Reliability Assessment of the Open-source Many-core Processor OpenPiton

Chifa Dammak

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Electrical and Computer Engineering) at

Concordia University

Montréal, Québec, Canada

August 2022

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Chifa Dammak**

Entitled: **Reliability Assessment of the Open-source Many-core Processor OpenPiton**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair/Internal Examiner
Dr. Sébastien Le Beux

_____ External Examiner
Dr. Mohammad Mannan

_____ Supervisor
Dr. Otmane Ait Mohamed

_____ Co-supervisor
Dr. Mounir Boukadoum

Approved by _____
Yousef R. Shayan, Chair
Department of Electrical and Computer Engineering

_____ 2022

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Reliability Assessment of the Open-source Many-core Processor OpenPiton

Chifa Dammak

The fast-growing demand for computational capacity has led to the emergence of large-scale systems where the parallel processing capabilities of many-core processors have made them an ideal solution to bridge the gap between the onboard processing throughput and the applications' increasing complexity. A many-core processor on a chip can enhance the overall performance without the need for higher clock frequencies and the associated cooling problems at a lower design cost and a smaller system footprint. These features have made the many-core processors an interesting platform to implement complex algorithms. The scaling of space applications from single-core to many-core processors is constrained by the estimation of the vulnerability of many-core COTS' reliability in the presence of radiation. Recent academic and industrial research efforts have focused on evaluating the reliability of many-core processors against radiation events in order to facilitate their integration in an avionic domain. The radiation environment is characterized by a high-energy particle that ionizes the processor's components causing potential system failure.

This work presents a scalable fault injection and SEU impact categorization tool. The proposed engine performs simulation-based fault injections at the register transfer level (RTL) level of the design. The injection campaigns insert bit-flips in the general-purpose registers as well as the instruction memory of the different cores in the target processor. The approach enables a fully automated analysis of the SEU effects early in the design time. The

soft error propagation through the many-core components is also evaluated to determine the potential impact on the memory elements of the multiple cores. This framework has been applied to evaluate the resilience of the open-source many-core processor OpenPiton.

Acknowledgments

First and foremost, I would like to express my appreciation to my supervisor, Dr.Otmane Ait Mohamed who has given me his excellent guidance and support. His mentorship, constant motivation, steadfast encouragement, and expert guidance make this journey a rewarding experience. Special thanks to my co-supervisor Dr. Mounir Boukadoum for his advice, help, and invaluable suggestions throughout my master's journey. I am truly grateful to have them as my thesis advisors in my journey at Concordia University.

I would like to acknowledge and thank my family for their belief, support, and love. It would be inappropriate if I omit to mention the name of my dear friends from Hardware Verification Group (HVG) who were there when I was away from my home and beloveds, and who never let things get dull and boring. Thank you for their ceaseless support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we provide a brief overview of many-core processor architecture. We state the context and the motivation of our conducted research. Subsequently, we discuss the incentive of integrating many-core processors in high computational applications. In addition, we highlight the radiation effects on computing systems and the underlying challenges of the evaluation of their impact on hardware systems. Afterward, we illustrate the problem statement and enumerate the thesis contribution. Finally, we conclude this chapter by presenting the thesis outline.

1.1 Context and Motivation

The increasing complexity of industry applications requires intensive computational power while providing optimal performance and power consumption. To accommodate the requirements of such applications, several optimizations are applied to the classical architectures with a single core processor. However, these efforts magnify the design complexity which leads to a larger area and higher delay. An alternative approach is to scale the number of cores integrated into a processor [3]. The advances in semiconductors technologies have improved considerably from 3µm in 1987 to 3nm in 2022 [4]. This progress has allowed the

integration of multi-core and many-core on a single chip [5]. This novel implementation enhances the computational capacity due to its parallel execution capabilities. The design's flexibility allows the implementation of different programming paradigms and processing modes.

The many-core processors are emerging in High-Performance Computing (HPC) systems e.g. The Tianhe-2 supercomputer integrates several instances of the Matrix MT2000+ many-core processors, where each instance is composed of 128 processors running at 2 GHz. Another example is the supercomputer Sunway TaihuLigh which is based on a heterogeneous Sunway SW26010 many-core processor [6]. Despite the wide application domain of the many-core processors, their employment in critical applications is insufficient [7].

The failure of a safety-critical system may lead to severe damage to property and human resources. These applications exist in various domains, such as medicine, transportation, avionics, etc. The system failure can originate from hardware and/or software bugs that remained undetected throughout the verification process [8] or from environmental effects such as radiation [9]. Safety-critical applications should be implemented in hardware devices that have been proven to be resilient to radiation effects. The radiation impact on semiconductors can range from temporary data corruption called soft errors to permanent device damage called hard errors [10]. However, the occurrence of soft errors is more frequent than hard errors. Therefore, the industry is more interested in evaluating the immunity of hardware devices in presence of soft errors. The soft errors cause transient bit-flip in memory or logic components of the circuit that can potentially lead to an undesirable state (e.g. system hang, generation of wrong results) [11]. Many-core processor architecture is complex and embedded a massive number of memory cells along with being manufactured with reduced technology size. Thus, they are more vulnerable to soft errors impact.

Applications performing mission-critical tasks including civil avionics, spacecraft, and

autonomous vehicles involve intensive computational effort. Scaling the number of cores per chip represents an alternative to single-core processors since it can handle the complexity of such applications while optimizing the performance. The challenge of incorporating many-core processors into safety-critical systems is to increase their reliability against the effects of soft errors.

1.2 Many-core Processors

Many-core processor is an interesting technology to cope with the increased performance demands. In order to increase performance while keeping reasonable power consumption rates, hardware designers have implemented multiple cores on a single chip. To further enhance the performance developers have benefited from the multiplicity of the cores to engage massive parallel execution. Therefore, many-core processors are able to minimize the execution time of an application without increasing the system frequency. According to [12], the power consumption is a function of the system frequency *f*, the voltage supply *Vdd*, the number of operating cores *n*, and two intrinsic parameters α and β that characterize the processor manufacturing technology.

$$P(f, Vdd, n) = n(\alpha Vdd^2 f + \beta Vdd) \tag{1}$$

According to Equation 1, the power consumption increases linearly with the number of cores per chip. On the other hand, voltage and frequency can cause a quadratic or cubic increase in power consumption. The power quantification equation is confirmed by the experiment results performed in [13]. In this work, an approximation of π using Riemann Method is performed respectively on 1 core, 2 cores, 3 cores, and 4 cores of the ARM Cortex A9 processor with a given frequency. The results have shown that increasing the number of cores consumes less power than increasing the system frequency. Thus, the

many-core processor provides an enhanced platform for boosting computation load while maintaining an acceptable power consumption.

In terms of performance scalability, the number of cores executing a program has a direct impact on the performance and execution time of the application. Engaging parallel processing can enhance the overall performance. However, scaling the number of cores introduce several parameters that can affect the performance. For instance, communication between the cores is essential to ensure multi-threading execution. However, this inter-core communication increases the latency and the average Worst-Case Execution Time (WCET) [14]. Therefore, the scheduling and partitioning techniques have a significant impact on the performance.

1.2.1 Many-core Processor Architecture

A many-core processor is a single chip that integrates many processors. The many-core processor can be homogeneous, which indicates that the implemented cores are of the same processor type. In addition, a many-core processor is called heterogeneous if the implemented cores are of at least two different types. The difference between the type of cores can be in both the instruction set architecture and the functionality. Each core on the chip functions as an independent processor.

The communication between them is ensured by a Network-on-Chip (NoC). These networks are responsible for handling memory requests originating from the different components of the many-core processor. In addition, these cores communicate with the outside world using I/O interface.

The structure of the interconnection between the cores can be established in different ways [1]. An interconnection bus can be used to manage memory resources. The bus is used to respond to memory requests between the cores and the shared memory. This methodology suffers from latency and bandwidth problems due to the long electrical wires

required to construct the bus. Other interconnection methods include cross-bar interconnection, multiple ring buses, and mesh structures. The choice of the interconnection approach depends on the number of integrated cores in the processor and the power constraints. Figure 1.1 shows the architecture of the different inter-connection methodologies.

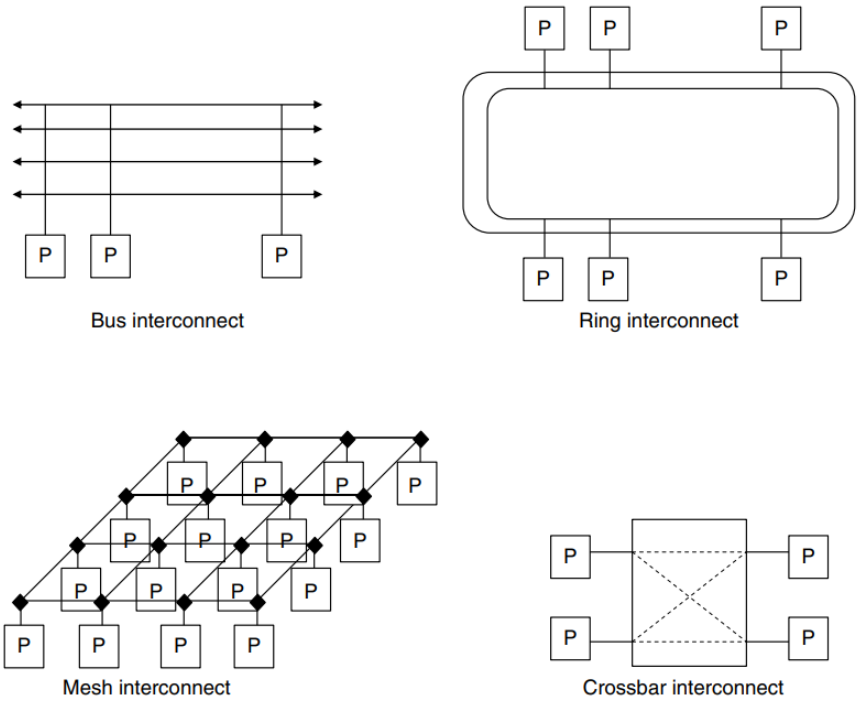

Figure 1.1: Inter-connection Methodologies [1]

In a many-core processor, the cores can be grouped into clusters. Each cluster is composed of a set of cores that interfaces with a local shared cache. The many-core processor architecture is given in figure 1.2. The memory hierarchy is composed of one or two levels of private caches integrated in each core and a shared cache memory. The role of private caches is to minimize access time and to reduce cache conflict. To ensure the memory consistency between the private caches, several cache coherence techniques can be used.

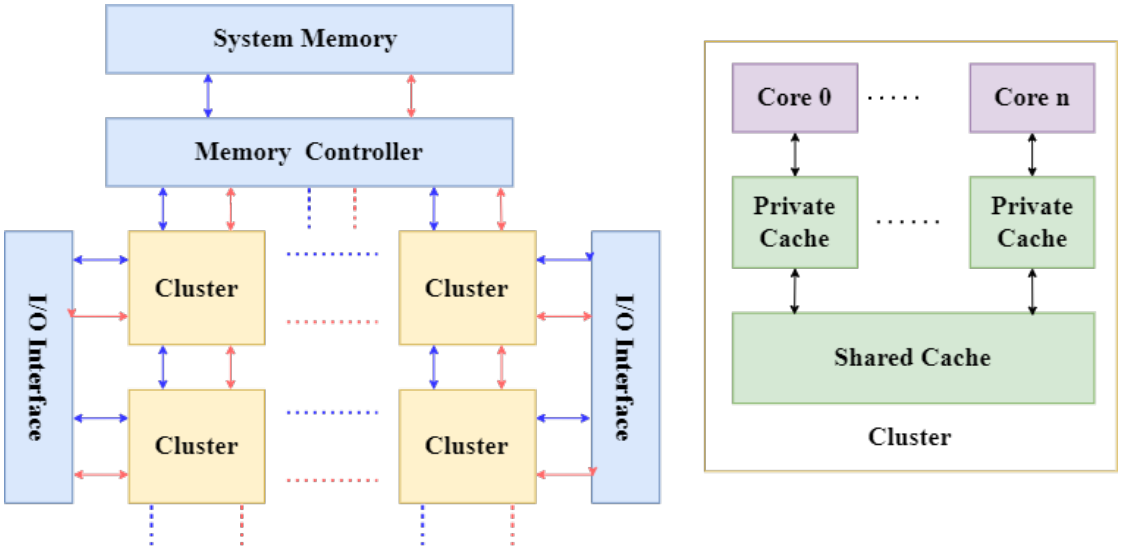

Figure 1.2: Many-core Processor Architecture

The shared cache facilitates the development of parallel applications since the cores can access the same address space. The memory controller handles the interfacing between the clusters and the system memory, such as DRAM. In order to increase throughput, this component typically employs advanced pre-fetching methods.

To further increase the performance, the core of a many-core processor can support multiple threads. This architecture allows the core to execute different hardware threads at the same time. The idea of implementing multi-threading in a many-core processor was to allow the core to switch to another thread in case the current thread is stalled due to the low throughput of the cache memories.

1.2.2 Many-core Processor Programming

The many-core processor offers great development flexibility. The developers can exploit different programming paradigms that allow them to reduce the execution time of their applications. A many-core processor can be programmed using two multi-processing modes: Asymmetric Multi-Processing (AMP) mode or Symmetric Multi-Processing (SMP)

mode [15]. In AMP mode, the many-core processor's cores are independent of one another. Despite the interconnection between the cores, each core has its private addressing space [16]. Furthermore, each core can be managed with or without an Operating System OS. However, for the SMP mode, the same operating system is managing all the cores and they share the same hardware resources and addressing space.

Software programmers are faced with several challenges to achieve the expected performance enhancement when programming a many-core processor. The data consistency issue causes the corruption of the memory content. This problem occur usually when the cache is a shared memory or while processing I/O data in parallel [17]. On the other hand, communication between the cores is a major concern for program developers. Scaling the number of cores can diminish the performance when the proper communication between the cores isn't ensured [18].

1.3 Radiation Effects

Hardware components operating in a harsh environment are susceptible to radiation effects. The radiation-induced errors can be caused by alpha particles and cosmic radiation in the space environment [19]. At the ground level, the hardware circuits are affected by the radiation caused by the high temperatures and the voltage fluctuations [20]. The radiation introduces a flux of energetic particles capable of ionizing the transistors of the hardware circuit device. This cumulative charge can cause a momentary spike of charge [21]. At this moment, a transistor can change its state. This phenomenon is represented by a bit-flip in the state architecture of the processor.

The vulnerability of hardware devices to radiation effects presents a significant concern for space and avionic applications (hardware devices are subject to natural space radiation environment) and for terrestrial safety-critical applications (hardware devices are subject

to natural terrestrial radiation environment or artificial radiations that can be found in military, industrial, and medical systems). Designers must be able to assess the radiation effects early in the design phase and implement fault tolerance techniques to enhance the reliability of the design. Hardware and/or software mitigation strategies can be employed. The hardware level mitigation techniques integrate additional components to cope with the radiation effects. An example of hardware mitigation techniques is the implementation of parity and Error Correction Code (ECC) to cope with soft errors in the internal memory of the device. In addition, the Triple Modular Redundancy (TMR) is a widely-used technique to improve the reliability of the design [22]. Despite the promising reliability enhancement, duplicating the hardware resources involves an increase in the overall area and power consumption. However, for many-core processors, the availability of the hardware components and moderate power consumption increase when scaling the number of cores motivates the implementation of the TMR technique. As for the software level mitigation techniques, the shielding is ensured by information redundancy. For example, the authors of [23] propose to create redundant tasks at the Operating System OS level to enhance the reliability in presence of transient faults.

1.4 Problem Statement

The high level of performance intrinsic to many-core architectures has made them the obvious successor to single-core processors in scenarios that require high computation [24]. Many-core COTS processors can achieve the performance and cost constraints required by future safety-critical avionic applications. However, the increase in compute units embedded within the many-core magnifies the vulnerability to soft errors, which may lead to system failure. Therefore, these low-cost alternatives present new challenges to validate their deployment in safety-critical applications. The integration of many-core COTS in avionics systems is limited due to the scarcity of research activities that aims to evaluate

the reliability of open-source many-core processors that could potentially be considered for aerospace applications. In the process of promoting the implementation of many-core COTS processors in spacecraft industries, several questions can help to orient the efforts of evaluating the reliability of a prospective man-core processor:

- What are the functionalities that should be performed by a simulation-based fault injection assessment technique at the RTL level of the design?
- How can we categorize the impact of Single-Event Upset (SEU) events on the reliability of a many-core processor?
- What are the effects of increasing the number of cores on the SEUs effects on a many-core processor?
- Does the executed program affect the behavior of the target many-core processor in presence of SEU events?
- Can Triple-Modular Redundancy (TMR) mitigation technique improve the overall accuracy of the system?

1.5 Thesis Contributions

The work presented in this thesis proposes a scalable fault injection engine for the open-source many-core processor OpenPiton. The tool performs automated simulation-based fault injection campaigns on the RTL design to provide insight about the impact of soft errors on each core of the processor and to evaluate their propagation across cores. The primary issues identified in the problem statement have led to the research work conducted in the context of this thesis, yielding the contributions listed below:

- Developing an automated simulation-based fault injection framework. It injects the faults at the RTL level of the design. Using this platform we can conduct automated fault injection campaigns that perform a regression of simulation on a specific fault injection location. The number of cores supported by this platform is scalable. The currently supported fault injection locations are the general purpose registers (we process the faults injected in the actual data of the register, and in the Error Correction Code bits separately to evaluate the impact of implementing the ECC mechanism), and the instruction register. Moreover, the tool allows the set-up of the fault injection duration. Additionally, the fault injection simulation results are extracted and compared with non-faulty results with the aim of assessing the impact of the injected faults. Subsequently, the impact of the SEU is automatically classified into predefined categories. Further, our methodology allows investigating the propagation of a fault injection effect from the targeted core to the state architecture of the rest of the cores in the processor.

- Evaluating the impact of scaling the number of cores of the SEUs effects on the case study many-core processor OpenPiton. We have established a comparison between a 2-core system and a 3-core system where we classify the results using six classes of fault injection based on the fault injection location and duration. The conclusion was drawn based on the soft error categories and the propagation metric that measures the cross-core impact of fault injection. Furthermore, a study is conducted to identify the application of the implemented application on the SEU results. For this experiment, three widely-known benchmarks were selected: Fibonacci Series, Matrix Multiplication, and Cyclic Redundancy Check.

- Extending the fault injection framework analysis to compute the improvement of the system accuracy using the Triple-Modular Redundancy (TMR) mitigation technique. We have estimated the effectiveness of implementing TMR in case of a soft error that

only causes a single bit flip in one core and when the soft error affects two cores by flipping one bit in each. To emulate this behavior, the fault injection framework was further extended to support two bit-flips per simulation with a random selection of the affected cores and fault injection parameters.

1.6 Thesis Outline

The rest of this thesis is organized as follows:

- In chapter 2 we provide a brief overview of the soft errors and their effects. Subsequently, we discuss the integration of many-core COTS processors. Finally, we compare the radiation assessment techniques.
- In Chapter 3 we concentrate on presenting the flow of the implemented fault injection methodology. Afterward, we detail the fault categorization approach.
- Throughout chapter 4 we introduce the OpenPiton processor platform and we motivate our choice of this many-core processor. Additionally, we provide a summary of the main architectural aspect of this processor.
- In Chapter 5 we present the experimental setup and the results of the different experiments. We evaluate the impact of the SEU on the system reliability.
- In Chapter 6 we summarize the work conducted in this thesis and provides an outline of the future research strategies.

Chapter 2

Preliminaries

In this chapter, we provide a brief overview of the preliminary concepts required for our work. We start by defining Single-Event Effects (SEEs) and discussing their various types. We explain their potential impact on the hardware circuits. Subsequently, we investigate the motivation for emerging many-core processor COTS in avionics applications, as well as the validation challenges. We conclude this chapter by outlining the state-of-art reliability assessment techniques and we highlight their advantages and limitations.

2.1 Single-Event Effects

The radiation impacts the functionality of semiconductor devices in various ways. The Single-Event Effects (SEEs) are the fundamental category of radiation effects. SEE errors results from a single high energetic particle. The SEEs events are divided into two categories: non-destructive SEEs known as soft errors and destructive SEEs known as hard errors [25]. The classification of SEEs errors is described in Figure 2.1.

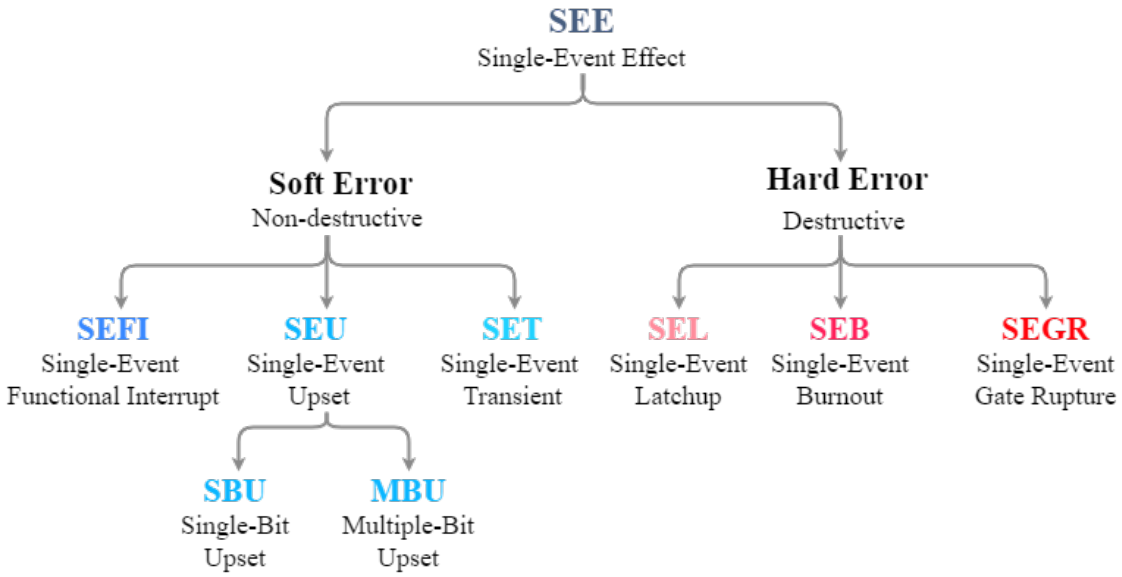

Figure 2.1: Single-Event Effects Types Classification

Soft errors are considered non-destructive because they do not cause damage to the circuit. Though, they temporarily change the state of the output or the internal data. Combinational circuits can self-recover from soft error impact once the disturbance caused by the energetic particle is removed. However, the radiation effects on sequential logic and memory components cause the corruption of a data state. The erroneous data persists in the state architecture until it's over-written.If the corrupted data is used by subsequent processes of the executed program, the system may fail or behave abnormally. There exist different types of soft errors such as:

- **Single-Event Transients (SETs)** are transient pulses at a single gate of a combinational circuit. These pulses can transit in the downstream processes to be eventually latched in a storage element.
- **Single-Event Upset (SEU)** is a particle strike caused by radiation that damages a storage element. This event can change the state of a single bit, in which case the

SEU is referred to as a Single-bit Upset (SBU), or multiple bits, in which case it is referred to as a Multiple-bit Upset (MBU).

- **Single-Event Functional Interrupts (SEFIs)** are SEUs that affect the control registers of an integrated circuit. This type of error causes the loss of functionality or cause incorrect execution. Product availability and failure rates are significantly more affected by SEFIs than SEUs. Since each SEFI results in a functionality perturbation, however, SEUs may have no impact on the device functionality and its output data.

Hard errors cause permanent damage to the circuit component. The observable impacts on the circuit functionality are the same as those of soft errors except that the circuit is physically destroyed. The hard errors can be categorized as follows:

- **Single-Event Latchup (SEL)** is linked to a significant increase in supply current. This event can be induced by a bipolar parasitic structure. The high current may overheat the integrated circuit causing permanent damage. However, when the maximum current of the circuit is limited, these errors can be considered as soft errors.

- **Single-Event Burnout (SEB)** is a destructive thermal runaway that effects the power transistors (e.g. Insulated Gate Bipolar Transistor (IGBT)).

- **Single-Event Gate Rupture (SEGR)** is associated with an increase in gate leakage current. It causes the degradation or the rupture of the resulting path.

The research community's activities are focused on addressing the impact of SEUs on the reliability of hardware devices (Figure 2.2). The memory elements are significantly vulnerable to errors induced by SEUs. Since the corrupted state can only be cleared when it's over-written, it can potentially result in a system failure. Therefore, the assessment of the reliability of the hardware devices should be conducted prior to the integration in

safety-critical applications. In this research work, we aim to evaluate the vulnerability of the OpenPiton many-core processor to SEUs.

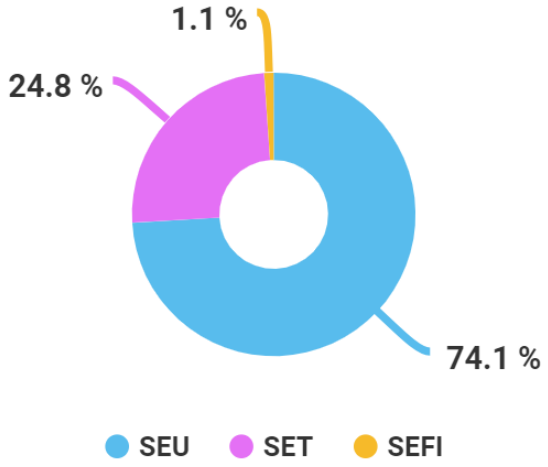

Figure 2.2: Ratio of research papers published in *IEEE Xplore* regarding SEUs, SETs, and SEFIs between 2000-2022

2.2 Many-core COTS in Avionics Domain

The classic architecture of avionic systems dedicates a single computer to perform an avionic function. For example, a computer is set to measure the flight coordinates function, at the same time the collision function is managed by another computer. Each computer has its private resources (sensors, actuator, etc.). In the literature, this concept is referred to as *federated architectures*. Despite the guaranteed robustness and the immunity of the system's overall functionality in presence of a computer failure, these architectures present various limitations [26]. For instance, the independence between the computers degrades

the response time. The point-to-point communication paradigm complicates the implementation of software changes on a single computer. Therefore, federated architecture has been phased-out due to its increased area and costly maintenance.

Nowadays, *Integrated Modular Avionic* (IMA) become a predominant architecture in avionic systems. Contrarily to federated architectures, IMAs allow resource sharing and the execution of multiple functions on the same computer. The main idea of IMA is to replace several distinct processors with a centralized processing unit. Recent research activities in the aviation and space industries have focused on integrating many-core processors in their systems due to their high performance and level of integration [27]-[28]. To further reduce the design cost, aerospace applications rely on Commercial-Off-The-shelf (COTS) components rather than custom electronics [29]-[30].

The design density and the huge number of memory cells implemented in many-core processors make them more vulnerable to soft errors [31]. In addition, systems operating in harsh environments are exposed to higher radiation rates. For instance, those destined for aerospace applications are affected by space radiation, which increases the fault occurrence rate in comparison to ground-level applications. The research work conducted by the authors of [32] measures the SEU rate on SRAMs of the Proba-II spacecraft for the duration of 3 years. The satellite has detected 0.48 SEU events/day/memory. The change in the state of a memory element introduced by a single event upset (SEU) can severely disturb the processing flow causing erroneous results or even system failure. Hence, the many-core COTS sensitivity to SEUs should be evaluated to certify their use in the space industry.

2.3 Reliability Assessment Techniques

There exist several techniques proposed by the research community to quantify the impact of SEUs on hardware devices. These techniques can be classified into radiation ground

testing, fault injection techniques, and formal methods. These approaches are covered in detail in the following subsections, along with each technique's benefits and downsides.

2.3.1 Radiation Ground Testing

To assess the upset rates induced by space radiation, ground radiation testing are often imperative [33]. The ground testing method exposes electronic components to a man-made radiation environment created by particle accelerators and laser beams that produce a higher magnitude flux than the natural space environment. The test results assist in identifying the impacts on the integrated circuit in a shorter amount of time than a real-world test. Given that they are carried out in dedicated facilities, these tests are quite expensive. Despite the accuracy advantage, this technique can only be applied to silicon chips. As a result, the vulnerability of the design is evaluated at the last stage of the manufacturing.

2.3.2 Fault Injection Methods

Fault injection techniques help to assess the resilience of the hardware circuit early in the design phase by artificially injecting bit-flips. The fault injection methodology can be conducted at different levels of abstraction [34]:

- **Fault Injection on a Manufactured Circuit** is the emulation of the faults by modifying the execution code of the device. To implement this type of fault injection, the internal resources of the manufactured circuits should be accessible. The fault injection is initiated by an exception that interrupts the execution code to inject the fault at a random position of the selected register or memory element. The application of this technique is limited to a set of hardware devices and can only be performed after the production phase.

- **Simulation-Based Fault Injection** is the injection of faults in the design phase of the

circuit using an HDL simulator. Simulation-based techniques have been proposed to assess system resilience to radiation events early in the design phase. The high-level approaches based on a behavioral design abstraction enhance simulation performance but suffers from reduced accuracy [35]. On the other hand, fault injection simulations targeting Register Transfer Level (RTL) have proven to be highly accurate [36].

- **Emulation-Based Fault Injection** the main interest of this technique is to boost the speed in contrast to the simulation-based fault injection by emulating the circuit design into an FPGA.

2.3.3 Formal Methods

The evaluation of hardware reliability to SEUs using formal methods implicitly searches for faults in the state space. The formalization of the architectural design using mathematical reasoning eliminate the need to exhaustively test all possible faults candidates. Therefore, the required time to evaluate the vulnerability of the design is more reasonable than simulation-based techniques.

The authors of [37] propose to combine fault injection and formal verification (Model Checking) to perform backward-tracing of the fault origin for a detected error. Using formal methods improves the required time to assess the reliability. However, establishing a mathematical model of the circuit design is time-consuming and requires expertise. In addition, the increased complexity of the architecture design results in state explosion. Further, the formalization of a large design can be complex.

2.4 Summary

In this chapter we present the background information required to comprehend our proposed approach. We addressed the effects of the radiation on the system functionality and

the potential failure risk associated with the different types of radiation events. The evaluation of the resilience of SEUs is an important metric to validate the implementation of the processor design in safety-critical applications. Subsequently, we discussed the design architecture used in the avionics domain and the new trend of centralizing the processing unit. We concluded that many-core processors present an interesting alternative to the federated architecture. However, their integration in avionics applications is restricted by the assurance of their resilience in radiation environments. Finally, we performed a literature review of SEUs assessment techniques. We have chosen to implement our assessment methodology using the simulation-based fault injection on the RTL level of the OpenPiton design. This choice is motivated by the accuracy of the results and the availability of the RTL code within the OpenPiton framework.

Chapter 3

Proposed Assessment Methodology

In this chapter, we present the fault injection methodology adopted to assess the reliability of the OpenPiton many-core processor. We explain the resilience evaluation strategy and identify the features that should be available in the proposed assessment environment. Subsequently, we provide the flow of the applied fault injection. Finally, we describe the process of categorizing the impact of SEUs and evaluating the propagation of the faults.

3.1 Single-Event Upset Evaluation Strategy

The observation of the system behavior in presence of hardware faults is necessary to conclude about the safety of the device. Reliability testing is performed while the processor is executing a task in presence of a fault. Among the methodologies to evaluate this safety metric early in the design phase is inducing such faults using fault injection. Based on the comparison of radiation assessment techniques performed in Chapter 2, we have chosen to elaborate our fault injection methodology using simulation-based fault injection targeting the Register Transfer Level (RTL) of the OpenPiton design. This decision is supported by the similarity of the results obtained using RTL fault injection and the radiation environment. A soft errors evaluation system based on fault simulation is composed of several

components:

- The ***Supervisor*** is responsible for launching and stopping the execution of the test.
- The ***Target System*** is the hardware system that implements the test application.
- The ***Application Handler*** provides the instruction stream that will be executed by the target system.
- The ***Fault Injector*** generates the time, location, and duration of the fault. It injects the fault into the target system based on the produced parameters.
- The ***Monitor*** monitors the execution flow of the test program on the target system.
- The ***Checkpoint Collector*** extracts and saves the checkpoints data at the end of the application execution.
- The ***Checkpoint Analyzer*** process the saved checkpoints data in order to evaluate the impact on the target system.

The approach of implementing a fault injection on the RTL design of the OpenPiton processor consists of the components shown in Figure 3.1. The target system is an instance of the OpenPiton processor generated based on the configuration parameter. The simulation is performed using the ModelSim simulator. The Tcl scripting language allows for automation of the fault injection campaign. The supervisor, monitor, fault injector, and checkpoint analyzer components are integrated within a Tcl script that exploits ModelSim commands to configure and insert the faults. Finally, the checkpoint analyzer is a python script that produces the classification of the impact of the soft errors.

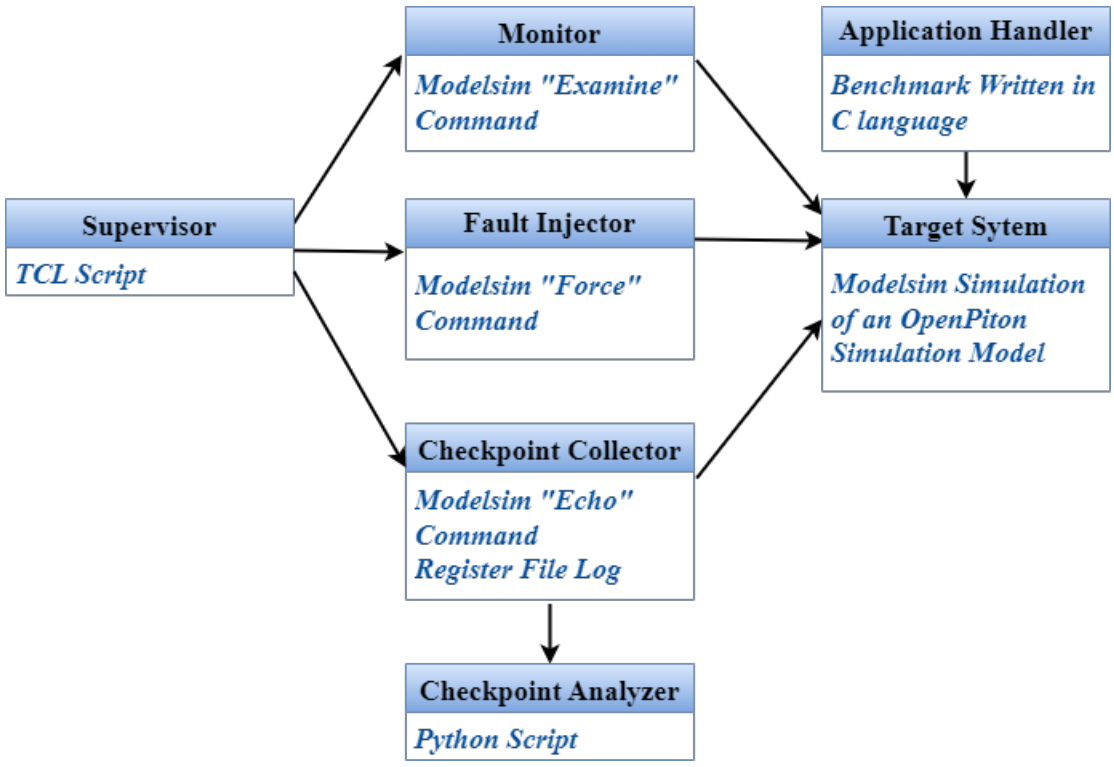

Figure 3.1: Simulation-based Fault Injection System Components

3.2 Proposed Reliability Analysis System

To validate the use of the many-core OpenPiton processor in safety-critical systems, an analysis of its resilience to SEUs must be conducted. This reliability study is based on the soft errors categorization and the propagation evaluation generated by the developed fault injection tool. A fault is modeled as a single-bit error that targets the general-purpose registers or the instruction register of either a single core or two cores at a time. This work is scalable in core number, and it can be extended to cover more fault injection targets. The entire process is automated. The proposed framework presents a custom fault injection and assessment environment for the OpenPiton processor design. It requires considerable time to identify the location of the fault target within the Verilog code of the design. Also, the

OpenPiton processor simulation infrastructure needs to be studied in order to integrate the fault injection process into its flow. The block diagram of the proposed engine is detailed in Figure 3.2. It is divided into a fault injection generator and a fault report analyzer.

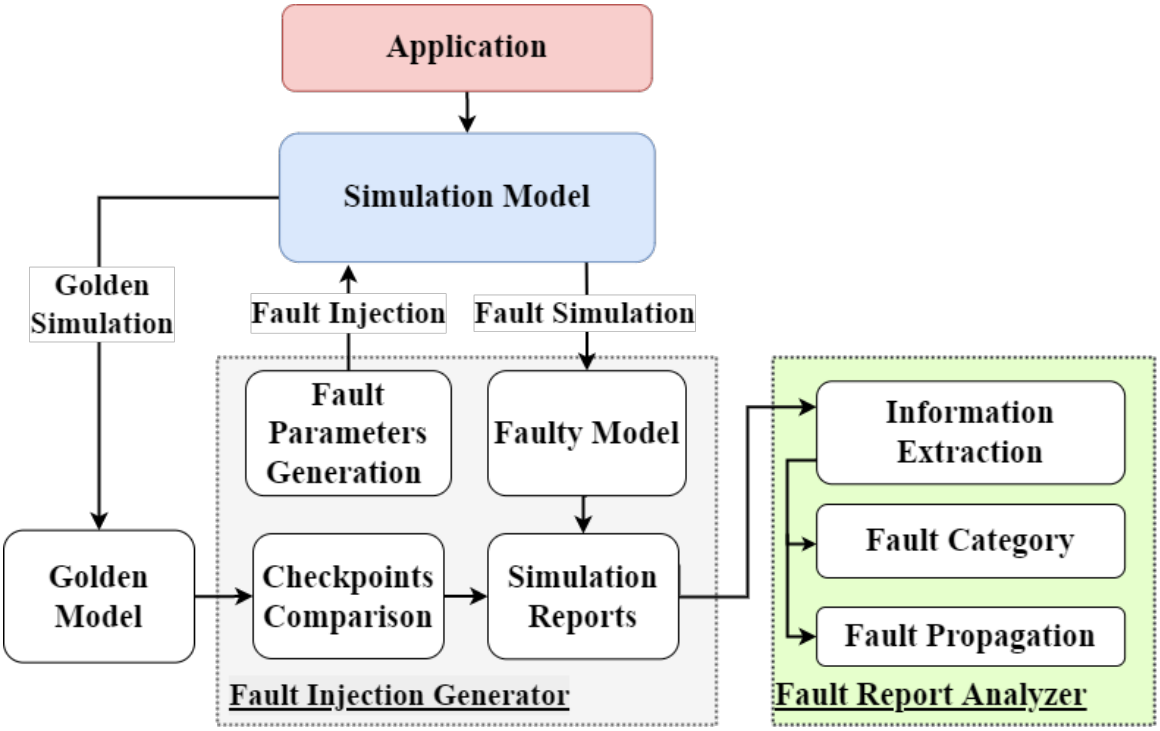

Figure 3.2: Proposed Fault Injection Environment

The fault injection generator module is responsible for launching, setting up, and monitoring the fault simulation, and it creates a report for each fault injection. The reports are then passed to the report analyzer to evaluate the propagation and the fault category. The source code of an application written in assembly or C language is the input to the many-core processor. Before starting the fault injection campaign, a fault-free simulation of the many-core processor is conducted to extract the golden model. This model is used as a reference to check the impact of the fault simulation. The golden checkpoints are the applications' outputs, the execution time, and the state architecture of the target at the end of the execution.

3.3 Fault Simulation Platform

Since the OpenPiton design is open-source, the injection of bit flips can be performed by accessing the targeted hardware components using a Hardware Description Language (HDL) simulator. An essential first step before building a fault injection framework is to identify the point of intervention within the OpenPiton simulation flow. The simulation of the OpenPiton is composed of two main steps [38]:

- **Building a simulation model:** During this step, an instance of the OpenPiton many-core is generated depending on the specified configuration options. It represents a 2D mesh composed of the OpenPiton tiles integrated on a single OpenPiton chip. The number of cores within the chip is defined by the x and y dimensions of the 2D mesh.

- **Running a C language test on a simulation model:** This phase consists of several automated steps that allow to set up the simulation environment. It builds the corresponding assembly code of the C test using the GCC cross compiler. Afterward, it generates the program memory image, links the symbol table, and finally calls the selected simulator to start the simulation. Different tools are responsible for performing the latter steps.

A simulation-based fault injection campaign is elaborated by monitoring the simulation and injecting a fault at the desired time and location. Therefore, we have adopted the second step of the OpenPiton simulation. The modifications are intended to interrupt the automated flow of executing a test on the simulation model. The fault injection generator module is now responsible for calling the HDL simulator and coordinating its workload.

3.4 Fault Injection Generator

The fault generator handles the fault simulation and reports generation. This part of the engine is written in the Tcl language, which was chosen for its wide use in industry and its interfacing capabilities with the ModelSim simulator. The functionalities of the components of this module are detailed in Section 3.1. The flow diagram of the fault injection generator is provided in Figure 3.3.

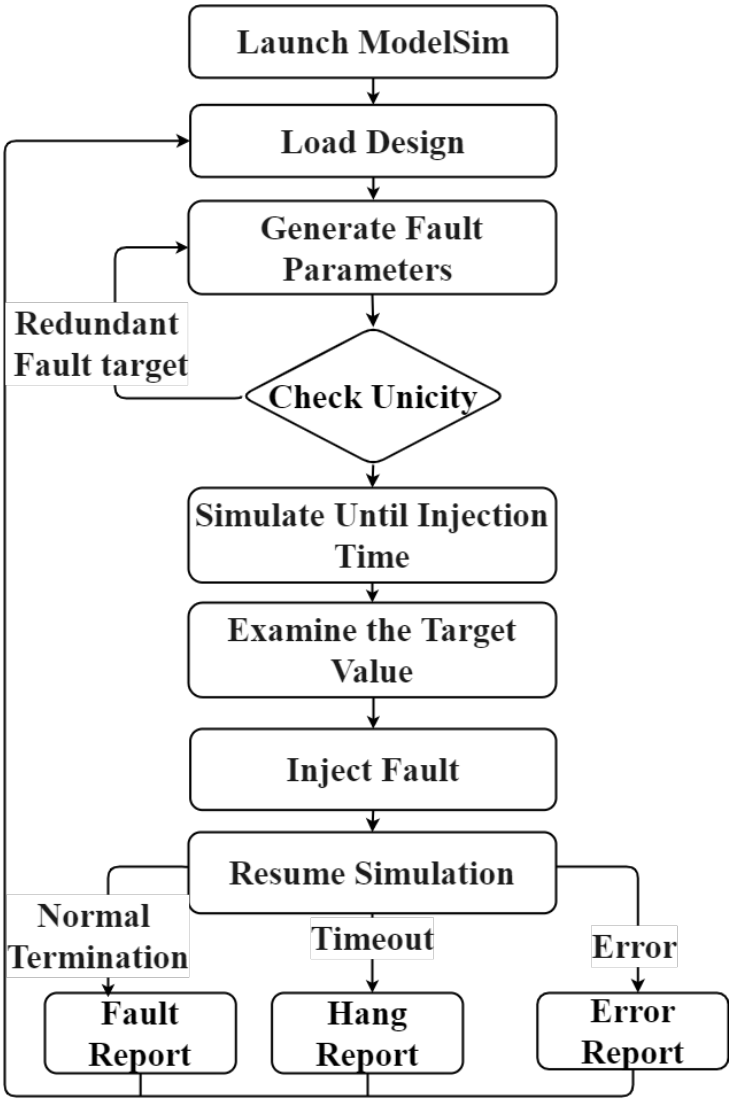

Figure 3.3: Fault Injection Generator Flow

After loading the OpenPiton simulation model, the fault injection parameters are randomly generated. This phase differs according to the desired fault target. In order to achieve full coverage of the possible faults scenarios, an exhaustive fault injection campaign that targets the total address space at different stages of the program execution is required. However, this approach is clearly unachievable due to the state explosion problem. To solve this issue, researchers have used uniform randomization to generate the fault parameters [39]. This technique allows to accurately estimate several radiation impacts at a lower computational cost. Afterward, the randomly generated fault configuration is compared with the fault history to avoid duplicating the same fault scenario. In case the fault is redundant, the generated parameters are discarded and the fault parameter generation is repeated. Once, the fault parameters are finalized, the simulation starts running until it reaches the injection time. The value of the injection location is checked to artificially insert the bit-flip. The evaluation of the current value of a register bit is performed using the ModelSim command **examine**. The Tcl script uses the ModelSim **force** command to inject the fault, with the duration of the bit-flip characterized by the force options *deposit* and *freeze*.

- The **deposit** parameter holds the fault injection until the program tries to override the value of the register.
- The **freeze** holds the forced bit flip in the target register until the end of execution.

The force parameters allow the emulation of different radiation scenarios. After inserting the fault, the simulator resumes the application's execution. Finally, the simulation model is reloaded, and the same flow is performed for the next fault injection. The following Listing provides a simplified version of the TCL script responsible for generating and injecting a fault in the general purpose registers. This pseudo-code doesn't cover the report generator functionality.

Listing 3.1: Simplified Version of the TCL script

```
1: for {set i 0} {$i < $loop_max} {incr i} {
2:     #Opening the file that saves the fault parameters history
3:         set fault_history [read [open "faults_history.mem" r]]
4:         while 1 {
5:             #Randomly generating the fault injection parameters
6:                 set tile [expr {int(rand() * $tile_number}]
7:                 set register [expr {int(rand() * $register_number)}]
8:                 set position [expr {int(rand() * $register_position)}]
9:                 set time [expr {int(rand() * $execution_time + 1)}]
10:                #Checking the unicity of the generated fault parameter
11:                set fault_history_unicity [lsearch $fault_history
12:                "${tile}${register}${position}${time}"]
13:                if {$fault_history_unicity == -1} {
14:                        set fault_history [linsert $fault_history
15:                        [llength $fault_list]
16:                        "${tile}${register}${position}${time}"]
17:                        break
18:                        }
19:        }
20:        #Running simulation until the fault injection time
21:        run $time us
22:        #Examining the value of the fault injection bit
23:        if {examine -binary $force_location]=="1'b0"} {
24:                set value "1"
25:                } else {
26:                set value "0"
27:                }
28:        #Injecting the bit-flip
29:        force -deposit $force_location 'b${value}
30: }
```

3.4.1 Fault Target Parameters

The developed fault injection environment considers the register file and the instruction register as fault targets. The register file is selected due to its criticality in the presence of transient faults. It is important to select the instruction register as the fault target, as there is a direct link between the instruction register and the program execution.

The faults targeting the register file are considered uniform as the fault is injected independently from the current register operation. However, the instruction register injection is only considered during the clock cycle in which it is being written.

In this environment, we consider two fault injection parameters: fault injection time and fault injection location. The generation of these parameters depends on the type of fault target. For the general-purpose registers, the fault injection starts after 10000 clock cycles to ensure that the application has been properly initialized (warm-up time). Therefore, the fault injection time is chosen randomly between the warm-up time and the final execution time extracted from the fault-free simulation. The fault injection location for the general purpose registers consists of the core, the register, and the position within the register.

The instruction register is holding the current instruction code at the Instruction Fetch Unit IFU level. This work only considers the instructions directly related to the application program. In order to identify these instructions, we extract the memory image of the application program. Subsequently, we use the assembly code generated by the GCC cross compiler from the C code of the application benchmark to identify the corresponding instruction of each assembly operation. Once we have identified the instructions of the application's main program, we extract their execution times from the fault-free simulation. Therefore, the fault injection time is a random selection from the list of the execution times of each main program instruction. The fault injection location for the instruction register consists of the core, the main program instruction, and the position within the instruction register. These parameters are arbitrarily generated.

3.4.2 Report Generation

Depending on how the application execution is terminated, the fault injection environment generates three distinct types of reports. The application's execution termination may be affected by the fault injection. There are three types of execution termination: a normal execution termination, an execution that finishes with an error indication, and an execution that doesn't terminate due to a processor halt. For the normal execution termination, when the application program reaches the stack return instruction, the OpenPiton monitors trigger ModelSim using the *stop* command to terminate the simulation. In this case, the developed fault injection environment generates the *fault report*. This report consists of the simulation time, the simulation state, the comparison between the content of the register file, the shared distributed L2 cache memory, and the outputs of the application with those of the golden model simulation. However, for an execution that terminates with a simulation error, we don't compare the checkpoints with those of the golden model. The *error report* contains only the error message and time. In case the simulation duration has exceeded the golden execution time by two times, we can conclude that the processor has reached a halt state. The Tcl script terminates the current simulation and creates a *hang report*. the simulator terminates the current execution and creates a hang report.

3.5 Fault Report Analyzer

The reports created during the fault injection simulations are passed to the report analyzer. The information extracted from the reports helps classify the faults into one of the following categories that we adopted from Cho et a. [40]:

- ***Vanished*** when the output result and the architecture state match those of the golden model.
- ***Output Not Affected (ONA)*** when the output result is correct, but one or more bit

from the architecture state is different from the fault-free model.

- ***Output Mismatch (OMM)*** when the output result is incorrect at the end of the execution.
- ***Unexpected Termination (UT)*** when the simulation exits with an error.
- ***Hang*** when the simulation time exceeds twice the golden execution time without terminating.

In addition, the propagation metric is evaluated to determine the impact of the fault injection on the other cores. The memory and the register file of all the cores except the target core are compared with the fault-free model. An error is characterized as propagated if one or more bit of the architectural state has changed. However, this parameter is not considered in case of a hang and an unexpected termination of the simulation.

3.6 Summary

The proposed assessment methodology is a custom fault injection generator and fault report analyzer built for the open-source many-core processor OpenPiton. The aim of building this environment is to evaluate the vulnerability of the OpenPiton processor in presence of soft errors. A soft error is represented by a bit-flip injected in the register elements. The fault injection generator inserts bit-flips into the register file and instruction register of a particular core of the OpenPiton processor. The duration of the fault injection differs according to the force command options. Depending on the execution termination type the fault generator module creates a report that summarizes the effect of the injected fault. Subsequently, the assessment environment classifies the impact of soft errors into several categories. The classification allows giving an overview of the resilience of the processor.

Chapter 4

OpenPiton Framework

This chapter presents the open-source many-core processor OpenPiton framework. It discusses the motivation for using the OpenPiton platform and its current application domains. We provide an insight into the various components of this processor's architecture. Afterward, we enumerate the available configuration options that can be implemented to alter the default architecture. Finally, we describe the execution flow of the OpenPiton framework.

4.1 OpenPiton Many-core Processor

The industries are increasingly incorporating many-core processors in their recent projects due to their promising performance enhancement. However, the industrial many-core processors are not budget friendly and they are commercialized in bare metal. These drawbacks limit the research activities that aim to experiment, evaluate and improve the design of this new generation of processors. Therefore, the academic research community is constrained by the lack of open-source processors that allows core scalability. As a result, there is a discrepancy between academic and industrial research efforts in this field.

The OpenPiton research framework is an open-source many-core processor developed

by the Princeton Parallel Group at Princeton University [41]. This framework gives the opportunity to explore the computational potential and the flexibility that comes with an extensible many-core processor. The support of an extensive number of cores and the configuration of the design justify the computational requirements of space applications. Nevertheless, the integration of this open-source in a safety-critical application depends on the efforts that aim to evaluate the resilience of this many-core processor to radiation effects.

4.1.1 Motivation of Using OpenPiton

The OpenPiton framework has been a focus of attention in recent research activities due to its maturity, active support, and continued release [42] -[43]. This multi-threaded many-core processor can scale from one core up to 500 million cores. The framework is built upon the commercial OpenSPARC T1 core developed by Oracle [44]. Therefore, it inherits its stability, supporting tools, and a large test suite in the assembler and C languages. In addition, the source code is written in the Verilog hardware description language, and it can be simulated by a wide array of simulators.

This open platform also provides configuration options for the number of cores, threads per core, cache size and associativity, and different cache coherence topologies. The flexibility of the design allows to investigate the resilience of many-core processors under different configuration scenarios. Furthermore, the code can be extended and reused to enable the implementation and reproduction of various applications, and OpenPiton is synthesizable and can be emulated on FPGA; a 25-core many-core processor of the OpenPiton architecture has already been taped out on a 32 nm die with over 460 million transistors [45]. Finally, this processor has been integrated into several educational and industrial research domains [2]. Given the above features, OpenPiton can be considered a strong candidate to launch the transition from single-core to many-core processors on existing space

applications.

4.1.2 Application Domain of OpenPiton Processor

The OpenPiton processor is mainly used in academic research activities. The framework is advantageous for research laboratories that lack access to adequate hardware resources. This platform facilitates the elaboration of their contribution. Some research works have focused on implementing novel computer architecture ideas based on OpenPiton. The authors of [46] have elaborated a heterogeneous many-core by integrating a RISC-V core into the OpenPiton infrastructure. The work presented in [47] aims to replace the NoC-based interconnection of the OpenPiton many-core processor with crossbars and meshes in order to evaluate the data interference. Another work [48] has focused on building a debugging compiler for many-core processors that have been implemented for OpenPiton. Several state-of-art research work have included OpenPiton framework as part of the experimental results validation [49]-[50]-[51]. However, there is a lack of research works that aims to evaluate the vulnerability of open-source many-core processor to soft errors. Therefore, this work presents a reliability analysis that attempts to motivate the integration of this COTS many-core processor in critical applications.

4.2 OpenPiton Processor Architecture

The OpenPiton processor architecture is tile structured. It is composed of a configurable number of chips that communicate with each other via a chipset and three P-Mesh NoCs. The chip is a cluster that hosts a 2D mesh network of tiles that host a core, a pipelined Floating-Point Unit (FPU), and two levels of cache. The chipset monitors the traffic of memory and I/O requests generated by the chip, which is connected to one chipset, the latter also connected to its neighboring chipsets. The chipset is connected to the chip using

a chip bridge. The connection is established by extending the three intra-chip NoCs. In order to ensure the data consistency between the tiles and the chips, the memory elements are protected by a cache coherence protocol.

The number of tiles within a chip is configurable from 1 to 65536, and each tile has three levels of cache. This processor is scalable both in the number of tiles inside the chip and the number of chips integrated into the processor. The Figure 4.1 given below provides an overview of the OpenPiton architecture.

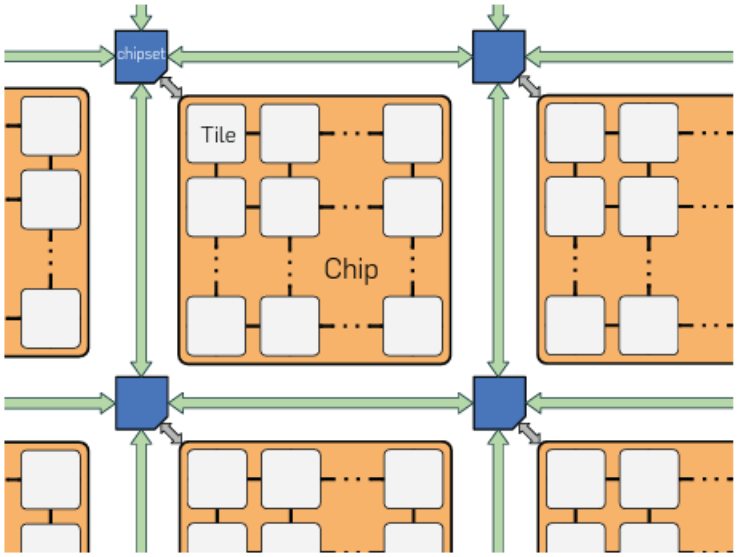

Figure 4.1: Overview of OpenPiton Architecture [2]

The large flexibility of the OpenPiton design allows its integration into a wide range of applications. The configuration targets the number of instances and the size of different components of the design. Also, it provides various interconnection schemes. Table 4.1 provides a summary of the main configuration options.

Table 4.1: OpenPiton Configurable Components [2]

Hardware Component	Configuration Option
Cores Per Chip	1 - 65536
Threads Per Core	1/2/4
L1 I-Cache	8/16/32KB
L1 D-Cache	4/8/16KB
L1.5 and L2 Caches	way associativity
Intra-Chip Topology	2D mesh, crossbar
Inter-Chip Topology	2D mesh, 3D mesh, crossbar, butterfly

4.2.1 OpenPiton Tile Architecture

The tiles consist of a single-core processor. This core is a revised version of the OpenSPARC T1 core. All cores are identical, therefore, the OpenPiton is a homogeneous many-core processor. The tile embeds a private L1.5 cache, a shared L2 cache, and the external FPU. The architecture of the tile is displayed in Figure 4.2. The two caches L1.5 and L2 communicate through the three P-mesh routers. The core is connected to its corresponding FPU and to the L1.5 cache via a crossbar arbiter (CCX). The L1.5 cache is responsible for managing the data flow between the crossbar and the NoCs.

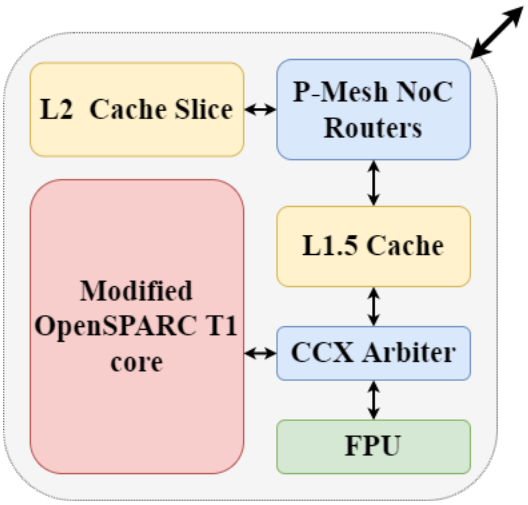

Figure 4.2: OpenPiton Tile Block Diagram

4.2.2 OpenPiton Core Architecture

The modified OpenSPARC T1 core integrated into the tile is a multi-threaded 64-bit processor that implements the SPARC V9 instruction set architecture (ISA). The core presents a good infrastructure choice due to its optimized area requirements and wide integration in industrial projects.

The modifications performed by the OpenPiton team on the OpenSPARc T1 processor aim to further reduce the area and the complexity of the design. For instance, the number of threads per core decreased from 4 to 2. Thus, the Translation Lookaside Buffer (TLB) has been also reduced. In addition, the Stream Processing Unit (SPU) was removed. To achieve a configurable architecture, several configuration registers were implemented in the design.

4.2.3 OpenPiton Cache Distribution

Each tile is composed of three levels of caches. The first level is a private L1 cache integrated into the core and comprising the instruction and data caches. The second level is a private L1.5 cache that is added to the OpenPiton processor to serve as a converter between the crossbar and the NoC communication interfaces. In case a date request to the L1 cache results in a miss, the L1.5 cache forwards the request to the L2 cache. If the latter also results in a miss, the request is forwarded to the chipset to provide the required data either from the off-chip memory or the other chips.

The three NoCs help routing requests between the L1.5 cache and the L2 cache. Also, they handle the communication between the L2 cache and the off-chip memory controller. To ensure cache coherence, the NoCs have different priorities, and each one handles a specific type of request. The L2 cache is distributed and inclusive of L1 and L1.5 private caches. By default, all caches are 4-way associative. The memory consistency model is based on a modified MESI coherence protocol.

4.3 Summary

This chapter highlights the motivation for incorporating many-core processors into recent industrial and academic projects. It addresses the underlying benefits of the overall performance. Afterward, we introduce the open-source many-core processor framework OpenPiton. We discuss the motivation for choosing OpenPiton as a potential many-core COTS for safety-critical applications. Further, we enumerated the current academic research activities performed using OpenPiton. We have observed the lack of research work that aims to perform reliability analysis of open-source processor. Finally, we describe the OpenPiton architecture and we emphasize the flexibility of the design.

Chapter 5

Experimental Setup and Results to Evaluate SEUs Impact on OpenPiton Many-core

In this chapter, we describe the case studies performed on the OpenPiton Many-core processor. The conducted experiments aim to evaluate the impact of SEUs on specific fault injection locations. The chapter begins by providing details about the experimental setup and defining some terminologies applied in the analysis of the results. Subsequently, we give an overview of the benchmark applications that have been implemented in the experiments. Next, we discuss the setup and the obtained results from three different experiments.

5.1 Environment Setup

The fault injection campaigns were executed on a Linux server with 160 CPUs clocked at 2.4GHz and 1TB RAM. During the experiments, the many-core processor is operating in AMP mode. Each core is operating independently from the other. The results of the fault injection of each experiment are grouped into six different classes according to the force

type and the fault location. The fault location can be an instruction register, a general-purpose register actual data, or the Error Correction Code (ECC) of a general-purpose register. The separation between the results of the injections in the register data and in the ECC is made to evaluate the behavior of the protection mechanism. The force command type can be either *deposit* or *freeze*. Hence, the six fault injection classes are:

- force -deposit in the register data
- force -freeze in the register data
- force -deposit in the register ECC
- force -freeze in the register ECC
- force -deposit in the instruction register
- force -freeze in the instruction register

The results of the experiments are analysed using the rates of the soft error categories defined in 3.5. These rates represent the percentage of occurrences of each category within the total number of fault injection simulations.

5.2 Benchmark Applications

Throughout the simulation-based fault injection campaigns, we have adopted three well-known benchmarks: *Fibonacci series*, *Matrix Multiplication*, and *Cyclic Redundancy Check*. The benchmark applications are written in C language and integrated into the Open-Piton Many-core processor simulation platform. The choice of the selected benchmarks is based on the wide use in the research experiments and the computational load executed by these programs.

5.2.1 Fibonacci Series Benchmark

The Fibonacci benchmark was chosen as a case study due to its extensive computational load [52]. The Fibonacci sequence is a recursive pattern that dependents on previous calculations. Therefore, the injected bit flip in the registers can propagate the error in the rest of the Fibonacci series. The pseudo algorithm for producing n element of the Fibonacci series is given in the Algorithm 1. During the experiments, the Fibonacci series application is set to calculate 8 elements.

Algorithm 1 Pseudo-code to Calculate n Elements of Fibonacci Series

$fibonacci\_serie[0] = 0$

$fibonacci\_serie[1] = 1$

for $i = 2 \rightarrow n - 1$ **do**

$fibonacci\_serie[i] = previous\_number + current\_number$

$previous\_number = current\_number$

$current\_number = fibonacci\_serie[i]$

end for

5.2.2 Matrix Multiplication Benchmark

The Matrix Multiplication (MM) is used to solve linear algebra problems. These calculations can be found within several safety-critical application programs. The MM benchmark is characterized by an extensive occupation of the memory [53]. The pseudo-code represented in Algorithm 2 performs an $n \times n$ matrix multiplication. In the experiments, we have implemented an 8×8 matrix multiplication. To facilitate the evaluation of the output result, the first matrix was composed of all 1's, and the second matrix was composed of all 2's.

Algorithm 2 Pseudo-code to Calculate $n * n$ Matrix Multiplication

```
for i = 0 → n − 1 do
    for j = 0 → n − 1 do
        C[i][j] = 0
        for k = 0 → n − 1 do
            C[i][j] += A[i][k] × B[k][j]
        end for
    end for
end for
```

5.2.3 Cyclic Redundancy Check Benchmark

The Cyclic Redundancy Check (CRC) is a generic benchmark that is based on simple logic operations as shown in Algorithm 3. The CRC is a protection technique used to detect data corruption in storage components. Hence, this application doesn't stress specific components of the processor. In the experiments, we calculate the CRC of a $9\ bytes$ message.

Algorithm 3 Pseudo-code to Calculate Cyclic Redundancy Check of a n byte Message

```
for byte = 0 → n − 1 do
    remainder = remainder xor (message[byte] << 8)
    for bit = 8 → 0 do
        if remainder & TOPBIT then
            remainder = (remainder << 1) xor POLYNOMIAL
        else
            remainder = (remainder << 1)
        end if
        CRC = remainder xor FINAL_XOR_VALUE
    end for
end for
```

5.3 Experiment 1: SEUs Impact Categorization for 2-core and 3-core System Implementing Fibonacci Benchmark

The objective of this experiment is to evaluate the impact of scaling the number of cores on the fault injection results. Therefore, the experiments were performed on a 2-core system and a 3-core system. In the 2-core system, the Fibonacci series is executed in two cores. In the 3-core system, the Fibonacci series is executed in three cores. Furthermore, the OpenPiton processor was set to the default configuration of one thread per core. In this experiment, we have performed 1000 fault injection simulations on the six classes defined in Section 5.1. The total number of injected faults is 12000.

5.3.1 Experimental Analysis

We have performed an experimental analysis based on the five soft error classes defined in Section 3.5: Vanished, ONA, OMM, UT and Hang.

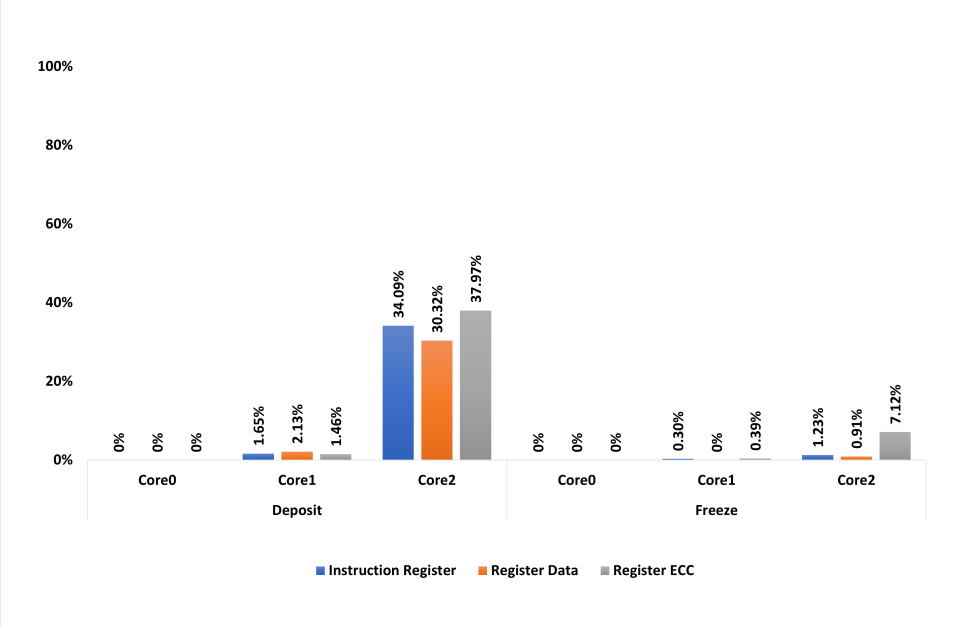

Figure 5.1: The Percentage of the *Vanished* Category of the 3-core System

Vanished: The *Vanished* category has a negligible occurrence rate in the 2-core system. Figure 5.1 represents the percentages of the *Vanished* category for the 3-core system using the force modes deposit and freeze. The percentages of the *Vanished* category for the 3 cores demonstrate that scaling the number of cores increases the *Vanished* rate due to the memory reuse. The bit-flip injected using the force deposit mode can be overwritten by the program in the subsequent execution processes. Therefore, the *Vanished* category has an increased rate in the force deposit mode. On the other hand, the injected bit-flip persists until the end of the execution in the force freeze mode which increases the probability of affecting the state architecture of the core. For this reason, the *Vanished* category rates are low using the freeze mode.

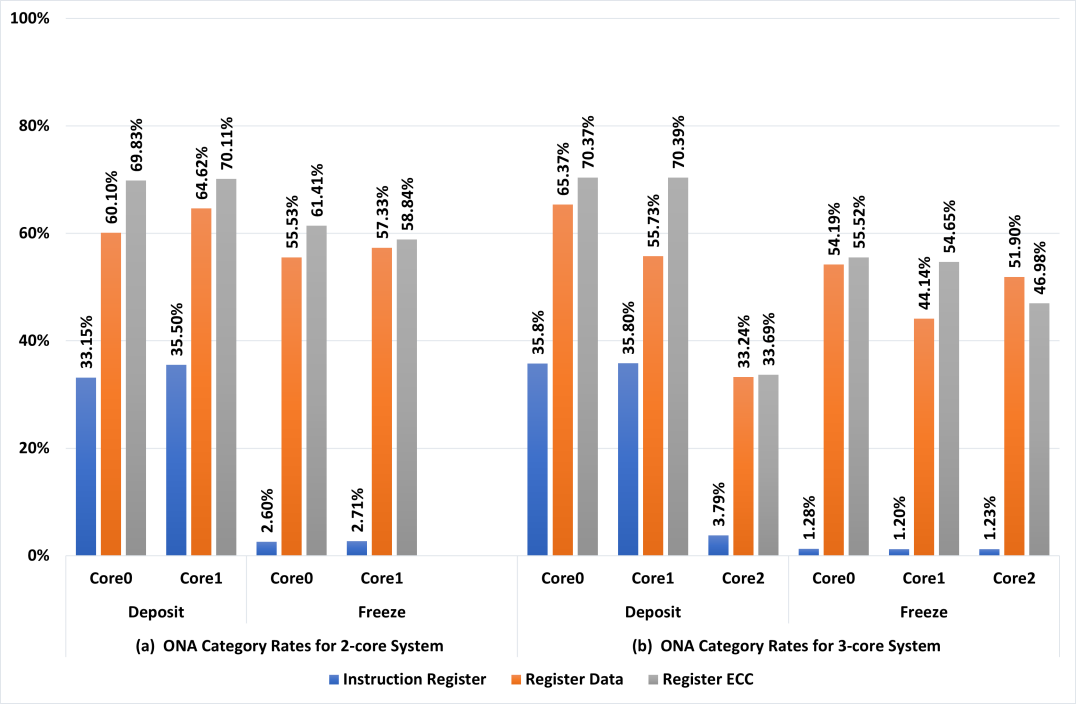

Figure 5.2: The Percentage of the *ONA* Category of the 2-core System (a) 3-core System (b)

Output Not Affected (ONA): the *ONA* rates for the 2-core system and the 3-core system using the force mode deposit and freeze are represented in Figure 5.2. The results have shown that scaling the number of cores presents similar correct output results rates. According to the graph, the instruction register has the lowest *ONA* rates compared to those injected in the register data and register ECC. We can conclude that the instruction register is the most vulnerable fault location. The instruction register stores the instruction code of the program, as a result, the fault injection in this register has a significant impact on the final output. The *ONA* rates for the register data and the register ECC are close. However, the register ECC have the highest *ONA*. The register ECC represents the detection and correction mechanism to protect the content of the register. Despite that the value of the register ECC does not represent the data of the program, the fault injection in these bits can result in changing the value stored in the register data and consequently affect the output result. Moreover, the three fault locations have revealed a decrease in the *ONA* occurrence rate when the fault is injected using the freeze mode compared to the deposit mode. When a bit-flip remains in the instruction register during program execution, it affects all subsequent instructions from the time of injection. Thus, the impact of freeze mode on the *ONA* percentage is greater for the faults targeting the instruction registers where the percentage has decreased from 30% on average to 2%. However, the freeze mode only decreases the *ONA* rate of the faults injected in the general-purpose registers by 5%.

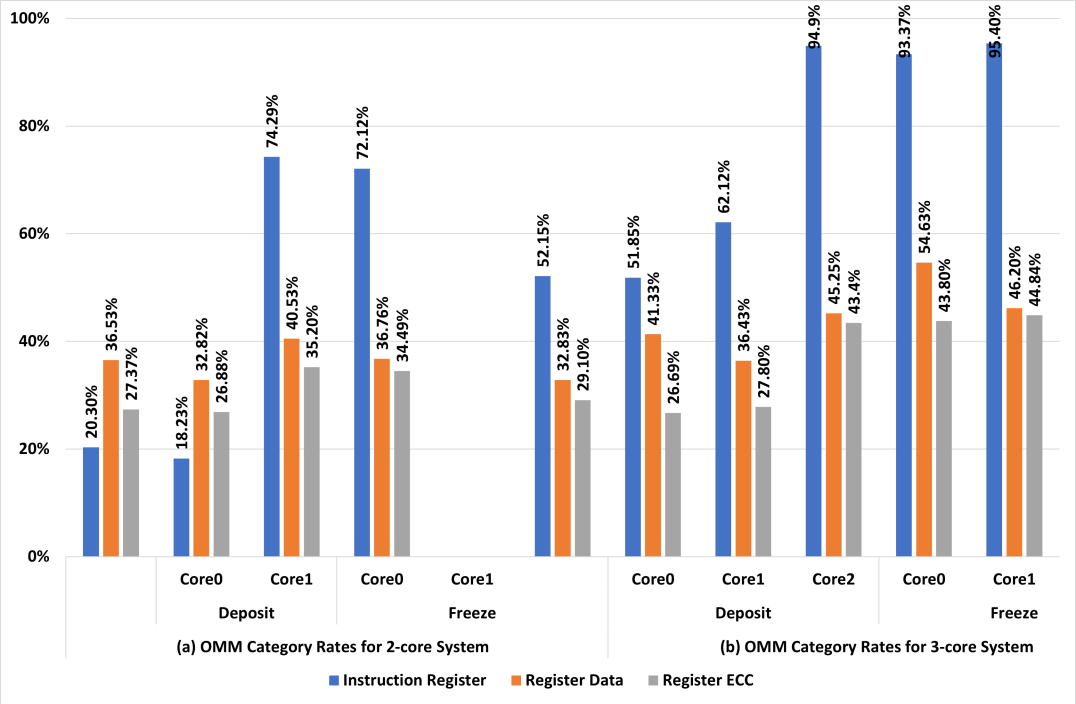

Figure 5.3: The Percentage of the *OMM* Category of the 2-core System (a) 3-core System (b)

Output Mismatch (OMM): This category represents a normal termination of the simulation with an incorrect final output result. The *OMM* category has the opposite behavior of the *ONA* category that represents a normal termination with a correct output result. The *OMM* category rates shown in Figure 5.3 for the 2-core system and the 3-core system demonstrates higher rates for the 3-core system. Based on the correlation between the *OMM* and the *ONA* rates, the instruction register faults have the highest occurrence of *Output Mismatch (OMM)*s followed by the faults injected in the general-purpose register data, and finally, the faults injected in the general-purpose register ECC due to the same reasons explained in the *ONA* paragraph. Like the *ONA* category, the freeze mode affects the *OMM* occurrence rate in all the fault targets. The faults injected in the instruction register faults have an *OMM* percentage of 55% using the deposit injection and 94% using the freeze

injection option. Whereas the output result of the Fibonacci application is erroneous in 32% and 46% of the fault injection simulations taking place in the general-purpose register, under the deposit and freeze mode respectively. The direct link between the instruction register and the output of the executed program explains the impact of soft errors on the correctness of the final result. The *OMM* rates during the 2-core system experiment (Fig. 4b) also increased in freeze mode in comparison to deposit mode. On the other hand, during the deposit mode, the percentage of *OMM*s in the instruction register faults is the lowest. This decrease is explained by the high UT occurrence using deposit mode as demonstrated in Figure 5.4.(a).

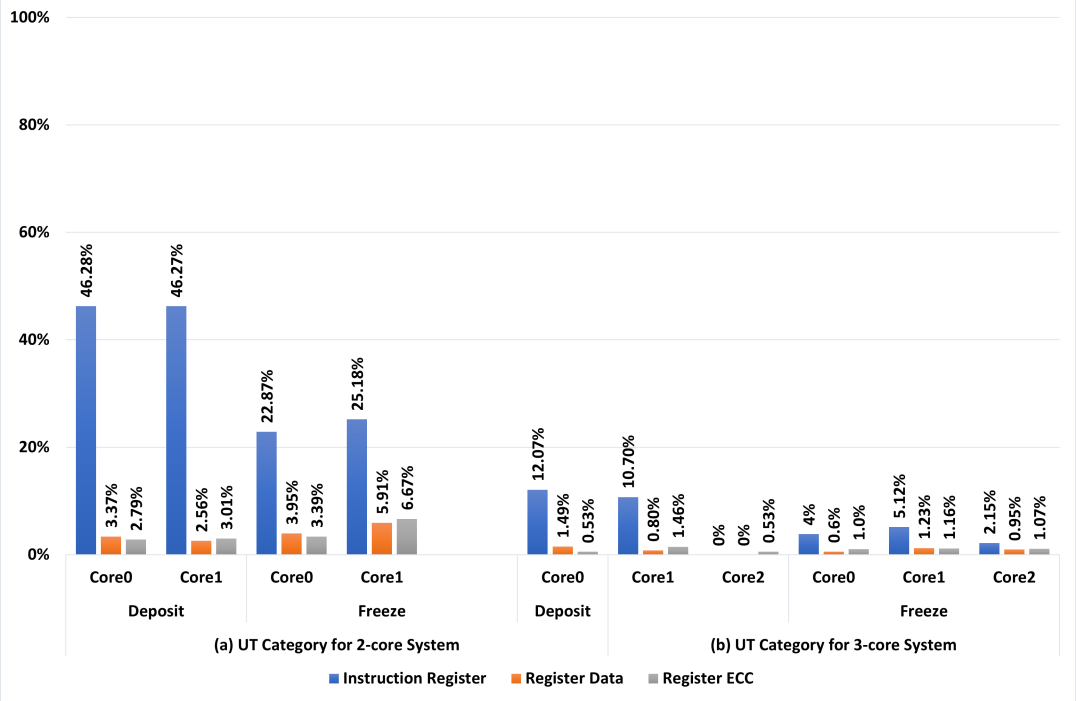

Figure 5.4: The Percentage of the *UT* Category of the 2-core System (a) 3-core System (b)

Unexpected Termination (UT): the *UT* rates for the 2-core system and the 3-core system using the force mode deposit and freeze are represented in Figure 5.4. Scaling the number of cores showed a decrease in the *UT* rates especially for the instruction register

fault location. This behavior explains the higher *OMM* rates for 3-core system compared to 2-core system. Most of *UT* occurrences have originated from the instruction register fault simulations. This rate is a result of fault injection in the instruction opcodes and in critical instructions E.g., branch, stack save, and stack return instructions, which leads to an abnormal termination of the execution. The interruption of the execution can affect the output result of the other cores. In fact, during the 2-core system experiment, the majority of the fault injections that caused a UT resulted in an incorrect output of the Fibonacci program in both cores. Only the faults targeting the instruction register presented a few cases where the abnormal termination didn't affect the result in both cores.

The 3-core system experiment showed that instantiating three cores can be interesting for redundancy techniques implementation. The faults injected in the instruction register using deposit mode had one core out of the three with a correct output result in all the *UT* occurrences. However, 39% of the *UT* occurrences of the "force -freeze in instruction register" class has one out of three cores that presents a correct output result.

Hang: the OpenPiton processor experienced very few *hang* cases throughout the entire set of fault simulations. The 2-core system experiment resulted in 5 *hang* cases after 6000 fault injection simulations, while the 3-core system experiment only reported one *hang* for the same number of fault injections. Therefore, the scale of the core number reduces the *Hang* cases since the redundancy of the cores helps to overcome the halt state.

SEUs propagation to the other cores showed a significant impact on the memory. The fault simulation of the 2-core system experiment resulted in a 100% propagation rate to the memory of the other core, but the register file didn't experience any propagation. During the 3-core system experiment, the injected error showed little impact on the third core memory, though the two other cores showed a 100% propagation rate to their memory. The register file experienced a negligible number of propagation cases. However, the propagation of

soft errors from the target core to the memory elements of the other cores did not disturb the execution flow of the program running on those cores.

5.3.2 Discussion

This set of fault simulations revealed that faults injected in the general-purpose register and those injected in the ECC had higher rate of correct output results when compared to the instruction register. Furthermore, the freeze mode decreases reliability, especially when the soft error is targeting the instruction register. Besides, the soft errors targeting the program instructions have a severe impact on the execution flow and the program result. This impact increases when the fault persists throughout the execution. Although the effect of the SEU on the instruction register is significant, the probability of the occurrence of such faults is low in reality. Finally, the implementation of the three cores revealed an improvement in vanished and hang categories. The 3-core system experiment revealed that scaling the number of cores improved the soft error effect on the final output result and the propagation rate. Despite, the high sensitivity of the L2 cache to the radiation effects. The corrupted data didn't effect the reliability of the overall system. In addition, very few cases of register file corruption of the other cores of the system have been reported.

5.4 Experiment 2: SEUs Impact Categorization Based on the Implemented Application

The key objective of this experiment is to compare the impact of SEUs depending on implemented benchmarks. The three implemented benchmarks (Fibonacci, MM, and CRC) have distinct characteristics based on the computational effort and the propagation of errors to the output result. When the final output result depends on the calculation of previous intermediate results, the bit-flip injected in an input variable can propagate in the

register file and corrupt the final result. In addition, the fault locations have an effect on the benchmark execution.

During this experiment, we instantiated one core of the OpenPiton processor with the default configurations. For each of the six fault injection classes (defined in 5.1), we have performed 3000 fault injection simulations. Therefore, the total fault simulations conducted in this experiment is 56000 simulations (3000 bit-flip injections x 3 benchmarks x 6 fault injection classes).

5.4.1 Experimental Analysis

To better understand the impact of the SEUs based on the application benchmark, we have established the result analysis based on the fault location: general-purpose register actual data (Register Data), Error Correction Code (ECC) of a general-purpose register (Register ECC), and instruction register.

Fault Injection In the Register Data

The general purpose registers data are responsible for holding the program variables. Therefore, the fault injections targeting this fault location have a direct impact on the final output result. When we analyze the vulnerability of the benchmarks based on their algorithms, we can easily realize that the MM benchmark occupies a larger space than the other two benchmarks which increases the possibility of being affected by fault injection. In addition, the output result is calculated using a triple for loops. Therefore, the injected bit-flip is likely to propagate to the final output result.

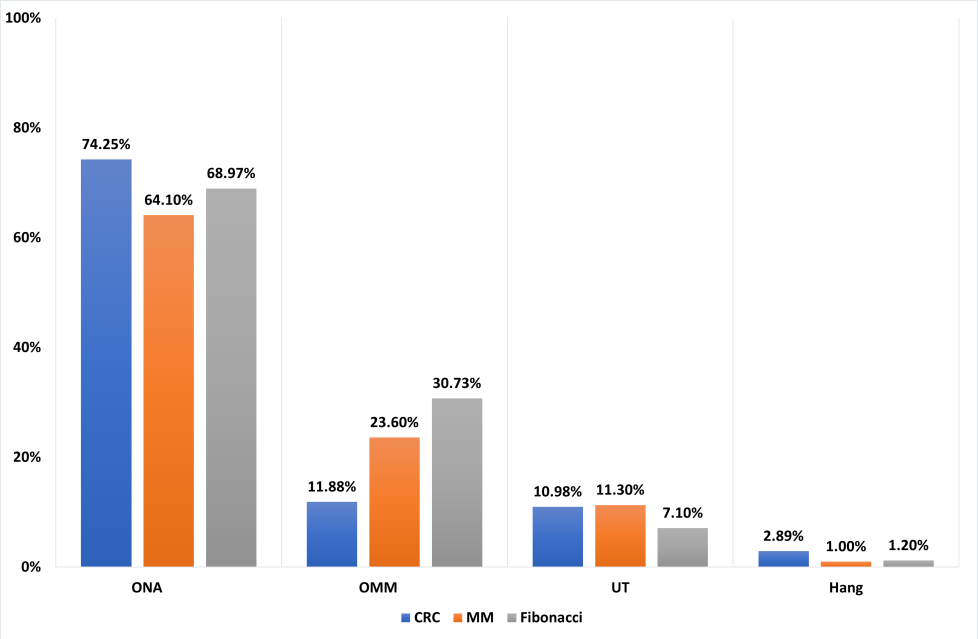

Figure 5.5: The Percentage of the Soft Error Categories for the *force -deposit in the register data* Fault Injection Class

force -deposit in the register data fault injection class results are displayed in Figure 5.5. The three benchmarks have exhibited close percentages of the soft error categories. However, the Matrix Multiplication benchmark has the lowest *Output Not Affected (ONA)* (74.25%, 64.1%, 68.97% respectively for CRC, MM, and Fibonacci benchmarks) due to the extensive data occupation problem that has been explained in the previous paragraph. The same explanation cane be applied to the percentages of the *Output Mismatch (OMM)* category where the MM benchmark has the highest occurrence rate. In addition, the CRC and MM benchmarks have higher *Unexpected Termination (UT)* percentages compared with the Fibonacci application (10.98%, 11.3%, 7.3% respectively for CRC, MM, and Fibonacci benchmarks). The algorithms of the benchmarks MM and CRC include several for loops and if conditions. These structures are translated in the assembly language into registers holding the counter register value and others storing the comparison results used

to evaluate the conditions of the branch and jump instructions. The bit-flips inserted in these critical variables can produce exceptions that lead to an abnormal termination of the program execution. The fault injection in the register data using the force deposit mode has resulted in a few cases of *Hang* SEU impact category. An average of 1.69% of *Hang* cases was reported.

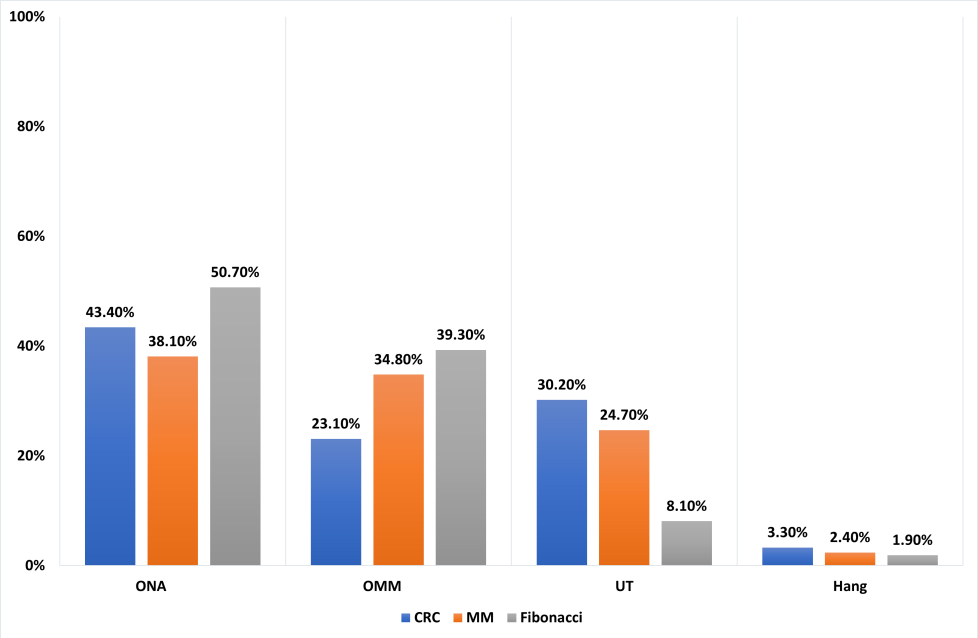

Figure 5.6: The Percentage of the Soft Error Categories for the *force -freeze in the register data* Fault Injection Class

force -freeze in the register data: Figure 5.6 shows that the results concluded for the *force -deposit in the register data* fault class are valid for the *force -freeze in the register data* (lower *ONA* rates, higher *OMM* rates for the MM benchmarks, and Less *UT* cases in the Fibonacci benchmark). However, the overall impact of the SEUs has been aggravated. The average of *ONA* rates are 25% less than those in the deposit mode. In addition, the average of the *UT* percentages has increased more than two times. Furthermore, the *Hang* cases have increased. This behavior shows the effect of the fault injection duration on the

vulnerability of the processor.

Fault Injection In the Register ECC

The Error Correction Code (ECC) is a mitigation technique used to detect and correct corrupted data. Nevertheless, this additional hardware can affect the system's behavior when exposed to a radiation environment. In this part of the experiment, bit-flips are injected into the ECC of the general purpose register. The ECC is independent of the application therefore three benchmarks have the same range of percentages of the soft errors impact categories.

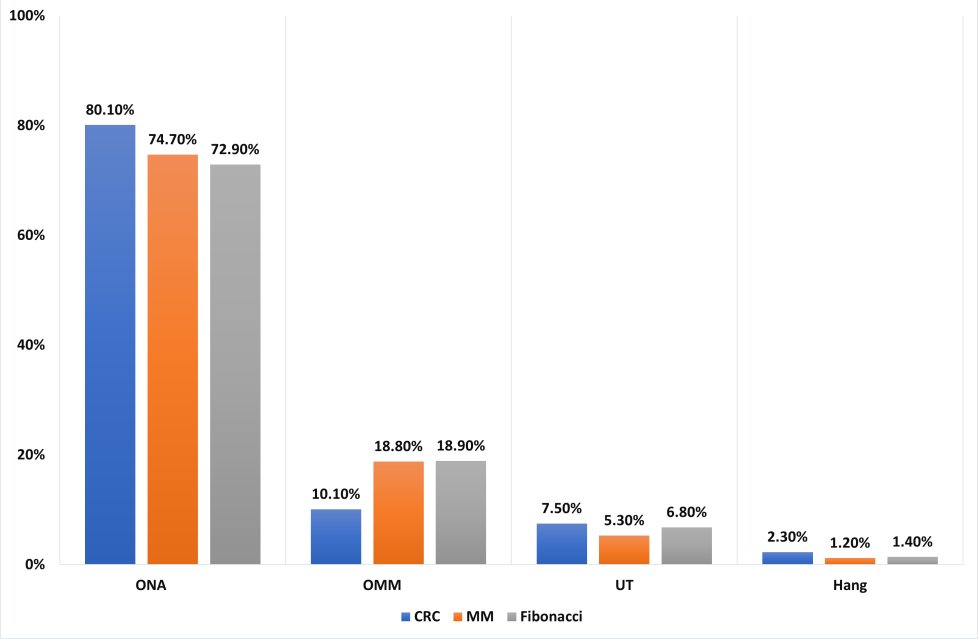

Figure 5.7: The Percentage of the Soft Error Categories for the *force -deposit in the register ECC* Fault Injection Class

force -deposit in the register ECC results demonstrated in Figure 5.7 show that the generated final output was erroneous in 15.93% of the fault injection campaigns using the three benchmarks. While the program execution has terminated with an error indication or an exception in 6.53% of the cases. Finally, the processor has reportedly entered a halt

state 1.69% on average.

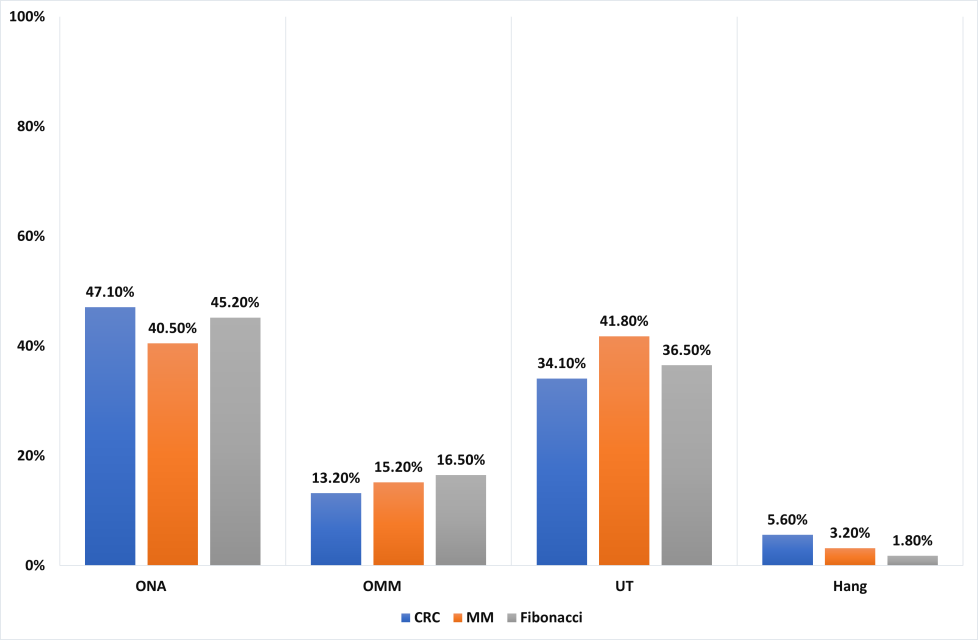

Figure 5.8: The Percentage of the Soft Error Categories for the *force -freeze in the register ECC* Fault Injection Class

force -freeze in the register ECC the Figure 5.8 shows that extending the duration of fault injection in the ECC bits significantly increase the percentage of *Unexpected termination*. The *UT* average percentage for the three benchmarks has increased from 6.53% to 37.47%. However, the program has generated a correct output result of 44.26% in the freeze mode compared to 75.6% in the deposit mode. The percentage of the *Hang* category has increased by 1.9%. The most concerning aspect of the inject a bit-flip in the ECC bits using the freeze mode is the abnormal termination of the execution. This behavior could cause a system failure if it's not handled properly.

Fault Injection In the Instruction Register

The instruction register store the instruction code in the Instruction Fetch Unit. The injecting time of the bit-flips corresponds with the clock cycle where it will be fed to the Decode stage. Therefore, this fault injection has a severe impact on the system's reliability. Regardless, the probability of occurrence of such faults is very low.

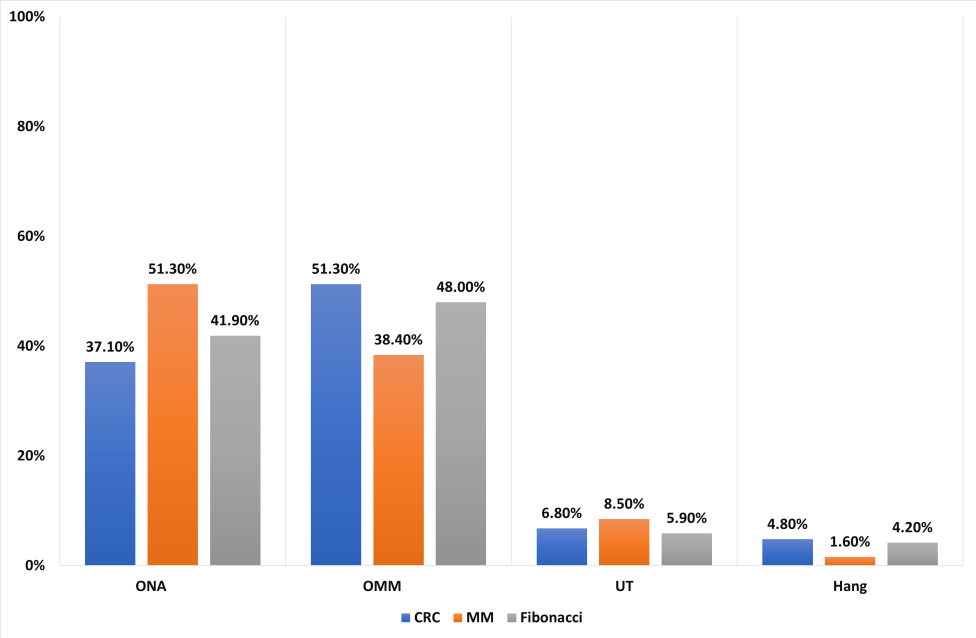

Figure 5.9: The Percentage of the Soft Error Categories for the *force -deposit in the instruction register* Fault Injection Class

force -deposit in the instruction register results are illustrated in the Figure 5.9. The fault injection campaign performed on the instruction register has demonstrated that the *ONA* percentage for the instruction register fault location is lower than those of register data and register ECC in the deposit mode. In addition, the MM benchmark has an increased *ONA* (51.3%) percentage compared to the CRC and the Fibonacci benchmarks (37.1% and 41.9% respectively). However, the *UT* soft error category has similar results to the register data and register ECC fault locations in the deposit mode.

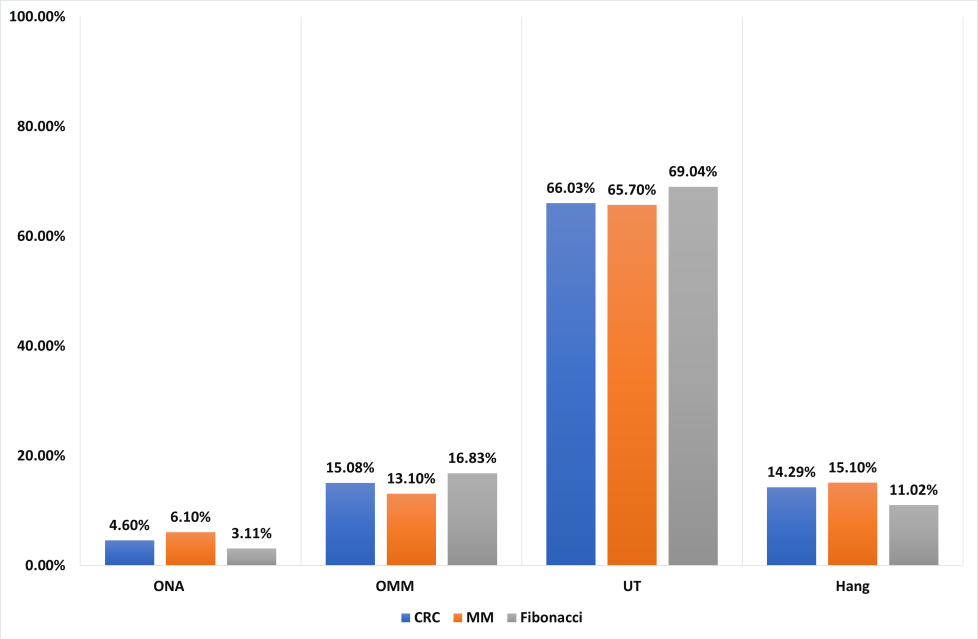

Figure 5.10: The Percentage of the Soft Error Categories for the *force -freeze in the instruction register* Fault Injection Class

force -freeze in the instruction register soft error categories percentages are presented in the Figure 5.10. The application has generated the expected output only in 4.6% of the cases. The *UT* category is the dominant SEUs impact reported in this fault injection campaign with an average of 66.92% percentage. Furthermore, the processor has hanged for 13.47% of the total simulations performed for this category. This behavior shows the effect of the fault injection duration on the vulnerability of the processor. The SEUs affecting the registers that store the instruction code have dangerous consequences on the functionality of the processor especially in the freeze mode.

5.4.2 Discussion

This experiment has demonstrated that the impact of the SEUs depends on the fault location. The percentages of the soft error categories for the register data depend on the

algorithm and the number of critical variables. Whereas, the register ECC is independent of the executed application. The instruction register is more vulnerable to soft errors impact. The results have shown that the percentage of *OMM* is reportedly higher than those of the rest of the fault locations. The fault injection for a long duration inserted using the ModelSim command *force -freeze* increases the percentage of producing an erroneous output result, an unexpected termination, and a processor hang. Although the percentage of *Hang* soft error category is low in most of cases, it represents a system failure. Once, the processor is hanged the normal execution can only be restored by a system reset or other implemented exception handlers.

5.5 Experiment 3: Evaluation of the Impact of Implementing Triple-Modular Redundancy on the Reliability

The analysis of the previous experiment results has shown that the OpenPiton many-core processor is vulnerable to SEUs events. The final fault injection campaign demonstrated the presence of erroneous results at the end of the simulation. The percentage of incorrect outputs depends on the fault injection location and the duration of the injection. Even though the architecture of the OpenPiton integrates mitigation techniques to increase the reliability of the system. For instance, the general purpose registers are protected by the Error Correction Code (ECC). However, the integration of the fault-tolerant technique is not enough to protect the application running on an independent core from producing erroneous outputs. Further, the SEUs targeting the ECC bits can originate errors in the output result. As a result, the present experiment aims to estimate the effectiveness of implementing a Triple-Modular Redundancy (TMR) on OpenPiton reliability.

The Triple-Modular Redundancy is a widely used mitigation technique to improve the

accuracy of the system [54]-[55]. The mechanism implements three instances of the sub-module responsible for computing the output and a voter as shown in Figure 5.11. The final output is selected by a voter that evaluates the outputs generated by the sub-modules and assigns the final output based on the majority analysis. Since the OpenPiton is a many-core processor we can benefit from the multiplicity of the cores to implement the TMR technique. In this case, each submodule represents an OpenPiton core executing the desired application.

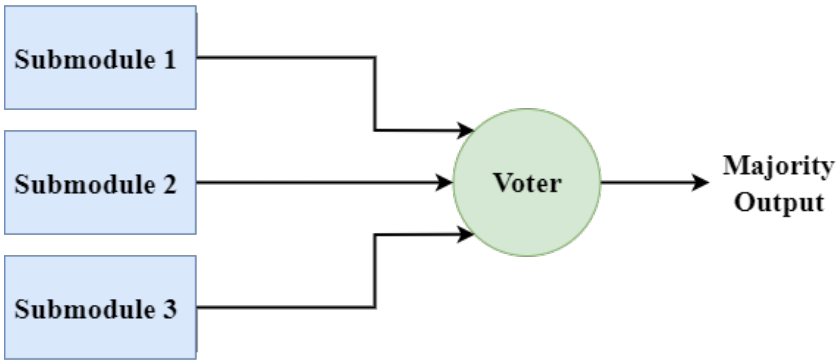

Figure 5.11: Triple-Modular Redundancy Block Diagram

The validation of the reliability improvement when implementing the TMR fault-tolerance technique on the OpenPiton is conducted by instantiating an OpenPiton processor that implements three cores. The three cores execute in parallel the same benchmark while our fault injection framework is responsible for injecting bit-flips to emulate the behavior of SEU event. The faults are injected in the same fault locations as the previous experiments (register data, register ecc, and instruction register).

5.5.1 Experimental Analysis

To evaluate the robustness of the TMR technique two fault injection campaign was performed on a three-core OpenPiton instance. The first campaign injects a single bit-flip

in one core at a time. An SEU event that affects a single bit of the hardware device is referred to in the literature as Single-Bit Upset SBU. The second campaign injects a single bit-flip in two different cores at the same fault injection simulation. The fault injection parameters: core, register, position, and time are chosen randomly. During this experiment, we instantiated three cores of the OpenPiton processor with the default configurations. For each of the six fault injection classes (defined in 5.1), we have performed 3000 fault injection simulations. Therefore, the total fault simulations conducted in this experiment is 72000 simulations (2000 bit-flip injections x 3 benchmarks x 6 fault injection classes x 2 set of experiments).

In the following analysis, we evaluate the percentage of the erroneous results that will be corrected when the TMR technique is implemented.

Table 5.1: Percentage of Accuracy Improvement by Implementing When the Fault is Injected in a Single-core

Fault Injection Class		**Cyclic Redundancy Check**	**Matrix Multiplication**	**Fibonacci Series**
Register Data	**Force -deposit**	100.00%	100.00%	100.00%
	Force -freeze	95.49%	100.00%	100.00%
Register ECC	**Force -deposit**	100.00%	100.00%	93.91%
	Force -freeze	95.38%	100.00%	100.00%
Instruction Register	**Force -deposit**	86.59%	84.31%	99.79%
	Force -freeze	92.81%	99.55%	96.63%
Average		95.05%	97.31%	98.39%

The fault injections targeting a single-core per simulation have shown promising results in implementing the TMR mitigation technique. The percentages of the accuracy

improvement ensured by including TMR in the design are collected for the six fault injection classes defined in 5.1. Table 5.1 presents the percentages of the corrected output results for the three benchmarks (CRC, MM, and Fibonacci) based on the fault locations and injection mode (deposit or freeze). On average, the TMR has selected the correct output result 95.05%, 97.31%, and 98.39% respectively for the cyclic redundancy check, the Matrix Multiplication, and the Fibonacci series benchmarks. The output errors produced during an *Unexpected Termination* and a *Hang* are unrecoverable. Therefore, the percentage of the corrected outputs is lower for the fault injection classes that exhibited a higher percentage of the *UT* and *Hang* categories. Furthermore, the output results for the Fibonacci series have been significantly improved by the TMR as it has lower *UT* percentages.

Table 5.2: Percentage of Accuracy Improvement by Implementing TMR When the Fault is Injected in Two Cores Per Simulation

Fault Injection Class		**Cyclic Redundancy Check**	**Matrix Multiplication**	**Fibonacci Series**
Register Data	**Force -deposit**	92.65%	76.98%	76.72%
	Force -freeze	61.63%	61.34%	71.71%
Register ECC	**Force -deposit**	83.87%	83.26%	83.08%
	Force -freeze	64.53%	65.57%	78.14%
Instruction Register	**Force -deposit**	54.75%	62.57%	71.15%
	Force -freeze	39.03%	41.69%	44.11%
Average		66.08%	65.23%	70.82%

With the continuous shrinking size of the transistors and the compactness of the design, a single particle strike can affect multiple memory bits. This behavior is called Multiple-bit Upset (MBU). Depending on the physical layout of the OpenPiton processor, the affected bits can belong to different cores. In addition, in harsh radiation enthronements, multiple

SBU events can occur on different cores during the execution of the application. Therefore, the objective of this fault injection campaign is to estimate the capability of the TMR technique to mask the erroneous outputs when a bit-flip per core is injected into two cores during one fault simulation. The results provided in Table 5.2 show that the average of percentage of the selected correct outputs by the TMR has decreased compared to the fault injection campaign with a single-bit flip on one core only. This behavior is due to the higher occurrence of the *UT* and *Hang* soft error categories when the bit-flips are injected into two different cores. The Cyclic Redundancy Check, the Matrix Multiplication, and the Fibonacci Series average percentages of faulty outputs corrected by the TMR are 66.08%, 65.23%, and 71.82% respectively.

5.5.2 Discussion

According to the results, the effectiveness of adopting the TMR mitigation technique to increase the OpenPiton many-core processor reliability depends on the fault injection location (Register Data, Register ECC, and Instruction Register), the duration of the fault injection, and the multiplicity of the bit-flips. The results estimate a significant improvement in reliability when implementing TMR as a fault-tolerant technique for a bit-flip on a single core. Although the injection of 2 bit-flips has reduced the effectiveness of the TMR technique, the likelihood of the occurrence of this phenomenon is considerably low.

5.6 Summary

This chapter presents three experiments performed to evaluate the resilience of the OpenPiton Many-core processor in a radiation environment. We are interested in evaluating the reliability of the design, particularly against SEU events that affect the storage elements. Three experiments were elaborated that targets the general purpose registers and

the instruction register in the IFU. The first experiment demonstrated a high occurrence rate of incorrect output results and unexpected termination of the execution for SEUs targeting the instruction register. In addition, scaling the number of cores has showed an improvement in the effect of the propagation rate to the memory of the other cores, and the *Vanished* category. The second experimental setup objective is to evaluate the SEUs' impact on the different fault injection locations based on various benchmarks. The results revealed that the soft error categorization percentages are in the same range for the three benchmarks. However, we concluded that the small variation in the register data results originates from the number of critical variables utilized in the application algorithm. Finally, we estimate the output result accuracy improvement when applying the TMR technique. The overview of the results shows that the TMR is a robust mitigation technique that remarkably improves the overall reliability of the system.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Recent activities in the avionics domain have shown interest in emerging to COTS many-core processors due to the promising performance and moderate power/cost budget. However, the process of scaling the number of cores is limited due to the lack of the many-core COTS reliability certification in radiation environments. On the other hand, the OpenPiton many-core processor is an open-source COTS that provides a wide range of scalability and configurable architecture. Chapter 4 provides an overview of the architecture of the open-source OpenPiton architecture. This processor is a potential candidate to be considered as an Integrated Modular Avionic for avionic domains. Therefore, this thesis presents a scalable fault injection engine developed for the open-source many-core processor OpenPiton.

The proposed work enables the evaluation of the impact of soft errors on the targeted core through categorization. In addition, it enables the assessment of the SEUs' propagation beyond the core subjected to the fault. In Chapter 3 we present our fault injection methodology. We have employed the OpenPiton architecture as a case study for the proposed framework.

Several experiments were described in the Chapter 5 that aim to assess the reliability of OpenPiton. The first experiment objective is to estimate the impact of the scalability of the cores on processor reliability. Also, during this experiment, we evaluated the propagation of the impact of a bit-flip in one core to the architecture state of the other cores of the processor. The results proved increasing the number of cores reduces the percentage of erroneous output results in the overall system and has less propagation rate to the memory and the register file of the other cores. The second experiment measures the variation of the impact of SEU based on the different benchmarks. The experiment demonstrated that only the general purpose registers reliability is related to the implemented application. The instruction register has critical consequences when affected by SEUs events. The mitigation techniques implemented in the OpenPiton were unable to protect the cores from producing faulty output results. In addition, the significant scalability of the OpenPiton design provides the opportunity to explore the implementation of redundancy techniques to improve reliability.Consequently, the third experiment estimates the potential impact of implementing the TMR mitigation technique on reliability. We have considered two case scenarios. In the first scenario, an SBU affects only a single core per simulation. In this case, the TMR has proved significant enhancement to the reliability of the processor. In the second scenario, a single bit-flip is injected into two distinct cores. In this case, the improvement is approximately 30% less than the first case scenario.

6.2 Future Work

The reliability assessment of the OpenPiton processor conducted in the context of this thesis presents an overview of the resilience of this architecture to radiation effects. The fault injection framework developed to estimate the vulnerability of many-core OpenPiton processor can be further enhanced to cover more fault locations and variations of the architecture configuration. Therefore, this work is the foundation for future improvement

of both the fault injection framework and the reliability evaluation methodology. The list below proposes future work that can improve the purposed contribution:

- Applying different configuration options, e.g., cache coherence topology, may impact the results. Further scalability, configuration, and benchmark diversity can allow for a wider view of the SEU effect.

- Extending the fault injection location to cover the caches, the Network-on-Chip, and the chipset. In addition to considering more than one chip.

- Enhancing the fault injection campaign time by exploring emulation-based fault injection.

- Engaging parallel processing using the SMP multi-processing mode. Where the implementation of an OS could allow for task scheduling and resource sharing between the cores.

- The certification to integrate an architecture in the avionics domain includes other parameters than the resilience to radiation effects such as security and availability. Consequently, the evaluation of these metrics could facilitate the integration of the OpenPiton processor in safety-critical applications.

Bibliography

[1] András Vajda. *Multi-core and Many-core Processor Architectures*, pages 9–43. Springer US, Boston, MA, 2011.

[2] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source hardware platform for your research. *Commun. ACM*, 62(12):79–87, nov 2019.

[3] Ye Liu, Shinpei Kato, and Masato Edahiro. Analysis of memory system of tiled many-core processors. *IEEE Access*, 7:18964–18977, 2019.

[4] TSMC Leading Semiconductor Manufactor logic technology. https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_3nm. Last Accessed: July 7, 2022.

[5] Yan Solihin. *Fundamentals of Parallel Multicore Architecture*. 11 2015.

[6] Mingzhen Li, Yi Liu, Hailong Yang, Yongmin Hu, Qingxiao Sun, Bangduo Chen, Xin You, Xiaoyan Liu, Zhongzhi Luan, and Depei Qian. Automatic code generation and optimization of large-scale stencil computation on many-core processors. In *50th International Conference on Parallel Processing*, 2021.

[7] Florian Kluge, Mike Gerdes, and Theo Ungerer. An operating system for safety-critical applications on manycore processors. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 238–245, June 2014.

[8] Sourav Sinha, Neeraj Kumar Goyal, and Rajib Mall. Survey of combined hardware–software reliability prediction approaches from architectural and system failure viewpoint. *International Journal of System Assurance Engineering and Management, Springer*, 2019.

[9] Shuo Cai, Caicai Xie, Yan Wen, and Weizheng Wang. A low-cost quadruple-node-upset self-recoverable latch design. In *2021 IEEE International Test Conference in Asia (ITC-Asia)*, pages 1–5, 2021.

[10] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, 2005.

[11] Zuhal Ozturk, Haluk Rahmi Topcuoglu, and Mahmut Taylan Kandemir. Studying error propagation on application data structure and hardware. 2022.

[12] Dawei Li, Jie Wu, Keqin Li, and Kai Hwang. Energy-aware scheduling on multiprocessor platforms with devices. In *2013 International Conference on Cloud and Green Computing*, pages 26–33, 2013.

[13] Antonio Paolillo, Paul Rodriguez, Nikita Veshchikov, Joël Goossens, and Ben Rodriguez. Quantifying energy consumption for practical fork-join parallelism on an embedded real-time operating system. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, page 329–338, 2016.

[14] Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. Methodologies for the wcet analysis of parallel applications on many-core architectures. In *2015 Euromicro Conference on Digital System Design*, pages 748–755, 2015.

[15] Pablo Ramos, Vanessa Vargas, Maud Baylac, Francesca Villa, Solenne Rey, Juan Antonio Clemente, Nacer-Eddine Zergainoh, Jean-Francois Méhaut, and Raoul Velazco. Evaluating the see sensitivity of a 45 nm soi multi-core processor due to 14 mev neutrons. *IEEE Transactions on Nuclear Science*, 63(4):2193–2200, 2016.

[16] E. Del Sozzo, G. C. Durelli, E. M. G. Trainiti, A. Miele, M. D. Santambrogio, and C. Bolchini. Workload-aware power optimization strategy for asymmetric multiprocessors. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 531–534, 2016.

[17] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 121–136, Santa Clara, CA, February 2020. USENIX Association.

[18] Farrukh Hijaz, Brian Kahne, Peter Wilson, and Omer Khan. Efficient parallel packet processing using a shared memory many-core processor with hardware support to accelerate communication. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 122–129, 2015.

[19] Edward Petersen. *Foundations of Single Event Analysis and Prediction*, pages 13–76. IEEE, 2011.

[20] Jin-Woo Han, M. Meyyappan, and Jungsik Kim. Single event hard error due to terrestrial radiation. In *2021 IEEE International Reliability Physics Symposium (IRPS)*, pages 1–6, March 2021.

[21] Jeffrey S. George. An overview of radiation effects in electronics. *AIP Conference Proceedings*, 2160(1):060002, 2019.

[22] Jie Li and Yue Wang. Transient fault tolerance on multicore processor in amp mode. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, pages 332–337, 2021.

[23] Seyyed Amir Asghari, Mohammadreza Binesh Marvasti, and Amir M. Rahmani. Enhancing transient fault tolerance in embedded systems through an os task level redundancy approach. *Future Generation Computer Systems*, 87:58–65, 2018.

[24] Takuya Azumi, Yuya Maruyama, and Shinpei Kato. Ros-lite: Ros framework for noc-based embedded many-core platform. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4375–4382, 2020.

[25] R. Baumann and Kirby Kruckmeyer. *Radiation handbook for electronics, available on TI.com (https://www.ti.com/applications/industrial/aerospace-defense/technical-documents.html)*. 01 2019.

[26] Aamir Mairaj. Preferred choice for resource efficiency: Integrated modular avionics versus federated avionics. In *2015 IEEE Aerospace Conference*, pages 1–6, March 2015.

[27] Fabrice Pancher, Vanessa Vargas, Pablo Ramos, Rodrigo Possamai Bastos, David Cx00E9;sar Ardiles Saravia, and Raoul Velazco. Nanosatellite on-board computer including a many-core processor. In *2021 IEEE 22nd Latin American Test Symposium (LATS)*, pages 1–6, 2021.

[28] Xavier Jean, Laurence Mutuel, and Vincent Brindejonc. Assurance methods for cots multi-cores in avionics. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–7, 2016.

[29] Anika Christmann, Adam Kostrzewa, Rolf Ernst, Marius Rockschies, Martin Halle, Frank Thielecke, Alexander Peuker, Alexander Kuzolap, Meiko Steen, Peter Hecker, Kai-Frederik Nessitt, and Selma Saidi. Integrating multi-/many-cores in avionics: Open issues and future concepts. In *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, pages 1–8, 2021.

[30] Michel Pignol, Florence Malou, and Corinne Aicardi. *COTS in Space: Constraints, Limitations and Disruptive Capability*, pages 301–327. 04 2019.

[31] Florian Kriebel, Semeen Rehman, and Muhammad Shafique. Studying aging and soft error mitigation jointly under constrained scenarios in multi-cores. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 139–142, 2019.

[32] M. D'Alessio, C. Poivey, V. Ferlet-Cavrois, H. Evans, R. Harboe-Sørensen, A. Keating, I. Lopez-Calle, F. X. Guerre, F. Lochon, S. Santandrea, A. Zadeh, M. Muschitiello, M. Markgraf, O. Montenbruck, A. Grillenberger, N. Fleurinck, K. Puimege, D. Gerrits, and P. Matthijs. Srams sel and seu in-flight data from proba-ii spacecraft. In *2013 14th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pages 1–8, 2013.

[33] Guideline for ground radiation testing of microprocessors in the space radiation environment ,pasadena, ca : Jet propulsion laboratory, national aeronautics and space administration, 2008. https://trs.jpl.nasa.gov/handle/2014/40790. Last Accessed: July 25, 2022.

[34] Luis Entrena, Mario García-Valderas, Almudena Lindoso, Marta Portela-Garcia, and Enrique Millán. *Fault Injection Methodologies*, pages 127–144. 04 2019.

[35] Tino Flenker, Jan Malburg, Görschwin Fey, Serhiy Avramenko, Massimo Violante, and Matteo Sonza Reorda. Towards making fault injection on abstract models a more accurate tool for predicting rt-level effects. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 533–538, 2017.

[36] J. Laurent, C. Deleuze, F. Pebay-Peyroula, and V. Beroulle. Bridging the gap between rtl and software fault injection. *J. Emerg. Technol. Comput. Syst.*, 17(3), may 2021.

[37] Bing Xue and Mark Zwolinski. Using formal methods to evaluate hardware reliability in the presence of soft errors. In *2022 17th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, pages 29–32, June 2022.

[38] OpenPiton Simulation Manual ,wentzlaff parallel research group. https://parallel.princeton.edu/openpiton/docs/sim_man.pdf. Last Accessed: July 16, 2022.

[39] Zhimin Li, Harshitha Menon, Kathryn Mohror, Peer-Timo Bremer, Yarden Livant, and Valerio Pascucci. Understanding a program's resiliency through error propagation. PPoPP '21, page 362–373, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A. Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, 2013.

[41] OpenPiton, Parallel Princeton Group Website. https://parallel.princeton.edu/openpiton/. Last Accessed: July 14, 2022.

[42] Michael B. Taylor. Technical perspective: Bootstrapping a future of open source, specialized hardware. *Commun. ACM*, 62(12):78, nov 2019.

[43] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. A comparative survey of open-source application-class risc-v processor implementations. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, CF '21, page 12–20, New York, NY, USA, 2021. Association for Computing Machinery.

[44] OpenSPARC T1, Oracle Website. https://www.oracle.com/servers/technologies/opensparc-t1-page.html. Last Accessed: July 14, 2022.

[45] Michael McKeown, Alexey Lavrov, Mohammad Shahrad, Paul J. Jackson, Yaosheng Fu, Jonathan Balkind, Tri M. Nguyen, Katie Lim, Yanqi Zhou, and David Wentzlaff. Power and energy characterization of an open source 25-core manycore processor. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 762–775, 2018.

[46] Katie Lim, Jonathan Balkind, and David Wentzlaff. Juxtapiton: Enabling heterogeneous-isa research with risc-v and sparc fpga soft-cores, 2018.

[47] Cheng Tan, Yanghui Ou, Shunning Jiang, Peitian Pan, Christopher Torng, Shady Agwa, and Christopher Batten. Pyocn: A unified framework for modeling, testing, and evaluating on-chip networks. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 437–445, 2019.

[48] Marcelo Ruaro and Kevin J. M. Martin. Manygui: A graphical tool to accelerate many-core debugging through communication, memory, and energy profiling. DroneSE and RAPIDO, page 39–46, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Yi-Chen Lu, Sai Pentapati, Lingjun Zhu, Gauthaman Murali, Kambiz Samadi, and Sung Kyu Lim. A machine learning powered tier partitioning methodology for monolithic 3d ics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2021.

[50] Neiel I. Leyva-Santes, Ivan Pérez, César A. Hernández-Calderón, Enrique Vallejo, Miquel Moretó, Ramón Beivide, Marco A. Ramírez-Salinas, and Luis A. Villa-Vargas. Lagarto i risc-v multi-core: Research challenges to build and integrate a network-on-chip. In Moisés Torres and Jaime Klapp, editors, *Supercomputing*, pages 237–248, Cham, 2019. Springer International Publishing.

[51] Lingjun Zhu, Lennart Bamberg, Anthony Agnesina, Francky Catthoor, Dragomir Milojevic, Manu Komalan, Julien Ryckaert, Alberto Garcia-Ortiz, and Sung Kyu Lim. Heterogeneous 3d integration for a risc-v system with stt-mram. *IEEE Computer Architecture Letters*, 19(1):51–54, 2020.

[52] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. Cold start in serverless computing: Current trends and mitigation strategies. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–7, 2020.

[53] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643, 2020.

[54] M Santhiya, S. Saranya, S. Vijayachitra, C. B. Lavanya, and M. Rajarajeswari. Application of voter insertion algorithm for fault management using triple modular redundancy (tmr) technique. In *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*, pages 578–583, Feb 2021.

[55] Jie Li and Yue Wang. Transient fault tolerance on multicore processor in amp mode. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, pages 332–337, Aug 2021.