical systems of today (details of this approach will be revisited in the background section under Section 2.4.1). However, the main problem of redundant approaches is the overhead: in time for the temporal redundancy, or hardware area and power consumption for the spatial redundancy. The hardware area overhead affects, for example, the per-unit price of the system, since more hardware must be employed to build it; meanwhile, the power and timing overheads affect the performance: the timing on the latency to provide results for computations, while the extra power may limit the maximum frequency in which a system can runs. Furthermore, another important aspect regarding hardware area and fault tolerance is that the bigger the hardware in use, the bigger the probability of this hardware to fail. Therefore, any fault-tolerant method that increases the hardware area must increase the fault tolerance of the system enough to compensate for its hardware increase. Hence, not all systems can *afford* such an overhead, and a good compromise must be found between the provided fault tolerance and the area and power overheads when using redundancy.

Looking once again into the safety-critical application domain, it is expected that not all tasks performed by the system are critical. For example, within a vehicle, the airbag systems are highly critical, in which a wrong activation or a failure to activate them at the correct time can lead to very harmful consequences such as human injuries or death; on the other hand, the system that enables the rear light or the fuel light - to advice about the low fuel level - have, in general, lower criticality attributed. The critical tasks are the ones in which a failure can lead to very harmful consequences either for the system itself or for its surrounding environment and humans. Because of these consequences, safety-critical systems are usually regulated by standards in which minimum safety requirements must be obeyed. For example, the ISO 26262 standard [44] regulates the functional safety assurance for road vehicles. It classifies the tasks of an automotive system in different safety-assurance levels according to parameters like: how important is the task for the system, how harmful can be the consequences of a failure, and the probability of a failure occurring. And depending on how the tasks fit into these parameters, maximum failure rates must be guaranteed. Furthermore, it is not seldom that critical and non-critical tasks share the same hardware platform, and are controlled by the same Operating System (OS), in which these tasks are only different processes within this OS. Therefore, this defines the mixed-critical scenario within the safety-critical application domain, which is intended to be *the context of this thesis*. A scenario like this can be found in multiple application fields such as aerospace, medical, and automotive industries. An electronic medical device, for example, can have life monitoring functions such as temperature and heart rate monitoring, as well as mundane and less critical functions, such as rendering the graphical user interface in a display, running in the same hardware platform, the same processor, and being controlled by the same OS.

Because embedded platforms within the safety-critical domain are usually designed for the worst cases in their application set, the aforementioned mix of critical and non-critical applications can lead to overestimated hardware and unnecessary use of power or hardware resources. Furthermore, it is not unusual that the critical functions of a mixed-critical system (e.g., the critical processes within an OS) run only for a small fraction of time, or they are only triggered by a special event. For example, in a vehicle, the airbag system is controlled by an electronic central unit that controls most of the electronic systems within a car, and the airbags will only be activated once a frontal crash is detected, in the remaining time, which is most of the lifetime, this central unit will be controlling other electronic functions of the car. Therefore, since redundancy is one of the most common ways to improve fault tolerance for the most critical functions in a safety-critical system, a system with redundant schemes always enabled would be wasting energy and hardware resources while executing its non-critical functions. To cope with that, this thesis proposes an approach that monitors the criticality of currently running tasks in a system and applies, at run-time, a proper redundant (or not) scheme. For example, it is possible to enable all redundant modules when the current criticalities of the tasks are high, or disable the redundancy in case only mundane functions are being performed. Such a strategy avoid unnecessary use of energy and hardware resources. Tackling this problem using such a strategy is one of the objectives of this thesis. This strategy was implemented within an approach in which the higher levels of the systems (e.g., the software and the OS level) hold the criticality knowledge and, with specifically implemented functions, properly manage the redundancy scheme throughout an implemented hardware infrastructure.

Run-time redundancy management for improved fault tolerance has been used in many domains and levels. For example, reconfigurable devices such as Field Programmable Gate Arrays (FPGAs) and Coarse-Grained Reconfigurable Arrays (CGRAs) enable from very fine-grained primitive elements reconfiguration (e.g., Lookup Tables (LUTs) and flip-flops) to processing elements, which can be a simple Arithmetical Logical Unit (ALU) or a complete capable processor core [29, 55, 104]. When it comes to processor designs, such redundancy management is mostly done at the core level [7, 87, 46]. However, other approaches using Very Long Instruction Word (VLIW) and superscalar processor architectures enable redundancy management at the level of their internal Functional Units (FUs) such as ALUs, multipliers, and dividers [68, 86]. These internal units can cover a big portion of the hardware area of a processor design. For example, for the RISC-V core CV32E40P (formerly known as RI5CY), approximately 40% of its area is covered by FUs [85], meanwhile for the Plasma processor core [75] (which is the one that is going to be used as a test vehicle in this dissertation), when disregarding its register file, the remaining 60% of its area, approximately, is covered by FUs. Therefore, if a fault-tolerant method is applied to these units, it is indeed true to say that a big portion of the design is being protected.

Furthermore, FUs are very easy to manage within pipelines because they are not used as storing elements, and, their control logic is very contained on its own. With such a property, it is possible to, for example, disable an unnecessary FU without flushing the whole pipeline of the processor in which the unit is inserted. Therefore, redundancy management within these units can be very agile and be just a matter of a few clock cycles. Moreover, still because of this agile property, once multiple units are available, it is possible to use them also for wear levelling strategies like, for example, performing load balancing among the FUs. Or, additionally, in case the architecture of the processor allows it, it is even possible to enable these additional units in parallel for performance improvements.

Moreover, it is known that application programs have different characteristics depending on their purpose. Based on these characteristics, some internal units of a processor design can be overused, meanwhile, others can stay in an idle state for a long time [90]. Additionally, for superscalar processors, for example, depending on the FUs allocation policy, certain units can be prioritised for execution, thus, leading again to non-homogeneous usage of the hardware resources [88]. Such heterogeneous usage causes as well a heterogeneous ageing of these units. However, it is indeed known that equalising the utilisation of FUs in a processor core can improve its lifetime reliability [37]. And, addressing this problem, multiple works are found in the literature proposing different methods for clock gating, power gating and Negative

Bias Temperature Instability (NBTI) aware instruction scheduling within processor designs [72, 73, 88].

The problem of the current approaches that are performing redundancy management at the FUs level is, for example, for the work done using VLIW processors, the run-time management is very compromised. Because, despite having multiple FUs, these processor architectures are very simple, and all the arrangements to distribute the operations throughout the available units are done before run-time by the compiler and included in the instruction words. Secondly, when a superscalar processor architecture is provided, although the dynamic scheduling of instructions throughout the FUs is then possible, current approaches do not provide a way to control the scheme in these units from the higher software levels, therefore they do not benefit from software level knowledge regarding the criticality of current tasks. Thereby, one of the aims of this thesis is to provide this interface that enables control of the processor's internal FUs from the software level, and manage, at run-time, these internal units scheme according to the criticality of tasks (or processes within an OS).

In summary, since the beginning of the down-scaling, electronic devices had increased their performance but also become more prone to fail. Additionally, with the advent of high-performance applications in the safety-critical domain, there is a need to improve the fault tolerance of these high-end devices so that minimum failure rates can be obeyed. Although redundancy has been a great solution for these problems, its disadvantages such as power and area overheads must be carefully considered, so that the price per unit does not increase beyond affordable limits, and the redundancy does not add more sources of error than it improves the fault tolerance. Therefore, looking for a compromise between fault tolerance, hardware area overhead, power and hardware usage, the redundancy at the FUs level seems to be a great choice because of their agile configuration properties, which can be very useful to avoid waste of power and hardware resources within the context of mixed-critical applications. Thus, the redundant configurations can be reserved only for the most critical applications, while in the remaining time other schemes can be used over the available units (e.g., for wear levelling or parallel schemes for performance improvement).

To conclude, *this thesis pushes the state-of-the-art presenting an approach for run-time management of redundancy among processor internal functional units, tackling to compensate the trade-offs between fault-tolerance, power consumption, hardware usage (ageing), and hardware area (cost).*

1.1 Objectives and Contributions of this Thesis

With all have said, and within the *mixed-critical application domain*, this thesis proposes a concept that enables, throughout an implemented hardware mechanism, software management of processor internal units towards improving the compromise between fault tolerance, power consumption, and hardware usage and area. Therefore, this theses has the following objectives:

- **Objective 1:** Enable fast and run-time software management of redundancy across internal units of a processor design.
- **Objective 2:** Increase fault tolerance, on-demand, of a device against soft errors, potentially caused by charged particles and neutrons hitting the silicon of an electronic device.
- **Objective 3:** Develop a processor design capable to enable and disable redundancy schemes for fault tolerance according to the criticalities of running processes. Thus, enabling the design to minimise additional power consumption and save hardware resources while not performing critical work.

Pursuing these objectives, this thesis presents a concept for a dynamic processor architecture capable to enable and disable redundancy on-demand of FUs. On the targeted system, the management of the units is done by the software (bare-metal or an OS) running on top of the platform, and the redundancy configuration is done according to the criticality levels of the tasks, or the processes in the case of the OS. This strategy, together with the hardware mechanism for fast FUs management, are the main contributions of this thesis, in which I aim to reach a platform that can increase and decrease, on-demand, its fault tolerance against soft errors (e.g., Single-Event Upsets (SEUs)), and at the same time, minimise power consumption and ageing while the platform is not performing any critical work.

1.2 Structure of this Thesis

After this introduction, the remaining chapters of this thesis are organised in such a way that Chapter 2 presents a review of the topic's state of the art. First, a background section reviews the main concepts and terms linked to this thesis. Thereafter, methods to handle reliability problems are shown, which goes from deep hardware to software and OS based methods.

In Chapter 3 the design concept for run-time management of processor internal FUs is presented. Targeting the mixed-critical scenario, this chapter shows the advantages of this fine-grained concept, as well as the possible foreseen configurations through the units. It also shows how a software (bare-metal or an OS) running over the intended platform can use the criticality information to enable or disable replication schemes of units towards improved trade-offs between fault tolerance, hardware overhead, and power consumption. Following this concept chapter, the details of the implemented test platform are presented in Chapter 4.

In Chapter 5 the implemented test platform is evaluated regarding its fault tolerance, hardware overhead, latency for the run-time configuration of the FUs, ageing, and power consumption incurred by the concept implementation.

Finally, Chapter 6 presents the concluding remarks about the whole approach. It evaluates the advantages and disadvantages of the presented concept within the intended mixed-critical scenario, and highlights whether the aforementioned objectives have been achieved. Furthermore, as future work, it also shows a preliminary study on the applicability, as well as its advantages, of this concept for a standard superscalar processor design.

CHAPTER 2

Domain Analysis

Since More's law started to set the pace of the electronics industry, and reliability problems started to emerge, a huge number of methods tackling these problems have been proposed. Techniques to improve fault tolerance can be found in all layers of electronic systems, from methods using low-level physical effects of electronic components, to high-level pure software approaches. In the next sections, we will first introduce and review important concepts, and later on, we will review the most relevant methods for fault tolerance within the context of this thesis.

2.1 Background, Concepts and Terminology

2.1.1 Dependable and Fault-Tolerant Computing

According to Avizienis et al., *dependability* of a computing system 'is the ability to deliver service that can justifiably be trusted' [5]. Alternatively, also by the same author: 'the dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable' [5]. The *service* is the behaviour as it is perceived by the user; a *user* is another system, physical or human, that interacts with the first; and finally, the *function* of a system is what it is intended for according to the system specifications [5].

A system delivers *correct service* when it implements its intended functions or behaviour. A *failure* is an event when the delivered service deviates from its correct service. A failure is usually a consequence of one or more deviations from the correct service system states. Such a deviation is called an *error*. Finally, the cause of an error is called a *fault*. If the fault becomes an error the fault is active, otherwise, it is dormant [5]. For example, if a charged particle hit an electronic device and generates a glitch or a fluctuation in its internal signals, we have a *fault*, if this wrong value propagates through the logic in such a way that it is saved in a register (flip-flop) or a memory cell, then the fault becomes an *error*. In case this error causes a wrong calculation or a system misbehaviour, the system fails to deliver its intended service, thus, a system *failure*. These failures can have multiple consequences, which can range from a simple calculation error to a total *system break-down*, in which the system stops working completely.

Regarding the ability of the system to attain dependability, a system is called *fault-tolerant* when it can avoid service failures in the presence of active faults (errors) - the ability to tolerate errors [5]. This ability can be intrinsic to the system, but it is usually enhanced with fault-tolerant methods to detect errors and correct them before generating a failure. Sometimes, these fault-tolerant methods can correct errors, but the system service suffers consequences, such as delaying its response to its intended service. Since this delay is planned, we do not call it a failure, but in these cases, we say that we have a *degraded service*.

The usual metrics to measure the dependability of a system quantify the time between correct and incorrect service. The *availability* of a system measures the *time between failures*, and the *reliability* measures the *time to failure*. The difference between these two metrics is that the last one considers the time at the beginning of the lifetime at instant zero ($t = 0$) [5]. Therefore, we can say that the reliability $R(t)$ of a system is the probability that it will deliver the correct service in the time interval $[0, t]$, given that it was performing correctly at instant zero ($t = 0$).

2.1.2 Soft Single-Event Effects

Faults and subsequent errors can appear on any electronic device. This thesis tackles problems specifically concerning soft Single-Event Effects (SEEs). Soft SEEs are events induced by a single radiation event [8], they are usually called soft errors in the literature, and they are called *soft* because the event does not cause permanent damage to the silicon of an electronic device, thus not a permanent error [8].

Soft errors can happen either in space or on Earth at ground level. In space, charged particles such as protons and alpha particles, and at the ground level, particles such as neutrons can hit the silicon area of an electronic device and accumulate enough energy to change or flip the logic level of a design element [8].

In regards to the ground level, highly energetic cosmic rays interact with the Earth's atmosphere and produce a complex cascade of nuclear reactions. In this cascade of reactions, secondary particles are produced, these secondary particles react again with the atmosphere creating tertiary particles, and this process continues until these particles reach the Earth's ground. It is mostly the sixth generation of particles, which represents less than 1% of the primary flux, that will reach the sea level where this flux is composed of muons, protons, neutrons, and pions. The neutrons are the ones with higher flux, and because neutron reactions with the silicon nucleus release ions with significantly higher energy, they are the most likely particles to cause effects in the silicon of electronic devices at the ground level [8, 32]. According to the Joint Electron Device Engineering Council (JEDEC) in its technical report JESD89A, it is expected that neutrons hit the Earth's ground at a rate of $13 neutrons/(cm^2 * h)$ at sea level [48]. Therefore, taking into consideration the enormous amount of electronic devices of today, exceeding the billion mark scale as reported by the International Roadmap for Devices and Systems (IRDS)™ on its 2021 report [42], such a rate *can not be neglected*.

When such a particle hits the combinational logic of an electronic device, it can cause a glitch or a temporary fluctuation in the logic value of a signal, thus generating a Single-Event Transient (SET). Furthermore, when these particles accumulate enough energy to change the state of, or the induced transient value is captured by, a logic device such as a latch, a flip-flop or a memory cell, we have a Single-Event Upset (SEU) [8]. These soft effects are the ones tackled in this thesis.

2.1.3 Mixed-Critical Systems

Criticality is the level of assurance against failure needed by a certain component of the system [99]. A mixed-critical scenario is one with multiple components (e.g., tasks or processes) of different criticalities running on a common hardware platform. A big portion of common and complex safety-critical systems of today - such as the automotive and avionics industry - are evolving into mixed-critical systems. Such evolution is mainly motivated by non-functional requirements such as power consumption, cost, and device area [12]. Furthermore, software standards, such as the AUTOSAR from the automotive industry [4], do address the mixed-critical domain.

Regarding the different levels of criticality, errors can cause different consequences within the safety-critical systems. Moreover, errors can cause dangerous consequences not only to the device's integrity but also to its surrounding environment and people. Therefore, standards regulate minimum safety requirements within this application domain (e.g., IEC 61508 [38], DO-178C [16], DO-254 [17] and ISO 26262 [44]). For instance, the ISO 26262 rules the automotive industry regarding functional safety. It states different levels of safety requirements - the Automotive Safety and Integrity Levels (ASILs) - with minimum failure rates according to parameters such as severity, time of exposure, and controllability of hazards. For example, systems like airbags, anti-lock brakes, and power steering require the highest degree of safety assurance because the risks associated with their failure are the highest. On the other hand, components like rear lights fit into the lowest level of safety requirements [94].

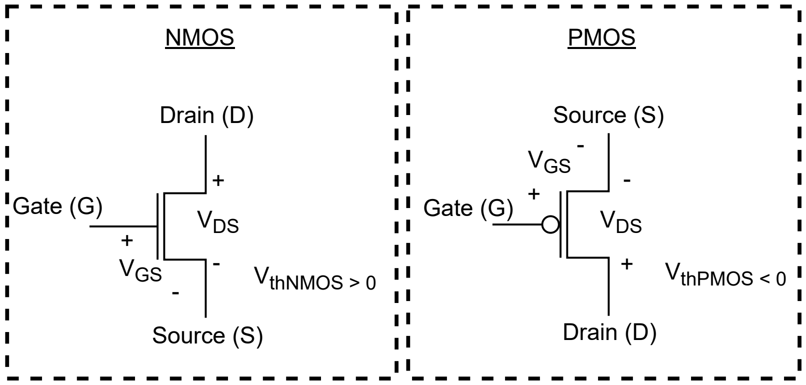
2.2 Basic Background on CMOS Transistors

Before looking into very specific details of electronic devices, it is important to explain the basic element that forms these electronics considered in this thesis: the Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET), or only MOS, for short. This is the main transistor type used in electronics of today, and in its majority, the Complementary MOSFET (CMOS) technology is used. Such technology uses the two complementary types of devices, the p-type MOSFET (PMOS) and the n-type MOSFET (NMOS).

Simplifying the understanding of these devices for simple switches, once a certain voltage level is applied to the *gate* terminal, current starts to flow between its *source* and *drain* terminals (a fourth terminal may also be present, and it is connected to the body of the device).

Figure 2.1 illustrates the two CMOS devices with their terminals and their main relational voltages, as well as their operating modes depending on the voltage applied to their terminals. To behave as an ON switch, the modulus of the voltage between the gate and source ($|V_{GS}|$) terminals must be higher than the modulus of the *threshold voltage* ($|V_{th}|$) of the device. Here the transistor is on its *linear regime*, so the current in the *drain* node (I_D) is linearly proportional to the modulus of the voltage between the *drain* and *source* terminals ($|V_{DS}|$). On the other hand, it is on its OFF (*cut-off*) state, when $|V_{GS}|$ is lower than the $|V_{th}|$, therefore the drain current is ideally zero. Furthermore, once the $|V_{DS}|$ reaches a certain value, the transistor enters on its *saturation mode*. This means that the I_D saturates, and even if increasing the $|V_{DS}|$, the current will remain ideally the same.

Please, note that the behaviour just described is regarding an ideal transistor. In real transistors, leakage currents are observed even when they are in the OFF state. However, this analysis goes beyond the scope of this thesis.



Cutt-Off	$\lvert V_{GS}\rvert < \lvert V_{Th}\rvert$
Linear regime	$\lvert V_{GS}\rvert \geq \lvert V_{Th}\rvert$ $\lvert V_{DS}\rvert < \lvert V_{GS}\rvert - \lvert V_{Th}\rvert$
Saturation	$\lvert V_{GS}\rvert \geq \lvert V_{Th}\rvert$ $\lvert V_{DS}\rvert \geq \lvert V_{GS}\rvert - \lvert V_{Th}\rvert$

Figure 2.1: Illustration of an NMOS and a PMOS transistor and their operating modes.

2.3 Background on Ageing of Integrated Circuits

Among the many effects and physical phenomena that increase ageing in a transistor, the most prominent ones that cause ageing by decreasing the switching speed of transistors are the Bias Temperature Instability (BTI) and Hot Carrier Injection (HCI). These two effects are particularly of interest because they gradually increase the transistor's threshold voltage - the necessary voltage applied to the transistor's gate pin starting the current flow between its source and drain pins -, therefore, decreasing the switching speed of the affected transistors, and consequently, the design in which they belong.

In the remainder of this section, the BTI and HCI effects are explained, and how these relate to ageing and the lifetime of electronic circuits.

Hot Carrier Injection

HCI is a phenomenon in which charge carriers with high kinetic energy (*hot*) travelling through the transistor channel gain enough energy to be injected (or collide with other atoms generating further particles that can be injected) in the dielectric layer of the transistor's gate (the gate-oxide). Such a phenomenon alter permanently the properties of the affected transistor, in particular, it increases the modulus of its *threshold voltage* ($|V_{th}|$) [1, 62, 92].

These highly energetic carriers are caused by strong electric fields in and around the transistor channel created by the potential difference between the transistor's drain and source. While in the linear regime this electric field is very small, in the saturation mode this electric field is very strong, therefore, the HCI effect becomes very prominent in this mode, causing irreversible effects in the affected transistor [92].

In CMOS based electronics, during normal operation, the transistors enter, for a short time, in saturation mode when the output of its respective gate is switching its logic level [57]. Therefore, the HCI effect is strongly related to the number of transitions (switching frequency) observed in the output pin of the corresponding gate of the evaluated transistor. Furthermore, the generated electric field in the transistor, and therefore the HCI effect, are also strongly related to the temperature and the supply voltage.

Bias Temperature Instability

BTI is a physicochemical phenomenon that causes wear out of transistors by affecting their *threshold voltage*. As its name says, it is a phenomenon caused by a certain bias in the transistor state and intensified by higher temperatures. It can be divided into two similar phenomena: Positive Bias Temperature Instability (PBTI) affecting the NMOS transistors, and Negative Bias Temperature Instability (NBTI) for the PMOS transistors [51]. Although the BTI effect is now known for years, the entire underlying physical and chemical processes are not completely understood. Since this phenomenon is happening inside transistors at the nanometer scale, it cannot be directly observed, and only indirect explanations based on measurements are available. These indirect explanations are very debated in the community, but the two of them, based on the *trapping-detrapping* [98] and in the *reaction-diffusion* [101] processes, are the most accepted by the community [91].

In both theories, the BTI phenomenon can be divided into stress and recovery phases. The *stress phase* happens when the transistor is in the ON state, which means that current is flowing in the linear regime between its *drain* and *source* pins. In this situation, among various physicochemical processes, the modulus of the *threshold voltage* ($|V_{th}|$) of the affected transistor increases. On the other hand, when the transistor is in the OFF state, we have the *recovery phase*. In this phase, the process that happened in the stress phase affecting the transistor's *threshold voltage* can be partially recovered, thus getting almost back to its previous threshold voltage condition. However, as it was just mentioned, the transistor is not fully recovered in this last phase, which causes the long term effects on the transistor operation, increasing the base level of its $|V_{th}|$. Therefore, the *threshold voltage* shift in the BTI effect is very dependent on the duty cycle between these stress and recovery phases.

Finally, it seems to be a consensus in the community that the *trapping-detrapping* process is responsible for the large and very fast modifications in the *threshold voltage* during stress and recovery phases. Meanwhile, the *reaction-diffusion* process is responsible for the gradual and long term effects [31], which are the ones of concern in this thesis.

Path Delay in Electronic Circuits

Combinational logic circuits are the ones with the property that at any point in time, the output of the circuit is related to its current input signals by a certain Boolean expression. These circuits are essentially formed by digital gates such as NAND, NOR and INV [78], and multiple different *paths* are possible from its inputs to output signals. Each of these paths goes through different gates, and each gate has its own propagation delay - the minimum time a signal must hold its state on the input pins of a gate so that it propagates thoroughly, reaching its outputs. Therefore, each *path* has its particular propagation delay based on the sum of the delays throughout its multiple gates, among also other factors such as wires/connections length. This path propagation delay is, sometimes, simply called as *path delay.* And the path throughout a combinational logic with the biggest delay is called the *critical path* of an electronic circuit.

In clock-driven electronic designs, the critical path delay plays a big role. Since the clock period determines the amount of time a signal will hold a certain value, this period should be at least as big as the critical path delay. Therefore, it is possible to say that the critical path delay, basically, determines the maximum operating frequency of a clock-driven electronic design. In the case of a processor core, the critical path is most likely located in between its pipeline stages, and its operating frequency will be optimised to reach (usually with a certain positive margin) the time required by this critical path delay.

From Path Delay to Ageing and Lifetime

Finally, effects caused by the BTI and HCI phenomena mentioned above can increase the threshold voltage, and consequently, decrease the switching speed of transistors, therefore, increasing the propagation delay of gates, and paths in which these affected transistors are included within a digital electronic device.

Since the path delay is strongly related to the maximum operating frequency of a clock-driven electronic device. In case the delay of a degraded path (e.g., affected by BTI and HCI) surpasses the clock period, it can directly affect the functioning of the device. Therefore, further techniques are needed to keep the electronic device, such as a processor, working. For example, in case of appropriate technology is available, it is possible to decrease the operating frequency of the processor, or the particularly affected block. As a result, the critical path delay can, again, fit within the clock period. However, such a decrease in frequency would, consequently, lead to performance degradation of the device. Furthermore, other techniques are also available to tolerate such an increase in the delay of critical paths [27, 30, 52, 64, 66]. However, in case none of these techniques is implemented, such a situation can potentially result in an *end of life* of the whole device or, at least, of the respective affected electronic block.

To Conclude

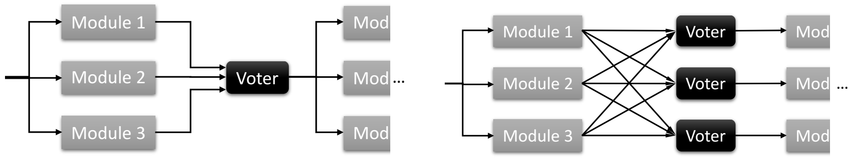
It was possible to notice that BTI and HCI phenomena do increase the $|V_{th}|$ of transistors depending on various parameters, but most importantly, on temperature, supply voltage, switching frequency, and duty cycle. Some of these parameters are very dynamic and payload dependent at run-time. For example, the individual switching activity and the duty cycle of the internal transistors of a circuit can be very different from the global clock frequency and duty cycle. Therefore, the degradation effects caused by these two phenomena can vary for different payloads, and directly affect the maximum operating frequency of electronic circuits shortening their lifetime, or, at least, decreasing performance. Hence, the study case presented in this thesis on its respective ageing evaluation (Section 5.5).

2.4 State of the Art Analysis

Methods to increase fault tolerance are found at different levels throughout the electronic systems stack. The next sections will go through these levels and present standard techniques, as well as the ones most relevant to the domain of this thesis.

2.4.1 N-Modular Redundancy

Back in the 1950s/60s, John Von Neumann envisioned the use of redundant system modules to make systems tolerate faults. He first proposed the Triple Modular Redundancy (TMR), a scheme with triplicated system modules and an additional voter (Figure 2.2a). The working principle of this scheme is that in case one of the modules produces a wrong output, the other two modules will likely remain working correctly and produce correct results on their output signals. Therefore, this enables the voter to *filter out* the wrong result by majority voting, and propagate only this value produced by the other, supposedly non-faulty, two modules.



(a) TMR scheme as it was envisioned by John Von Neumann.

(b) TMR scheme with triplicated voters.

Figure 2.2: TMR schemes.

The internal logic of a very simple voter is shown in Figure 2.3, the AND ports only produce logic level *one* on their outputs when their both two inputs have logic level *one* as well, otherwise, they produce logic level *zero* on their outputs. The OR port produces logic level *one* on its output when any of its three inputs are in logic level *one*, it will only produce logic level *zero* when all of its three inputs are in logic level *zero* as well. As explained before, the voter is capable to filter out one input signal that is not matching the other two inputs. Figure 2.4 shows two examples of non-matching inputs, the intermediate values, and the respective output values. Note that the output signals follow the rule just mentioned, filtering out the non-matching value of the inputs. Furthermore, Table 2.1 shows the complete truth table for all possible digital inputs and their respective output values for such a voter. A similar TMR scheme with triplicated voters is also proposed in the literature (Figure 2.2b). This one is based on the same principle as the simple TMR, but, in this case, a fault in one of the voters would also be covered by the triplication scheme [58].

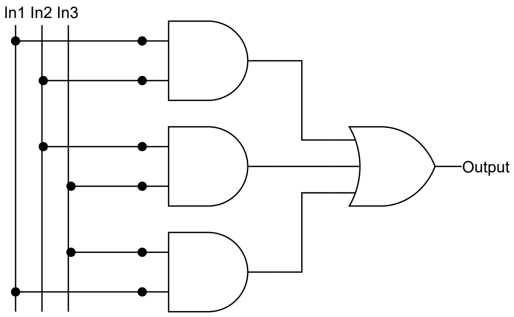
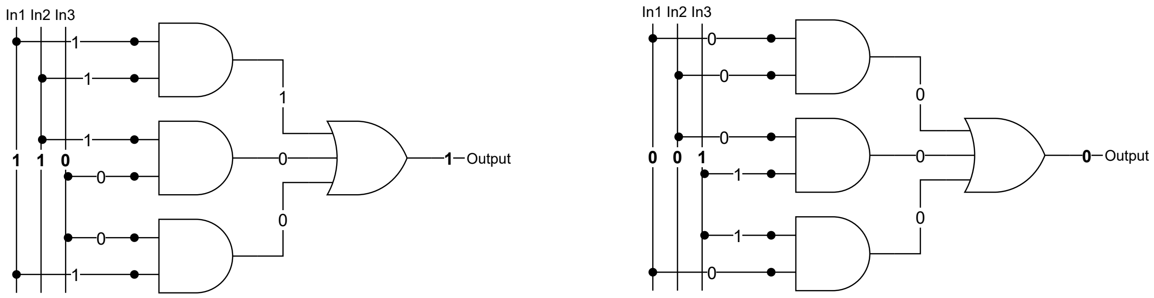


Figure 2.3: The internal logic of a simple voter.



(a) Non-matching inputs example 1. **(b)** Non-matching inputs example 2.

Figure 2.4: Examples of a non-matching input and its respective output value.

Table 2.1: The truth table for a general three inputs voter.

Input 1	**Input 2**	**Input 3**	**Output**
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
1	0	0	**0**
0	1	1	**1**
1	1	0	**1**
1	0	1	**1**
1	1	1	**1**

Since the first TMR scheme was envisioned, multiple schemes using redundancy have been proposed to increase the fault tolerance of electronic systems. Variations such as Double Modular Redundancy (DMR), with only duplicated modules plus a comparator, enabled fault detection instead of correction, but with less overhead. Therefore, the name of the technique was generalised by some authors to N-Modular Redundancy (NMR), where N stands for the number of replicated modules.

Regarding the replication of modules, not only spatial but also temporal replication is possible. For example, in the spatial model, the voter is mostly done in hardware and the physical components are replicated (e.g., transistors, flip-flops, processor units, and cores). Therefore, the spatial replication has hardware area as its main penalty, and a very low, if any, execution time overhead. Furthermore, because of the physical replication, this scheme would continue working in case one of its redundant modules becomes permanently faulty. On the other hand, in the temporal replication schemes, instead of replicating the hardware modules physically, the calculation is the one performed redundantly over the time, one after the other, using mostly the same hardware. In this case, the control of the replication is usually done in software with no dedicated hardware to perform the majority voting. Therefore, the temporal approaches present a much lower area penalty, if any. In contrast, since the replication is distributed over time, these schemes usually result in a bigger execution time. Moreover, the temporal scheme would stop working in case its only one physical module becomes faulty, thus, a redundant scheme that does not protect against permanent faults.

In the work presented in this thesis, the spatial replication scheme is used. It means that this work benefits from the very low execution time overheads and, although not the main goal tackled in the thesis, from the protection against permanent faults.

2.4.2 Pure Hardware approaches

Pure hardware approaches for fault tolerance are found in many levels of granularity. From transistor and gate-level approaches, to complete processors with multiple fault-tolerant techniques applied to their components. At transistor and gate-level, for example, optimised versions of flip-flops were designed for better resilience against ageing effects (such as BTI) and supply voltage fluctuations. These flip-flops were then integrated into an electronic circuit design replacing only its most critical flip-flops (e.g., time-critical flip-flops on the design's critical path) [30]. Still in the same abstraction level, a multiple-input hysteresis flip-flop (usually called C-Element), whose one output only changes when all of its inputs agree, is used to increase fault tolerance over electronic designs. Although it was originally used in the design of asynchronous circuits, in combination with a delay line and a comparator, it can be used to detect and correct SETs, and also to detect increased propagation delays of the logic in front of the C-Element [52, 64, 66]. Once more at gate-level, a flip-flop based datapath was converted into a two-phased latch based one [27]. Targeting the detection and correction of timing errors (possibly caused by ageing), as well as avoiding hold time issues and enabling larger speculation windows, the authors replaced each flip-flop of the pipeline using two-phased latches. These latches are phased with each other in such a way that when one latch is open, the other two proximal ones (the preceding and the previous one) are already closed. Such a replacement enabled an increase in the speculation window for possible late signals to reach the open latch throughout the current datapath. Therefore, possible and eventual timing errors in the propagation of the signals through the datapath are corrected. However, although these methods do improve fault tolerance, they are usually static, and no optimisation can be performed at run-time.

Another common method to make electronic designs tolerate faults and correct errors is the use of Error Correction Codes (ECCs) on storage elements such as memory cells and registers. ECC methods can be embedded directly into the memory devices, or also into the control logic of processor designs. For example, the fault-tolerant versions of Leon3 and Leon4 processors have Error-Detection and Correction (EDAC) units that use ECCs to correct possible errors on their internal buffers and registers [28]. Once more, ECC is usually applied permanently to storage elements within processor designs. Therefore, there is usually no dynamic configuration of modules.

2.4.3 Software Approaches and Dependable Operating Systems

Going further in the electronic systems' stack, we can find software-based methods for fault tolerance. In this domain, stand-alone approaches work independently of an Operating System (OS) for example; or also other approaches that are applied directly into the kernel of OSs and other ones that simply run on top of it as user applications.

In the work developed in [79], the authors developed a replica-aware co-scheduling mechanism for mixed-criticality systems. Such a mechanism explores the co-scheduling to provide short response times for replicated tasks without affecting the remaining unprotected tasks. The co-scheduling consists of scheduling interacting tasks/threads to be executed simultaneously in different cores, which allow these to communicate more efficiently by reducing the waiting and blocking time during synchronisation. Therefore, the time needed for synchronisation and comparison of the replicated tasks is reduced. A specialised layer is introduced in the OS to manage these replicated executions and the check-pointing and rollback of the protected applications [18]. Therefore the replication turns to be transparent to the applications, and the Processing Elements (PEs) (e.g., the processor cores) can just perform the computations of the assigned tasks as usual. Furthermore, with such a system the applications that are not critical can simply run as usual tasks/threads without the need to be co-scheduled.

Regarding fault-tolerant OSs, there is a good amount of work done and literature proposing different approaches and systems. For example, professor Tanenbaum and his chair have developed Minix 3, an open-source OS designed to be very reliable [35]. A part of other components for processes monitoring, it is mainly based on the microkernel approach that only a tiny portion of the system is running in kernel mode, and the rest of the OS runs as isolated and protected processes in user mode, in which an error would not compromise the whole system, but only its own process [35]. Therefore, it considerably decreases the time spent on sensitive software in which an error could potentially endanger the whole system activity.

In the work [36], the authors proposed the dOSEK, a Real-Time Operating System (RTOS) dedicated to safety-critical systems. It is strongly based on two pillars: strict fault avoidance, by static tailoring of the kernel towards a concrete application and hardware platform, thus minimising vulnerable runtime states; and redundancy integration for fault detection and containment within the kernel execution path, by employing arithmetic encoding to realise control-flow errors across the RTOS execution path.

Yet, other commercial RTOSs aiming for safety-critical systems can be found, such as the SAFERTOS [102] and Erika Enterprise RTOS [21]. The most common approaches to avoid failures in these RTOSs are the following: the restriction of dynamic memory operations, therefore, avoiding run-time memory problems such as memory leaks and memory overloads; the manipulation of Memory Protection Units (MPUs) to provide task granular separation of memory addresses, therefore avoiding unintentional or accidental access to incorrect memory regions, which can potentially lead the whole system to crash. Furthermore, another usual feature is the definition of safety cores within a multi-core environment by the RTOS. In these configurations, the system is partitioned in such a way that the non-safety cores do not interfere with the safety ones. Thereby, the safety cores can run their safety-critical functionalities as well as the monitoring and verification of the other non-safety cores, while these last ones can transparently run any other mundane application.

2.4.4 Dependable Processor Designs

Regarding complete processors, while some designs disable certain internal elements of their pipeline so that critical instructions spend less time in unprotected internal buffers [47]. Other works applied a coarse-grained redundancy approach, such as the lock-step processors [41, 46, 89]. These processors have their cores running in parallel and executing the same instructions redundantly, and an additional combinational logic is responsible for keeping it synchronous, comparing the core outputs, detecting, and possibly, correcting faults. Further comparisons with these lock-step processor approaches will be made in the evaluation section of this thesis. Therefore, Section 2.4.5 goes more into the details of these techniques. Other designs use dynamic scheduler architectures based on Tomasulo's algorithm [34]. These designs benefit from the multiple Functional Units (FUs) available on their architectures so that different operating modes are proposed to tackle fault detection and fault correction using double or triple re-execution of instructions over the processor FUs [67, 68].

In the work [54], the authors proposed a reliability-heterogeneous architecture. This architecture offers different types of reliability modes in different cores. For this, each core has different stages of its pipeline hardened with different mitigation techniques for dependability threats. The final and definitive set of cores of the architecture is defined at design-time and optimised for the application scenario. At run-time, an adaptive manager is used to estimate the reliability requirements of the applications and map their threads to a set of hardened cores.

Still regarding fault-tolerant processor designs, the author of [25] developed a fault-tolerant processor design based on RISC-V Instruction Set Architecture (ISA). The target of the work is to enhance the execution stage of the pipeline of the design with fault-tolerant methods. To detect and correct both transient and permanent errors, the author applied TMR in the executions units of the design. Furthermore, the author also added a fourth unit allowing the system to maintain the TMR scheme in case one of the units becomes permanently faulty. In addition, the voter was modified to not only correct errors but also to detect them, therefore the scheme can account the number of errors detected in each of the execution units, and once a predefined threshold is reached, the unit is considered to be in a faulty state. With such methods, the system can detect and correct both transient and permanent errors, and track the current state of the execution units. However, no link with higher software levels is provided.

Complete systems, from software to hardware are also proposed. For example, the authors of [87] presented a fault-tolerant self-aware platform. A multi-core processor with tightly coupled monitoring infrastructure. Such infrastructure monitors not only physical parameters (such as temperature), but it also tackles fault detection and correction in the internal elements of the processor such as its control registers. Furthermore, a system health map is maintained with the information provided by the hardware monitors, and an OS running on top, uses the data from this database to schedule tasks through the cores of the processor design.

2.4.5 Core Lock-Step Processors

Core Lock-Step processors are designs with their cores running in parallel performing the same computation. The simplest version of this approach is the Dual Core Lock-Step (DCLS) processors, these are the mainstream solution for terrestrial safety-critical applications, and it is possible to find them in many Commercial off-the-shelf (COTS) System on Chips (SoCs) [3, 95]. The working principle of this scheme is strongly based on the DMR concept, in which two cores run in parallel redundantly to each other, therefore performing the same computations redundantly. An extra hardware mechanism is responsible for comparing the outputs of these redundant cores and issuing an error signal in case these output signals present different values. In the DCLS scheme, the recovery from detected errors is mainly done using software approaches with checkpoints and rollback strategies [100]. It means that assumed error-free checkpoints are created systematically when no difference is detected in the output of the cores. When a difference is detected, a signal is raised by the fault detection mechanism, and a recovery process can start rolling back the processor states to the last saved checkpoint. However, check-pointing and rollback are expensive processes with big penalties in execution time. Although it is very dependent on the amount of data to be saved in the checkpoints, recent work shows execution time overheads ranging from 17% to 53% [74]. Furthermore, the recovery process of rolling back the processor states may also vary, but, depending on the architecture and the amount of data to be retrieved, it can take up to a few seconds [82].

Another version of this approach is the Triple Core Lock-Step (TCLS) scheme. Here, instead of two, there are three cores performing parallel computations redundantly to each other. Thus, a TMR scheme with a voter is observed across the cores that allows this design to perform at least fault correction at run-time without bigger penalties in execution time. A good example of a TCLS design is the one presented by Iturbe *et al.* [46]. In this design, the system uses the TMR scheme to correct errors in the output ports of the processor cores. In this design, the memories are detached from the cores, and every data goes through the TMR voter before going to the data cache. Therefore such a scheme prevents an erroneous value to be saved in memory. Furthermore, an additional hardware is responsible to detect non-matching outputs between the cores, and in the event of an error detection, the processor can still keep itself running with the remaining cores (the two non-faulty ones) until a proper time is available to recover the faulty core. Here, the recovery mechanism halts the cores, saves the internal states of the non-faulty ones, resets them all, assigns the saved states to the three cores and resumes the execution. Because of the TMR, and the detachment of the caches from the cores, the caches are assumed to be safe from the detected error. Therefore, there is no need to create checkpoints, and no execution time overhead is expected for normal operation. Also, in the case of an error detection, the recovery procedure is optimised, because the cache memories (data and instructions) are decoupled from the cores, only their internal states need to be retrieved.

In another variation of these DCLS and TCLS approaches the redundant cores are delayed (unsynchronised) by a few clock cycles (usually one or two instruction cycles) from each other (Figure 2.5). In this way, a possible error, caused by a fault event with a smaller duration than the lockstep delay, affecting both of the cores at the same time would not affect the very same calculation. Therefore, the comparison, or the majority voting, would still detect or correct these errors.

Although these approaches present high protection against soft errors, the resulting hardware overhead is considerably high. Because of the entire core replication and its additional control logic, plus the recovery procedures, these schemes tend to present an area increase beyond 100% for DCLS and 200% for TCLS. In fact, the TCLS scheme proposed in [46], resulted in 218% of total area overhead, 200% for the cores and 18% for the control logic. Such a large area overhead tends to increase power consumption, generate more heat, and, as consequence, increase the ageing of the electronics. Therefore, for most of the application domains, and especially for embedded systems with power restrictions, such an overhead is very undesirable. Therefore one of the main goals of this thesis is to *decrease the resulting area overhead* while still *increasing the fault tolerance* of the processor design.

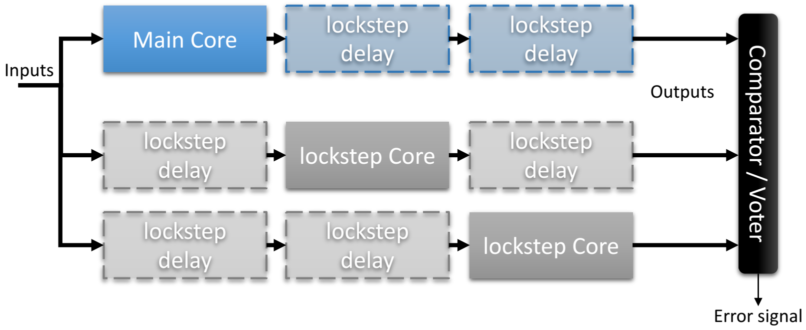


Figure 2.5: Example of a TCLS scheme with delayed cores.

2.4.6 Very Long Instruction Word and Superscalar Processors

Very Long Instruction Word (VLIW) and superscalar processor architectures present multiple copies of their execution paths. Usually, each of these execution paths presents multiple FUs such as Arithmetical Logical Units (ALUs), multipliers, dividers, and others. In the VLIW architecture, the instruction-level parallelism is provided by these multiple execution paths. However, these are very simple processors, and there is no logic for dynamic allocation of instructions over these paths, thus, the instructions must be arranged before run-time to enable the desired parallelism. Therefore, these processors have very large instructions words with multiple individual operations bundled together in only one instruction, and each operation is then executed by its corresponding execution path (Figure 2.6). These instructions are created by very sophisticated compilers with very little help, if any, from the programmer [26]. On the other side, the superscalar processor architectures usually have multiple components, for instance, register files, instruction queues, reservation stations and instruction schedulers or dispatchers to enable dynamic allocation of instructions to be executed in each of the available FUs [34, 93]. The parallelism provided by both of these architectures is originally intended for performance improvement so that multiple operations or instructions can be fetched from memory at once and executed in parallel.

However, these architectures can also be used in the strive for fault tolerance. An example of such a design is presented in [86], in which a VLIW processor architecture is used to enable redundant execution of instructions by different hardware units. For such a strategy, the compiler duplicates all the software instructions, and with the help of additional control bits, schedules each duplicated instruction to a different execution path. Furthermore, the processor architecture is also extended to enable the comparison of the result of each of these duplicated instructions, with its original. In case the result does not match, the duplicated instruction is re-scheduled and executed again, and the third result is used for majority voting. Therefore, it provides fault detection and correction. However, to achieve such a state, all the software instructions must be duplicated in memory. Furthermore, the scheduling of parallel and/or redundant instructions must be arranged by the compiler before run-time, which can be affected by any change in the state of the hardware during run-time. For example, if one of the units becomes permanently faulty, the mentioned mechanism can still re-arrange the execution of the instruction in other units, but at the price of a bigger run-time overhead.

Another example is presented in [68], in which the authors presented a superscalar processor architecture based on Tomasulo's algorithm [34]. It uses its multiple functional units not only for performance purposes but also, to enable redundant execution of operations. When required, the design can execute the same instructions twice using different hardware units, later on, the results of these operations are compared. If these results differ, a third execution of the instruction can be triggered, thus enabling fault correction by majority voting over the three results.

The problem of these approaches, performing dynamic or static scheduling of instructions for fault tolerance, is that they do not usually explore the knowledge available at the software levels, and neither provide a clear way (e.g., a mechanism or an interface) on how to do that. This means that high valuable knowledge of the criticality of the running tasks or processes, usually only available at the higher software levels, is not used for the scheduling of instructions among the units.

To conclude, taking into consideration the multiple functional units already available in the superscalar processor architectures, these turn out to be good candidates for the approach proposed in this thesis (further evaluation on this is presented in Chapter 6, Section 6.1). Furthermore, this thesis still pushes the state of the art by proposing an approach in which the criticality knowledge available from the higher software levels is used to organise at run-time the execution of instructions over internal processor FUs. Thus, evolving from the VLIW architecture based schemes enabling a run-time strategy for FU allocation. And from the usual superscalar based approaches enabling software level decisions regarding the allocation of units.

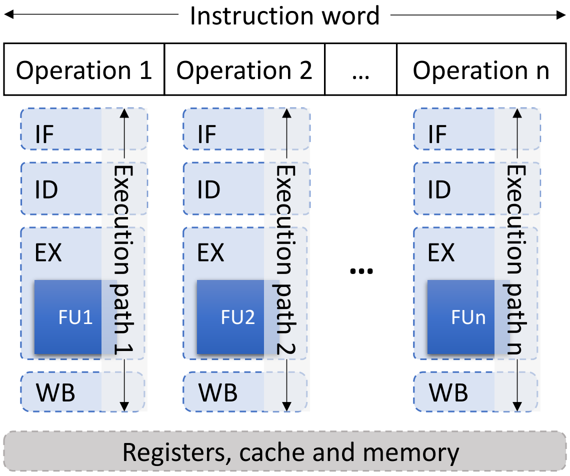


Figure 2.6: A simplified representation of a VLIW processor architecture with its multiple execution paths and bundled operations.

2.5 Summary

As it can be seen, fault tolerance is a topic that has been evolving together with electronics for decades. As more complex electronic systems tend to be, more robust and well planned the fault-tolerant methods must be. Therefore, there is a good portion of work already done in most of the known subtopics. However, new methods are continuously needed as hardware technologies continue to evolve. Furthermore, regarding hardware and software, most of the approaches stay in their own domain. Thus, the hardware and software approaches are usually very separated, or not well integrated and evaluated.

Although improving fault tolerance using redundant modules is a common technique, not many approaches have tried to go at the level of processor internal units. Furthermore, from the best of my knowledge, I have not seen approaches providing redundancy at this level and enabling run-time software control of the desired redundancy scheme, therefore, providing means to avoid unnecessary power consumption and hardware wear out. If so, no proper evaluation on this was given. Hence, the approach presented in this thesis proposes a design concept, with a hardware infrastructure and software extensions, to enable run-time management of multiple (redundant or not) internal processor FUs (e.g., ALUs, multipliers and shifters).

CHAPTER 3

The Design Concept for Functional Units Management

As introduced in the first two chapters of this thesis, the technology of electronics is constantly evolving reaching feature sizes of $3nm$ and below. As more complex and smaller an electronic system becomes, the more prone to fail it tends to be. Most likely because effects such as Bias Temperature Instability (BTI) and Hot Carrier Injection (HCI) become more prominent, and also because of new computing paradigms such as lower supply voltages, and near transistor threshold operation, less energy is necessary to change the electric charge (digital value) stored in memory cells, logic latches, or flip-flops. Consequently, particles hitting the silicon of an electronic device can easier cause a bit-flip in its logical components [8]. Therefore, the fault-tolerant methods need to evolve together with the electronics, and new methods and strategies are needed to overcome the challenges these new technologies impose.

Within the safety-critical domain, earlier applications could rely on more reliable and consolidated technology with bigger feature sizes (e.g., $130nm$ and $90nm$). However, as the applications within this domain have also evolved, computation power has been required as never before. Therefore, migrating to these new technologies and creating new methods to overcome the vulnerabilities of these electronic devices are current issues. At the same time, it is not unusual that embedded systems need to deal with limited power, which is also true for most safety-critical systems. However, in this context, the power constraints and the fault tolerance requirements may work against each other. In other words, the design may need to spend more power to increase its fault tolerance (using Triple Modular Redundancy (TMR) for example), however, as a power-limited system, it is desirable to save as much power as possible. Therefore, a compromise between these parameters is usually required.

Taking this into account, the following research question guides the remainder of this thesis: **How can we increase the fault tolerance of an electronic system based on a processor design without fully triplicating its power consumption? Is it possible to manage the system elements, according to the application demand,**

and optimise run-time parameters such as fault tolerance, power consumption and hardware degradation (ageing)?

To answer this question, this chapter introduces the concept of processor internal units management, in which fault tolerance can be improved at run-time as the system demands. Furthermore, when no improved fault tolerance is needed, the system can disable unnecessary internal units, therefore avoiding unnecessary power consumption and ageing.

3.1 The Fine-grained Management Concept

The main idea of this concept is to enable run-time management of available Functional Units (FUs) (spares or not) - such as Arithmetical Logical Units (ALUs), multipliers and dividers - towards improved trade-offs between fault tolerance, power consumption, and ageing. In this concept, the management of the available units is done by a software (bare-metal or via an Operating System (OS)) running over the platform, which can take into account parameters such as the criticality of running tasks (or processes in case of the OS), increased susceptibility to faults (e.g., due to ageing), increased fault rate, and health state of monitored units.

Other approaches do offer such a level of units management. However, it is usual that these strategies are not available at run-time, or they add a high overhead in execution time. One of the main properties of my approach is the fast and easy management of units at run-time. Among different possibilities to reach these mentioned properties, very specific design and implementation decisions were made. First, instead of using memory mapping, the Instruction Set Architecture (ISA) of the processor design, and consequently its microarchitecture, was extended enabling the desired run-time management of FUs. More about this decision, as well as its implementation details, will be given in Chapter 4.

Second, since the configuration is performed only over the FUs, and, as it is usually the case for any other processor design, these units are not used as storage elements within the pipeline (e.g., registers or memory). Thereby, it is expected that no other pipeline stage will access any data stored within the FUs for the upcoming instructions. Consequently, there is no need to clear registers and neither to flush the processor pipeline to migrate from one unit to another. Furthermore, in case of any data or control hazard, it will be the case that a resulting value of a calculation performed by the FUs is needed in the preceding pipeline stages. However, the needed value is forwarded to other pipeline stages only after the

FUs, this pipeline connection is usually done only in the stage that follows the Instruction Execute (EX) stage. Therefore, such a connection would not prevent, and neither affect the flexibility of the desired configuration at the FUs. Thereby, the configuration process takes the same as any other simple instruction of the processor: *one instruction cycle*.

The fast management of FUs enables a broad applicability scenario. For example, in very seldom cases in which a system may be subjected to an increased rate of faults (e.g., due to an increased rate of particles hitting its silicon), replication schemes of units can be enabled to increase its run-time fault tolerance. Furthermore, it is also possible to enable/disable replication schemes within very fast loop bodies of programs, in which the result of a very punctual calculation may need to be assured by increasing the design's fault tolerance. Moreover, while configured with no redundancy, load balancing can be performed over the available FUs replica, therefore enabling wear levelling strategies on these units. Last, but not least, because of the multiple redundant FUs available, permanently defective units can be masked out, however, at a cost of a reduction of the possible redundant schemes. Further discussions on these topics are going to be presented in the next sections of this chapter.

Essentially, the idea is to enable a processor design with redundant schemes among its internal FUs, and, with a run-time capable management mechanism, enable and disable these units to configure triple, dual, or none redundancy. Therefore, providing error correction, detection, or none, respectively. With this concept in mind, the management mechanism can provide different configuration schemes, which are going to be explained in Section 3.1.1 that follows.

3.1.1 Units Configuration Schemes

This section explains the main capabilities of the proposed design. In essence, it is possible to manage, at run-time, the internal FUs of the platform to increase fault tolerance or avoid its unnecessary usage. Therefore, it can minimise ageing and power consumption, or also enable the load balance over the available units.

A processor design has usually multiple types of FUs such as ALUs, multipliers, dividers, and Floating Point Unitss (FPUs). For the matter of understanding, in the remainder of this chapter, I will refer to FU as a group of one unit of each type available in the processor design. As it can be seen in Figure 3.1, there are essentially three major schemes in which the units can be configured, and other minor variations within each scheme also. Therefore, the schemes are the following:

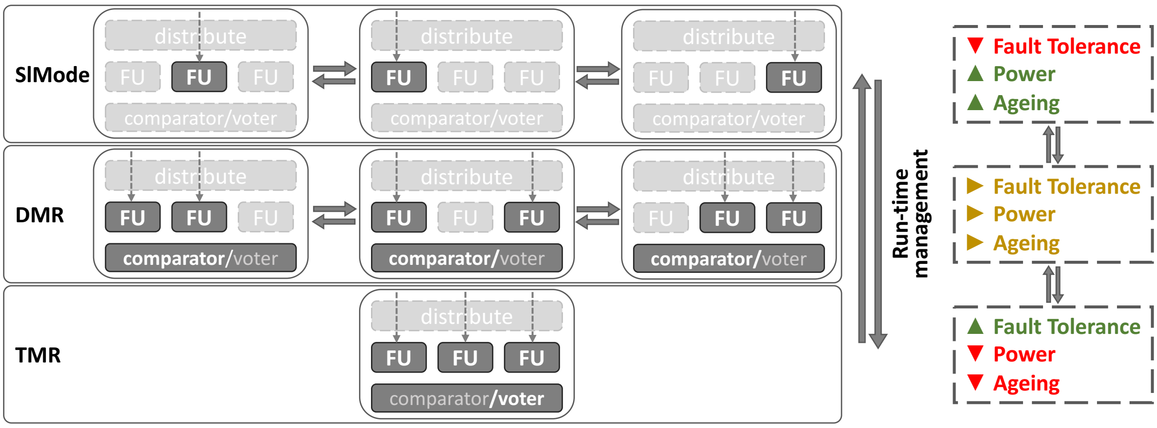


Figure 3.1: Run-time management capabilities of the design.

- **Single Unit Mode (SlMode) Scheme**

  In this scheme, only one FU is enabled at a time. Therefore no replication scheme is enabled, neither fault detection nor correction is enabled. For that, the hardware modules responsible for comparison and error detection/correction are transparent performing no function in this scheme.

  This scheme can be used to avoid unnecessary power consumption, since all redundant units are disabled, it is expected that this mode will consume the least. However, at a price of decreased fault tolerance.

  Furthermore, although only one FU is used at a time, it is still possible to perform load balancing throughout the units. Therefore, with a proper algorithm, interchanging the unit that is being used, it is possible to keep the usage of all available units at a similar level. Such an algorithm can be either agnostic to the usage of the units, scheduling a random unit to be used every time window (or every configuration's switch), or it can actually monitor the usage time of each unit, and schedule the units to balance their usage.

  Finally, for future improvements, and if supported by the processor architecture, this mode can also be used for performance improvement by enabling the FUs in parallel.

- **Double Modular Redundancy (DMR) Scheme**

  In this scheme, two of the FUs are enabled at a time. Therefore, the dual replication scheme is enabled, which enables error detection through Double Modular Redundancy

(DMR). For that, the hardware modules, responsible for comparison and error detection, consider the output of these two units and detect any difference between them.

This scheme can still be used to avoid unnecessary power consumption since only dual redundancy is enabled, it is expected that this mode will present medium consumption. However, no error correction is provided here, but only detection. Therefore, this is an intermediary configuration scheme, because, although its error detection capability may be enough for some applications, it can also be used with other strategies, supported by software mechanisms, for error correction on demand. Such strategies will be explored more in detail in Section 3.5.

Furthermore, although two FUs are used in this scheme, it is again still possible to perform load balancing throughout the available units.

Again, if supported by the processor architecture, this mode can also be used for performance improvement by enabling the remaining FUs for parallel execution.

- **Triple Modular Redundancy (TMR) Scheme**

  In this scheme, all three FUs are enabled. Therefore, the triple replication scheme is enabled, which enables error correction through TMR. For that, the hardware modules, responsible for the majority voting and, thus, the error correction, consider the output of these three units and propagate only the value that wins the voting.

  This is the scheme in which power is used the most, and all the units are under stress of usage. However, since this scheme is actually intended for improved fault tolerance, its price is acceptable once it is reserved only for the cases in which, for instance, the criticality of the running application is at the highest level. Therefore, the extra power expenditure is justified.

  In case there are only three FUs available for management, no load balancing is possible in this scheme. However, in case other spare units are available, load balancing would again be possible using these extra units. Likewise, the same logic would apply to possible parallel schemes for performance improvement.

It is important to notice that the non-used units in each scheme can be disabled in different ways. They can, for example, remain active (no clock and neither power gating) but with their inputs forced to a low logic level, therefore performing no computations. Another possibility is to apply clock-gating [15] and deactivate the input clock of the non-used units.

However, it depends very much on the processor design, whether clock gating is possible (or not) at the intended level for the intended units. In both cases, although not at the same amount, power will be saved by disabling the non-used units.

Regarding power gating, the reactivation (warm-up) time needed to stabilise the voltage level through a disabled circuit block, would increase considerably the latency to change between the proposed FUs schemes, hence, not desirable. However, power gating could be implemented if more than three units are available. Therefore, a fourth spare unit could, for example, be gated out of the power minimising its power consumption while not in use. Another possibility would be if it is known that certain units will remain inactive for a minimum amount of time. Therefore, it would be possible to enable the power on these units beforehand so that once needed they are already ready to use without warm-up delays.

3.2 Units Management within an Operating System with Multiple Processes and Criticalities

Let us assume we have the mixed-critical scenario just as presented in Section 2.1.3, and our scenario is based on an OS running multiple processes, each one with a different criticality. The idea is to embed this criticality information into the processes data-set of the OS, and using a proper hardware management infrastructure based on the concept from Section 3.1, manage the processor FUs according to this parameter. With this new criticality parameter, the mechanism of the OS to switch processes was modified to, before releasing a new process for execution, enable the proper scheme of the FUs according to the criticality of this upcoming process. As a result, the desired criticality-aware management of FUs for mixed-critical processes within an OS is enabled. The implementation details to enable an OS with this FUs management mechanism at processes granularity will be explained in the Section 4.3.

Regarding the redundancy scheme attributed to each criticality level, essentially, the user can configure the desired scheme for the actual criticality levels within the system. However, in general, the most critical processes and everything executed in kernel context (OS functions), should run using the TMR scheme. On the other hand, the least critical ones should be configured to use only one FU at a time (the Single Unit Mode (SlMode)). Finally, intermediate critical processes can be customised using DMR. Because of the fault detection provided by this scheme, with an additional implementation, this arrangement can work together with strategies for instruction re-execution on-fault-detection. Such strategies can potentially correct faults by including pipeline rollback and execution rescheduling of the instructions whose results differ in the DMR module. In case these strategies are implemented, further timing analyses would indeed be necessary to make sure that there is no change in the real-time characteristics of the system. Section 3.5 explores these possibilities in detail.

After all, it is expected that very few and punctual modifications would be necessary over an OS kernel to enable it to control the hardware mechanism for FUs management. Therefore, these modifications are very feasible and portable for a great variety of systems. In Chapter 4, a real implementation example using the Plasma Real-Time Operating System (Plasma-RTOS) is presented, where the few necessary modifications are discussed in detail.

3.3 Further Opportunities for Functional Units Management

Within the mixed-critical context, run-time management of hardware resources to better fit into the current criticality requirements can help systems to deliver proper fault tolerance when needed, and also avoid power and hardware resources from being wasted. My concept enables such a feature at the FUs level. Furthermore, its low latency for the management of the hardware units opens possibilities to change the unit's schemes, not only between processes but also in a multi-granular way. Therefore, two more possible scenarios, in which my concept would be useful, are explored in the next subsections below.

3.3.1 Functional Units Management within Programs - Loop Bodies and Algorithms

Once more within the mixed critical scenario, but now instead of changing the unit's scheme at process switching time, the idea is to enable and disable redundancy schemes within processes algorithms, for instance, at loop bodies.

For example, in case a programmer wants to improve the chances that upcoming computations present no errors, the newly added instructions can be used directly in the program code to enable, for example, the TMR scheme over the FUs. Once the need for correctness is relieved, another instruction can be used, still within the same program body, to enable one of the other possible schemes (SlMode or DMR).

This scheme is only possible because of the very fast management strategy adopted (Section 3.1). With that, migrating between the replication schemes within a program would increase the execution time by a rate of only one instruction cycle per migration performed.

3.3.2 Functional Units Management at an Increased Fault Susceptibility Zone

Still exploring the system, let us say that at this time, the processes are agnostic to parameters like criticality and power consumption. However, one would like to run the system as reliable as possible at some point in time, and avoid unnecessary power consumption whenever high reliability is not needed. For example, in aerospace applications such as satellites, it is expected that within the normal orbit there will be zones with a high incidence of charged particles such as the *South Atlantic Anomaly*, in which the Earth's radiation Belts are closer to the ground [2]. Therefore, a satellite, or any other object or device in orbit, as it passes over the South Atlantic Ocean, will receive a larger than average dose of radiation, thus increasing the occurrence of events such as Single-Event Transients (SETs) and Single-Event Upsets (SEUs). Therefore, an error monitor attached to the FUs can monitor the incidence rate of these events, and once this rate reaches a certain level, it can be assumed that the system is passing over this high incidence area.

Monitoring the error rate can be done, for example, using the provided DMR scheme. However, other schemes and monitors, such as the ones presented preliminary by the author in [23], can also be attached to the units. Once this error monitor is in place and monitoring the error rates, an increase in these rates can be detected. In these situations, it can be assumed that the satellite is entering a zone of high incidence of particles. Therewith, the platform can enable TMR over its FUs, thus, increasing its fault tolerance. Once the error rate drops again, the platform can return to its previous scheme, therefore avoiding unnecessary power consumption and hardware resources.

3.4 Opportunities for Load Balancing over the Functional Units

As it was mentioned in Section 3.1.1, some of the schemes of the design do allow strategies to perform load balancing over the FUs. In the ones that do not use all available units (e.g., the SlMode and DMR), it is possible to *shift* from one unit to another while keeping the same configuration regarding the redundancy scheme. This shift can be done in a multi-granular way depending on the application. For example, in case the platform is running only bare-metal code, it is possible, for example, to shift from the current units to the others every time the main loop of the running program reinitiates, or also, even within a loop body.

Another possibility is in case there is an OS running on the platform, the units shift can be done, for example, every time the OS switches between processes. Similarly to the idea presented in Section 3.2, every time the system switch between processes, besides being able to choose the best replication scheme of the FUs, it is also possible to balance the load over the units and keep the usage of all available units at the same level. For this, different approaches can be used, for instance, it is possible to simply enable a different unit every time the unit shift is performed, or also choose the units randomly. Furthermore, in a more sophisticated approach, it would be possible to, for example, monitor the usage of the units, and always enable the least used ones, therefore, keeping the usage at a similar level.

Once more, such strategies for load balancing, shifting from currently used units to others, are only possible because of the very low latency of the management mechanism proposed in this thesis.

3.5 Design Space Exploration

As it is stated in the previous sections, the proposed design enables run-time management of redundancy schemes of the FUs, Figure 3.1 illustrates its run-time capabilities. As already explained in previous sections, it can provide improved fault tolerance through a TMR scheme, or avoid unnecessary usage of power enabling only a few (e.g., one or two) of its available units. However, other variations are still possible, and some of them are being discussed in the next subsections.

DMR and Error Correction On- Error Detection

An example of such variation would be the DMR configuration scheme with strategies for error correction on error detection. This means that, once an error is detected using the comparator of the DMR scheme, a strategy for on-demand error correction can be implemented. Such a strategy can include rollback and re-execution of the instructions which were in the execution stage of the pipeline while the error was detected. For this, the erroneous result should not be committed into the memory or registers and the Program Counter (PC) should be rolled back by at most one instruction. If the rollback is needed for more than one instruction, assumed non-faulty checkpoints must be created because it would be very difficult to prevent erroneous results to be committed into memory or registers. Thus, creating these checkpoints and retrieving them in case of an error, would result in bigger and undesirable run-time penalties.

Within this approach, the instruction re-execution can also be explored as, for example, Figure 3.2 suggests. Once an error is detected for the first time on a specific instruction, a simple re-execution of this instruction using the very same unit can be performed. In case an error is detected again in the same instruction, a different unit can then be used, thus, avoiding any particular problem (permanent or temporary) of the previously used units. Therefore, in case three units are available in total, it would be possible to change the units two times, in the second and in the third attempt of execution. Finally, in case the error still persists, a re-execution using the full TMR scheme can then be done, thus minimising the possibility of a new error. Furthermore, these re-execution possibilities can be explored according to the criticality level of the processes. For example, if one needs to guarantee that the instruction must be corrected within, at most, its second attempt, the TMR must be used in this attempt to avoid extra latency. Otherwise, the other options just explained can be explored to avoid unnecessary usage of hardware units.

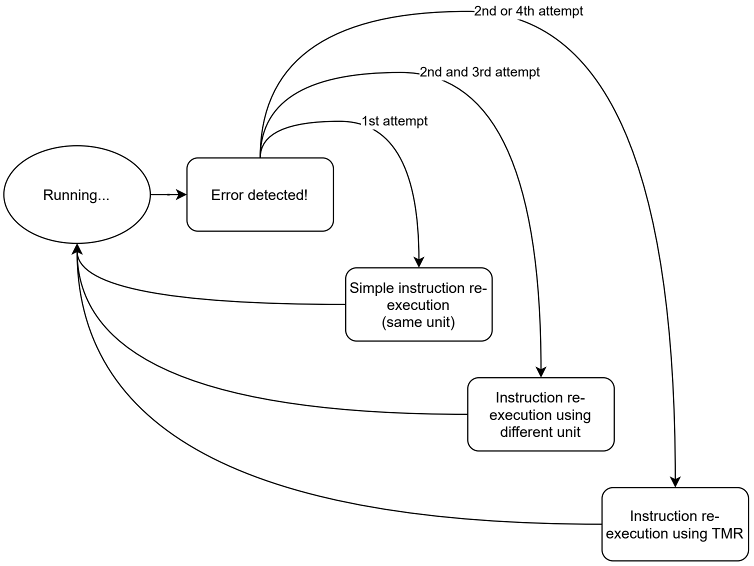


Figure 3.2: Possible instruction re-execution strategy once an error is detected.

These multiple attempts of error correction at run-time can potentially affect the real-time characteristics of a system. For example, while no error is detected, no extra time is needed, but in case an error is detected, additional time is necessary to re-execute the required instructions. As a consequence, it can potentially lead to run-time variation (jitter). Therefore, in case implemented, further analysis must be performed in this regard. However, for a worst-case analysis, the fourth attempt can be considered as the upper bound for the re-execution of each instruction. And, although theoretically increasing the run time upper bounds by four times its original, these would still be very deterministic, which would fit again as a real-time system.

Temporal Redundancy using SIMode

Temporal replication schemes are also possible on this platform. For example, with additional assembly instructions, it is possible to perform multiple and redundant calculations distributed over time. For example, the pseudo-assembly code from Listing 3.1 shows a very simple version of this temporal redundancy used for the multiplication. In this example, the multiplication is performed first with the first group of FUs (*Cfg 001*), after that the pseudo-instruction *Cfg 010* is used to disable this group and enable the second group of units. Therefore, performing the same multiplication again, but using this other group of units. Then, the instruction *BranchEq R3 R4 "equal1"* compares the results of the two redundant multiplications, and, if it equals, it branches to the label *equal1*, which stores the result of the multiplication and jumps back from the current context. In case the comparison of this branch (line 6) does not succeed, the branch will not be taken, and the lines that follow will be executed. In this case, the pseudo-instruction *Cfg 100* is used to enable the third group of FUs (and disable the others), and perform the same multiplication for the third time using this group of units. Here, to simplify the operation, and since no further action is foreseen in this example, the result from this third multiplication is considered correct. Therefore, the next two lines that follow save the last multiplication result and exit from the current context. With such a piece of code, temporal DMR, and TMR on-demand are provided.

On the other hand, as it can be seen from the example, despite the memory overhead to store those extra instructions, it can be expected at least 6 additional instructions per desired instruction to be protected over the temporal redundancy mechanism. Therefore, such an approach can result in a considerable run-time overhead, which makes this approach only applicable to a very small and specific set of instructions within a program. Additionally, still regarding time, such an approach would only make sense to protect instructions that take more time than the additional ones for configuration controlling and comparison. For

Listing 3.1: Example of a pseudo assembly instructions for temporal redundancy.

```
FaultTolerantMulti:     ;Fault-tolerant multiplication
Cfg 001                 ;Enable group 1 of FUs
Mult R3 R1 R2           ;Multiply R1 by R2 and store on R3
Cfg 010                 ;Enable group 2 of FUs
Mult R4 R1 R2           ;Multiply R1 by R2 and store on R4
BranchEq R3 R4 "equal1" ;If R3 e R4 are equal, branch to label equal1
Cfg 100                 ;Enable group 3 of FUs
Mult R5 R1 R2           ;Multiply R1 by R2 and store on R5
Add R R5 R0             ;store R5 on R
JumpBack                ;Jump back from this current context
equal1: Add R R3 R0     ;label equal1 - store R3 on R
JumpBack                ;Jump back from this current context
```

the mentioned example, it only makes sense if the expected run-time for one multiplication is bigger than the time needed for the *Cfg*, the *BranchEq*, the *Add* and the *JumpBack* instruction. Otherwise, unless protected by other mechanisms, these additional instructions would be adding more possible sources of errors than the instruction it wants to protect.

However, in case the application context can afford such a run-time overhead, and the mentioned timing conditions are satisfied, the *great advantage* of this mechanism is, in case some of the FUs become permanently faulty, it is still possible to perform the temporal replication scheme while non-faulty units remain operating.

Permanent Fault Correction

As it is already stated, permanent fault correction is possible. However, not only when running temporal replication schemes, but whenever one of the FUs become faulty, the remaining units can be used to keep the system running using the other still possible schemes (e.g., the SlMode or the DMR scheme).

Units Health Monitoring and Re-Adaptation when Degraded

Another very interesting scenario is when hardware units are degraded. In this scenario, the OS running over the platform can be used to monitor the health state of the hardware units of the platform. This can be done by tracking the number of errors detected in the units, and once a certain threshold or rate is reached, it can be assumed that the units are not anymore in their original health state. Once a change in this state is detected, it is possible to change the scheme that was previously attributed to the existent criticality levels of the application scenario. Therefore, it is possible to compensate for an increased incidence of errors, due to ageing for example, by attributing stronger fault tolerant measures (e.g., migration from DMR to TMR) to the existent criticality levels. A working example of this scenario is demonstrated in Section 4.4.

3.6 Summary

In this chapter, the concept for the management of processor internal FUs was introduced, in which essentially three major configurations schemes are foreseen: SlMode, DMR, and TMR. Furthermore, together with this concept, the mechanism for criticality-aware management of FUs for mixed-critical processes within an OS was introduced.

Moreover, an illustration of other opportunities for FUs management is provided for different scenarios. Here, the satellite case was presented, in which the TMR scheme is needed for zones of increased susceptibility of particles, such as the *South Atlantic Anomaly*. Still in this line, it also presented the case of very fine-granular management of FUs within program loops, and the opportunities for load balancing and wear levelling over on the FU.

Finally, this chapter closes with a brief illustration of other possibilities in which such a concept for FUs management would be applicable.

CHAPTER 4

From Conceptual Design to the Test Platform

This chapter introduces the implementation details of the test platform used to evaluate the hardware and software concept for run-time management of Functional Units (FUs). It shows how extensions were made over a processor design and an Operating System (OS), the Plasma processor and the Plasma Real-Time Operating System (Plasma-RTOS) respectively. It shows how this system is used to enable and disable replication schemes of processor internal FUs at run-time, for the safety-critical domain in the context of mixed-critical applications. Therefore, enabling tailored on-demand optimisations such as improved fault tolerance, and decreased hardware usage and power consumption. Therefore, the next sections will first show a quick overview of the baseline of the Plasma processor and, afterwards, illustrate the extensions made in this processor design and in the Plasma-RTOS.

4.1 The Plasma Processor

As already mentioned, Plasma was used as the baseline of the test platform implemented to comply with the design concept proposed in this thesis. Plasma is a synthesisable 32-bits Reduced Instruction Set Computer (RISC) microprocessor that executes the MIPS-I user mode Instruction Set Architecture (ISA) [75]. The block diagram of the Plasma microarchitecture and its corresponding pipeline stages are presented in Figure 4.1. The *PC_next* and the *Mem_ctrl* are part of the Instruction Fetch (IF), the *Control* block and the *Bus_mux* responsible for the Instruction Decode (ID), the FUs (the *Mult*, the *Arithmetical Logical Unit (ALU)* and the *Shifter*) participate in the Instruction Execute (EX), and, finally, the *Reg_bank* is part of the Write Back (WB). From the block diagram, it is possible to notice that the IF is in the first pipeline stage, but the ID, the EX, and the WB, are all part of the second stage, therefore, these three operations are performed all in the same cycle.

As it is possible to see from the block diagram, the Plasma processor has a very simplified microarchitecture with few internal blocks, and only two pipeline stages. Despite its simplicity, it can, however, execute a good portion of programs supporting GNU Compiler Collection (GCC) and a comprehensive subset of the American National Standards Institute (ANSI) C library. There is also the Plasma-RTOS, which is a fully preemptive Real-Time Operating System (RTOS) that is available for this processor. Therefore, its low complexity together with its library support were the reasons this processor was chosen as a baseline for the test platform going to be presented in the next sections.

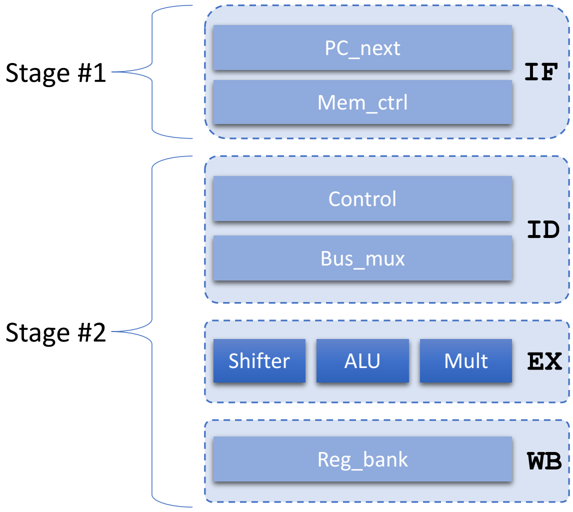


Figure 4.1: Block diagram and pipeline stages of Plasma processor design.

4.2 The Extended Plasma Processor

The core of this processor was modified as shown in Figure 4.2 to allow the intended fast management of its internal FUs. The following sections go through each block of this design and explain these in detail.

4.2.1 The Extended Instruction Decoder and the New Instructions

First, instead of an additional hardware component as presented in previous work [22], the *Instruction Decoder* was extended in such a way that additional instructions were introduced and enabled software control over the redundancy scheme of the FUs. These new instructions extended the ISA of the processor design, and to avoid any modification in the compiler, an *assembly volatile* declaration was used - $asm\ volatile(".byte < instruction > ")$. Therefore,

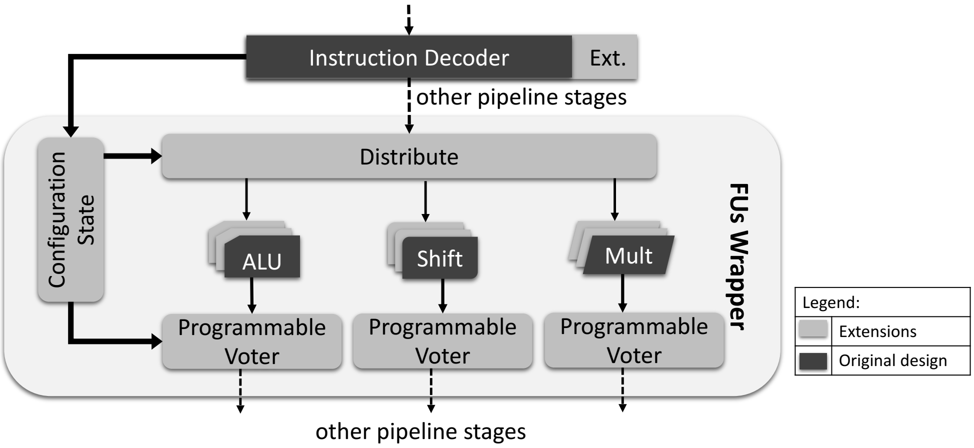


Figure 4.2: Extensions made over the hardware design showing the extended decoder, the extra functional units and the control logic of the FUs Wrapper.

the compiler keeps the command stated in the < *instruction* > field unmodified after compilation.

The created instructions were carefully planned to not overwrite any of the existing ones in the ISA of the processor design (MIPS-I). As we can see in Table 4.1, we follow the convention of the MIPS ISA, and the highest six bits of the instructions are the operation identifier (the Operation Code (OpCode)), so the instruction decoder can easily identify these new instructions. The lowest bits of the instructions, which are usually used for immediate values and offsets, index the FUs, in which each existent unit could be indexed by a different bit in this instruction. However, for bigger designs, it might be that there will be more available units than available bits in the instruction. Therefore, to make it more general, and for simplified operation, the different units were joined in groups, in such a way that each group has one ALU, one multiplier, and one shifter (the different available FUs in the Plasma processor). Therefore, for this current implementation, the lowest three bits of the instruction index the groups of FUs, and not them individually. With this, the remaining bits that are not used by now are reserved for future extensions.

The possible schemes follow the concept presented in Section 3.1.1, and are indexed by the lowest three bits of the instruction just as it follows (Table 4.1):

- **001, or 010 or 100** → it indexes the Single Unit Mode (SlMode), enabling only one group of FUs for computation, therefore, there is no replication in this scheme;
- **011, or 110 or 101** → it enables the Double Modular Redundancy (DMR) over the

FUs. Therefore, two groups of units are working in parallel redundantly to each other, and an additional hardware block (the programmable voter) is comparing the output signals enabling error detection.

- **111** → it enables the Triple Modular Redundancy (TMR) scheme, which means that all the three groups of FUs are working redundantly with each other, and the programmable voter is performing the majority voting over the output signals of the FUs, correcting any possible error produced individually by one of the units.

It is important to say that the new instructions only trigger the management actions, and for the rest of the pipeline modules it is considered as a NOP instruction. NOP stands for *No-Operation* and, as its name says, it is an operation that does nothing. This means, for the FUs, that a NOP instruction is being executed, therefore, there is no operation in the FUs while these are being configured. Furthermore, the process triggered by one of these new instructions and the NOP instruction are equally executed in one instruction cycle. Thus, *this process does not interfere with any other instruction in the pipeline.*

At this point, the reader might be wondering again: *but why not use memory mapping instead of extending the processor ISA and, therefore, its instruction decoder?* To answer this question, we first must look at the usual implementation of a processor's pipeline. It is mainly divided and ordered as follows: IF, ID, EX, Memory Access (MEM) and WB. The MEM stage might be skipped if no memory operation is necessary. However, in the case of memory mapping, this is exactly the stage in which our memory-mapped instructions would be redirected to the management mechanism for appropriate controlling of the FUs. While, in the meantime, another instruction would have already reached the FUs in the EX stage, preventing the management mechanism from changing the FUs scheme without at least waiting until the current operations in the units are finished and forwarded to the next pipeline stage. Therefore, stalling the pipeline would be necessary in this case. In such a situation, the management of the FUs would no longer be done in one instruction cycle, but in at least, two to three extra cycles.

Load Balancing over the Functional Units

The new instructions were created to enable control, not only of the replication scheme, but actually of the desired units to be used. Therefore, one can choose to operate in SlMode or DMR and explicitly define which units should be enabled. Therefore, one can use these commands to, at every certain time window, migrate from the currently enabled units to the other ones while still using the same replication scheme. Thus, enabling the user to create load balancing strategies over the FUs.

Table 4.1: The created instructions for software control of the processor FUs and their respective configuration schemes.

Instruction bit-fields			**Configuration**			
OpCode (31 downto 26)	**Not used** (25 downto 3)	**Group Units Index** (2 downto 0)	**Units Group**			**FUs Scheme**
			FUs3	FUs2	FUs1	
010101	don't care	001	-	-	✓	SlMode
	don't care	010	-	✓	-	
	don't care	100	✓	-	-	
	don't care	011	-	✓	✓	DMR
	don't care	110	✓	✓	-	
	don't care	101	✓	-	✓	
	don't care	111	✓	✓	✓	TMR

4.2.2 Functional Units Wrapper

The processor core was extended in such a way that more FUs were added to the design. In fact, two extra units of each type (two ALUs, two multipliers and two shifters) were added allowing the configuration of the additional DMR and TMR schemes among these units.

Furthermore, these units were encapsulated into the *FUs Wrapper*. This wrapper is a very tiny layer in between the units and the rest of the processor core, which controls the group of units to be used, therefore the desired scheme (with redundancy or not) among the FUs. Within this wrapper we can find the *Configuration State* block, the *Distribute* block and the *Programmable Voter*. As we saw in Section 4.2.1, the instruction decoder is responsible for decoding the newly created instructions for FUs management, and retrieving the requested group of units to be used from the last three bits of the command. The *Configuration State* block receives this configuration from the instruction decoder, writes it in a local configuration register, which may be read from software for status control, and send the control signals to the *Distribute* block and the *Programmable Voter*. The *Distribute* block receives these control signals and it configures its internal logic to distribute the incoming pipeline signals to one, two, or all three groups of units accordingly. Finally, the *Programmable Voter* also receives these configuration signals and, as its name says, it is not only a voter, but a hardware block that can be transparent (just forward the incoming signals) when using only one group of FUs, a fault detector (a comparator) when in the DMR scheme, or a three inputs voter (just as Table 2.1 describes) when TMR is enabled.

For the Plasma processor, the ALU and the shifter are not directly clock-driven, they are instead placed within the pipeline control logic. The only clock-driven FU is the multiplier. Therefore, to disconnect the non-used units from the pipeline, the *Distribute* block places a switch between the input pins of the FUs and the incoming pipeline signals. In this way, units can be disconnected by forcing their input pins to zero logic level. Otherwise, the switches just capture the incoming pipeline signals and reproduce them in the input pins of the active units. Furthermore, since the multiplier is the only directly clock-driven unit, besides these mentioned switches on the inputs, clock gating was also used, thus, maximising the power savings on these units when disabled.

4.2.3 The Low Latency Process to Manage the Units Configuration

As it was mentioned, the process to manage the scheme on the FUs is triggered by one of the newly added instructions (Table 4.1). So, once one of these instructions is fetched from memory, the following happens.

- The instruction is fetched from memory and reaches the extended instruction decoder.
- The instruction decoder recognises this instruction, retrieves the requested scheme for the FUs, and sends the control signals to the *Configuration State* block.
- The *Configuration State* writes this configuration in a local register, and sends these signals to the *Distribute* and the *Programmable Voters*.
- The *Distribute* and the *Programmable Voters* receive these signals and configure their internal logic to use the requested units.

Because of the management mechanism just presented in the sections above, and the strategy to use dedicated instructions to manage the units, the design can go from one configuration scheme to the other within a very low latency. This process takes only *one instruction cycle*, which is the cycle the processor design takes to execute any of its simple instructions (e.g., ADD and AND instructions). Furthermore, because the Plasma processor core has a very flat pipeline, once the instruction is fetched from memory and reaches the instruction decoder, the process to change the scheme of the units takes only *one clock cycle*. Figure 4.3 illustrates the instruction flow within the pipeline of the Plasma processor core. It shows three general instructions (*inst1, inst3, inst4*), and one instruction to change the FUs scheme (*cfg2*). As we have explained in previous sections, one particular implementation of the

Plasma pipeline is that the ID, the EX and the WB stages are merged within the same control logic so that no extra clock cycle is spent within these stages. Therefore, it is possible to see that, once the *cfg2* instruction reaches the ID stage, it is internally translated into a NOP opcode, with that the FUs within the EX stage and also the WB stage do not perform any operation in this clock cycle. Furthermore, in the next clock cycle, as soon the *inst3* reaches the EX stage, the FUs from group 2 are already enabled and ready to perform the required operations, meanwhile the FUs from group 1 are already disabled. This concept, and its provided very low latency, opens possibilities for the usage of this mechanism in a very broad way, enabling the management of processor FUs even within small program loops as suggested in Section 3.3.1.

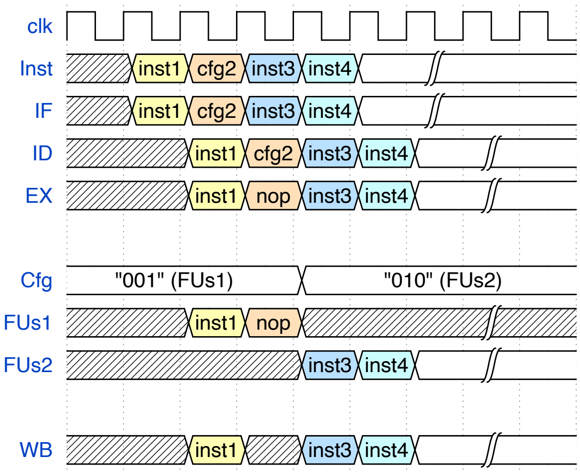


Figure 4.3: Instructions within the pipeline of the Plasma processor core while disabling a group FUs and enabling another one.

4.3 The Extended Operating System

The baseline OS used for the implementation of this test platform is the Plasma-RTOS. This RTOS was created by Rhoads to run in the Plasma processor. It supports interrupts, threads, semaphores, mutexes, message queues, timers, heaps, and preemptive context switching [80]. For this thesis, the Plasma-RTOS was extended to perform at run-time the desired management of the processor FUs. Figure 4.4 shows the main software components which participate in the management process.

Due to the hardware extensions in the instruction decoder of the processor design, and the new set of instructions (Table 4.1) presented in Section 4.2.1, the software layer can use these

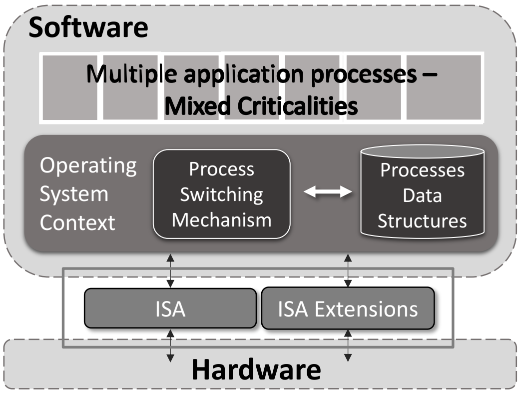


Figure 4.4: Software and OS blocks that participate in the management process.

instructions to control the configuration of the FUs. These new instructions are represented in Figure 4.4 as the *ISA Extensions*.

The per *Process Data Structure* that stores information regarding its corresponding process (such as process id, name, priority, and state) was extended with a criticality level field (Listing 4.1).

Furthermore, the *processes interface function* that creates new processes for the operating system was extended with an additional criticality parameter. As it can be noticed in the Listing 4.2, this interface is a function that creates the processes (which are called threads within the Plasma-RTOS). This function has usual parameters to create a new process such as the process *name*, the pointer to its actual code/function (*funcPtr*), respective arguments that its respective process may expect, the *priority* of the process that is used by the scheduling algorithm to order the processes executions, the respective desired *stack size*,

Listing 4.1: Plasma-RTOS *Process Data Structure* extended with the criticality level field.

```
struct OS_Thread_s {
    const char *name;           //Name of thread
    OS_ThreadState_e state;     //Pending, ready or running
    [...]
    uint32 criticality;         //Criticality of thread
                                // (0=low, 255=high)
    [...]
    //Linked list of threads by priority
    struct OS_Thread_s *next;
    struct OS_Thread_s *prev;
};
```

and finally, the newly added *criticality* value. This extension allows the operating system, right from the creation of the process, to attribute the desired value to the criticality level field in the Process Data Structures just mentioned.

Furthermore, once this per process criticality parameter is available, the *Process Switching Mechanism* of the Plasma-RTOS was modified to, before releasing a new process for execution, enable and/or disable the proper group of FUs according to a user predefined configuration that matches the criticality of each process. In summary, the inline function from Listing 4.3 was added at the very end of the *Process Switching Mechanism.* The *inline* definition was here used to prevent the compiler from creating a function call by actually adding the content of this inline function directly at the address it was called. Therefore, this avoids the additional overheads, incurred by a normal function call, of saving context and jumping instructions.

Nevertheless, despite being inline, this function has its additional internal instructions. For example, extra memory read operations (load instructions) are necessary to read the criticality value from the per *Process Data Structures*, and additional supporting instructions (branches and comparison with immediate values) are needed to compare with reference values and branch to the correct place where the corresponding instruction to enable the proper FUs group is located. These additional instructions for loading data and branching to correct places, together with the extra instruction for FUs management, result in memory and execution time overheads. The incurred memory overhead is actually only a few lines of assembly code that do not take much space. However, the execution time overhead will include the cycles needed to execute these additional instructions and increase the time needed to perform the process switching. Nevertheless, such an overhead will still represent a small fraction of the whole time used by the *Process Switching Mechanism* as it will be presented in the Evaluation Chapter in Section 5.1. This process switching procedure, together with the FUs scheme configuration, is further detailed in Section 4.3.1.

Finally, assuming that all operations executed in the kernel context are considered at the

Listing 4.2: Operating system interface to create a process extended with the criticality parameter.

```
//Extending the ThreadCreat to assign a criticality value
OS_Thread_t *OS_ThreadCreate(const char *name,
                             OS_FuncPtr_t funcPtr,
                             void *arg,
                             uint32 priority,
                             uint32 stackSize,
                             uint32 criticality);
```

Listing 4.3: Prototype of the inline function created to switch the FUs scheme.

```
inline void switch_FUsMode(uint32 criticality);
```

highest criticality, the instruction to enable the TMR scheme in the FUs was placed at the very beginning of the assembly code of the Interrupt Service Routine (ISR). Therefore, once an interruption is raised calling the system to enter in the kernel context, its first instruction will enable the TMR scheme over the FU, thereby, any further instruction will be executed with this scheme. Furthermore, the mechanism will keep this TMR mode until the Plasma-RTOS switches to a new process, then, right before jumping to this process, the instruction to enable the proper units' configuration for this upcoming process is issued.

Although switching the FUs scheme to enter in kernel context could be done directly in hardware, it would need further modifications in the core of the processor design, which would make the implementation of the mechanism much more intrusive, opening possibilities for more implementation errors. Secondly, a configuration done directly in hardware would not comply with my current concept, in which the hardware only provides the management mechanism, but the software is the one responsible to control and manage the hardware units.

Regarding the mapping between the FUs scheme and the criticality of the processes present in the system, it can actually be done by a user before run-time, Table 4.2 illustrates an example of such a configuration. However, in summary, the most critical processes should run using the TMR scheme, while the other less critical processes can be mapped to the DMR or SlMode.

Table 4.2: Configuration example for the criticality levels and its correspondent redundancy scheme.

Criticality level	Criti 0	Criti 1	Criti 2	Criti 3
Redundancy scheme	SlMode	SlMode	DMR	TMR

4.3.1 The Process Switching Procedure

As stated, the Plasma-RTOS supports priority and preemptive context switching. Thus, each application process runs its portion of time (time-slice), and the next process in the ready state with the highest priority should be released for execution. Figure 4.5 shows the intermediate states of the procedure for process switching with the management of the hardware FUs in between. While the application process is running, an end of time-slice interruption is raised, the application process stops and the ISR is called. At the very beginning of the ISR, there is an instruction to enable the TMR scheme among the FUs of the design. The TMR is then enabled, and the process switching mechanism performs its remaining operations normally.

Later on, before releasing the next process for running, the criticality level of the respective upcoming process is read (load instruction) from the respective process data-set structure, and, according to this criticality level, the expected instruction to enable the proper FUs scheme is issued. Here, in case the new configuration scheme does not use all available units (e.g., the SlMode or the DMR), the choice of the units to be used can obey an algorithm for load balancing, thus balancing the usage of the units and enabling wear levelling between them.

Finally, once the proper scheme is enabled, the application process is released and will now run using the units it was programmed for.

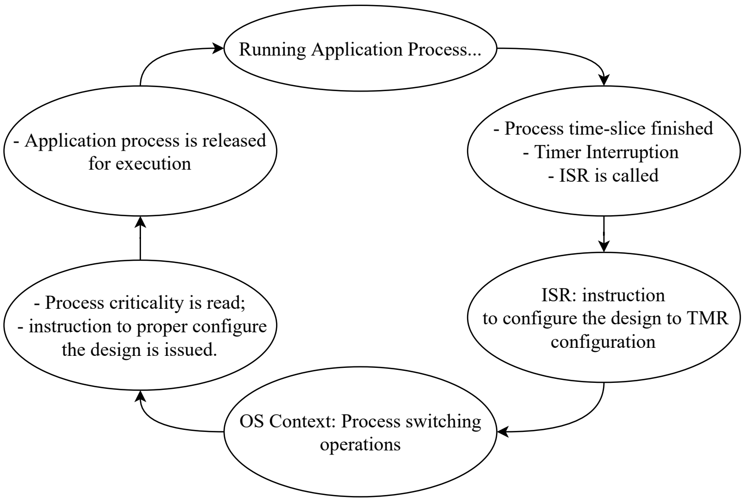


Figure 4.5: Procedure for the process switching with the management of hardware FUs in between.

4.3.2 Portability Analysis

The extensions made over the baseline code of the Plasma-RTOS used very few extra lines of code. Moreover, the necessary modifications were very punctual and very easy to port. Therefore, the extensions explained in the sections above can be easily portable to other systems.

As an example, the *FreeRTOS* was studied. *FreeRTOS* is an RTOS distributed freely under the MIT open source license. It supports more than 35 architectures and provides core real-time scheduling functionality, inter-task communication, timing, and synchronisation primitives only. Additional functionality, such as a command console interface, or networking stacks, can then be included with add-on components [40].

Looking into the *FreeRTOS* implementation, it has a good number of similarities with the Plasma-RTOS. For example, it also has a per *process data structure* called *Task Control Block* that saves processes state information. So this structure could also be expanded with the criticality level field as it is done for the Plasma-RTOS.

Regarding the *Process Switching Mechanism*, the *FreeRTOS* has special functions to perform the context switch. These functions are usually architecture dependent, and, in a general way, the top level of these functions looks like the one presented in the Listing 4.4. Here the implementation of the top-level function used for manual context switch (*vPortYield()*) is presented. This function calls other specialised functions that save the current context, switch the processes, restore the context for the new process, and jump into its new context for execution. So, we could use the *switch_FUsMode()* function right before jumping to the new process context, so that it would configure the FUs scheme a few instructions before starting its execution. Other implementations are also possible here, however, a deeper and specialised implementation analysis over other platforms is out of the scope of this thesis.

To change to the TMR scheme when entering in RTOS context, the assembly of the ISR in the *FreeRTOS* can also be modified in the same way as in the Plasma-RTOS. Therefore, the instruction to enable the TMR scheme can be placed at the very beginning of the assembly part of the *FreeRTOS* ISR.

Listing 4.4: Implementation of the *FreeRTOS* context switch function.

```
/* Manual context switch. */
void vPortYield( void )
{
        /*Save the context of the current task.*/
        portSAVE_CONTEXT();
        /*Switch to the highest priority task that is ready to run.*/
        vTaskSwitchContext();
        /*Start executing the task we have just switched to.*/
        portRESTORE_CONTEXT();
}
```

Finally, the *processes interface function* of the *FreeRTOS* can also be extended similarly as it was done for the Plasma-RTOS. Therefore, the criticality level parameter can be added to the function so that it can be attributed right at the time in which the processes are created.

4.4 Test Case - Units Monitoring and Re-Adaptation when Degraded

In this section, a test case of the platform just described in the sections above is presented. However, the management of the FUs takes into account two parameters: the criticality levels of the application processes running on top of the Plasma-RTOS, and the health state of the FUs. The experiment going to be described was presented in detail in [24], and evaluates the system's capability for run-time re-adaptation according to detected changes and degradation in the health states of monitored FUs.

First, we divided the applications into three different criticality levels: ordinary, medium, and critical. Second, for the health state classification, the intermittent soft errors (e.g., Single-Event Upsets (SEUs)) were considered, which are the ones that usually increase their occurrence as the electronic elements start to age [33, 37]. Therefore, each unit has its own health state classified according to the number of errors detected by assumed monitors attached to each unit. Furthermore, to go from one state to another, thresholds were defined as presented in Figure 4.6, once the number of detected faults reaches the *Medium Healthy*, the *Intermittent* or the *Faulty* threshold, the unit changes its state, respectively, from *Healthy* to *Medium*, from *Medium* to *Intermittent* and from *Intermittent* to *Faulty.* It is important to notice that no actual fault monitors are attached to the functional units. Although it can be done while the design is in DMR or TMR scheme, this experiment does not intend to evaluate individual fault monitoring techniques nor hardware fault detection mechanisms. However, an example of such monitors with preliminary hardware overhead results is presented by the author in [23]. Thereby, the simulation presented in Figure 4.7 shows the platform behaviour during this experiment, which consists of a series of increments in the individual unit registers ($units\_regs[< FU - index >]$), which accounts the errors detected in the FUs. Since these registers are the ones responsible for counting the number of events that led to errors, these registers were incremented, in the experiment, in such a way that the FUs changed their initial attributed health states. Thus, it can be observed that the platform chooses the healthier units to execute its application processes.

For this experiment, the health state thresholds (Figure 4.6) were configured as the following: the *Medium Healthy* threshold to 10 (0x0A) faults, the *Intermittent* threshold to 100 (0x64), and the *Faulty* threshold to 1000 (0x3E8). It is important to notice that these are only demonstration numbers, and linking the number of errors detected in hardware units to health states can be the subject of a completely separate study.

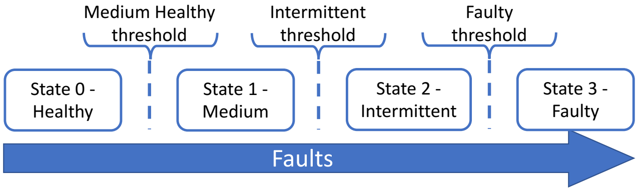


Figure 4.6: Health states of the FUs and their respective thresholds.

The signals in Figure 4.7 represent, from top to bottom, the clock signal (clk), the configuration status registers that store the current units' configuration (*stats_reg*), the application processes with ordinary (Process1), medium (Process2) and high (Process3) criticality, and, finally, the units registers: *units_regs*(0) representing the accounted events for FU 1, the *units_regs*(1) for FU 2 and the *units_regs*(2) for FU 3. As it is highlighted in the figure by the dashed red lines and squares, once the number of accounted events in a specific FU reaches one of the health thresholds, the configuration scheme attributed to each criticality - and consequently to the processes - is updated taking into account the new health state of the units. As a result, the further executions of the processes are done using a new configuration, either by replacing a faulty FU or shifting from DMR to TMR. For example, in the figure, Process1 begins using FU 1. As the register responsible to save events in this unit is incremented and reaches the Intermittent threshold, FU 1 goes to the Intermittent health state. Therefore, in the next execution of Process1, the system uses one of the other healthier units, as in the case of the figure, the FU 2.

Following the other processes in Figure 4.7, we can see similar behaviour with Process2. The design shifts from FU 1 and FU 2 to FU 2 and FU 3 as the FU 1 goes to the Intermittent health state. However, a different situation is observed when there are no more FUs in the Healthy state. In this case, the system needs to re-adapt its default configurations, and, to compensate for its increased susceptibility to intermittent faults due to assumed ageing, it increases its fault tolerance and goes from DMR to TMR scheme as it can be noticed in the third execution of Process2 in the figure.

Finally, still in Figure 4.7, the last case is noticed when we look at Process3. Since it is a highly critical process, it starts running with TMR. However, as the simulation evolves, the system enters in a state in which one of its FUs is classified as Faulty. Therefore, the system needs to re-adapt again, and the TMR is replaced by a degraded service configuration using DMR over the remaining non-faulty units. It is important to say that, although not yet implemented, this degraded DMR configuration can be upgraded with a scheme for instruction re-execution once a mismatch is detected over the outputs of the units, thus, potentially correcting a soft error that may occur in the remaining units.

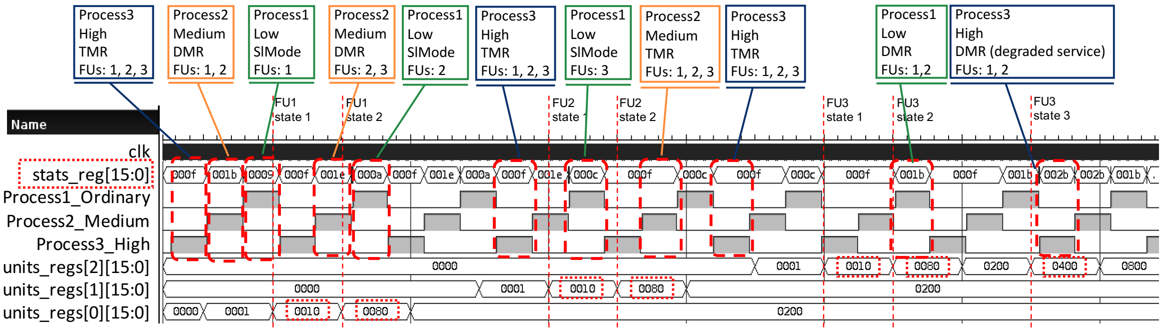


Figure 4.7: The fault simulation performed over the platform. The fault counter registers are incremented from time to time simulating the detection of faults in the FUs. Adapted from [24].

4.5 Summary

In this chapter, we saw the details of the test platform implemented to reach the concept proposed in Chapter 3. First, the extended Plasma processor design was presented. New instructions were created and, with these, the instruction decoder was extended. Together with also added FUs and a wrapper, these new instructions were used to change from one scheme to another with a very low latency. Not only changing the redundancy scheme, but load balance can also be done using the new instructions. Furthermore, the extended Plasma-RTOS was presented. It shows how the per process data-set was incremented with criticality information, and the process switching mechanism extended, to enable criticality-aware management of FUs for mixed-critical processes. Finally, a use case test was presented, in which the degradation of FUs is considered, and the platform adapts its FUs scheme to the new degraded state.

CHAPTER 5

Evaluation

This chapter evaluates the hardware overhead, latency, and power consumption incurred by the concept implementation over the test platform. Furthermore, it shows the fault tolerance for the different configuration schemes and, also, the hardware ageing incurred by the criticality-aware management of Functional Units (FUs).

5.1 The Units Management Latency

First, the latency of the hardware platform while running bare-metal code was evaluated. For this, one of the new instructions stated in Table 4.1 was manually introduced in the source code of the testing program. The test program used in this case was a bubble sort algorithm, and the instruction was introduced at the very beginning of the algorithm, right before the loops that perform the ordering. In this situation, the latency observed was of *one instruction cycle*. As already stated, our processor design has a very flat pipeline with only two stages. This means that once our instruction reaches the instruction decoder, the units are already properly configured in the next clock cycle.

Such a low latency can enable even finer granular changes in the configuration of the units. For example, a change of the redundancy scheme can be performed even within program loops adding only one extra instruction cycle per change. Therefore, a very focused increase in fault tolerance is possible targeting, for example, specific calculations within a program loop.

The next step was to measure the latency (increase in execution time) introduced over the Plasma Real-Time Operating System (Plasma-RTOS) due to the implemented extensions presented in Section 4.3. As explained, the process switching mechanism was extended to change the units' scheme at every process switch. The mechanism reads the criticality level field in the processes data structure and, according to this level, it will issue the correct instruction to enable the proper scheme over the FUs. For this procedure, extra instructions are needed, for example, to load the criticality value from memory, and to compare, branch, and issue the correct instruction to configure the FUs. These additional instructions take extra time that is summed to the overall execution time of the process switching mechanism.

To realise the latency added by these extra instructions, the execution time of the raw process switching mechanism was measured and compared with the same when the above mentioned extra instructions are included (the extended version). However, since the execution time of the process switching mechanism depends on various internal states of the operating system, it varies each time it runs. Therefore enough measurements were done, so it was possible to see a convergence for a common value. After these measurements, it was possible to see that the extended process switching mechanism increased its execution time by *approx. 15 clock cycles* (value calculated by a geometric mean of the acquired values).

Nevertheless, altogether, these extra clock cycles represent just a small fraction of the whole switching mechanism. In fact, for the current implementation of the process switching mechanism within the Plasma-RTOS, these 15 extra cycles represent *approx. 2.4%* of the cycles needed by the whole mechanism.

5.2 Hardware Area Overhead

The Plasma processor core and its extensions were synthesised using the Cadence® Genus™ tool and mapped to the Open Cell Library (OCL) 15*nm* [61], the resulting area is shown in Table 5.1. This table also shows the total area covered by the Plasma design with no modification (Original), and a further row with the area of the Plasma processor with its internal FUs triplicated, but with no additional control logic for dynamic management, so the design has Triple Modular Redundancy (TMR) scheme always enable (static) with no further possibility for configuration of other schemes.

It is important to notice that the original Plasma processor core has about 22% of its area covered by functional units, and when triplicating these units, they add an overhead of about 44% to the design. This overhead is the very same presented in both designs: the Plasma TMR static and the Plasma Extended for dynamic management. However, comparing these two modified designs we can notice a slightly bigger overhead in the control logic of the Plasma Extended design for dynamic management (from approx. 2% for the *static* TMR, to approx. 6% for the *dynamically* configurable design). This extra overhead is due to the additional control logic to make the design dynamically configurable at run-time, such as the extensions in the Instruction Decoder and in the other blocks inside of the FUs Wrapper just as it is presented in Figure 4.2. Therefore, we can conclude that in case the design is already made to be fault-tolerant with triplicated FUs, the control logic overhead to make the design dynamically configurable would be approximately 4% (comparing the overhead of approx. 2% for the static TMR, with the approx. 6% of overhead for the dynamically configurable design).

Table 5.1: Hardware area overhead for the Plasma Original design, the Plasma with static TMR, and the Plasma extended for dynamic management of FUs.

Design	Total Area $[um^2]$	Total Area Overhead $[um^2]$	Total Area Overhead	FUs Area Overhead $[um^2]$	FUs Area Overhead	Control Logic Overhead $[um^2]$	Control Logic Overhead
Plasma Original	3522.13	-	-	-	-	-	-
Plasma TMR static	5122.33	1600.24	45.43%	1530.99	43.47%	69.25	1.97%
Plasma Extended (dynamic managm.)	5269.29	1747.16	49.61%	1530.99	43.47%	216.17	6.14%

The overhead just presented in Table 5.1, although not negligible, is still less than other full core replication approaches such as core lock-step schemes [45, 89]. In these schemes, the whole processor core is under double or triple modular redundancy, and a control logic is built on top of these cores to administrate the redundancy and perform error detection or correction among the core output signals. Therefore, since these approaches place the whole processor core over the redundancy scheme, it is expected a better fault tolerance performance when compared to my mechanism for fine-grained management of FUs. However, the price of hardware overhead for full core replication is usually above 100% for duplication and 200% for triplication. For example, in [45] the authors claim to have the overhead for replicating the cores plus an additional 18% of control logic overhead. Meanwhile, for my approach, the overhead is very dependent on the area distribution of its internal units, but, in summary, tends to be much smaller. To better illustrate such differences, Table 5.2 presents the hardware overhead for the Plasma Extended with the mechanism for dynamic management of FUs and an estimation for applying this same mechanism to the RI5CY processor [85]. Then it is compared with the expected minimum overhead for full core lock-step schemes based on the numbers presented by other authors [45]. As it is shown, my approach presents an overhead between 50% to 82%, meanwhile, the core lock-step scheme goes above 200%.

To summarise, it is notable that the approach for replications and management of FUs presented in this thesis would result in less overhead compared to full core replications schemes, but also less protection concerning fault tolerance. Hence, our approach fits as an intermediate level solution that is less costly in terms of hardware area, although not providing full core protection. Furthermore, in processors in which multiple FUs are already present such as superscalar processors, only the additional control logic to enable the dynamic management would be needed to apply the concept for dynamic management of FUs presented in this thesis. Therefore, in this case, it is expected that the resulting overhead would be even lower, and closer to the only 6% of control logic overhead needed for the Plasma processor.

Table 5.2: Area overhead comparison between designs implemented with the mechanism proposed in this thesis and the Triple Core Lock-Step (TCLS) scheme.

Design	Design area FUs participation	Proposed scheme total overhead	TCLS scheme total Overhead
Plasma	22%	∼50%	>200%
RI5CY	40%	∼82% (estimated)	>200%

5.3 Critical Path Delay

After evaluating the hardware overhead, we need to check how much the critical (longest) path delay increased due to the implementation of the management mechanism. For this, the critical path delay was compared between the Plasma Original version and the Plasma Extended. For this synthesis, instead of using $500MHz$, I tried to push the designs more to closer to their operating limits, and for this, the synthesis was performed targeting an operating frequency of $1GHz$. Therefore, the signals throughout the electronic design must propagate thoroughly before $1000ps$.

Table 5.3 presents the results of this timing analysis. It can be noticed that, under this scenario, the critical path delay increased by 6% in the Plasma Extended design. Despite this increment, both of the designs met the timing constraints to operate at $1GHz$.

Table 5.3: Increase in the critical path delay for a synthesis targeting $1GHz$.

Design	**Critical Path Delay**
Plasma Original	$866ps$
Plasma Extended	$920ps$
Path Delay increase	**$54ps$ (6.2%)**

5.4 Power Overhead

Going further in the evaluation of the platform, the power overhead generated by the additional hardware in the design was verified. For this evaluation, I used the same synthesised designs as in Section 5.2 (mapped to OCL $15nm$ [61]), and targeted an operating frequency of $500MHz$.

For this evaluation, after the synthesis, the design needs to be simulated for a certain amount of time using ModelSim. This simulation needs to be captured and stored in a Value Change Dump (VCD) file, which is a human-readable dump file that stores all states of all elements of the design for the desired simulation time. The VCD file is then used to generate the power profile of the design in the Cadence® Tempus™ tool. Thereby, the same tool is used to generate the power report of the design and get its power consumption based on this power profile.

Since the power profile takes into account the switching activity of the signals throughout the processor design, the simulation time to correctly represent the general switching activity of the signals depends strongly on the payload. Moreover, the captured simulation time in the VCD file should be enough to represent a fraction of a long run.

The Payload and the Scenarios

The payload used for these power measurements consisted of the Plasma-RTOS running three extra application processes. These application processes are identical and all the three are performing the same bubble sort algorithm. For this payload in particular, the initialisation phase of the design spends most of the time in the operating system context, and only after that the processes start to run periodically. In a long run, the resources used in this initialisation time would not be significant. However, in this short simulation time, the extra power spent in this time would indeed lead to inaccurate results. Thus the first two milliseconds of the simulation were discarded. The $\sim 1.6ms$ that follows is then captured in the VCD file and used to generate the power profile of the hardware design.

The power report was generated for four different scenarios. In the first one, the original Plasma design with no modifications was evaluated. Because the original processor design has no management possibility of its FU, the Plasma-RTOS was also used with no modifications. In the second scenario, the extended processor design was evaluated, but in this scenario, the design runs with the TMR scheme always enabled over the FUs, which would be the case if no dynamic management is possible. Again, since there is no change in the FUs scheme in this scenario, the Plasma-RTOS was used with no modifications. It is important to mention, that the normal state of the design is in the TMR scheme, which means that as long it receives no instruction to change its units scheme, it will remain running in the TMR. Therefore, no change in the hardware design is needed to pin it to the TMR scheme.

In the third scenario, the extended processor design was evaluated again, but in this scenario, it is performing a dynamic configuration of the FUs scheme according to the criticality of the running processes. Here, the extended Plasma-RTOS as it was explained in Section 4.3 was used. Therefore, the design changes its FUs scheme every time the system switches between processes. In summary, for this scenario, each process was configured with a different criticality level: one with the lowest possible criticality level configured to run in Single Unit Mode (SlMode); one with an intermediate level of criticality configured to run in the Double Modular Redundancy (DMR) scheme; and the last one with the highest possible criticality level, therefore configured to run using the TMR scheme. Everything else, besides these three processes, was considered operating system context, also considered highly critical and configured to run using TMR as well. It is important to say that for this scenario, the amount of time spent in the different FUs scheme highly impact the power profile generated later on. Therefore, to produce accurate results regarding the power consumption of the design, the execution window captured in the VCD file must present all the processes equally distributed over the time. Figure 5.1 shows how the processes, the Operating System (OS) context, and the configuration schemes of the FUs are distributed over the execution time window captured for this scenario. As it can be noticed, in this execution time, all the processes run for one time-slice and the OS context is equally distributed over the time performing the process switching. Therefore, this execution window represents a factor of other bigger execution times so that longer execution times will be only repetitions of this. Hence, this execution window can be used to generate the power profile of the design. Table 5.4 illustrates the different schemes running in this third scenario and the time spent in each of the processes and in the operating system context.

In the fourth scenario, the extended processor design using the extended Plasma-RTOS is again evaluated. Once more, three application processes are used. However, in this scenario, these three processes are all considered of the lowest criticality level, therefore, they all run in SlMode. Only the OS functions will be executed using TMR. Figure 5.1 shows also, the schemes distributions for this fourth scenario. As well as Table 5.4 shows the run time spent in each of the schemes configurations for this scenario.

It is important to notice that for the first two scenarios, in which there is no change in the schemes of the FUs, none of the hardware blocks is presenting considerable time without any switching activity. Therefore, the power profile is not considerably affected in case of small differences in the distributions of the processes throughout the captured execution window. Because of this behaviour, we could use the same execution time window for all the evaluated scenarios.

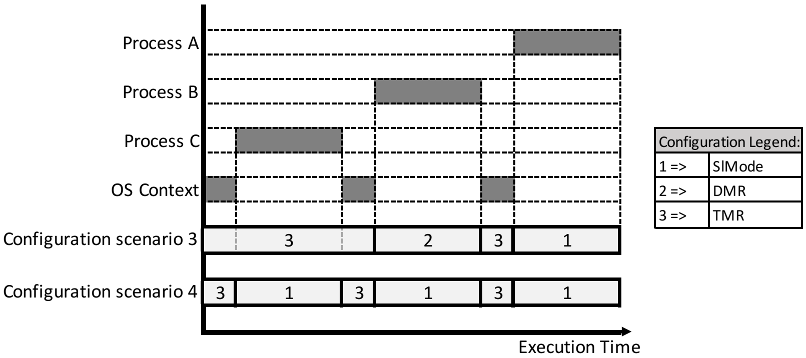


Figure 5.1: Execution time window when changing the FUs scheme for every process.

Table 5.4: Processes run time and its redundancy scheme configuration for the third and fourth scenarios.

Running context	**FUs Scheme**		**Run time**
	scenario 3	**scenario 4**	
Application process A	SlMode	SlMode	522984 ns
Application process B	DMR	SlMode	522996 ns
Application process C	TMR	SlMode	522966 ns
Operating system	TMR	TMR	3918 ns

The Results

Finally, Table 5.5 illustrates the power consumption for the original and the extended version of Plasma design through the four different evaluated scenarios as it was explained above. As it can be noticed, the worst case for power consumption, presenting a power overhead of approximately 98%, is the second scenario in which the extended design is evaluated and no dynamic management of its triplicated FUs is enabled. However, as the dynamic management is enabled, the power consumption decreased reaching an overhead of approximately 57% for the third scenario and 16% for the fourth scenario.

Table 5.5: Power savings when doing dynamic management of the redundancy scheme over the FUs.

Evaluated scenario	**Design**	**Total power [mW]**	**Total power overhead**
1 - no dynamic managm.	Plasma Original	2.68	-
2 - no dynamic managm.	Plasma Extended *fixed in TMR*	5.29	97.7%
3 - dynamic per process managm.	Plasma Extended	4.22	57.6%
4 - dynamic per process managm.	Plasma Extended	3.11	16.2%

Power Consumption Comparison

Table 5.6 summarises the power overhead for different approaches. It is notable that the resulting power overhead for triplicating only the FUs is much lower than for triplicating, or even for only duplicating, the whole core which would be the case for the core lock-step approaches. Furthermore, while the dynamic management of FUs enabled fault tolerance improvement at run-time, it enabled even lower power consumption than other mechanisms that apply always enabled redundant schemes.

Table 5.6: Power overhead comparison related to the single-core Plasma original design.

Design	**Total Power overhead**
Plasma with TCLS	>200%
Plasma with Dual Core Lock-Step (DCLS)	>100%
Plasma Extended with dynamic per process managm.	**from 16% to 98%**

To Conclude

The results presented in this section clearly illustrate that the FUs triplication, together with the dynamic management of units, could avoid the overall triplication of the power consumption for a system in which the TMR scheme is enabled dynamically.

5.5 Ageing Evaluation

The main goal of this evaluation is to show *the reduced ageing provided by the criticality-aware management of the FUs enabled by the software and hardware mechanism proposed in this thesis.* To this end, payload dependent ageing and its effects need to be evaluated.

As we have seen in Section 2.3, from multiple effects that can increase the ageing of electronic devices, the Hot Carrier Injection (HCI) and the Bias Temperature Instability (BTI) are the ones that most affect the switching speed of transistors, by increasing the modulus of their threshold voltage ($|V_{th}|$). Therefore, these effects can possibly decrease the working speed of big blocks within an electronic circuit, or even a complete device such as a processor core. Consequently, these effects are often taken into account when estimating the degradation of electronic devices due to transistor ageing [11, 43, 49, 72].

To estimate ageing effects caused by these two phenomena, proper mathematical models are necessary. For this purpose, this thesis uses widely accepted and already validated models for the BTI and the HCI phenomena described in [9, 101] and [72] respectively. These models describe the long-term transistor threshold voltage shift (ΔV_{th}) for these two effects. The respective equations to calculate the transistor threshold voltage shift at time t due to HCI ($V_{th_{HCI}}(t)$) and BTI ($\Delta V_{th_{BTI}}(t)$) are summarised in Table 5.7.

Furthermore, once the threshold voltage shift is calculated for the two considered effects, the respective relative delay degradation ($\Delta^{rel} d$) at time t can be obtained using Equation (5.1).

Table 5.7: Ageing models for the threshold voltage shift due to BTI and HCI effects, and their respective parameters, variables, and constants [71].

$\Delta V_{th_{BTI}}(t) \leq A_{BTI}\left(\frac{\sqrt{D_H \delta T_{cyc}}}{1-\beta^{1/(2n)}(t)}\right)^{2n}$ $\beta(t) = 1 - \frac{\xi_1 t_{ox} + \sqrt{\xi_2 D_H (1-\delta) T_{cyc}}}{2t_{ox} + \sqrt{D_H t}}$ $A_{BTI} = \frac{q}{C_{ox}}\left(\left(K \exp\left(\frac{E_{ox}}{E_0}\right)\right)^2 C_{ox}\left(V_{gs} - V_{th}\right)\right)^{1/(2n)}$ $n = 1/6$	Long-term BTI model for V_{th} shift
$\Delta V_{th_{HCI}}(t) \approx A_{HCI} \cdot \exp\left(\frac{E_{ox}}{E_1}\right) \cdot \exp\left(\frac{-E_a}{kT}\right) \cdot \sqrt{\alpha \cdot f \cdot t}$	Long-term HCI model for V_{th} shift
$D_H = \gamma e^{-E_a/kT}$ $\gamma = 10^8, E_a = 0.13\text{eV}, k = 8.6174 \cdot 10^{-5}\text{eV/K}$	Reaction/Diffusion constants
$\xi_1 = 0.9, \xi_2 = 0.5$	Back-diffusion constants [6]
$t_{ox} = 0.9\ nm$	Transistors oxide thickness [6]
$q = 1.602 \cdot 10^{-19}\ C$	Elementary charge
$C_{ox} = t_{ox}^{-1} \cdot 3.45 \cdot 10^{-22}\ F/nm$	Oxide Capacitance
$E_{ox} = \frac{V_{gs} - V_{th,0}}{t_{ox}}$	Electrical field
$E_0 = 0.08\ V/nm$	Technology dependent constant [6]
$E_1 = 0.8\ V/nm$	Technology dependent constant
$V_{th,0} = \begin{cases} 0.317\ V & \text{for PMOS} \\ 0.271\ V & \text{for NMOS} \end{cases}$	Default transistors threshold voltage
δ	Stress time to total cycle (duty cycle)
α	Transistor switching activity per clock cycle of the circuit frequency
T_{cyc}	Stress-recovery cycle time
T	Temperature
K	Electric field influence
f	Clock frequency of the design
A_{HCI}	Technology dependent constant

$$\Delta^{rel} d(t) = \frac{d(t)}{d(0)} = \frac{(V_{dd} - V_{th,0})^{\sigma}}{(V_{dd} - V_{th,0} - \Delta V_{th}(t))^{\sigma}} = \left(1 - \frac{\Delta V_{th}(t)}{V_{dd} - V_{th,0}}\right)^{-\sigma} \tag{5.1}$$

where: $\Delta^{rel} d(t)$ = relative delay degradation at time t

$\Delta\ V_{th}(t)$ = is the superposition of the threshold voltage shift due to BTI and HCI ($\Delta V_{th_{BTI}}$ and $\Delta V_{th_{HCI}}$ respectively)

V_{dd} = supply voltage

$V_{th,0}$ = transistor threshold voltage at time $t = 0$

σ = technology dependent constant (for modern technologies $\sigma \approx 1.3$ [39])

In this model, it is important to note that the electric field influence K is a proportionality constant in the BTI model, and its exact influence can only be determined experimentally. The same is valid for the technology dependent constant A_{HCI} from the HCI model, it can only be determined experimentally. Therefore, these two constants were determined in such a way that under worst-case conditions (e.g., permanent stress and 100° C), the delay degradation is at most 10% after 3 years, which follows previous results presented in [70, 50].

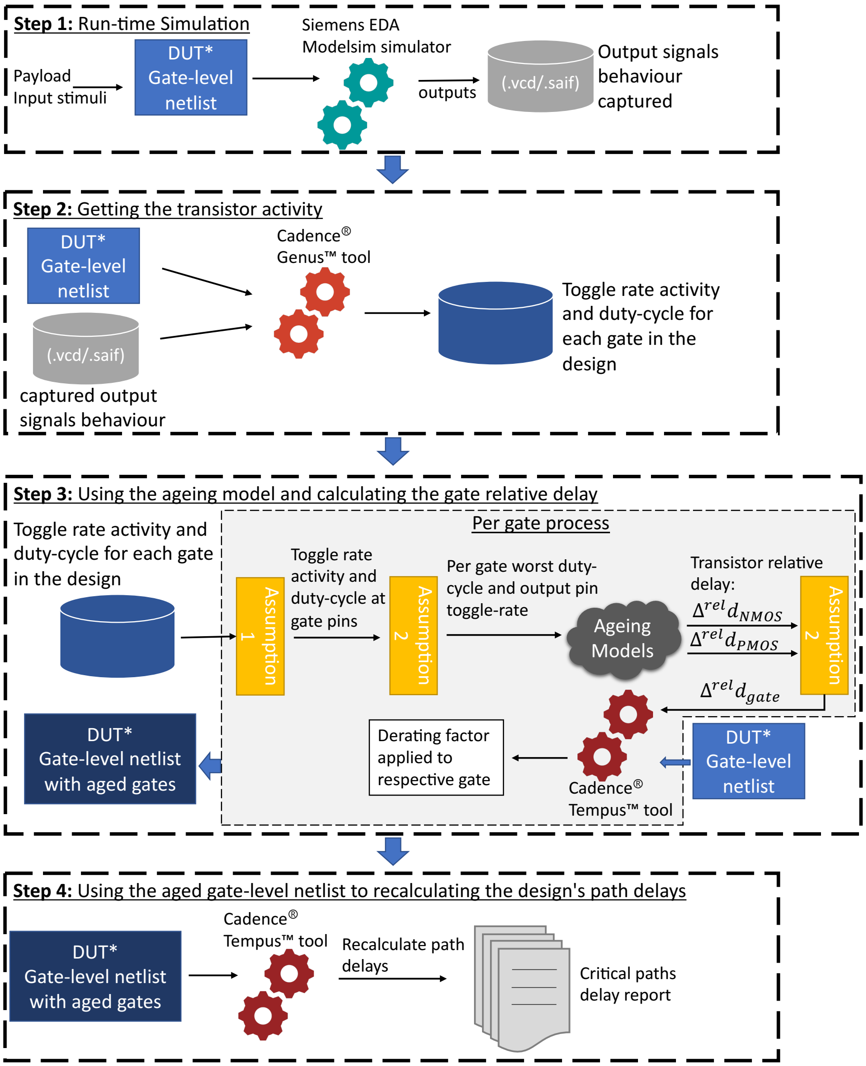
Therefore, it was just described, at the transistor level, the long-term threshold voltage shift (ΔV_{th}) caused by ageing degradation due to the BTI and the HCI effects. And using Equation (5.1) above, it just relates these degradation effects over the threshold voltage to its respective relative delay degradation ($\Delta^{rel} d$) in the transistor. Moreover, using the developed framework described in the next section, this transistor delay is translated into gate delay, and later to path delay. Once reaching this level, the design's critical path can be recalculated and its respective propagation delay as well.

Furthermore, as already explained in Section 2.3, this critical path delay is strongly related to the maximum operating frequency of an electronic circuit, in such a way that its respective period must be, at least, as big as the propagation delay over the critical path. Therefore, in case effects such as BTI and the HCI increase the critical path delay so that it exceeds the clock period of the circuit, special techniques such as graceful degradation, modified flip-flops, or others [27, 30, 52, 64, 66] must be applied to maintain the affected device in operation. After all, the critical path delay is, therefore, the metric used in this section to evaluate ageing effects over the test platform used in this thesis.

Moreover, since the mentioned BTI and HCI models take into account the transistor switching activity (e.g., toggle rate, and frequency) and stress time (e.g., duty cycle), they can be used to estimate the threshold voltage shift and its respective delay degradation according to the design's switching activity of specific payloads. Therefore, the next sections first describe the implemented framework to estimate ageing accordingly to different payloads. And, afterwards, the results of the ageing estimation are presented.

5.5.1 The Ageing Estimation Framework

Together with an implementation of the HCI and the BTI models from Table 5.7, an ageing estimation framework was built, which allowed payload-specific estimation of ageing at the design level based on the transistor ageing models mentioned above. This framework is Tool Command Language (TCL) based and, as it can be seen from Figure 5.2, it calls different software tools for digital design throughout its workflow, which is described in the multiple steps below.



*DUT - Device under test

Figure 5.2: Workflow of the ageing estimation framework.

Step 1: Run-Time simulation

First, using Modelsim, a gate-level design (the design's netlist) is simulated using the desired payload, and its signals switching activities are then saved in a VCD or a Switching Activity Interchange Format (SAIF) file. The intention here, is to capture these signals activities for certain payloads to, at the end of this evaluation, compare the estimated ageing for different payloads.

Step 2: Getting the activity

The same design and the just generated VCD/SAIF activity file are then loaded in the Cadence® Genus™ tool. Once we have the design and the activity file properly loaded, we use the tool to generate a human-readable power report with the input and output pins activity of each gate (e.g., NAND gate, NOR, and INV) in the design. In this stage, we are especially interested in the *toggle-rate* and the *duty-cycle* of the gate pins in order to use these in the ageing models.

Step 3: Using the ageing model and calculating the gate relative delay

Once we get the *toggle-rate* and the *duty-cycle* of the pins for each gate in the design, we use these values in the ageing model to get the respective voltage threshold shift ($\Delta\ V_{th}$), and therefore, the respective relative delay degradation ($\Delta^{rel}d$) for each gate in the design. However, the equations presented in Table 5.7 model the BTI and the HCI effects for *only one transistor*. And, as we know, digital gates are composed of multiple transistors, and these are connected differently depending on the function of the gate. Therefore, a first assumption must be made regarding these internal transistor connections within a gate. Such an assumption should allow us to use the obtained *toggle-rate* and the *duty-cycle* in the input pins of a certain gate, to estimate the respective activity on its internal transistors. Thus, enable the usage of the ageing models from Table 5.7. To this end, the following is then defined:

- **Assumption 1:** It is assumed that all transistors in a gate are directly connected to, at least, one of its inputs. Therefore, there is no intermediate transistor in between the inputs and the output of the gates, and they are all switching together with the switching activity in the input pins of the gate. Figure 5.3 shows a NAND gate as an example in which this assumption is valid. As it can be noticed, all transistors in the gate are directly connected to, at least, one input of the gate. Although this assumption might not be true for all gates used in a design's combinational logic, it holds for the gates that are used the most (e.g., inverters, NAND, and NOR gates). For instance, approximately 80% of the gates in the combinational logic of the core of the Plasma processor fulfil this assumption.

From this first assumption, once we get the switching activity at the input pins of a gate, we then have the activity for all transistors within this gate. In this case, transistors connected to the same pin/pins will age equally. And transistors connected to different inputs in the gate will have different activity rates, leading to different ageing rates. Furthermore,

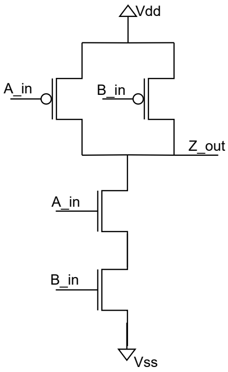


Figure 5.3: Circuit implementation of a NAND gate in Complementary MOSFET (CMOS) technology.

from the ageing models presented in Table 5.7, different $\Delta\ V_{th}$ are expected for p-type MOSFET (PMOS) and n-type MOSFET (NMOS) transistors. Therefore, at the end of this phase, there will be two different relative delays (one for the PMOS and another for the NMOS transistors) for each input of the gate under evaluation.

However, in the used software tools it is only possible to deal with gate-level delays, therefore, it is necessary to find a respective gate-level delay with the just obtained transistor-level delay rates. Therefore, a second assumption is made to cover this situation:

- **Assumption 2:** It is assumed that all transistors in a gate will age homogeneously equal to the most degraded transistor relatively to the switching activity at the input pins of this gate. Although this assumption might not again be true for all gates in a design, it gives us a good estimation of a gate's worst-case degradation based on its input activity.

To follow this assumption first: the *toggle-rate* in the output pin of the respective gate of the transistor under evaluation is taken, so it gives the worst case for the HCI effect; and the highest and lowest *duty-cycle* are taken for the NMOS and the PMOS transistors respectively, so that the highest *duty-cycle* represents the biggest amount of time spent in the BTI stress-phase for the NMOS transistor, the lowest *duty-cycle* represents the same for the PMOS transistor.

Thereafter, using the models from Table 5.7, two different $\Delta\ V_{th}$ are calculated: one for the NMOS and the other for the PMOS transistors. These represent the threshold voltage variation for the most degraded NMOS and the most degraded PMOS transistor within the gate. Secondly, these two different $\Delta\ V_{th}$ are used to calculate their respective relative delay degradation for the NMOS ($\Delta^{rel}d_{NMOS}$) and PMOS ($\Delta^{rel}d_{PMOS}$) transistors using Equation 5.1. Finally, for these two different relative relays, the second assumption is used again getting its worst-case estimation, and the one with the highest relative delay is chosen to represent the relative delay of the entire gate ($\Delta^{rel}d_{gate}$).

Afterwards, this $\Delta^{rel}d_{gate}$ must be applied to the respective gate in the design. For this, the Cadence® Tempus™ tool is used to apply a *derating* factor to the desired gate. This *derating* factor is used to change the original timing information of a desired gate, therefore, it can make a gate faster or slower by applying a factor smaller or bigger than 1.0, respectively. After all, such a *derating* factor is exactly the $\Delta^{rel}d_{gate}$ that was just calculated using Equation (5.1) from the ageing model.

Step 4: Using the aged gate-level netlist to recalculate the design's path delays

Step 3 above needs to be repeated for each and every existent gate in the design, thereby, a relative delay is properly applied to all gates in the design, resulting in an aged gate-level netlist. Once it is done, the Cadence® tool is used again to recalculate the path delays of the design and generate the respective timing reports.

Finally, with these timing reports, it is possible to compare, for different payloads, the delay increase over the design's critical paths due to the estimated ageing. Therefore, it is possible to compare the criticality-aware run-time management of the FUs approach proposed in this thesis with, for example, a case in which redundancy is always enabled over the FUs.

5.5.2 The Ageing Estimation Results

This section intends to evaluate the benefit of the criticality-aware management of FUs for mixed-critical scenarios proposed in this thesis. For this purpose, the four scenarios used for the power overhead evaluation (Section 5.4) are again used here. Quickly describing these scenarios again, we have as it follows:

- **Scenario 1:** Plasma Original design running the Plasma-RTOS without any modification.
- **Scenario 2:** Plasma Extended design fixed in the TMR scheme and running the Plasma-RTOS without any modification.

- **Scenario 3:** Plasma Extended design running the extended Plasma-RTOS and performing the per process management of FUs. Here application process A is running in SlMode, application process B in DMR, and application process C, as well as the Real-Time Operating System (RTOS) functions, are running in the TMR scheme (Figure 5.1).

- **Scenario 4:** The same as scenario 3 with Plasma Extended design and the extended Plasma-RTOS, but here the application processes A, B and C are running in SlMode, the RTOS functions are the only ones running in the TMR scheme (Figure 5.1). Furthermore, in this scenario, a load balancing approach is being used, and each application process runs in a different FU.

Running the ageing estimation framework, described in Section 5.5.1 above, for these four scenarios, we can evaluate their correspondent delay differences, due to ageing, over the critical paths of the design. Table 5.8 shows the critical path delay before (non-aged) and after (aged) applying the ageing estimation framework for the four described scenarios. The *Non-aged* delay presented in the table was the critical path delay calculated right after synthesis. And the *Aged* was the critical path delay calculated after applying the ageing estimation framework for an estimated ageing of 3 years. From the table, it is possible to see that the ageing impact over the critical path delay decreased as the dynamic management of FUs is enabled. For example, the delay degradation (the difference before and after ageing) decreased from 287.99ps for scenario 2, to 222.37ps for scenario 3, and to 214.71ps for scenario 4 in which the dynamic management of FUs is performed. Such a decrease in the delay degradation can have a positive impact on the long-term performance of the electronic device. This means that as long we can avoid the critical path delay to increase, signals are still able to propagate throughout this path within the clock period specified at design-time. Once the critical path delay becomes larger than the intended period, there is not enough time for the propagation of a signal throughout this path. Consequently, if no further technique is applied, the device is no longer able to operate at the frequency specified at the design-time. After all, under the evaluated scenarios and for the tested design, we can say that the criticality-aware management of FUs is indeed able to decrease the ageing rate by reducing the degradation over the critical path delay.

Table 5.8: Critical path delay before and after applying the ageing estimation framework for the four described scenarios.

Evaluated scenario	**Design**	**Critical path delay**		**Delay difference**	**Delay (ageing) reduction**
		Non-aged[1]	**Aged**	**due to ageing**	**due to dynamic managm.**
1 - no dyn. managm.	Plasma Original	922.00ps	1172.79ps	250.79ps	-
2 - no dyn. managm.	Plasma Extended *fixed in TMR*	1139.00ps	1426.99ps	287.99ps	-
3 - dynamic per process managm.	Plasma Extended	1139.00ps	1361.37ps	222.37ps	65.61ps (scenario 3 vs. 2)
4 - dynamic per process managm.	Plasma Extended	1139.00ps	1353.71ps	214.71ps	73.27ps (scenario 4 vs. 2)

[1] results obtained directly after synthesis of the design, therefore, it is no payload dependent.

5.6 Fault Tolerance Evaluation

In this section, we will first see a theoretical evaluation regarding the reliability of the different FUs schemes proposed in this thesis. Later on, a practical and comprehensive fault tolerance evaluation is presented using fault injection campaigns.

5.6.1 Theoretical Evaluation

A standard method to evaluate the fault tolerance of a system is in terms of its *reliability*. As we have seen, the reliability $R(t)$ of a system is the probability that it will deliver the correct service in the time interval $[0,t]$, given that it was performing correctly at instant zero ($t = 0$). Following an exponential lifetime distribution, and for a constant failure rate λ, we have that the reliability of a system at a specific point in time ($R(t)$) is defined as Equation (5.2) [53].

$$R(t) = e^{-\lambda t} \tag{5.2}$$

When a group of hardware modules is working together we can calculate the reliability of the entire system (composed of these modules). For such a calculation, one must take into account how these multiple modules interact with each other. First of all, we must assume that the modules are independent of each other, therefore, a failure in a particular module would not generate another failure in the other neighbour modules, and neither affects their individual reliability. This assumption is taken for all the upcoming reliability models mentioned in this section.

A system that fails if one of its participant modules fails, is called a *series system*, and its resulting reliability is obtained by the product of the individual reliability of the n participant modules, as it is defined by Equation (5.3).

$$R_{series}(t) = R_{mod_1}(t) \cdot R_{mod_2}(t) \cdot \ldots \cdot R_{mod_n}(t) = \prod_{i=1}^{n} R_{mod_i}(t) \tag{5.3}$$

In case we have n identical modules, the reliability for a series system is as stated in Equation (5.4).

$$R_{series}(t) = (R_{mod}(t))^n \tag{5.4}$$

Another possibility is a *parallel system*, which is defined as a set of n modules working together so that it requires that all participant modules fail for the system to fail as well. Therefore, to calculate the reliability of the system, we first calculate its *unreliability* (the probability for all modules failing), which is the product of the individual unreliability of each module ($\prod_{i=1}^{n}(1 - R_{mod_i}(t))$). Finally, for the final system reliability, we subtract this unreliability from the whole probability space. Therefore, the resulting reliability of a parallel system is as it is stated in Equation (5.5).

$$R_{parallel}(t) = 1-(1-R_{mod_1}(t))\cdot(1-R_{mod_2}(t))\cdot\ldots\cdot(1-R_{mod_n}(t)) = 1-\prod_{i=1}^{n}(1-R_{mod_i}(t)) \tag{5.5}$$

And, again, in case the system is formed by n identical modules, the reliability for this parallel system is as stated in Equation (5.6).

$$R_{parallel}(t) = 1 - (1 - R_{mod}(t))^n \tag{5.6}$$

However, not all systems fit into the series or parallel model, so that there are other models in the literature to cover these. Here, we are particularly interested in models that fit into the TMR and DMR replication schemes used in this thesis. First, regarding the TMR scheme, it is classified as an M-of-N system, which consists of a system of N identical modules that fails when fewer than M modules are working correctly, thus, the TMR is a 2-of-3 system. With this, we have that the reliability of any generic TMR system ($R_{TMR}(t)$), is composed in terms of the individual reliability of its redundant modules ($R_{mod}(t)$) and the voter ($R_{voter}(t)$), as it is defined by Equation (5.7) [53].

$$R_{TMR}(t) = R_{voter}(t) \cdot (3R^2_{mod}(t) - 2R^3_{mod}(t)) \tag{5.7}$$

Continuing in this line, to calculate the reliability of any standard DMR scheme, we must add the coverage factor C that represents the probability of the faulty module to be correctly diagnosed, identified and corrected. With this, we have the reliability of such a system ($R_{DMR}(t)$) defined as Equation (5.8) in terms of this factor, and the individual reliability of the modules ($R_{mod}(t)$) and the comparator ($R_{comp}(t)$) [53].

$$R_{DMR}(t) = R_{comp}(t) \cdot (R^2_{mod}(t) + 2CR_{mod}(t)(1 - R_{mod}(t))) \tag{5.8}$$

Regarding the respective schemes proposed in this thesis, we can say that they follow the equations (5.7) and (5.8) above. Therefore, the modules are actually the FUs, and the voter for the TMR and the comparator for the DMR are represented by the *Programmable Voter*, which is present in the two schemes and is responsible for both the voting and the comparing in the TMR and the DMR schemes respectively.

Finally, we are still missing the SlMode scheme, in which we have only one module working in series with the *Programmable Voter*. It is important to notice here, that the *Programmable Voter* is not doing any voting, and neither comparing any signal. But in this scheme, it is only bypassing the incoming signals from the working module (one of the FUs) to the outgoing hardware modules in the remaining of the design. Nevertheless, this voter is still in between the modules and the remaining of the design, thus, a series system. Therefore, the reliability for this scheme ($R_{SlMode}(t)$) follows the simple series model as it is shown in Equation (5.9).

$$R_{SlMode}(t) = R_{voter}(t) \cdot R_{mod}(t) \tag{5.9}$$

With these three equations ((5.7), (5.8), and (5.9)) we can calculate the theoretical reliability for the three different configuration schemes proposed in this thesis. Therefore, to calculate this, let us assume that the failure rate is constant, and at a certain point in time t, the individual reliability of each FU is $R_{mod}(t) = 0.90$. With this, we calculate the reliability for the three different schemes taking into consideration different values for the reliability of the

Programmable Voter up to an ideal case, though unreal, with voter reliability $R_{voter}(t) = 1$ (Table 5.9).

Table 5.9: Theoretical reliability estimation.

$R_{voter}(t)$	$R_{mod}(t)$	$R_{SlMode}(t)$	$R_{DMR}(t)^*$	$R_{TMR}(t)$
0.80	0.90	0.720	0.792	0.778
0.90	0.90	0.810	0.891	0.874
0.91	0.90	0.819	0.900	0.884
0.92	0.90	0.828	0.911	0.894
0.93	0.90	0.837	0.921	0.903
0.94	0.90	0.846	0.931	0.913
0.95	0.90	0.855	0.940	0.923
0.98	0.90	0.882	0.970	0.952
0.99	0.90	0.891	0.980	0.962
1.00	0.90	0.900	0.990	0.972

$^*C = 1$

It is important to notice here that for the DMR scheme, the probability factor C was considered to be one, which means that a possible high-level mechanism would always be able to, once detected, correctly diagnose, identify and correct a possible fault. Disregarding time, such behaviour provides the same benefit as the TMR, but with less hardware. Therefore, from the reliability point of view, it means fewer sources of errors, which justifies the higher resulting reliability, from Table 5.9, for the DMR scheme when compared with the TMR. However, in real applications, this probability factor C is actually smaller than one. Furthermore, this correction provided by a possible high-level mechanism, and assumed to be working together with the DMR scheme, would also take additional time, which is not the case for the TMR. After all, since evaluating these high-level mechanisms is not the objective of this thesis, this assumption is done so we can better compare the reliability of the different configuration schemes.

As it can be noticed from Table 5.9, once the reliability of the voter is high enough (e.g., higher than or equal to 0.92 for the DMR, and higher than or equal to 0.93 for the TMR for the cases presented in the table), the reliability of the DMR and the TMR schemes are as such that they have a higher value than the reliability of one module alone. Therefore, we can conclude that, if certain conditions are met (e.g., from the case in the table $R_{voter}(t) \geq 0.95$, and $R_{mod}(t) = 0.90$), *these two schemes do increase the reliability of a module*, as it is proposed for the FUs in this thesis.

5.6.2 Practical Evaluation

A framework based on ModelSim simulator was implemented to perform the fault injection campaigns, and its workflow is illustrated in Figure 5.4. As it shows, it first runs a *golden* simulation with no interference. After that, it runs several other *faulty* simulations, in which a fault is injected on each simulation round. The fault is injected within the simulation time window at a randomly selected point in time, in a randomly chosen design signal. Since the target of this thesis is to cope with soft Single-Event Effect (SEE), the actual fault model performs a bit-wise inversion of the value of a signal (bit-flip). Thus, it is equivalent to a Single-Event Transient (SET), or a Single-Event Upset (SEU) if the inversion is captured by a latch or flip-flop. Finally, it uses ModelSim again to compare both the *golden* and the *faulty* simulations. In the end, we have the number of injected faults that generated an error in the output ports of the design.

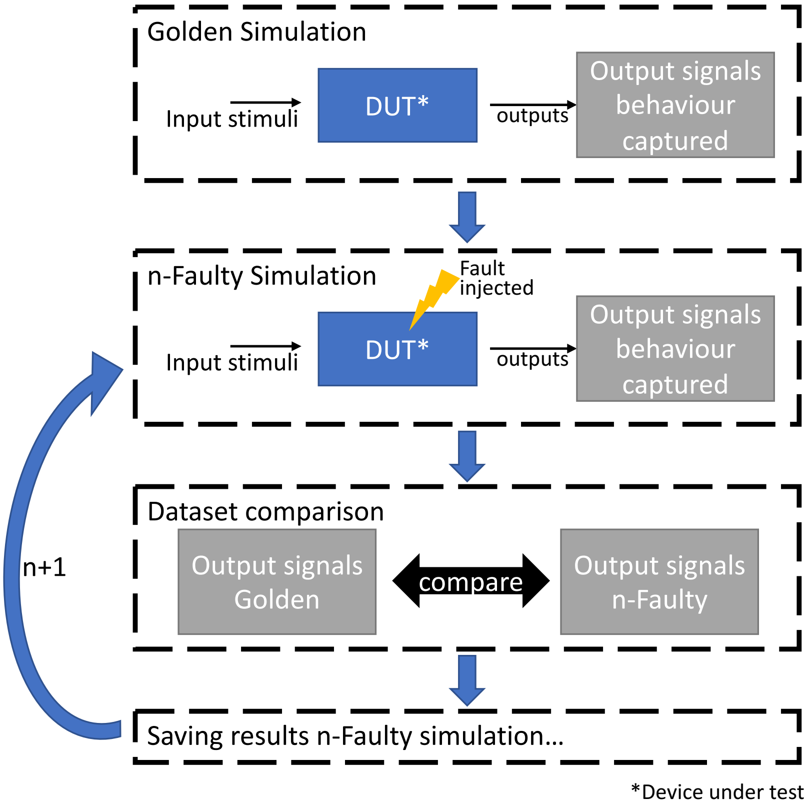


Figure 5.4: Workflow of the fault injection campaign. First, a golden simulation, then n-faulty simulations are performed.

However, due to the large size of electronic designs, it becomes very time consuming, or even unfeasible, to run a fault injection for all design signals at each unit of time. For the core of the Plasma processor design, for example, there are around two hundred thousand signals, and considering that a fault can cause different effects at each clock cycle, there would be around *a hundred billion* of possible points to inject a fault for one millisecond of simulation time, considering an operating frequency of $500MHz$. Therefore, to shorten the time spend on these campaigns, the design must be sampled. For this sampling, the initial population (N) consists of these billions of possible points to inject a fault (all possible signals at any clock cycle). It is assumed that the characteristics of this population follow a normal distribution, which means that each individual (e.g. a fault possibility at a given clock cycle) from the initial population must have the same probability to be selected in the sample [56]. Such a sample must be big enough to represent, with a reasonable error margin, the whole set of signals and possibilities for injection of faults throughout the simulation time. Therefore, from statistical theory, the Equation (5.10) can be used to find the sample size (n) of a finite population of elements (N) that follow a normal distribution for given different error margins (e) and confidence levels (t). This approach of defining the sample size for fault injections campaigns was presented by Leveugle et al. in [56], and has been used and acknowledged by many authors in the test community in works such as [13, 14, 19, 20, 63, 77, 97].

$$n = \frac{N}{1 + e^2 \cdot \frac{N-1}{t^2 \cdot p \cdot (1-p)}} \tag{5.10}$$

where: n = represents the sample size (number of faults to randomly select for injections;

N = the initial population;

p = the estimated probability of faults resulting in a failure;

e = the error margin;

t = the cut-off point (or critical value) corresponding to the confidence level, this level is the probability that the exact value is actually within the error interval. The cut-off point is computed with respect to the Normal distribution (quantile table).

The calculated sample sizes for different error margins and confidence levels are presented in Table 5.10. It is important to notice that the design is being simulated at gate-level, in which, again, there are this very big amount of signals to be simulated by the tool (∼200000 signals). Therefore, each simulation round can take a considerable amount of time, from several seconds to a few minutes. Looking for a feasible sample size, in which the fault injection campaigns could be performed using a reasonable amount of time, the option for an error margin of 1% with a confidence level of 99% was chosen: 16586 samples.

Table 5.10: Calculated sample sizes for different error margins and confidence levels obtained using Equation (5.10).

	$t = 1.96$ (95% conf.)	$t = 2.5758$ (99% conf.)	$t = 3.0902$ (99.8% conf.)
$e = 5\%$	384	663	955
$e = 1\%$	9604	**16586**	23873
$e = 0.1\%$	959180	1655052	2379811

In fact, the real amount of time for each simulation depends not only on the design, but also on the payload. Regarding this, the open-source Powerstone benchmark suite [59] was taken. This benchmark contains a collection of embedded and portable applications, therefore, suitable for the intended application domain of this thesis. However, due to limitations of the software library provided by the Plasma CPU project, not all benchmarks could be compiled for the design. Furthermore, there were also additional benchmarks that were compiled successfully, but their required run time were too long, therefore not feasible for the several gate-level simulations needed by this fault injection campaign, in which a few milliseconds of a single virtual simulation can take several minutes or even hours depending on the simulated design. After all, four different benchmarks from the Powerstone benchmark suite, and one test program available together with the Plasma CPU project [75] were used as payloads for the fault injection campaign. These five benchmarks include applications performing shift and adding, graphics functions, cyclic redundancy check (CRC) calculation, paging communication protocols and a test program testing all MIPS I opcodes supported by the Plasma processor. Table 5.11 summarises these benchmarks and shows the required virtual simulation time for each of them.

Regarding the different designs tested in this section. The fault injection campaigns were performed through four designs. These four designs consist of the Plasma unmodified version (original), and the remaining three are of the Plasma extended design fixedly running in SlMode, DMR and TMR scheme. There are different run times for each design and each payload.

The fault injections were performed in a server machine with its cores running at $2.3GHz$ (its Central Processing Unit (CPU) model and vendor will not be disclosed due to confidentiality). An individual simulation cannot be divided into different processes, therefore each simulation runs only on one processor core. However, the n-Faulty simulations are each a different process that can be distributed to run in parallel through the cores of a machine. Table 5.12 illustrates the average amount of time used per CPU core in each simulation for the five benchmarks and each design tested. Finally, this table also shows a summary of all CPU time used for these fault injections campaigns using the calculated sample size from above. It can be noted that taking into account all the designs simulated, all the five different payloads, and the number of simulations that needed to be performed to reach the calculated sample size n, the total amount of CPU time used for the fault injection campaign was approximately 8307 hours.

Table 5.11: Benchmarks used as payloads for the fault injection campaigns and their required simulation time.

Benchmark	**Description**	**Simulation time**
Opcodes2	Test all supported MIPS I opcodes	0.2ms
BCNT	Bit shifting and anding through 1K array	2.0ms
CRC	Cyclic redundancy check	3.0ms
Blit	Graphics application	3.5ms
POCSAG	Communication protocol used to transmit data to pagers	4.5ms

Table 5.12: CPU time used for each benchmark simulation and the full fault injection campaign.

Benchmark	CPU Time			
	Plasma Original	**Plasma Ext. SlMode**	**Plasma Ext. DMR**	**Plasma Ext. TMR**
Opcodes2	~3 s	~3 s	~4 s	~5 s
BCNT	~60 s	~66 s	~89 s	~135 s
CRC	~69 s	~76 s	~105 s	~148 s
Blit	~87 s	~95 s	~121 s	~175 s
Pocsag	~95 s	~113 s	~150 s	~204 s
Total time (x16586)	~5208004 s	~5854858 s	~7778834 s	~11062862 s
Total CPU time	**~299045558 s (~8307 hours)**			

As already mentioned, the fault injection campaign was performed through the unmodified (original) version of the Plasma processor design, and the extended version presented in Section 4.2 in three different configurations: in SlMode, DMR and TMR scheme. These designs were synthesised and mapped to OCL 15nm [61], and, therefore, these are the evaluated Devices Under Test (DUTs), as presented in Figure 5.4. After performing the fault injections, it was possible to get the amount of injected faults that propagated through the design and generated an error in the output ports of the Plasma processor core. Therefore, the rate at which faults propagate through the design or, in other words, the vulnerability of the design. Such a metric, when attributed to a specific structure of an electronic design (e.g., an internal module, or the whole processor core) is called to be the Architecture Vulnerability Factor (AVF). This factor, together with two further introduced metrics, are used to illustrate in the next sections the design's fault tolerance regarding the intended fault model (SETs and SEUs).

Architecture Vulnerability Factor Evaluation

The AVF is defined as the probability that a fault in a particular structure will result in a visible error in the final output of a program [69]. In the fault injection campaigns, this probability is actually the rate of injected faults, in a given design or structure, that resulted in a visible error in the output signals of the tested designs - therefore a failure - (Equation (5.11)).

$$AVF = \frac{failures}{injected\ faults} \tag{5.11}$$

Figure 5.5 shows the calculated AVFs for all the tested designs and for each of the prepared payloads, with additional columns to illustrate the average AVF values of the designs. Notably, the overall vulnerability of the design decreased for the Plasma Extended processor in all evaluated configurations. However, as expected, the most prominent vulnerability improvement was over the extended design when the TMR scheme was enabled. This means that it is less likely that a fault will become an error for this design while in this scheme.

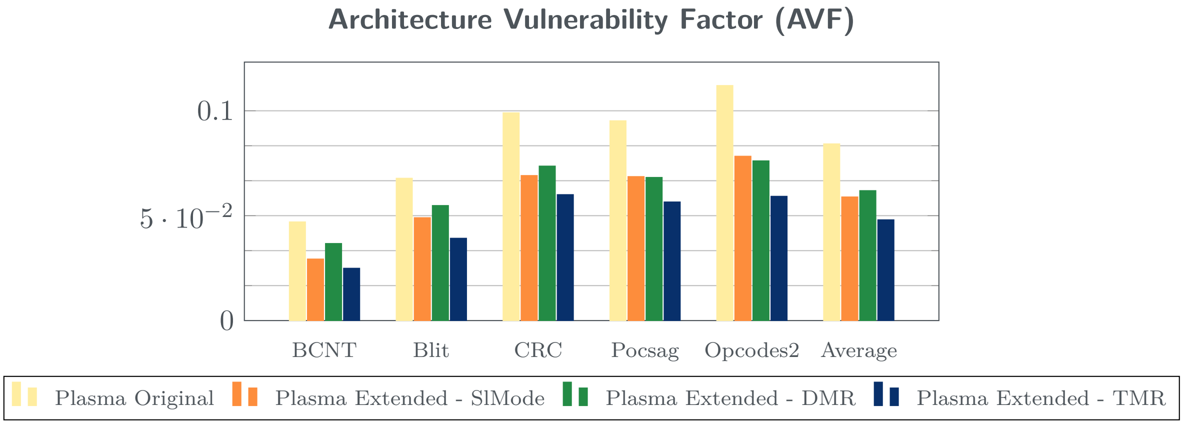


Figure 5.5: AVF for Plasma original and the extended design in different configuration schemes.

Figure 5.6 shows a deeper vulnerability analysis of the internal modules of the processor design. The internal modules are shown as described in section 4.1. What is important to notice here, is the "*FUs Module / FUs Wrapper*" column that evaluates the vulnerability for: all the FUs of the Plasma Original processor; and the extended FUs module of the Plasma Extended design that contains all FUs and the additional control logic for units management (such a module is represented by the *FUs Wrapper* in Figure 4.2). It is very important to notice in this column that the extended FUs module, when configured with the TMR scheme, presented a near null vulnerability factor. This means that for this scheme, it is very unlikely (*with a near zero probability*) that a fault will propagate generating an error in the module and an erroneous behaviour in the output pins of the processor core, thus generating a failure.

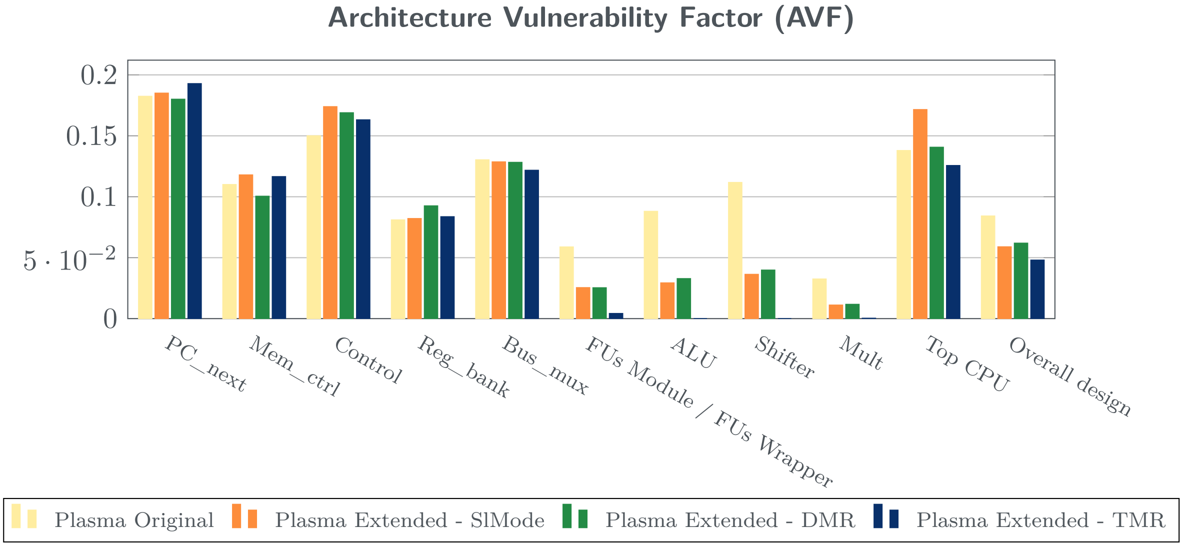


Figure 5.6: Average AVF of the internal modules of Plasma original and the extended design in different configuration schemes.

However, since we are evaluating the vulnerability of the design regarding faults caused by external particles hitting the device silicon (e.g., SEE). It is also necessary to use a metric that takes into account the area of the tested designs. Because an increase in silicon area would, consequently, increase the chances of a charged particle or neutron hitting the device. Therefore, the cross-section metric is used for this evaluation, which takes into account the AVF and the area of the tested designs.

Cross-Section Evaluation

In the context of electronic devices testing, the cross-section σ is defined as the *radiation-sensitive area* of a device [84]. It is usually obtained experimentally by fault injection experiments using the Equation (5.12) defined below.

$$\sigma = \frac{\lambda}{\phi} \tag{5.12}$$

where: σ = cross-section;
λ = observed error output rate, thus the failure rate
ϕ = particle flux

Where λ is the observed failure rate (e.g., the number of failures per unit of time), and ϕ is the particle flux, which represents the rate of particles hitting the device silicon per unit of area. Therefore, the units attributed to each of these metrics are the illustrated in Equation (5.13) for λ and in Equation (5.14) for ϕ.

$$\lambda = \frac{[failures]}{[time]} \tag{5.13}$$

$$\phi = \frac{[particles]}{[area \cdot time]} \tag{5.14}$$

Putting these all together in Equation (5.12) we have as it shows in Equation (5.15), and as it shows, after working a little on the formula, we have defined our cross-section in terms of what it actually represents.

$$\sigma = \frac{\lambda}{\phi} = \frac{[failures \cdot time^{-1}]}{[particles \cdot area^{-1} \cdot time^{-1}]} = \frac{[failures]}{[particles \cdot area^{-1}]} = \frac{[failures \cdot area]}{[particles]} \tag{5.15}$$

If we consider the number of faults injected in the fault injection experiments as the *particles* from the equations above, and the number of failures generated by these injected faults as the *failures* from the equation above. We have then defined a way to calculate the cross-section of the tested designs, and its final result would represent the *fault-sensitive area* of that given design. Furthermore, as we have seen above, the AVF is exactly this factor between the observed failures and the injected faults. Therefore, we can finally calculate the cross-section σ in terms of the AVF and the respective area of a design, as defined by the Equation (5.16).

$$\sigma = \frac{[failures]}{[injected\ faults]} \cdot area = AVF \cdot area \tag{5.16}$$

Therefore, we can calculate the cross-section σ of the four tested designs by multiplying the obtained AVF from the section above by their corresponding area obtained as a result of the synthesis presented in Section 5.2.

The resulting cross-sections for each of the payloads and the tested designs are presented in Figure 5.7. It is notable that the fault-sensitive area of the device did decrease for the extended design when enabled with TMR over its FUs. This means that despite being bigger than the original design, it is less likely that a particle hitting the design will cause a failure in this extended version when enabled with this scheme.

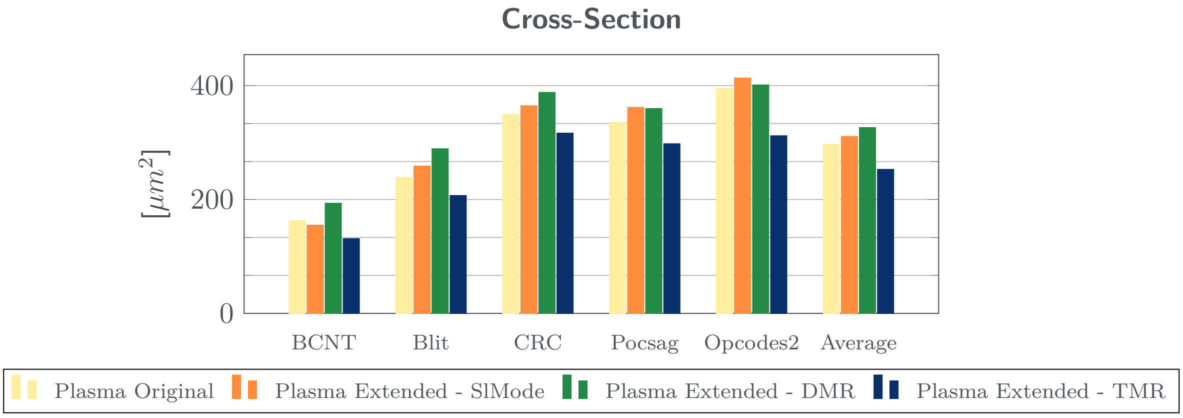


Figure 5.7: The cross-section for Plasma original and the extended design in different configuration schemes for the different benchmarks.

Figure 5.8 shows a deeper analysis of the cross-section for the internal modules of the Plasma processor design. From the figure, it is noticeable that the most relevant differences are in the cross-section size of the "*FUs Module / FUs Wrapper*" and in the overall processor core. The overall design improved just as we have explained. But for the FUs module, its fault sensitive area did decrease reaching almost none area.

On the other hand, looking at the figures it is notable that the design got an average increase on its cross-section, while configured with SlMode and DMR scheme. It is, however, expected because the extended version of the Plasma processor is bigger than its original. And since

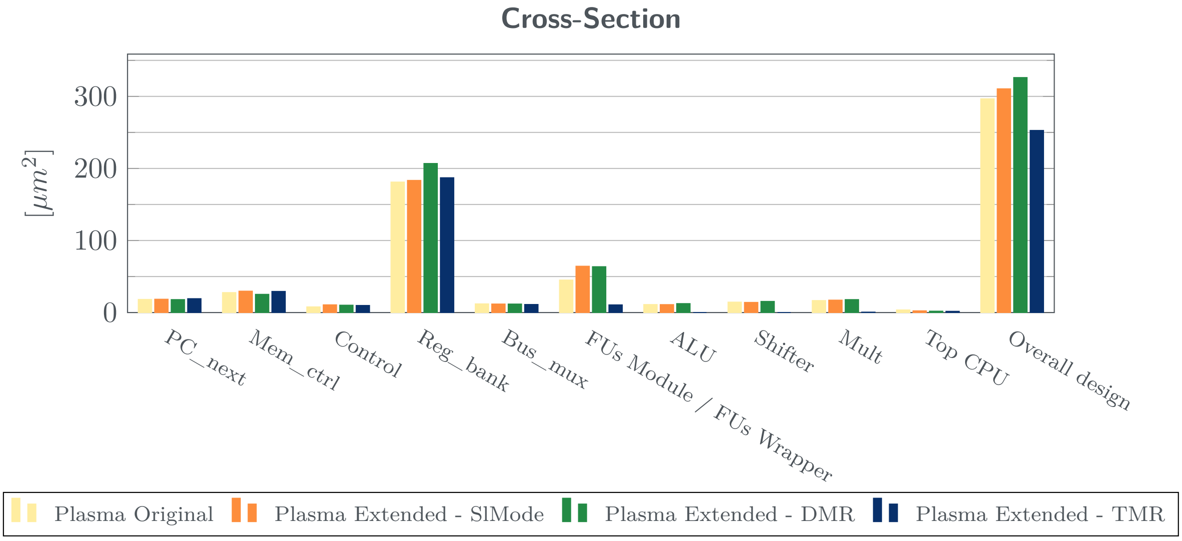


Figure 5.8: Cross-section average value of the internal modules of Plasma original and the extended design in different configuration schemes.

no error is being corrected in these modes (only error detection is enabled in the DMR), the device only became more susceptible to faults, since it is bigger and easier for a particle to hit its silicon. Another interesting analysis is regarding the cross-section increase while the design is in the DMR scheme when compared to the SlMode one. This increase happens because a bigger portion of the design is now active. Therefore, since the DMR scheme enables two units, while the SlMode enables only one, it is more likely that a particle will hit an area that is being used, which will easier activate and propagate the fault. However, as it is stated in previous sections, the DMR scheme not only enables error detection, but also the usage of strategies for error correction on-demand, which can compensate for the increased susceptibility to faults of this scheme. Therefore, it can be used, for example, with applications that can accept the time overhead of these strategies but still need to tolerate faults by correcting them on-demand.

Failure Rate at Sea Level

As it is already mentioned in Section 2.1.2, at sea level, the neutron particles are the ones that cause the most prominent effect over electronic devices and, therefore, are more likely to produce soft SEE. Furthermore, according to the Joint Electron Device Engineering Council (JEDEC), in its technical report JESD89A, it is expected that neutrons hit the Earth's ground at sea level at a rate of $13 neutrons/(cm^2 * h)$ [48]. Additionally, from

Equation (5.12) we have that the failure rate of a device can be calculated as defined by Equation (5.17) below. Finally, entering in this equation the obtained cross-section σ and the flux ϕ of neutron particles at sea level, it is possible to calculate the expected failure rate λ at sea level for the above tested processor designs. Furthermore, the corresponding Failure In Time (FIT) rate can also be obtained by multiplying this failure rate by 10^9, which represents the number of detected failures in one billion hours of device operation.

$$\lambda = \sigma \cdot \phi \tag{5.17}$$

Figure 5.9 illustrates the calculated failure rate (failures/hour) and the corresponding FIT rate for the evaluated designs and each of the payloads as well. Furthermore, a comparison was done between the unmodified version of the processor design (Plasma Original) and the three different configuration schemes of the Plasma Extended. Figure 5.10 shows these rate comparisons for the different payloads and an average result. In the figure, it is possible to see that, for the TMR scheme, the failure rate at sea level decreased for all payloads. As a result, it presented an average decrease in this rate of approximately 15%.

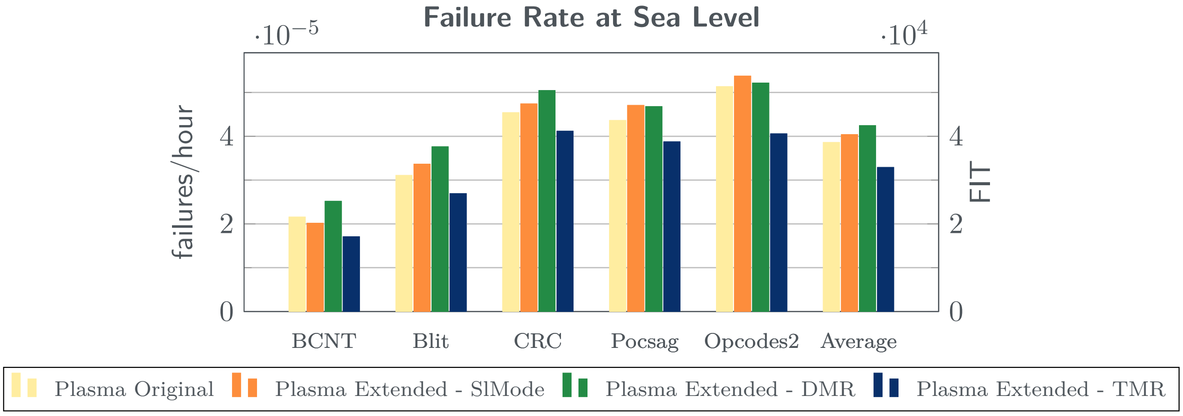


Figure 5.9: Calculated failure rate at the evaluated designs at sea level for the obtained cross-section and the given flux ϕ of neutron particles.

However, if we look closer at the internal modules again, Figure 5.11 illustrates the average calculated failure rate (failures/hour) and the corresponding FIT rate for the internal FUs Module and its units within. Furthermore, Figure 5.12 shows again a comparison of the Plasma Original and the Extended, but it is now highlighting this mentioned internal FUs module. From the figure, it is possible to see that the failure rate decreased by a factor of approximately 80% for this module when configured with the TMR scheme, and regarding its internal units, these get their sensitive area decreased by almost 100%. Therefore, it is

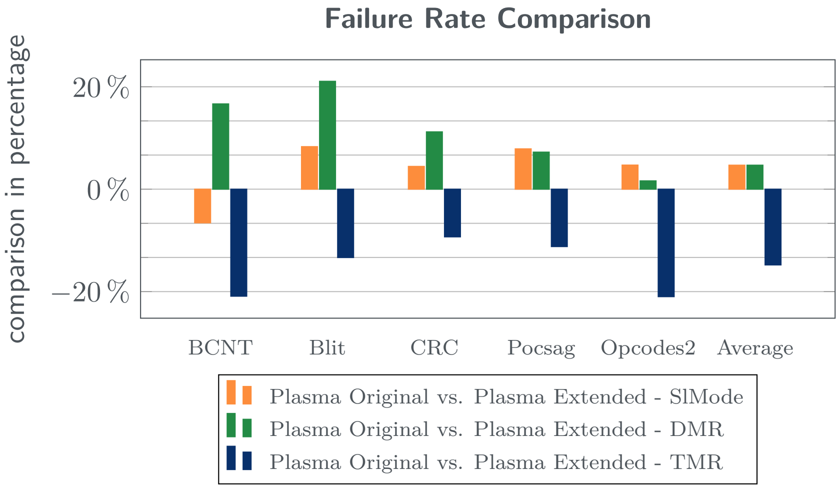


Figure 5.10: Failure rate at sea level comparison for different benchmarks and the evaluated designs.

possible to say that, by adding the necessary control logic for the management of the units, and when configured with TMR, this module became *approximately 80% less sensitive to the studied faults*, if compared to its original and non-fault tolerant implementation.

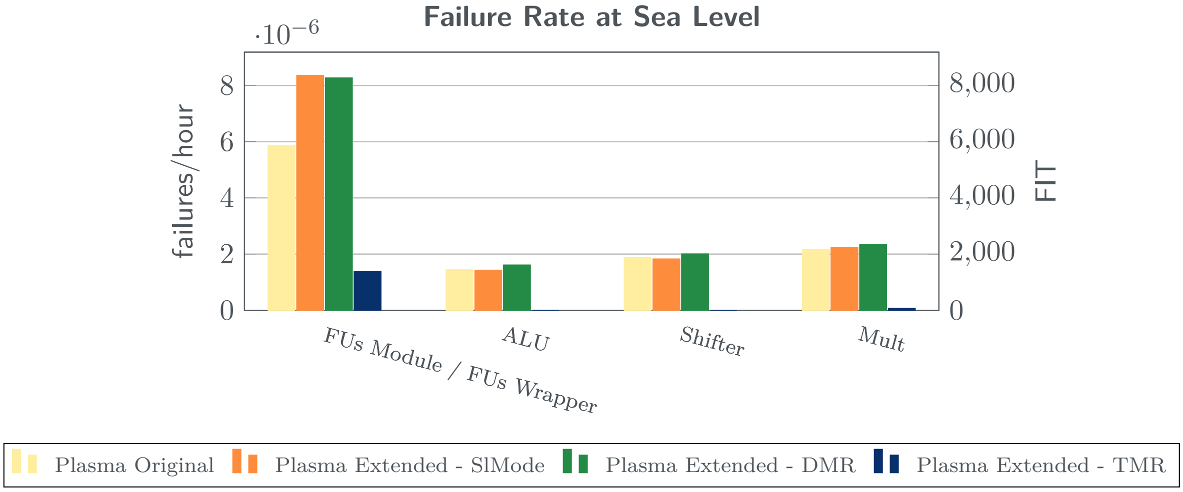


Figure 5.11: A closer look at the FUs module calculated failure rate at the evaluated designs at sea level for the obtained cross-section and the given flux ϕ of neutron particles.

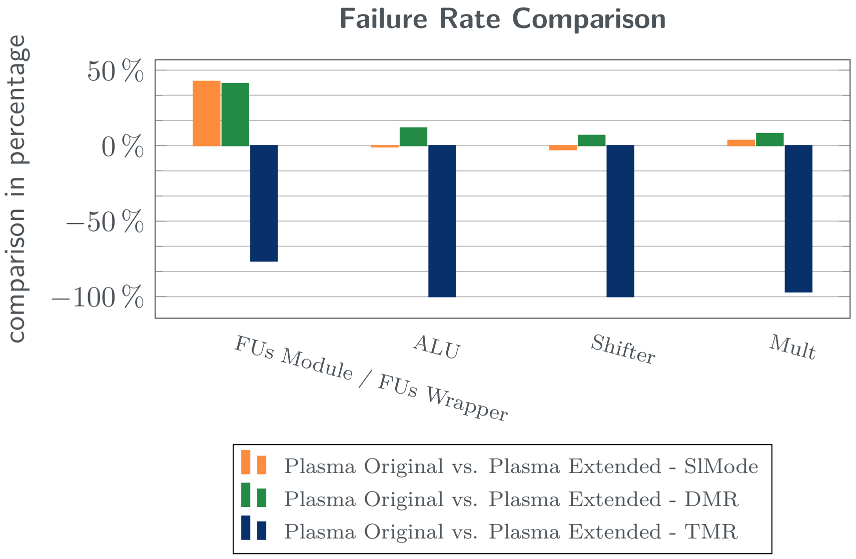


Figure 5.12: Failure rate at sea level comparison for the FUs module of the evaluated designs.

To Conclude

To summarise, we could see throughout this fault tolerance evaluation section that the replication method applied to the FUs did decrease the failure rate of not only its internal FUs module, but also the whole design. Given the evaluated scenario, if we assume that the remaining units and registers of the processor can be protected by other methods (e.g., Error Correction Code (ECC) in the register file), it is possible to say that the mechanism proposed in this thesis *can decrease the failure rate of the FUs module by approximately 80%* when TMR is enabled on this module. Still, if nothing else is done to improve the fault tolerance of the processor design, my mechanism can, *alone*, improve the failure rate of the *whole processor by approx. 15%* when the same triplication scheme is used.

5.7 Summary

This chapter evaluates the implemented test platform in many different aspects. It first presented the latency to change between the FUs scheme, in which two cases were presented: while running bare-metal code, and while running an OS. For the first, we could see that a latency of only *one instructions cycle* was necessary to change the FUs scheme. For the OS we could see that only a few clock cycles were added to the process switching mechanism of the OS, accounting approx. 2.4% of latency overhead to switch between processes. Secondly, the area overhead to implement the hardware management mechanism was presented. It was possible to see that the extra FUs were responsible for the most for the area increase. However, if considering only the control logic overhead, we could see that the area increased by about $216.2um^2$, which means less than 6.2% of overhead compared to the original design of the core of the Plasma processor. Nevertheless, taking altogether this scheme's overhead is still below standard replication schemes such as core lock-step approaches. Continuing, the power overhead was evaluated, and it was possible to see that the run-time management of FUs can decrease and optimise the power consumption of the proposed system for the proposed mixed-critical scenario. Furthermore, regarding the increase in the critical path delay. Despite the 6% increase in the critical path delay, the design could still reach its desired operating frequency at design-time.

Regarding fault tolerance, a very comprehensive evaluation was presented. Through fault injections campaigns at gate-level, this chapter presented how the implementation of the management mechanism affected positively and negatively the design. Nevertheless, under the evaluated scenario, the average failure rate of the internal FUs module, when configured with the TMR scheme, decreased by approximately 80% if compared to its original version. Furthermore, evaluating the whole design, it was possible to see that the mechanism proposed in this thesis can, *alone*, decrease the failure rate of the *whole processor design* by approximately 15% when configured with the same triplication scheme.

Finally, when evaluating ageing. A framework was implemented that enabled the measurement of critical path delay increase regarding payload-dependent ageing effects. With this framework, it was possible to notice that the criticality-aware management of FUs could successfully decrease the ageing rate over the design's critical path.

CHAPTER 6

Conclusions

Systems targeting reconfiguration for fault tolerance can be especially interesting in the domain of mixed-critical systems. Because tailored designs for this domain tend to be overestimated for the worst-case scenarios, changing system configuration or redundancy schemes to match the current criticality of running tasks can avoid unnecessary power consumption and usage of hardware resources. Within the mixed-critical context, but in the domain of processor designs, management of hardware resources has been done mostly at the level of cores, and very few further approaches deal with internal units of the processor core, for example. Furthermore, even fewer works link these low-level hardware units with software-level management, and if so, no proper study is presented to evaluate the effects that such a level of management can provide. To cover this gap, this thesis just presented the concept and its implementation details for a software-managed hardware mechanism that enables run-time management of processor internal Functional Units (FUs). With such a mechanism, it was possible to enable, on-demand, redundancy schemes of the internal units of the processor design, therefore, increasing the system's fault tolerance.

Targeting soft Single-Event Effects (SEEs), Section 5.6 presented a fault tolerance evaluation. It is shown that once Triple Modular Redundancy (TMR) is enabled over the FUs, the expected failure rate over the FUs module decreased by approximately 80% if compared to its original implementation. Furthermore, when evaluating the whole processor, it was possible to see that the proposed mechanism could, *alone*, decrease the failure rate by approximately 15% when compared to the Plasma Original version (reaching *objective 2*).

Additionally, as Section 5.1 presented, the proposed concept enabled very agile management of hardware FUs, taking *only one instruction cycle* to configure the desired scheme of the units down the hardware (reaching *objective 1*).

Furthermore, using the presented design, it was possible to enable the desired scheme of the FUs not only fast, but on-demand at run-time. And when complemented with an Operating System (OS) and the proposed extensions, the presented platform was able to perform criticality-aware management of its FUs for mixed-critical processes at a cost of approximately 2% of increase in the execution time of the process switching mechanism of the Plasma Real-Time Operating System (Plasma-RTOS) (reaching *objective 3*). This was only possible because of the very fast mechanism based on the newly created instructions, which enabled this low latency to enable or disable a desired redundancy scheme of the units down the hardware. Such a low latency could enable even smaller configuration granularity, for example, inside program loops.

Moreover, because of the criticality-aware management of FUs, it was possible to enable and disable the replication schemes as the criticality of the running applications changed. For example, in Section 5.4, it was possible to see that the switching between different replication schemes as the criticality of the currently running process changed, decreased the overall power consumption if compared to permanently enabled replication. The actual decreasing factor depends on the amount of time the design spends on each of the possible replication schemes. But for example, for one of the cases presented in Section 5.4, instead of adding a power overhead of approximately 98%, the dynamic per process management strategy allowed an overhead of only 16.2%. Such a decreased power overhead was observed in the scenario in which only the OS is executed in TMR, while the remaining processes are in Single Unit Mode (SlMode).

Furthermore, when evaluating ageing, we could see that the criticality-aware management of FUs could decrease the ageing rate over the design's critical path, by lowering the degradation rate of its respective propagation delay.

Regarding the impact over the hardware, Section 5.2 presented that the overall hardware overhead (∼50%) to implement the hardware mechanism is very dependent on the area of the FUs and, a part of the extra area incurred for triplicating these units, the overhead for the control logic was only about to ∼6%. This means that in case the FUs are already part of the intended design, the only significant overhead would be this small additional control logic. Furthermore, because of the additional hardware elements in the pipeline of the processor design, there was an increase in the critical path delay. However, when pushed closer to its operating limits, targeting an operating frequency of 1GHz, the propagation delay of the critical path increased by approximately 6%. Nevertheless, despite this increase, the synthesis tool was still able to synthesise the design so that the timing constraint required for this frequency was met.

Finally, after performing all the evaluations described above, it was possible to see that my concept was able to increase fault tolerance on-demand of a processor design, while minimising its power consumption and hardware usage (ageing) penalties for its intended mixed-critical scenario. Such a feature was provided at a cost of moderately low hardware overhead when compared to other course-grained replication approaches.

6.1 Future Work - Applying the Concept to Superscalar Processors

Superscalar processors have multiple copies of their datapath hardware so that instructions can be executed simultaneously. To enable this feature, special hardware elements must be available in the processor's microarchitecture, and multiple FUs are expected to be found in these processors. These microarchitectures usually have register files, instruction queues and instruction schedulers to allocate instructions for being executed on each of the available FUs [68, 93].

Due to their internal microarchitecture, these processors are good candidates for my concept of fine-grained control of FUs. I estimate that after a proper extension in the instruction decoder with instructions and signals to control the operations, very few further modifications would be necessary for the remainder of the design. So, my approach would benefit from the already in place FUs and the logic to control these.

Thinking in a simplified way, the instructions scheduler would need to be extended with additional control bits that would allow it to allocate instructions for multiple executions. These executions could be done in parallel or in series using either the different available FUs, or a simple series of re-executions under the same unit. Additionally, other control bits would need to be aggregated to the operations and their results, so that, additional combinational logic would be necessary to perform the comparison of the results or, when triple redundancy is enabled, the majority voting.

To conclude, the overhead to implement my approach is potentially smaller for superscalar processors than for simple processors as I have presented for the implemented test platform in Chapter 4. Therefore, I see a high potential for improvements in fault tolerance for these designs. However, since these processors present different internal structures, further evaluations would still be necessary to confirm my expectations about the hardware overhead and its possible improvement in fault tolerance.

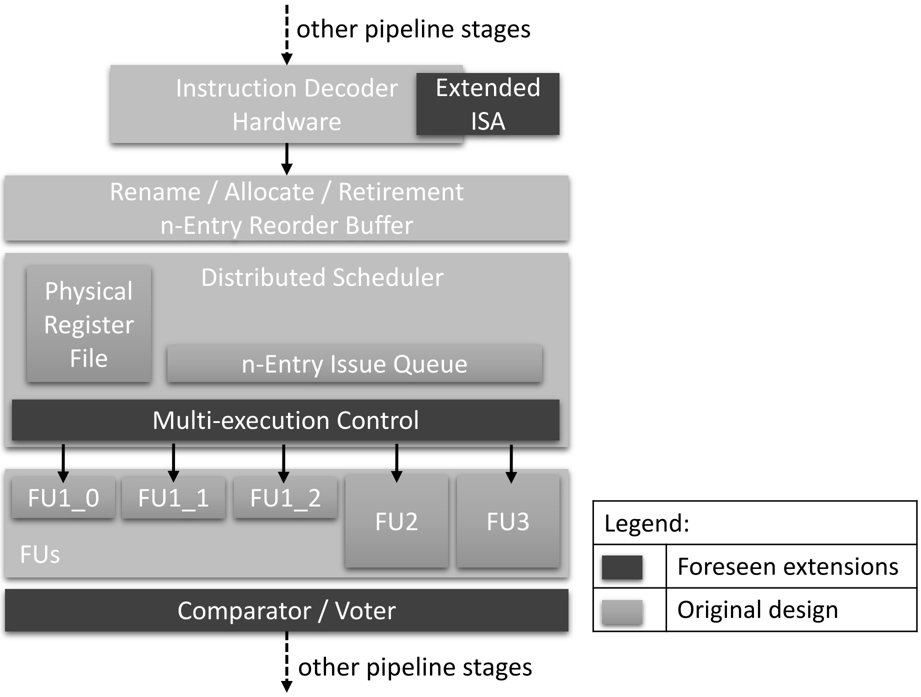


Figure 6.1: Simplified superscalar processor design with its extension points highlighted in dark Gray.

6.2 Own Publications Used in this Thesis

Raphael Segabinazzi Ferreira and Jörg Nolte. ‘Low latency reconfiguration mechanism for fine-grained processor internal functional units’. In: *LATS 2019 - 20th IEEE Latin American Test Symposium*. 2019, pp. 1–6. ISBN: 9781728117560. DOI: `10.1109/LATW.2019.8704560`

Raphael Segabinazzi Ferreira et al. ‘Configurable Fault Tolerant Circuits and System Level Integration for Self-Awareness’. In: *Proceedings of the Work in Progress Session held in connection with SEAA 2019, the 45th EUROMICRO Conference on Software Engineering and Advanced Applications and DSD 2019, 22nd EUROMICRO Conference on Digital System Design*. SEA-Publications, 2019. ISBN: 978-3-902457-54-7. DOI: `10.26127/BTUOpen-5050`

Raphael Segabinazzi Ferreira et al. ‘Run-time Hardware Reconfiguration of Functional Units to Support Mixed-Critical Applications’. In: *21st IEEE Latin-American Test Symposium, LATS 2020*. 2020. ISBN: 9781728187310. DOI: `10.1109/LATS49555.2020.9093692`

Randolf Rotta, Raphael Segabinazzi Ferreira and Jörg Nolte. 'Real-Time dynamic hardware reconfiguration for processors with redundant functional units'. In: *Proceedings - 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing, ISORC 2020*. 2020, pp. 154–155. ISBN: 9781728169583. DOI: `10.1109/ISORC49007.2020.00035`

6.3 Dissemination and Presentations

In the context of this thesis, besides presenting the published papers above, the author presented his work in the following events:

R. Segabinazzi Ferreira. 'A fine-grained HW/SW co-supervisor running under dependable OS – The Concept'. Poster presented in the Ph.D. forum of the 8th BELAS - Biannual European-Latin American Summer School on Design, Test and Reliability, held in Tallin, Estonia, June, 2018.

R. Segabinazzi Ferreira. 'Hardware and Software Cooperation for Improved Dependability - A Reconfiguration Mechanism'. Poster presented in the Ph.D. forum of the Biannual European-Latin American Summer School 2019 on: Design, Test and Reliability, held in Frankfurt (Oder), Germany, June, 2019.

R. Segabinazzi Ferreira. 'ESR2.2 (BTU) Innovative real-time operating system for error management for single- and multi-core units'. Presented in the RESCUE Virtual PhD Forum, 2020.

R. Segabinazzi Ferreira. 'Run-time Management of Hardware Redundancy for Mixed-Critical Applications'. Presented in the TuZ 2021 - Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen (virtual), Erfurt, Germany, 2021.

R. Segabinazzi Ferreira. 'Run-time Dynamic Configuration of Functional Units Redundancy for Mixed-Critical Scenarios'. Presented in the RESCUE Highlights session of the RESCUE 2021 - Virtual Workshop on Interdependent Challenges of Reliability, Security and Quality, organized as a DATE 2021 Friday Workshop, February, 2021.

R. Segabinazzi Ferreira. 'Run-time Management of Hardware Redundancy for Mixed-Critical Applications'. Poster presented in the 2021 MCAA - Marie Curie Alumni Association - Annual Conference, March, 2021.

R. Segabinazzi Ferreira. ‘A successful collaboration between Brazil and Germany within my fellowship in the RESCUE-ETN project’. Presented in the II Latin America MCAA Conference, March, 2021.

APPENDIX A

Acronyms

ALU	Arithmetical Logical Unit
ANSI	American National Standards Institute
ASIL	Automotive Safety and Integrity Level
AVF	Architecture Vulnerability Factor
BTI	Bias Temperature Instability
CGRA	Coarse-Grained Reconfigurable Array
CMOS	Complementary MOSFET
COTS	Commercial off-the-shelf
CPU	Central Processing Unit
CRC	cyclic redundancy check
DCLS	Dual Core Lock-Step
DMR	Double Modular Redundancy
DUT	Device Under Test
ECC	Error Correction Code
EDAC	Error-Detection and Correction
EX	Instruction Execute
FIT	Failure In Time
FPGA	Field Programmable Gate Array
FPU	Floating Point Units
FU	Functional Unit
GCC	GNU Compiler Collection
HCI	Hot Carrier Injection
ID	Instruction Decode
IF	Instruction Fetch
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
IRDS	International Roadmap for Devices and Systems
JEDEC	Joint Electron Device Engineering Council
LUT	Lookup Table
MEM	Memory Access
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MPU	Memory Protection Unit
NBTI	Negative Bias Temperature Instability
NMOS	n-type MOSFET
NMR	N-Modular Redundancy
OCL	Open Cell Library
OS	Operating System
OpCode	Operation Code
PBTI	Positive Bias Temperature Instability

PC	Program Counter
PE	Processing Element
PMOS	p-type MOSFET
Plasma-RTOS	Plasma Real-Time Operating System
RISC	Reduced Instruction Set Computer
RTOS	Real-Time Operating System
SAIF	Switching Activity Interchange Format
SEE	Single-Event Effect
SET	Single-Event Transient
SEU	Single-Event Upset
SlMode	Single Unit Mode
SoC	System on Chip
TCLS	Triple Core Lock-Step
TCL	Tool Command Language
TMR	Triple Modular Redundancy
VCD	Value Change Dump
VLIW	Very Long Instruction Word
WB	Write Back

Bibliography

[1] E. Amat et al. ‘Channel hot-carrier degradation in pMOS and nMOS short channel transistors with high-k dielectric stack’. In: *Microelectronic Engineering* 87.1 (2010), pp. 47–50. ISSN: 0167-9317. DOI: `10.1016/j.mee.2009.05.013`.

[2] P. C. Anderson, F. J. Rich and Stanislav Borisov. ‘Mapping the South Atlantic Anomaly continuously over 27 years’. In: *Journal of Atmospheric and Solar-Terrestrial Physics* 177 (2018), pp. 237–246. ISSN: 1364-6826. DOI: `https://doi.org/10.1016/j.jastp.2018.03.015`.

[3] ARM Ltd. *Cortex-R5 and Cortex-R5F. Technical Reference Manual.* 2011.

[4] AUTOSAR - Automotive Open System Architecture GbR. *AUTOSAR Specification of operating system (version4.3.1).* Tech. rep. 2017.

[5] A. Avizienis et al. ‘Basic concepts and taxonomy of dependable and secure computing’. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: `10.1109/TDSC.2004.2`.

[6] Aneesh Balakrishnan et al. ‘Modeling Soft-Error Reliability Under Variability’. In: *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT).* 2021, pp. 1–6. DOI: `10.1109/DFT52944.2021.9568295`.

[7] Alessandro Baldassari, Cristiana Bolchini and Antonio Miele. ‘A dynamic reliability management framework for heterogeneous multicore systems’. In: *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT).* IEEE, Oct. 2017, pp. 1–6. ISBN: 978-1-5386-0362-8. DOI: `10.1109/DFT.2017.8244440`.

[8] Robert C. Baumann. ‘Radiation-induced soft errors in advanced semiconductor technologies’. In: *IEEE Transactions on Device and Materials Reliability* 5.3 (Sept. 2005), pp. 305–315. ISSN: 15304388. DOI: `10.1109/TDMR.2005.853449`.

[9] Sarvesh Bhardwaj et al. ‘Predictive Modeling of the NBTI Effect for Reliable Design’. In: *IEEE Custom Integrated Circuits Conference 2006.* 2006, pp. 189–192. DOI: `10.1109/CICC.2006.320885`.

[10] Mark T. Bohr and Ian A. Young. ‘CMOS Scaling Trends and beyond’. In: *IEEE Micro* 37.6 (Nov. 2017), pp. 20–29. ISSN: 02721732. DOI: `10.1109/MM.2017.4241347`.

[11] Marcelo Brandalero et al. ‘Proactive Aging Mitigation in CGRAs through Utilization-Aware Allocation’. In: *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*. DAC ’20. IEEE Press, 2020. ISBN: 9781450367257.

[12] A. Burns and Robert I. Davis. *Mixed Criticality Systems - A Review*. Twelfth ed. 2019, p. 81. URL: `https://www-users.cs.york.ac.uk/burns/review.pdf` (visited on 26/01/2022).

[13] Douglas Maciel Cardoso et al. ‘Improving Software-based Techniques for Soft Error Mitigation in OoO Superscalar Processors’. In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2019, pp. 201–204. DOI: `10.1109/ICECS46596.2019.8964749`.

[14] Athanasios Chatzidimitriou et al. ‘Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments’. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019, pp. 26–38. DOI: `10.1109/DSN.2019.00018`.

[15] Xinghao Chen and Nur A. Touba. ‘Fundamentals of CMOS design’. In: *Electronic Design Automation*. Ed. by Laung-Terng Wang, Yao-Wen Chang and Kwang-Ting (Tim) B T - Electronic Design Automation Cheng. Boston: Elsevier, 2009, pp. 39–95. ISBN: 978-0-12-374364-0. DOI: `10.1016/B978-0-12-374364-0.50009-6`.

[16] *DO-178C - Software Considerations in Airborne Systems and Equipment Certification*. Tech. rep. Radio Technical Commission for Aeronautics - RTCA, Dec. 2011.

[17] *DO-254 - Design Assurance Guidance for Airborne Electronic Hardware*. Tech. rep. Radio Technical Commission for Aeronautics - RTCA, Apr. 2000.

[18] Björn Döbel, Hermann Härtig and Michael Engel. ‘Operating System Support for Redundant Multithreading’. In: *Proceedings of the Tenth ACM International Conference on Embedded Software*. EMSOFT ’12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 83–92. ISBN: 9781450314251. DOI: `10.1145/2380356.2380375`.

[19] Mojtaba Ebrahimi et al. ‘CLASS: Combined logic and architectural soft error sensitivity analysis’. In: *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2013, pp. 601–607. DOI: `10.1109/ASPDAC.2013.6509664`.

[20] Mojtaba Ebrahimi et al. ‘Event-driven transient error propagation: A scalable and accurate soft error rate estimation approach’. In: *The 20th Asia and South Pacific Design Automation Conference*. 2015, pp. 743–748. DOI: `10.1109/ASPDAC.2015.7059099`.

[21] *Erika Enterprise RTOS v3*. 2022. URL: `http://www.erika-enterprise.com/` (visited on 28/01/2022).

[22] Raphael Segabinazzi Ferreira and Jörg Nolte. 'Low latency reconfiguration mechanism for fine-grained processor internal functional units'. In: *LATS 2019 - 20th IEEE Latin American Test Symposium*. 2019, pp. 1–6. ISBN: 9781728117560. DOI: `10.1109/LATW.2019.8704560`.

[23] Raphael Segabinazzi Ferreira et al. 'Configurable Fault Tolerant Circuits and System Level Integration for Self-Awareness'. In: *Proceedings of the Work in Progress Session held in connection with SEAA 2019, the 45th EUROMICRO Conference on Software Engineering and Advanced Applications and DSD 2019, 22nd EUROMICRO Conference on Digital System Design*. SEA-Publications, 2019. ISBN: 978-3-902457-54-7. DOI: `10.26127/BTUOpen-5050`.

[24] Raphael Segabinazzi Ferreira et al. 'Run-time Hardware Reconfiguration of Functional Units to Support Mixed-Critical Applications'. In: *21st IEEE Latin-American Test Symposium, LATS 2020*. 2020. ISBN: 9781728187310. DOI: `10.1109/LATS49555.2020.9093692`.

[25] Luca Fiore. 'Design of a fault tolerant RISC-V instruction execute stage for safety critical applications'. MA thesis. Politecnico di Torino, 2021. URL: `http://webthesis.biblio.polito.it/id/eprint/17869` (visited on 26/01/2022).

[26] Joseph A. Fisher, Paolo Faraboschi and Cliff Young. 'VLIW Processors'. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 2135–2142. ISBN: 978-0-387-09766-4. DOI: `10.1007/978-0-387-09766-4_471`.

[27] Matthew Fojtik et al. 'Bubble Razor: Eliminating Timing Margins in an ARM Cortex-M3 Processor in 45 nm CMOS Using Architecturally Independent Error Detection and Correction'. In: *IEEE Journal of Solid-State Circuits* 48.1 (Jan. 2013), pp. 66–81. ISSN: 0018-9200. DOI: `10.1109/JSSC.2012.2220912`.

[28] J. Gaisler. 'A portable and fault-tolerant microprocessor based on the SPARC v8 architecture'. In: *Proceedings International Conference on Dependable Systems and Networks*. 2002, pp. 409–415. DOI: `10.1109/DSN.2002.1028926`.

[29] Zana Ghaderi and Eli Bozorgzadeh. 'Aging-aware high-level physical planning for reconfigurable systems'. In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2016, pp. 631–636. DOI: `10.1109/ASPDAC.2016.7428082`.

[30] Mohammad Saber Golanbari et al. 'Selective Flip-Flop Optimization for Circuit Reliability'. In: *Dependable Embedded Systems*. Ed. by Henkel Jörg and Nikil Dutt. Cham: Springer International Publishing, 2021, pp. 337–364. ISBN: 978-3-030-52017-5. DOI: `10.1007/978-3-030-52017-5_14`.

[31] Tibor Grasser et al. 'The Paradigm Shift in Understanding the Bias Temperature Instability: From Reaction–Diffusion to Switching Oxide Traps'. In: *IEEE Transactions on Electron Devices* 58.11 (2011), pp. 3652–3666. DOI: `10.1109/TED.2011.2164543`.

[32] Tino Heijmen. 'Soft Errors from Space to Ground: Historical Overview, Empirical Evidence, and Future Trends'. In: *Soft Errors in Modern Electronic Systems*. Ed. by Michael Nicolaidis. Boston, MA: Springer US, 2011, pp. 1–25. ISBN: 978-1-4419-6993-4. DOI: `10.1007/978-1-4419-6993-4_1`.

[33] Jörg Henkel et al. 'Reliable On-chip systems in the nano-era: Lessons learnt and future trends'. In: *Proceedings - Design Automation Conference*. 2013. ISBN: 9781450320719. DOI: `10.1145/2463209.2488857`.

[34] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[35] Jorrit N. Herder et al. 'MINIX 3'. In: *ACM SIGOPS Operating Systems Review* 40.3 (July 2006), p. 80. ISSN: 01635980. DOI: `10.1145/1151374.1151391`.

[36] Martin Hoffmann et al. 'dOSEK: the design and implementation of a dependability-oriented static embedded kernel'. In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, Apr. 2015, pp. 259–270. ISBN: 978-1-4799-8603-3. DOI: `10.1109/RTAS.2015.7108449`.

[37] Hyejeong Hong et al. 'Lifetime reliability enhancement of microprocessors: Mitigating the impact of negative bias temperature instability'. In: *ACM Computing Surveys* 48.1 (Sept. 2015). ISSN: 15577341. DOI: `10.1145/2785988`.

[38] *IEC 61508:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems*. Tech. rep. International Electrotechnical Commission - IEC, Apr. 2010.

[39] Hyunsik Im et al. 'Physical insight into fractional power dependence of saturation current on gate voltage in advanced short channel MOSFETs (alpha-power law model)'. In: *Proceedings of the International Symposium on Low Power Electronics and Design*. 2002, pp. 13–18. DOI: `10.1109/LPE.2002.146701`.

[40] Amazon Web Services Inc. *Why RTOS and What is RTOS?* 2019. URL: `https://www.freertos.org/about-RTOS.html` (visited on 24/01/2022).

[41] Infineon Technologies AG. *Infineon Technologies AG - 32-bit TriCoreTM Microcontroller.* 2022. URL: `https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/` (visited on 28/01/2022).

[42] *IRDS 2021 Update - Executive Summary.* Tech. rep. International Roadmap For Devices and Systems, 2021.

[43] *IRDS 2021 Update - More Moore.* Tech. rep. International Roadmap For Devices and Systems, 2021.

[44] *ISO 26262 Road Vehicles—Functional Safety.* Tech. rep. International Organization for Standardization - ISO, Dec. 2018.

[45] Xabier Iturbe et al. 'A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications'. In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W).* IEEE, June 2016, pp. 246–249. ISBN: 978-1-5090-3688-2. DOI: `10.1109/DSN-W.2016.57`.

[46] Xabier Iturbe et al. 'The Arm Triple Core Lock-Step (TCLS) Processor'. In: *ACM Trans. Comput. Syst.* 36.3 (June 2019). ISSN: 0734-2071. DOI: `10.1145/3323917`.

[47] Xabier Iturbe et al. 'Work-in-progress: A "high resilience" mode to minimize soft error vulnerabilities in ARM Cortex-R CPU pipelines'. In: *Proceedings of the 2017 International Conference on Compilers, Architectures and Synthesis for Embedded Systems Companion, CASES 2017.* Association for Computing Machinery, Inc, Oct. 2017. ISBN: 9781450351843. DOI: `10.1145/3125501.3125509`.

[48] JEDEC. *Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semi-conductor Devices.* Tech. rep. JESD89A, JEDEC Standard, Oct. 2006. URL: `https://www.jedec.org/standards-documents/docs/jesd-89a` (visited on 26/01/2022).

[49] Maksim Jenihhin et al. 'Identification and Rejuvenation of NBTI-Critical Logic Paths in Nanoscale Circuits'. In: *Journal of Electronic Testing* 32.3 (2016), pp. 273–289. ISSN: 1573-0727. DOI: `10.1007/s10836-016-5589-x`.

[50] Kunhyuk Kang et al. 'Estimation of statistical variation in temporal NBTI degradation and its impact on lifetime circuit performance'. In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. 2007, pp. 730–734. DOI: `10.1109/ICCAD.2007.4397352`.

[51] Andreas Kerber and Eduard Cartier. 'Bias Temperature Instability Characterization Methods'. In: *Bias Temperature Instability for Devices and Circuits*. Ed. by Tibor Grasser. New York, NY: Springer New York, 2014, pp. 3–31. ISBN: 978-1-4614-7909-3. DOI: `10.1007/978-1-4614-7909-3_1`.

[52] Tobias Koal, Stefan Scharoba and Heinrich T. Vierhaus. 'Combining Correction of Delay Faults and Transient Faults'. In: *Proceedings - 2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2015*. Institute of Electrical and Electronics Engineers Inc., Aug. 2015, pp. 99–102. ISBN: 9781479967803. DOI: `10.1109/DDECS.2015.23`.

[53] Israel Koren and C. Mani Krishna. 'CHAPTER 2 - Hardware Fault Tolerance'. In: *Fault-Tolerant Systems*. Ed. by Israel Koren and C. Mani Krishna. Burlington: Morgan Kaufmann, 2007, pp. 11–54. ISBN: 978-0-12-088525-1. DOI: `https://doi.org/10.1016/B978-012088525-1/50005-5`.

[54] Florian Kriebel et al. 'Fault-Tolerant Computing with Heterogeneous Hardening Modes'. In: *Dependable Embedded Systems*. Ed. by Jörg Henkel and Nikil Dutt. Cham: Springer International Publishing, 2021, pp. 161–180. ISBN: 978-3-030-52017-5. DOI: `10.1007/978-3-030-52017-5_7`.

[55] Amit Kulkarni et al. 'Pixie: A heterogeneous Virtual Coarse-Grained Reconfigurable Array for high performance image processing applications'. In: *CoRR* abs/1705.0 (May 2017). URL: `http://arxiv.org/abs/1705.01738` (visited on 26/01/2022).

[56] R. Leveugle et al. 'Statistical fault injection: Quantified error and confidence'. In: *Proceedings -Design, Automation and Test in Europe, DATE*. DATE '09. Leuven, BEL: European Design and Automation Association, 2009, pp. 502–506. ISBN: 9783981080155. DOI: `10.1109/date.2009.5090716`.

[57] Ping-Chung Li, G. I. Stamoulis and I. N. Hajj. 'A probabilistic timing approach to hot-carrier effect estimation'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.10 (1994), pp. 1223–1234. DOI: `10.1109/43.317465`.

[58] R. E. Lyons and W. Vanderkulk. 'The Use of Triple-Modular Redundancy to Improve Computer Reliability'. In: *IBM Journal of Research and Development* 6.2 (Apr. 1962), pp. 200–209. ISSN: 0018-8646. DOI: `10.1147/rd.62.0200`.

[59] A. Malik, B. Moyer and D. Cermak. 'A low power unified cache architecture providing power and performance flexibility'. In: *ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No.00TH8514)*. 2000, pp. 241–243. DOI: `10.1145/344166.344610`.

[60] Elie Maricau and Georges Gielen. *Analog IC reliability in nanometer CMOS*. Springer New York, Jan. 2013, pp. 1–198. ISBN: 9781461461630. DOI: `10.1007/978-1-4614-6163-0`.

[61] Mayler Martins et al. 'Open cell library in 15nm FreePDK technology'. In: *Proceedings of the International Symposium on Physical Design*. Vol. 29-March-2. ISPD '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 171–178. ISBN: 9781450333993. DOI: `10.1145/2717764.2717783`.

[62] W. McMahon, A. Haggag and K. Hess. 'Reliability scaling issues for nanoscale devices'. In: *IEEE Transactions on Nanotechnology* 2.1 (2003), pp. 33–38. DOI: `10.1109/TNANO.2003.808515`.

[63] Shahrzad Mirkhani et al. 'Rethinking error injection for effective resilience'. In: *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2014, pp. 390–393. DOI: `10.1109/ASPDAC.2014.6742922`.

[64] S. Mitra et al. 'Robust system design with built-in soft-error resilience'. In: *Computer* 38.2 (Feb. 2005), pp. 43–52. ISSN: 0018-9162. DOI: `10.1109/MC.2005.70`.

[65] Subhasish Mitra, Pia Sanda and Norbert Seifert. 'Soft Errors: Technology Trends, System Effects, and Protection Techniques'. In: *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*. Institute of Electrical and Electronics Engineers (IEEE), Aug. 2007, pp. 4–4. DOI: `10.1109/iolts.2007.61`.

[66] Subhasish Mitra et al. 'Soft Error Resilient System Design through Error Correction'. In: *2006 IFIP International Conference on Very Large Scale Integration*. IEEE, Oct. 2006, pp. 332–337. ISBN: 3-901882-19-7. DOI: `10.1109/VLSISOC.2006.313256`.

[67] Felix Muhlbauer, Lukas Schroder and Mario Scholzel. 'A fault tolerant dynamically scheduled processor with partial permanent fault handling'. In: *2018 IEEE 19th Latin-American Test Symposium, LATS 2018*. Vol. 2018-January. Apr. 2018, pp. 1–6. ISBN: 9781538614723. DOI: `10.1109/LATW.2018.8349676`.

[68] Felix Muhlbauer, Lukas Schroder and Mario Scholzel. 'On hardware-based fault-handling in dynamically scheduled processors'. In: *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*.

IEEE, Apr. 2017, pp. 201–206. ISBN: 978-1-5386-0472-4. DOI: `10.1109/DDECS.2017.7934572`.

[69] Shubhendu S. Mukherjee et al. ‘A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor’. In: *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*. Vol. 2003-Janua. MICRO 36. USA: IEEE Computer Society, 2003, pp. 29–40. ISBN: 076952043X. DOI: `10.1109/MICRO.2003.1253181`.

[70] T. Nigam, B. Parameshwaran and G. Krause. ‘Accurate product lifetime predictions based on device-level measurements’. In: *2009 IEEE International Reliability Physics Symposium*. 2009, pp. 634–639. DOI: `10.1109/IRPS.2009.5173322`.

[71] Fabian Oboril. ‘Cross-Layer Approaches for an Aging-Aware Design of Nanoscale Microprocessors’. PhD thesis. 2015. DOI: `10.5445/IR/1000045647`.

[72] Fabian Oboril and Mehdi B. Tahoori. ‘ExtraTime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level’. In: *Proceedings of the International Conference on Dependable Systems and Networks*. 2012. ISBN: 9781467316248. DOI: `10.1109/DSN.2012.6263957`.

[73] Fabian Oboril et al. ‘Negative Bias Temperature Instability-Aware Instruction Scheduling: A Cross-Layer Approach’. In: *Journal of Low Power Electronics* 9.4 (Dec. 2013), pp. 389–402. ISSN: 15461998. DOI: `10.1166/jolpe.2013.1284`.

[74] Ádria Barros de Oliveira, Gennaro Severino Rodrigues and Fernanda Lima Kastensmidt. ‘Analyzing Lockstep Dual-Core ARM Cortex-A9 Soft Error Mitigation in FreeRTOS Applications’. In: *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design: Chip on the Sands*. SBCCI ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 84–89. ISBN: 9781450351065. DOI: `10.1145/3109984.3110008`.

[75] OpenCores.org. *Plasma - most MIPS I(TM) opcodes - Overview*. 2019. URL: `https://opencores.org/projects/plasma` (visited on 28/01/2022).

[76] Eric Peters. *Can a Self-Driving Car Glitch Threaten Passenger Safety?* 2016. URL: `https://www.govtech.com/fs/can-a-self-driving-car-glitch-threaten-passenger-safety.html` (visited on 28/01/2022).

[77] Oskar Pusz, Christian Dietrich and Daniel Lohmann. ‘Data-Flow-Sensitive Fault-Space Pruning for the Injection of Transient Hardware Faults’. In: *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers,*

and Tools for Embedded Systems. LCTES 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 97–109. ISBN: 9781450384728. DOI: `10.1145/3461648.3463851`.

[78] Jan M. Rabaey, Anantha P. Chandrakasan and Borivoje Nikolić. *Digital integrated circuits: a design perspective*. Vol. 7. Pearson education Upper Saddle River, NJ, 2003. ISBN: 9780130909961.

[79] Eberle A. Rambo and Rolf Ernst. 'ASTEROID and the Replica-Aware Co-scheduling for Mixed-Criticality'. In: *Dependable Embedded Systems*. Ed. by Jörg Henkel and Nikil Dutt. Cham: Springer International Publishing, 2021, pp. 57–84. ISBN: 978-3-030-52017-5. DOI: `10.1007/978-3-030-52017-5_3`.

[80] Steve Rhoads. *Plasma Real-Time Operating System*. 2019. URL: `http://plasmacpu.no-ip.org:8080/rtos.htm` (visited on 28/01/2022).

[81] Randolf Rotta, Raphael Segabinazzi Ferreira and Jörg Nolte. 'Real-Time dynamic hardware reconfiguration for processors with redundant functional units'. In: *Proceedings - 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing, ISORC 2020*. 2020, pp. 154–155. ISBN: 9781728169583. DOI: `10.1109/ISORC49007.2020.00035`.

[82] H. Saito et al. 'Index: piggy-back satellite for aurora observation and technology demonstration'. In: *Acta Astronautica* 48.5 (2001), pp. 723–735. ISSN: 0094-5765. DOI: `10.1016/S0094-5765(01)00079-0`.

[83] Sayeef Salahuddin, Kai Ni and Suman Datta. 'The era of hyper-scaling in electronics'. In: *Nature Electronics* 1.8 (Aug. 2018), pp. 442–450. ISSN: 25201131. DOI: `10.1038/s41928-018-0117-x`.

[84] Thiago Santini et al. 'Beyond cross-section: Spatio-temporal reliability analysis'. In: *ACM Transactions on Embedded Computing Systems* 15.1 (Dec. 2016). ISSN: 15583465. DOI: `10.1145/2794148`.

[85] Pasquale Davide Schiavone et al. 'Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for internet-of-things applications'. In: *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation, PATMOS 2017*. Vol. 2017-January. Institute of Electrical and Electronics Engineers Inc., Nov. 2017, pp. 1–8. ISBN: 9781509064625. DOI: `10.1109/PATMOS.2017.8106976`.

[86] Mario Schölzel. 'HW/SW co-detection of transient and permanent faults with fast recovery in statically scheduled data paths'. In: *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*. 2010, pp. 723–728. DOI: `10.1109/DATE.2010.5456957`.

[87] Konstantin Shibin et al. 'Health Management for Self-Aware SoCs Based on IEEE 1687 Infrastructure'. In: *IEEE Design & Test* 34.6 (Dec. 2017), pp. 27–35. ISSN: 2168-2356. DOI: `10.1109/MDAT.2017.2750902`.

[88] Taniya Siddiqua and Sudhanva Gurumurthi. 'A multi-level approach to reduce the impact of NBTI on processor functional units'. In: *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*. 2010, pp. 67–72. ISBN: 9781450300124. DOI: `10.1145/1785481.1785498`.

[89] Felipe Augusto da Silva et al. 'Special Session: AutoSoC - A Suite of Open-Source Automotive SoC Benchmarks'. In: *2020 IEEE 38th VLSI Test Symposium (VTS)*. 2020, pp. 1–9. DOI: `10.1109/VTS48691.2020.9107599`.

[90] Michael A. Skitsas, Chrysostomos A. Nicopoulos and Maria K. Michael. 'DaemonGuard: Enabling O/S-Orchestrated Fine-Grained Software-Based Selective-Testing in Multi-/Many-Core Microprocessors'. In: *IEEE Transactions on Computers* 65.5 (May 2016), pp. 1453–1466. ISSN: 0018-9340. DOI: `10.1109/TC.2015.2449840`.

[91] J.H. Stathis and S. Zafar. 'The negative bias temperature instability in MOS devices: A review'. In: *Microelectronics Reliability* 46.2 (2006), pp. 270–286. ISSN: 0026-2714. DOI: `10.1016/j.microrel.2005.08.001`.

[92] Alvin W. Strong et al. *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. 2009. DOI: `10.1002/9780470455265`.

[93] David Suggs, Mahesh Subramony and Dan Bouvier. 'The AMD 'Zen 2' processor'. In: *IEEE Micro* 40.2 (2020), pp. 45–52. ISSN: 19374143. DOI: `10.1109/MM.2020.2974217`.

[94] Synopsys Inc. *What is ASIL?* May 2022. URL: `https://www.synopsys.com/automotive/what-is-asil.html` (visited on 28/01/2022).

[95] Texas Instruments. *Hercules TMS570 Microcontrollers.* Tech. rep. 2014.

[96] *The Autonomous Car's Big Challenge: Using the Hyperscale Server Fleet to Train AI Neural Networks.* Dec. 2018. URL: `https://www.designnews.com/electronics-test/autonomous-car-s-big-challenge-using-hyperscale-server-fleet-train-ai-neural-networks/196274523759943` (visited on 24/01/2022).

[97] Rafael Billig Tonetto et al. ‘A Knapsack Methodology for Hardware-based DMR Protection against Soft Errors in Superscalar Out-of-Order Processors’. In: *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. 2019, pp. 287–292. DOI: `10.1109/VLSI-SoC.2019.8920350`.

[98] Jyothi Bhaskarr Velamala et al. ‘Compact Modeling of Statistical BTI Under Trapping/Detrapping’. In: *IEEE Transactions on Electron Devices* 60.11 (2013), pp. 3645–3654. DOI: `10.1109/TED.2013.2281986`.

[99] Steve Vestal. ‘Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance’. In: *Proceedings - Real-Time Systems Symposium*. 2007, pp. 239–243. ISBN: 0769530621. DOI: `10.1109/RTSS.2007.47`.

[100] Massimo Violante et al. ‘A Low-Cost Solution for Deploying Processor Cores in Harsh Environments’. In: *IEEE Transactions on Industrial Electronics* 58.7 (2011), pp. 2617–2626. DOI: `10.1109/TIE.2011.2134054`.

[101] Wenping Wang et al. ‘Compact Modeling and Simulation of Circuit Reliability for 65-nm CMOS Technology’. In: *IEEE Transactions on Device and Materials Reliability* 7.4 (2007), pp. 509–517. DOI: `10.1109/TDMR.2007.910130`.

[102] WITTENSTEIN high integrity systems Ltd. *Pre-Certified Safety RTOS – SAFERTOS*. 2022. URL: `https://www.highintegritysystems.com/safertos/` (visited on 28/01/2022).

[103] Junko Yoshida. *EETimes - Toyota Case: Single Bit Flip That Killed*. 2013. URL: `https://www.eetimes.com/toyota-case-single-bit-flip-that-killed/` (visited on 28/01/2022).

[104] Hongyan Zhang et al. ‘Module diversification: Fault tolerance and aging mitigation for runtime reconfigurable architectures’. In: *2013 IEEE International Test Conference (ITC)*. 2013, pp. 1–10. DOI: `10.1109/TEST.2013.6651926`.