Evaluating System Robustness Against Single Effect Upsets with Triple Modular Redundancy Integration

Aya Khaled Galal Mohammed

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

October 2024

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Aya Khaled Galal Mohammed**

Entitled: **Evaluating System Robustness Against Single Effect Upsets with Triple Modular Redundancy Integration**

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. Rodolfo Coutinho

_____ External Examiner
Dr. Mohsen Ghafouri

_____ Supervisor
Dr. Otmane Ait Mohamed

_____ Co-supervisor
Dr. Abdelwahab Hamou-Lhadj

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

_____ 2024

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Evaluating System Robustness Against Single Effect Upsets with Triple Modular Redundancy Integration

Aya Khaled Galal Mohammed

Many-core Commercial-off-the-shelf (COTS) processors offer a promising solution for meeting the performance and cost requirements of safety-critical avionics applications. However, the increased transistor count in these processors makes them more prone to soft errors, potentially leading to system failures. This raises challenges in integrating their use in critical applications due to limited research on how to mitigate the radiation-induced soft errors on open-source many-core processors. Various mitigation strategies have been proposed, including hardware and software techniques. Despite these efforts, a significant gap remains, largely due to the inefficient use of multi-core processor resources.

Our work addresses this gap by integrating Triple Modular Redundancy (TMR) into OpenPiton, an open-source many-core processor while maintaining minimal time and area overhead. This integration enhances the system's robustness against soft errors, leveraging unused cores for redundant threads, thus improving overall reliability. Our methodology includes detailed implementation strategies for multi-threaded TMR, addressing key issues such as synchronization across cores and race conditions. This study contributes to the field by introducing a novel TMR implementation for many-core processors and advancing the integration of fault tolerance mechanisms in complex computing environments, offering a robust solution for critical applications.

Acknowledgments

First and foremost, I wish to express my profound gratitude to God for granting me strength, wisdom, and perseverance throughout this research.

I extend my deepest thanks to my supervisor, Dr.Otmane Ait Mohamed, for their invaluable guidance, support, and encouragement. Their expertise and insightful feedback were instrumental in shaping this work and advancing my academic journey.

I am also sincerely grateful to my co-supervisor, Dr. Abdelwahab Hamou-Lhadj, for their constructive suggestions, support, and continuous motivation. Their contributions were crucial in refining the research and ensuring its success.

My heartfelt thanks go to my family, whose unwavering love and support have been a constant source of strength. To my husband, Ibrahim, and my son, Adam, your Love, patience, and encouragement gave me the motivation to complete this work. I couldn't have achieved this without you by my side. I love you endlessly.

I also wish to acknowledge my parents, Khaled & Dalia, for their enduring love, support, and belief in my abilities. Your sacrifices and encouragement have been a driving force behind my achievements.

I would like to extend my gratitude to Chifa Dammak, whose work on the fault injection framework served as a valuable baseline for my research.

To everyone who has supported and believed in me, thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we provide a brief overview of many-core processors, TMR implementation techniques, and their effect on boosting system robustness against Single Event Upsets (SEUs). We begin by presenting the motivation for our research. Following this, we discuss the rationale behind the industry's shift towards using commercial off-the-shelf (COTS) many-core processors. We also provide an overview of the various radiation effects on safety-critical applications and the associated challenges in enhancing their reliability against these effects. Subsequently, we highlight our problem statement and the contributions of this thesis. We briefly explain how we exploited the vast resources of many-core processors to implement TMR. Finally, a breakdown of the thesis chapters is provided.

1.1 Motivation

Due to the increasing complexity of industrial systems, single-core processors have failed to achieve optimal performance. This is why the industry has shifted towards many-core processors to meet these requirements [3]. Multi-core processors enable many activities to run simultaneously without suffering substantial performance bottlenecks, leading to faster computations, increased productivity, and quicker response times[4].

Despite the significant improvements brought by multi-core processors, these advancements come at a cost. Power consumption and heat dissipation are the main challenges [5]. According to Moore's Law, announced in 1975, the transistor count—a measure of an electronic system's capability—doubles every two years. The transistor count has exceeded this limit, shrinking the channel length from 3µm in 1987 to an expected 2nm in 2025 [6]. This miniaturization has allowed the integration of multi-core processors on a single chip, achieving outstanding performance and computational capabilities.

However, the limited robustness of multi-core processors against radiation effects has restricted their use in many safety-critical applications such as aerospace systems, banking systems, automated railway systems, and mission-critical embedded applications[7]. Radiation effects are categorized into permanent effects, known as "Hard Errors," and temporary effects, known as "Soft Errors." Soft errors occur more frequently than hard errors[8]. Consequently, the industry is keen to assess and improve system robustness against soft errors. A soft error can be modeled as a transient bit flip, potentially altering the system's control flow, which can have catastrophic consequences, especially for safety-critical systems [9].

In safety-critical systems, such as aerospace systems, boosting system reliability against soft errors is essential due to the significant radiation effects these systems experience[8]. A variety of techniques have been proposed to mitigate these effects, ranging from hardware mitigating techniques to software and hybrid mitigating techniques[10].

1.2 Multi-core Processors

With the industry's shift towards leveraging the extensive resources of multi-core processors, we provide a more in-depth analysis of this trend. Multi-core technology has facilitated parallel processing, enabling higher performance without increasing system frequency, thus reducing application execution time.

Performance Optimization With Enhanced RISC, known as "POWER," is the world's

first multi-core processor, launched by IBM in 2001. This innovative architecture allowed two on-chip processors to operate collaboratively with large on-chip memories at a very high bandwidth. Furthermore, combining four microprocessors into an 8-way module set a new industry standard by achieving a record clock speed of 1.3 GHz [11]. While multi-core processors offer numerous advantages, challenges such as inter-core communication and race conditions during program execution arise[12].

1.3 Radiation Effects

Radiation effects are a major concern for the system robustness of safety-critical applications, including aircraft avionics and nuclear power plants. Radiation effects can be classified into cumulative damage from multiple energy particles and damage caused by a single particle[13]. Single-event effects (SEEs) result from the damage caused by a single particle. When a charged particle passes through a device, it ionizes atoms along its path, generating charged particles like electrons and holes. SEEs can be further classified as either Hard SEEs, which cause permanent damage, or Soft SEEs, which are transient and can be corrected [14]. More details will be discussed in Chapter 2.

1.4 Mitigation Techniques

Extensive research has been conducted to address the challenges posed by these effects, resulting in several mitigation techniques. These techniques can be categorized into three groups: hardware mitigation techniques, software mitigation techniques, and hybrid mitigation techniques.

Software mitigation techniques often involve duplicating instructions or tasks and employing dual-use Error Detection and Correction Codes (ECC). By redundantly processing tasks or using ECC, these techniques enhance the system's ability to detect and correct

errors at the software level.

Hardware mitigation techniques focus on solutions at the circuit, logic, and architectural levels. Strategies include gate sizing, increasing capacitance, and implementing resistive hardening. These hardware-level interventions enhance system robustness by fortifying physical and logical components.

Hybrid mitigation techniques integrate both hardware and software approaches, combining hardware-based solutions with software strategies like redundant multi-threading and parallel processing. Hybrid techniques aim to detect and recover from soft errors more effectively, leveraging the full potential of multi-core processors [10].

1.5 Problem Statement and Thesis Contributions

Industries, particularly in aerospace applications, have adopted commercial off-the-shelf (COTS) processors to achieve performance and cost objectives. However, integrating multi-core COTS processors into safety-critical applications presents challenges due to their limited robustness against radiation-induced soft SEEs. Various mitigation techniques have been proposed, ranging from hardware to software solutions.

This thesis introduces a novel implementation of multi-threaded Triple Modular Redundancy (TMR) on the OpenPiton framework. The novelty of this work lies in integrating multi-threaded TMR within a highly configurable and resource-intensive architecture like OpenPiton[15]. Unlike conventional single-threaded TMR implementations, the multi-threaded approach introduces unique challenges, such as synchronization, race conditions, and parallel fault detection across multiple cores.

The primary objective of this research is to develop and integrate a multi-threaded TMR approach tailored for OpenPiton, leveraging its architectural features to enhance error detection and correction capabilities.

Additionally, this work addresses the challenge of integrating robust error mitigation

with OpenPiton's scalable architecture. It ensures that the system can handle faults across multiple cores while preserving the architectural flexibility of OpenPiton. By doing so, this research extends the boundaries of existing fault tolerance methodologies, bridging the gap between high-performance many-core processors and robust fault tolerance mechanisms for real-world critical systems.

1.6 Thesis Outline

The thesis is organized as follows:

- Chapter 2 discusses various topics, including soft errors and the OpenPiton multi-core processor used in this research. We provide an overview of the reliability assessment framework employed, explore different reliability assessment techniques, and offer a discussion on Triple Modular Redundancy (TMR) and its historical development.

- Chapter 3 provides an in-depth discussion of the TMR (Triple Modular Redundancy) implementation, introducing key components such as the golden model, the majority voting mechanism, the fault injection framework, and the process of results generation. Challenges encountered during the implementation are also addressed.

- Chapter 4 presents the experimental setup and the results of various experiments. The discussion covers the impact of TMR implementation across different cores and benchmarks, providing detailed insights into the outcomes of each experiment.

- Chapter 5 concludes the thesis with a comprehensive summary of key findings. Suggestions for future work are also provided, outlining potential directions for further exploration and improvement.

Chapter 2

Preliminaries

In this chapter, we provide an overview of the foundational concepts relevant to our work. We begin with a discussion of various SEEs, followed by a review of reliability assessment techniques. Next, we outline the framework used in this study, and introduce the many-core processor, OpenPiton. Lastly, we examine soft error mitigation strategies, with a focus on the TMR technique employed in our work.

2.1 Single-Event Effects

Radiation impacts the functionality of semiconductor devices in multiple ways. SEEs are caused by a single charged particle and can be categorized into two types: destructive effects, referred to as ”Hard Errors,” and non-destructive effects, known as ”Soft Errors.” A comprehensive classification of SEEs is shown in Figure 2.1.

Soft errors are non-destructive and result from transient bit flips in registers or memory cells, or transient pulses in logic cells. These errors can be easily overwritten by the system, and their effects can disappear if properly handled. In contrast, hard errors cause permanent damage to the circuitry. They can be triggered by Single-Event Latchup (SEL), Single-Event Burnout (SEB), or Single-Event Gate Rupture (SEGR) .

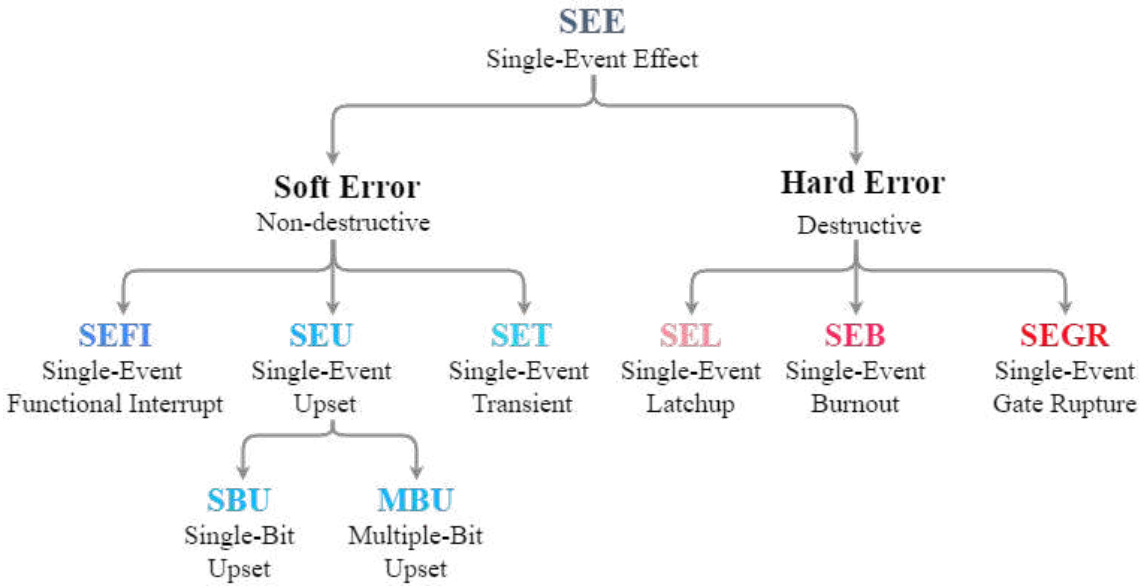

Figure 2.1: classifications of Soft Errors[1]

Soft errors, being non-destructive, can vary in their impact depending on the type of circuit affected. In combinational circuits, the effects of a soft error may disappear once the transient bit flip or pulse dissipates. However, in memory components and sequential logic, such errors can propagate through data states, potentially causing system failures in critical parts of the circuitry [1].

Soft errors are generally classified into three types: Single-Event Transients (SETs), Single-Event Upsets (SEUs), and Single-Event Functional Interrupts (SEFIs) [1].

- **SETs** occur as voltage spikes caused by high-energy particles, which lead to brief fluctuations in voltage or current.
- **reg** shows the exact register where the faults are injected.
- **SEUs** are cosmic ray-induced bit flips in memory storage, which may result in a single-bit upset (SBU) or a multiple-bit upset (MBU) as the particle passes through multiple bits in the device.
- **SEFIs** are more severe, often resulting from an upset in a control bit or register that

disrupts the system's functionality. These are considered the most critical form of soft errors.

- **SEFIs** are more severe, often resulting from an upset in a control bit or register that disrupts the system's functionality. These are considered the most critical form of soft errors.

In contrast, hard errors are destructive and cause permanent damage to the circuitry, which cannot be corrected or overwritten by the system.

- **SELs** occur when a charged particle passes through the parasitic silicon-controlled rectifier between the p-type and n-type regions of a CMOS transistor, causing a current flow from the power supply to the ground. If left unaddressed, this can permanently damage the circuit.
- **SEBs** are destructive breakdowns often occurring in MOSFET transistors, especially power transistors.
- **SEGRs** involves damage or breakdown of the gate oxide layer in a MOSFET or similar transistor due to a high-energy particle.

Given the critical impact that soft errors can have, especially in safety-critical applications such as aerospace, storage systems, cryptography, and embedded systems, extensive research has been dedicated to mitigating their effects.

2.2 Reliability assessments Techniques

As mentioned before in 1.3, soft errors have critical effects on electronic systems of safety-critical applications such as aerospace. Thus, that created many motives to mimic the soft error effects in various ways such as Radiation Ground Testing, and Fault injections

2.2.1 Radiation Ground Testing

Radiation Ground Testing is regarded as one of the most accurate methods for simulating soft errors. However, it is a time-consuming and costly approach. This technique involves exposing the silicon chip to artificial radiation environments, such as proton, neutron, or heavy ion beams, subjecting the system to radiation levels even higher than those encountered in real-life outer space. As a result, it provides a more realistic assessment of the effects of soft errors [16].

2.2.2 Fault Injection Methods

Another technique to mimic soft errors is Fault Injection at different abstraction levels such as Hardware-Based fault injection, Software-Based fault injection, Simulation-Based fault injection, Emulation-Based fault injection, and Hybrid fault injection.[17]

- **Hardware-based fault injection** is performed at the physical level by introducing faults directly into the circuit. These faults are injected into the system's hardware through various means, such as environmental parameters, power supply disturbances, or laser fault injection.

- **Software-based fault injection**'s primary purpose is to replicate the effects of hardware-based fault injection by simulating the same variations in pins at the software level.

- **Simulation-Based Fault Injection** occurs in high-level models, thus fault injection is done in the designing phase of the circuit. By employing several description languages, it targets various abstraction levels. It gives an overview of system reliability at the early stages before actual hardware is manufactured.

- **Emulation-Based Fault Injection** depends on exploiting the Field Programmable Gate Arrays (FPGAs) to emulate the system model, thus spending a shorter time

than Simulation-based Fault injections, giving the designer an opportunity to study the actual behavior to different errors.

- **Hybrid Fault Injection** allows the mixing of the features of software-based fault injection with observing the effects on the actual hardware.

2.3 OpenPiton

OpenPiton is the world's first open-source, general-purpose, multi-threaded many-core processor. It gained a lot of interest and focus, because of its maturity, continuous assistance, and ongoing release.

OpenPiton, developed by the Princeton Parallel Group at Princeton University, addresses the gap between the high cost of industrial many-core processors and the research community's need for an open-source framework that is scalable, configurable, and compatible with various verification tools. This one-of-kind multi-core processor can scale from one core to half a billion cores, allowing it to be used from small embedded systems to large data centers.

This 64-bit architecture framework is based on the OpenSPARC T1 core designed by Oracle [18], thereby benefiting from its stability, along with its supporting tools and comprehensive test suite available in both assembler and C languages.

2.3.1 OpenPiton Processor Architecture

Figure 2.2 illustrates the architecture of OpenPiton. This tiled-structured framework enables the integration of multiple tiles (cores) within a single chip, with the potential for scaling across multiple chips. Intra-chip communication is maintained through the Network on Chip (NoC) in 2D mesh topology. NOC router can have address space up to 256 tiles in one OpenPiton Chip

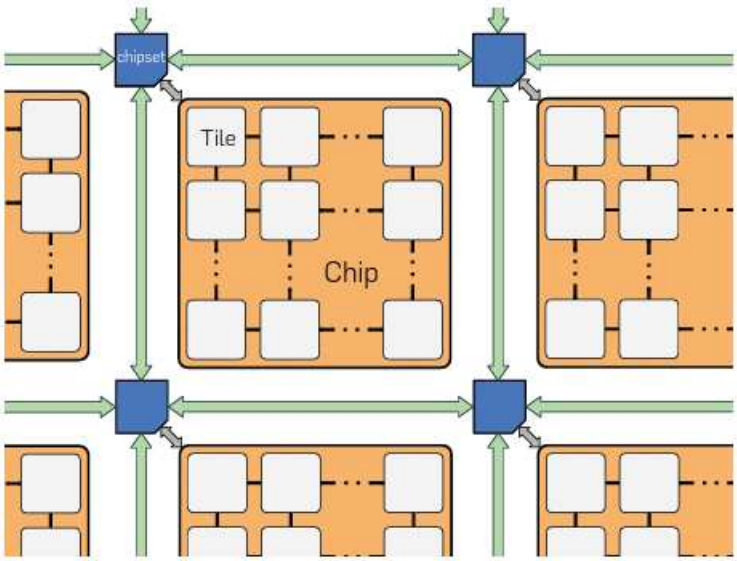

Figure 2.2: Overview of OpenPiton Architecture [2]

while inter-chip (off-chip) communication is facilitated by the chipset as shown in figure 2.3. The chipset logic is connected to the chip (tile array) via chip bridge as shown in figure 2.4. The chipset logic in addition to another three NOC routers allows seamless connection of multiple chips creating an integrated large system.

The cache architecture in OpenPiton is distributed across three levels, consisting of private L1 and L1.5 caches for each core, and a distributed shared L2 cache. The L1 cache is largely inherited from the industrial-grade OpenSPARC T1 and is split into a data cache and an instruction cache. The L1 data cache is an 8KB write-through cache, featuring a 16-byte line size and 4-way set-associativity. In contrast, the L1 instruction cache follows a similar structure but has a 32-byte line size.

Since the L1 cache uses a write-through policy, multi-core processors such as OpenPiton may experience network-on-chip (NOC) congestion. To mitigate this, an intermediate cache, referred to as the L1.5 cache, was introduced. This local write-back cache manages MESI protocol states and streamlines communication with the NOCs.

The L2 cache, shared across all tiles, is distributed across the system. By default, each tile is configured with a 64KB L2 cache, maintaining 4-way set-associativity [15].

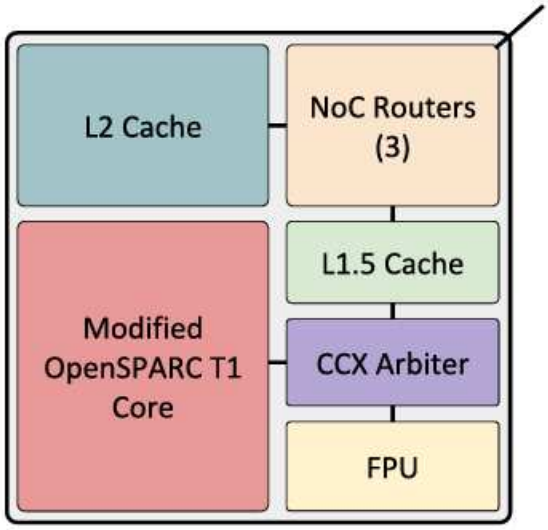

Figure 2.3: Architecture of a tile

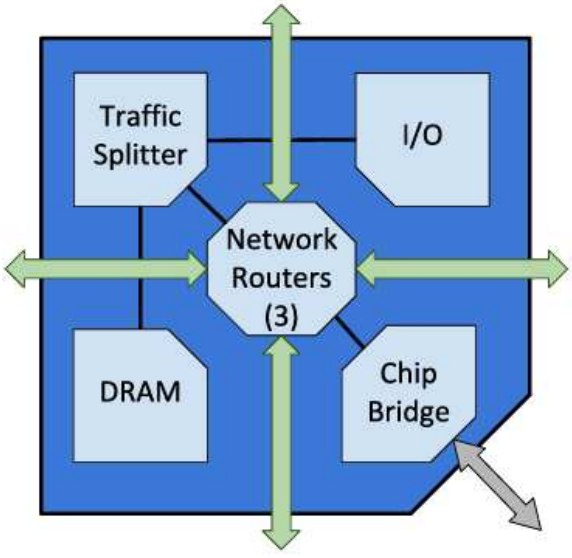

Figure 2.4: Architecture of a chipset

2.3.2 OpenPiton Configurability

One of the key features that has given OpenPiton significant interest is its flexibility and configurability, which can be tailored to suit various applications. This versatility allows customization at multiple levels. Core configurability enables users to scale up to four hardware threads, allowing for the adjustment of TLB size, activation of the FPU and SPU, and customization of thread counts. Cache configurability offers options to adjust the sizes of the L1 cache for both instructions and data. Additionally, users can configure the L1.5 and L2 caches in terms of sets and ways, providing a high degree of customization. All configuration options are summarized in table 2.1.

Table 2.1: 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

2.4 Benchmark Applications

During the fault injection simulations, we have used two well-known benchmarks:*Fibonacci series* and *Matrix Multiplication*. These benchmarks are written in assembly language and integrated into the OpenPiton Many-core processor simulation platform. The selection of the benchmarks is based on how widely these programs are used in research projects and how much processing power they consume.

2.4.1 Fibonacci Series Benchmark

Due to its high computational load, the Fibonacci benchmark was selected as the first case study . The Fibonacci sequence is a classic recursive algorithm where each term is dependent on the values of the preceding two terms. This recursive nature introduces a cumulative dependency, meaning that any small error or fault in intermediate computations, such as a fault in a register or memory cell, can propagate through the entire computation. As the sequence grows, the impact of any such fault becomes more pronounced, making the Fibonacci sequence particularly sensitive to errors[19]. Moreover, the Fibonacci benchmark represents a common workload in various applications requiring recursive computations. These include optimization problems, financial modeling, cryptography, and even certain algorithms in computer graphics. Given its relevance and sensitivity to faults, the Fibonacci series offers a valuable benchmark for evaluating how well a system, such as a multi-core processor, can handle fault injection while maintaining computational correctness. The pseudo-code for producing the n^{th} element of the Fibonacci series is illustrated in Algorithm 1.

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

$fib[0] = 0$

$fib[1] = 1$

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

$fib[i] = previous\_number + current\_number$

$previous\_number = current\_number$

$current\_number = fib[i]$

end for

2.4.2 Matrix Multiplication Benchmark

Matrix Multiplication (MxM) is a core operation in linear algebra and is foundational in various domains, including scientific computing, engineering, artificial intelligence, and machine learning. It is utilized to solve a wide range of computational problems, such as systems of linear equations, transformations in graphics processing, and neural network operations. Given its fundamental role, the correctness and efficiency of MxM are critical in safety-critical applications like aerospace, defense, cryptography, and medical imaging[20].

As a benchmark, MxM is particularly valuable due to its intensive memory and computational requirements. The algorithm demands significant memory access patterns and intensive data manipulation, making it an ideal test case for evaluating both processing power and memory bandwidth. Furthermore, MxM's sensitivity to faults makes it an excellent candidate for reliability testing. Any error in the intermediate calculations can propagate through the entire result, potentially affecting system behavior in real-world applications.

MxM's widespread use, coupled with its complexity and high resource demand, motivates its frequent selection as a benchmark in evaluating the reliability, fault tolerance, and performance of computing systems, particularly in multi-core and high-performance processors like OpenPiton [20].

The pseudo-code shown in Algorithm 2 performs an $n \times n$ matrix multiplication.

Algorithm 2 Pseudo-code to Calculate Matrix Multiplication of matrix X and Matrix Y yielding in matrix Z

```
for i = 0 → n − 1 do
    for j = 0 → n − 1 do
        Z[i][j] = 0
        for k = 0 → n − 1 do
            Z[i][j]+ = X[i][k] × Y[k][j]
        end for
    end for
end for
```

2.5 Soft Errors Mitigation Techniques

The significant shift of safety-critical applications, such as those in aerospace, towards the use of FPGAs and many-core processors—driven by their substantial performance and resource advantages—has introduced a major challenge: susceptibility to soft errors. Consequently, assessing their robustness against these errors and developing effective mitigation strategies has become a prominent area of research.

Extensive research has been conducted to mitigate these effects, which can be classified into hardware, software, and hybrid-based mitigation techniques.

Hardware-based mitigation techniques primarily involve duplicating or triplicating various circuit components and incorporating additional voters or checkers. While these methods enhance system robustness, they come with significant trade-offs, including increased overhead in terms of time, space, and power consumption.

On the other hand, Software-based mitigation techniques, also known as Software Implemented Hardware Fault Tolerance (SIHFT), involve duplicating instructions executed by the processor without requiring hardware modifications. This approach reduces costs,

space requirements, and power consumption overhead. However, it results in a significant increase in memory usage and execution time.

Thus, a hybrid approach is often more appealing, as it combines the advantages of both software and hardware techniques. This approach aims to leverage the strengths of each method, optimizing robustness while managing costs, space, and power consumption [21].

The primary objective of this work is to propose the application of Software Implemented Hardware Fault Tolerance (SIHFT) to multi-threaded many-core processors, effectively creating what can be termed Multi-Threaded TMR. This approach aims to exploit the vast often unused resources (i.e., cores) available in these processors for enhanced fault tolerance. There have been a lot of automatic mitigation tools like Trikaya [22] and COAST[23], however, they deal with bare-metal single-threaded codes only and they depend mainly on temporal and spatial redundancy.

This work presents a novel use for TMR on applications running on top of Linux OS with OpenPiton many-core processor, where we exploit the unused cores to implement multi-threaded TMR that is able to mitigate the errors up to 17% in comparison when TMR is not integrated within the OpenPiton. this will be discussed more in the upcoming chapters.

2.5.1 TMR

TMR can be implemented in either bare-metal or commercial operating systems (OS). Work presented in [24] has utilized bare-metal applications where no OS is used, allowing for direct control of the hardware. Without OS overhead, system performance benefits in terms of speed and resource utilization.

However, as system complexity increases, achieving a bare-metal implementation becomes more challenging. Managing all aspects of system hardware directly becomes more difficult, and higher complexity can lead to greater error susceptibility, making the system

less reliable.

Alternatively, operating systems are widely used and extensively evaluated by researchers and industry groups. As a result, OSs are expected to be nearly error-free. Consequently, much research, such as that in [25], has shifted towards OS-based applications. This work utilizes the Linux OS due to its robustness and the extensive research conducted on it, as demonstrated in studies like [26].

TMR can be classified as either single-threaded or multi-threaded. Single-threaded TMR, also known as time-based TMR, involves the sequential execution of the same task three times on a single or different cores. A majority voting mechanism is then used to determine the correct output. While this technique offers advantages such as low resource usage with only one active thread and reduced implementation complexity, it has drawbacks, including slower execution speed and increased susceptibility to errors.

In contrast, multi-threaded TMR, also known as concurrent TMR, involves the parallel execution of the same task across multiple threads. This method is relatively immune to errors, offering reduced execution time but at the cost of increased complexity and resource usage [27].

In conclusion, our proposal fully leverages the threading capabilities of modern microprocessors by distributing the execution of the same program across all available processing cores. This approach allows multiple instances of the program to run in parallel, without requiring communication between them, thus maximizing efficiency and resource utilization. The following chapter will discuss the implementation of multi-threaded TMR in detail.

Chapter 3

Multi-threaded TMR Implementation Methodology

In this chapter, we discuss the proposed multi-threaded TMR implementation on various benchmarks running on OpenPiton. We used the fault injection framework proposed by Dammak et al. [1] with some modifications to test the system's reliability before and after integrating TMR. A detailed flow of the proposed methodology is presented starting from running the golden model, followed by integrating TMR, then injecting faults, and subsequently, calculating the improvement percentage for each case study.

3.1 Multi-threaded TMR Implementation

In our study, we exploit the many-core processor's vast resources to build a TMR framework to enhance system reliability against SEUs. Our proposal takes advantage of the threading capabilities of modern microprocessors by distributing the execution of the same program across all available processing cores. This allows multiple instances of the program to run in parallel, with no communication between them, except for a small segment of code dedicated to stall and synchronization purposes.

TMR involves replicating the application three times and incorporating a simple majority voter to mask any errors occurring in one of the three replicas. Due to the configurable number of cores of OpenPiton from 1 to 500 million cores, TMR can easily be done without the substantial hardware and cost overhead, as a lot of these cores remain unused for the majority of applications.

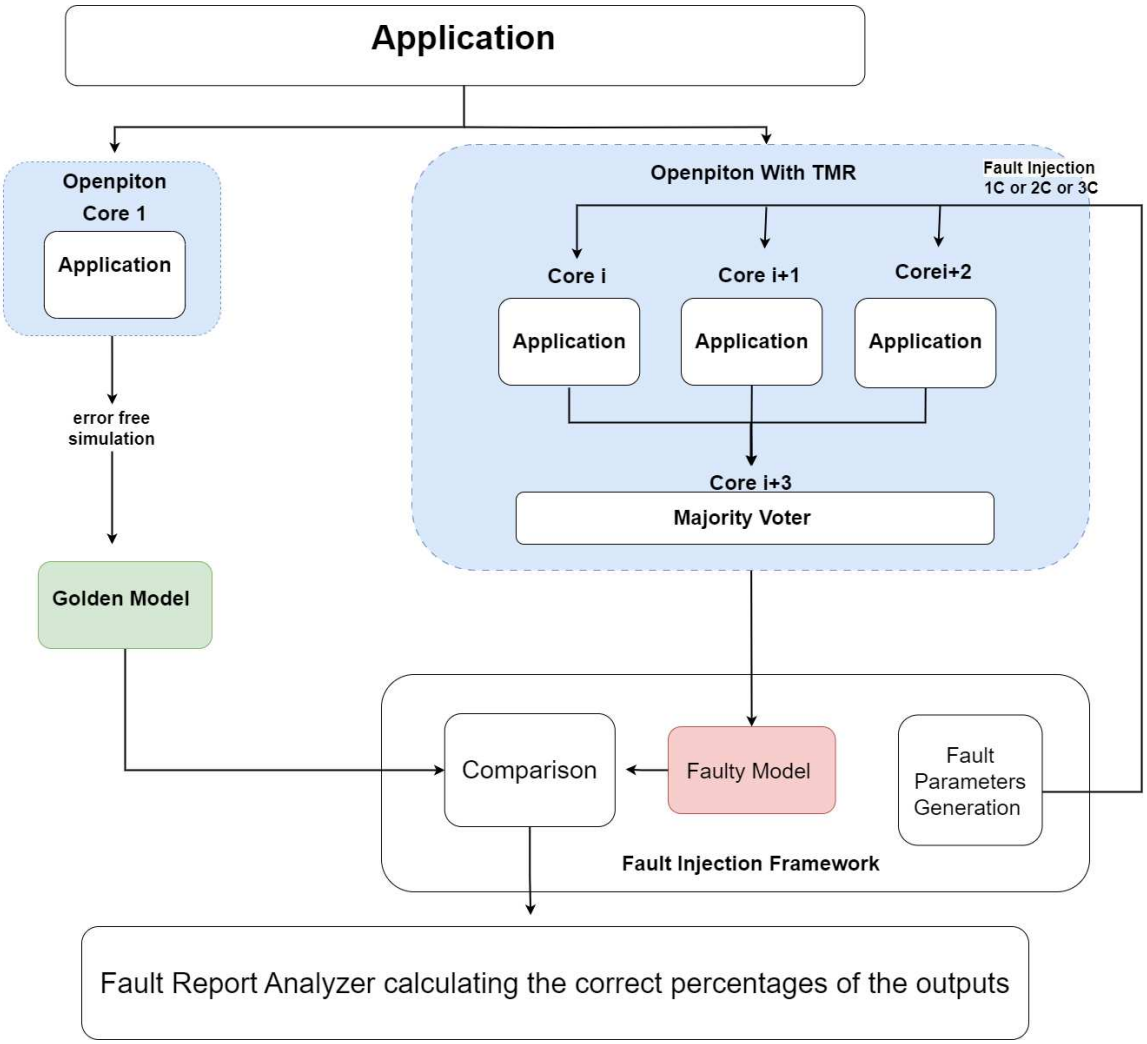

Figure 3.1: Summarized Methodology

Figure 3.1 provides a summary of the methodology discussed in this chapter. The first step involves running a golden model simulation, which executes the application without

any errors being injected. This provides a baseline for comparison with the outputs generated by the TMR framework. By comparing the error-free outputs from the Golden Model with those from the TMR system, we can assess the system's robustness by evaluating the correct percentages obtained and calculating the TMR mitigation percentages. In this golden run, only one core is instantiated with no fault injections.

Following this, TMR was implemented using OpenSPARC T1 assembly language. This implementation involves triplicating the same benchmark application code across three different cores and creating a majority voter in a fourth core, which is assumed to be error-free—a valid assumption in this context. Two key aspects of TMR to be discussed in detail are the independent execution of each application in a single thread on separate cores and the implementation of the majority voter.

Two key aspects related to TMR that will be explored in greater detail are:

- The independent execution of each application as a single thread on separate cores, ensures minimal interaction between cores.
- The majority voter Implementation.

These aspects are crucial for the successful implementation of TMR, as they play a pivotal role in enhancing fault tolerance and improving overall system reliability.

In this setup, four instances of OpenPiton tiles are instantiated within the chip. Three of these cores are designated to run identical copies of the target application concurrently. The fourth core is reserved for implementing a majority voter mechanism. This majority voter core operates by comparing the output values produced by the three application-running cores and selecting the majority result—i.e., the output that appears most frequently across the three cores. The majority voter algorithm functions as a fault tolerance measure, ensuring that transient faults or errors in one of the cores do not impact the final system output, as the most common value is assumed to be correct. The Majority voter Algorithm is shown below:

Listing 3.1: Majorty Voter Algorithm

```
function majority_voter(Core0_output, Core1_output, Core2_output):
    if (Core0_output == Core1_output) or
    (Core0_output == Core2_output):
        return Core0_output
    else if (Core1_output == Core2_output):
        return Core1_output
    else:
        return error # or some default value/error signal
```

Our ultimate goal is to make each core run each thread independently of the other cores and then allow the fourth core (the majority voter) to communicate with the other three cores and access the outputs of each of the three cores.

3.2 Communication between Cores

3.2.1 Memory Mapping

The proposed option was that we use shared memory rather than memory passing, meaning that if a core wants to send data to another one, it would write the data to an address that is shared by all other cores.

The OpenPiton configuration is done in a way to make every thread run independently on different cores where each core has a private L1 and L1.5 cache and a shared L2 cache that can be accessed by all cores. In each Thread, the benchmark application code is written and the output is stored in a different address in the shared global L2 cache memory. Thereafter, The majority voter loads these three outputs and compares them and the output will be the majority of the 3 core outputs.

A brief outline of the assembly code can be shown below:

Listing 3.2: Simplified Version of the assembly code

```
# start by using fork
th_fork(th_main)
th_main_0:
 # thread 0 run by core 0
 # Benchmark application code to be written here
   stx  $(output of the benchmark application)$,$location of L2 shared
        cache memory L0
th_main_1:
 # thread 1 run by core 1
 # Benchmark application code to be written here
   stx  $(output of the benchmark application)$,$location of L2 shared
        cache memory L1
th_main_2:
 # thread 2 run by core 2
 # Benchmark application code to be written here
   stx  $(output of the benchmark application)$,$location of L2 shared
        cache memory L2
th_main_3:
ldx L0, $any general purpose register g0
ldx L1, $any general purpose register g1
ldx L2, $any general purpose register g3
 # compare between the three outputs using Majority voter
   compare value 1 with value 2,
   if (g0 = g1) , output = g0
   elseif (g0=g2), output = g0
   elseif (g1=g3), output= g1
   else output =0 # all three are mismatched
```

th_fork macro is used to manage thread execution in OpenPiton. It begins by reading the thread ID through the *rdth_id* function, followed by comparing various thread masks until the correct target thread is identified. Once identified, the corresponding instructions

are executed via *fork_expand* The M4 code definition of *th_fork*can be shown in this code snippet.

Listing 3.3: th_fork M4 code

```
define(th_fork, `
    rdth_id
    fork_expand($1)
    nop
    ta T_BAD_TRAP
')dnl
```

One problem that we faced was **Race Condition**, when the majority voter reads the shared L2 Cache memory before one of the other 3 cores writes in it, as the three threads are being run simultaneously. Thus, we had to dig deep into OpenPiton architecture.

3.2.2 Race Condition

A race condition occurs in multi-threaded systems when one thread depends on the sequence or timing of instructions in another thread, leading to unpredictable system behavior. This issue is particularly critical in systems such as aerospace and medical applications.

In our research, to ensure proper synchronization we need to make sure that the majority voter reads data only after all three cores had completed writing. Moreover, we need to ensure that no memory loads are done during the cores storing the outputs in the memory. The first issue was addressed by mimicking Python's *Event.set()* and *Event.wait()* methods using memory-based synchronization techniques in OpenSPARC T1 assembly. *Event.set()* signals an event has occurred. When called, it sets an internal flag to True, meaning any thread waiting on this event will be unblocked and allowed to continue, while *Event.wait()* makes a thread wait until the event's internal flag is True. If the flag is already set, the thread continues immediately. If not, the thread is blocked until another thread calls event.set().

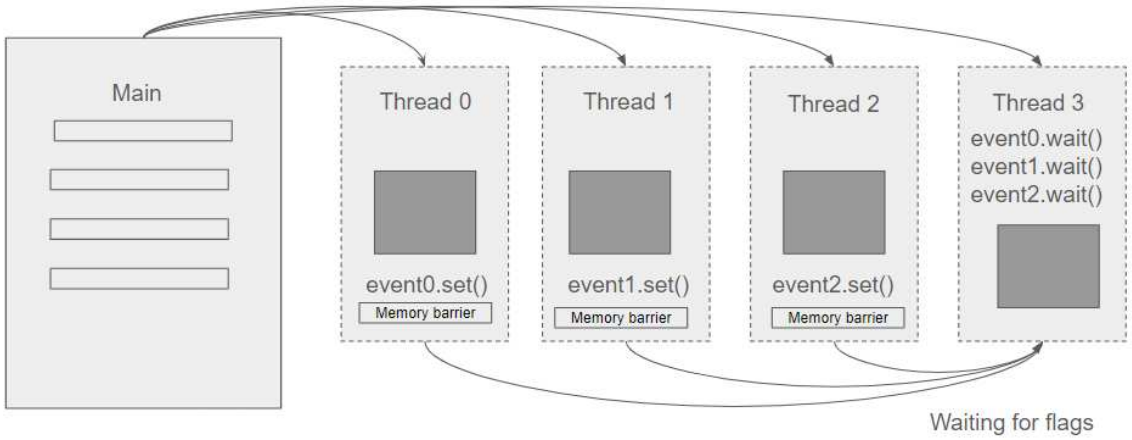

Figure 3.2: Proposed proper synchronization approach

The second issue was resolved by introducing memory barriers (*membar #StoreLoad*), ensuring that all previous stores were completed before any subsequent loads were initiated. The implemented proposed approach is shown in figure 3.2

To be able to implement Python's *Event.set()* in OpenSPARC T1 assembly, the assembly code used is as shown below :

Listing 3.4: Event.set() in OpensparcT1 assembly implementation

```
!defining global lock flags
.global ready_flag_0
 ready_flag_0:
 .word 0,0
thread_0:
   ! After the benchmark finishes its task
    set ready_flag_0, %g3
    add %g0, 0x1, %g4
    !setting the flag to 1
     stx %g4,[%g3]
     ! using memory barrier ensuring no further loads are done until
        the store is finished first
      membar #StoreLoad
```

The Python's *Event.wait()* can be implemented in OpenSPARC T1 assembly as shown below:

Listing 3.5: Event.wait() in OpensparcT1 assembly implementation

```
!in the Majority Voter thread
    thread_3:
    wait_for_ready_flags:
   ! Load ready flags and check
   set ready_flag_0, %o1
   ldx [%o1], %o2
   cmp %o2, 1
   bne wait_for_ready_flags    ! Wait if thread 0 is not done
   nop
```

In a nutshell, We addressed these race conditions by employing synchronization primitives along with memory barriers. To synchronize threads, where the first three threads store values in memory and the fourth thread reads these values, we ensured proper coordination between write and read operations. A "ready flag" was used to signal when writing was complete, allowing the reading thread to access the data safely afterward. Each thread is assigned a specific flag variable to indicate the status of its operations. Upon completing the data-writing operation, a thread sets its respective flag to a predefined value (e.g., 1), indicating that the writing process has finished. Following that, the majority voter continuously checks the status of these flags to determine whether the data-writing operation has been completed by all relevant threads before proceeding with reading the data.

The following code explains the handling of the race condition.

Listing 3.6: Race condition Handling pseudo code

```
!defining global lock flags
.global ready_flag_0
 ready_flag_0:
 .word 0,0
```

```
.global ready_flag_1
ready_flag_1:
.word 0,0
.global ready_flag_2
ready_flag_2:
.word 0,0

thread_0:
  ! After the benchmark finishes its task
   set ready_flag_0, %g3
   add %g0, 0x1, %g4
   !setting the flag to 1
    stx %g4,[%g3]
    ! using memory barrier ensuring no further loads are done until
       the store is finished first
     membar #StoreLoad

   thread_1:
  ! After the benchmark finishes its task
   set ready_flag_1, %g3
   add %g0, 0x1, %g4
   !setting the flag to 1
    stx %g4,[%g3]
    ! using memory barrier ensuring no further loads are done until
       the store is finished first
    membar #StoreLoad

    thread_2:
  ! After the benchmark finishes its task
   set ready_flag_2, %g3
   add %g0, 0x1, %g4
   !setting the flag to 1
```

```
 stx %g4,[%g3]
 ! using memory barrier ensuring no further loads are done until
    the store is finished first
 membar #StoreLoad

 !in the Majority Voter thread
 thread_3:
 wait_for_ready_flags:
! Load ready flags and check
set ready_flag_0, %o1
ldx [%o1], %o2
set ready_flag_1, %o3
ldx [%o3], %o4
set ready_flag_2, %o5
ldx [%o5], %o6

cmp %o2, 1
bne wait_for_ready_flags    ! Wait if thread 0 is not done
nop
cmp %o4, 1
bne wait_for_ready_flags    ! Wait if thread 2 is not done
nop
cmp %o6, 1
bne wait_for_ready_flags    ! Wait if thread 4 is not done
nop
```

3.3 Fault Injection Framework

In the Fault Injection Framework, we have used the framework mentioned here [1]. Multiple modifications have been added to the used framework such as the fault parameters

generation, and the yielded results. Figure 3.3 summarizes the fault injection framework used.

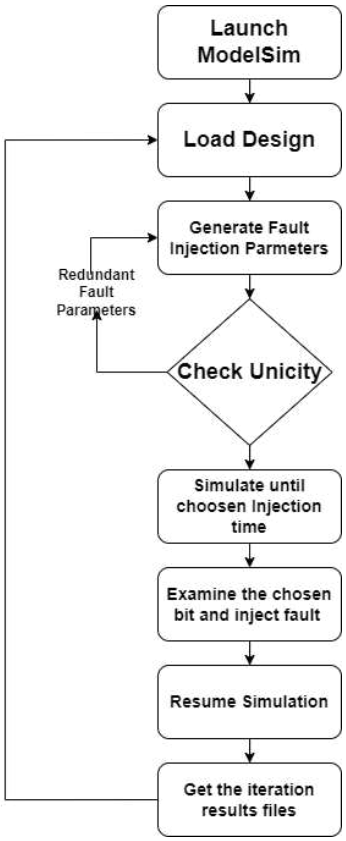

Figure 3.3: Summarized Fault injector Framework [1]

The fault injection framework is implemented in ModelSim, where OpenPiton is loaded with the benchmark being used. Initially, various fault parameters such as injection time, fault location, and core selection are determined. If the selected combination of fault injection parameters has been used previously, it is discarded, and a new, unused combination is chosen. The simulation then begins, running until the specified injection time, at which point faults are injected using the force deposit feature. This involves identifying the randomly chosen bit and flipping it. The simulation then continues until completion. In the end, critical data required for calculating the results are stored securely.

our work focuses on injecting the faults randomly either into the actual data of the

general-purpose registers (RD) or into the Error Correction Code (ECC) associated with these registers (RECC). The force *deposit* feature is utilized to induce transient bit flips, allowing for subsequent alterations to the affected bits. Fault injection for each case is done for 1000 iterations.

In all of our test cases, four cores of OpenPiton are instantiated, Thereafter, fault injection is done into one core (1C), two cores (2C), and a random combination of three cores (3C Random). more details on each test case type can be found in section 4.1 which discusses environment setup.

3.4 Result Generation

The Python script begins by uploading the saved files from the simulations, with each iteration corresponding to a specific register file for each core. Critical data, including injection time, fault location, core selection, and benchmark outputs, are then extracted. A comparison is made between the benchmark outputs and the golden model (error-free) outputs for all 1,000 iterations, resulting in the calculation of correct and false percentages. Our system reliability metric, termed "correct," represents the number of iterations where the majority voter outputs match the error-free outputs, as defined by the following equation.

$$Correct = (\frac{K}{total\ no\ of\ iterations}) * 100 \tag{1}$$

where K= no of iterations where the majority voter outputs match with the error-free outputs

The Python script also compiles all the extracted information into an Excel sheet. An example of Python script used can be found in B

The table 3.1 shows a sample of the Excel sheet containing detailed info about each

Table 3.1: Experimental Results of MxM RD Experiment with TMR

iteration_no	tile	reg	pos	val	time	t0_out	MM_result_1	MM_result_2	category
0	0	25	13	1	215	1	0xh1d	0xh0a	CORRECT
1	2	15	11	1	149	1	0xh01d	0xh00a	CORRECT
2	0	23	62	1	151	1	0xh01d	0xh00a	CORRECT
3	2	17	15	1	110	1	0xh01d	0xh00a	CORRECT
..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..
996	1	18	18	1	265	1	0xh01d	0xh00a	CORRECT
997	0	16	55	1	235	1	0xh01d	0xh00a	CORRECT
998	2	05	34	1	38	1	0xh01d	0xh00a	CORRECT
999	1	21	09	1	104	1	0xh01d	0xh0a	CORRECT

iteration of the experiment:

- **tile** (core) shows the core where the faults are injected.
- **reg** shows the exact register where the faults are injected.
- **pos** shows the position where the bit is being flipped in the target register.
- **val** shows the bit value after being flipped due to fault injection.
- **t0_out** indicates the comparison output between the benchmark outputs and the golden (errorfree) model, 1 if they match, 0 otherwise.
- **MM_result_1 MM_result_2** shows the benchmark outputs.
- **category** is the same as t0_out (right=1),(wrong=0).

We then compare the correct percentages obtained with TMR to those achieved without TMR implementation, generating the TMR mitigation percentages.

3.5 Summary

To provide a detailed summary of this chapter, the framework development process begins with the creation of a golden model, which serves as an error-free baseline. This

model runs the benchmark application without any faults, establishing a reference for subsequent comparisons. After generating this model, the benchmark application is replicated and deployed across three separate cores in the OpenPiton platform. Each core executes its own independent instance of the application, ensuring no inter-core interaction during runtime. This isolation allows for individual fault injection analysis across the cores.

In addition to the three application-running cores, a fourth core is instantiated to perform majority voting. The majority voter algorithm compares the outputs from the three cores and selects the most frequently occurring output, producing a TMR-masked result. This configuration allows us to effectively mitigate faults by leveraging the redundancy provided by TMR.

Several issues were encountered throughout the implementation, such as race conditions. We addressed these issues by applying specific synchronization techniques to prevent incorrect behaviors in multi-core execution. These solutions were critical in ensuring the framework operated reliably under fault injection conditions.

We also provided an in-depth discussion of the fault injection framework, covering the key parameters used, such as fault timing, location, type, and the technical specifications that shaped the injection scenarios.

Finally, the chapter detailed how the results from the TMR implementation were processed through a fault report analyzer. By comparing the outputs of the TMR system with those from the golden model, we were able to calculate the ”correct” output percentages. This comparison enabled us to determine the TMR mitigation percentages, offering insights into the system’s robustness and fault tolerance. The results demonstrated how effectively the TMR framework enhanced the system’s reliability under various fault scenarios.

Chapter 4

Experimental Setup and Results to Evaluate the Impact of Implementing TMR in OpenPiton Many-core on the System Reliability against SEUs

In this chapter, we discuss the case studies simulated on the OpenPiton processor with and without the TMR implementation. The tests aim to show the effect of implementing the TMR in increasing the reliability of the many-core processor OpenPiton against radiation effects. The chapter starts by giving detailed information about the experimental setup, a summary of the benchmark applications used, and the yielded results from these various experiments.

4.1 Environment Setup

The fault injection experiments were run on a Linux server with 160 CPUs clocked at 2.4GHz and 1TB RAM. OpenPiton is operating in Asymmetric Multiprocessing mode

(AMP) mode during all these tests where each running core is operating independently from the other. As mentioned before, We used the injection fault framework mentioned here [1].

TMR is integrated in OpenPiton in such a way that the benchmark application is triplicated in three different cores and a majority voter is implemented in a fourth core that we assume is error-free (which is a valid assumption). Two main aspects that we need to consider: the independent run of the three first cores and the majority voter implementation in a way to ensure correct synchronization between cores avoid=ing any race conditions happening.

The results of the TMR effect on the different benchmarks are classified into two groups according to the fault location which can be either in a general-purpose register actual data or the Error Correction Code (ECC) of a general-purpose register. The force command type used is *deposit* to mimic the soft error effect, this feature causes an instantaneous bit flip and any external effects can cause any changes to that bit, thus mimicking soft errors caused by radiation.

The objective of all experiments is to evaluate the impact of integrating TMR to OpenPiton while simulating different Benchmarks on system reliability and robustness against SEUs Faults are being injected firstly only in one core (1C), then two cores (2C), then a random combination of three cores (3C Random). Our experiments were simulated for 1000 iterations for each different case. The first step is to simulate a simple error-free model to get the error-free result to which we later compare our results, it is a simple 1-core simulation on OpenPition.

In all experiments, The OpenPiton processor instantiates 4 cores and is set to the default configuration of one thread per core. The only changes from one experiment to another are the number of cores that the faults are injected into(1C, 2C, 3C Random)

In 1C, a fault is injected only in one core. In 2C fault is injected in two cores. In 3C

Random, a binary combination is generated from 0 to 7, where when a core is assigned 1, a fault injection is injected randomly into it, and if it is assigned 0, no fault is injected into it. For example, if the binary combination is 5 which translates to (101) this means both core 0 and core 2 will have faults injected into them, while core 1 remains error-free. All combinations are shown in 4.1.

Table 4.1: Binary Combination of 3C random fault injection

Binary combination	C2	C1	C0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

3C fault injection script can be found in A. The yielded results are achieved from 1000 iterations in each case study. The correct percentages achieved explain how many iterations where the yielded result matched the golden model result out of these 1000 iterations. The yielded results showed promising percentages of how TMR can increase the system's robustness.

4.1.1 Running Simulation With OpenPiton

To be able to run a simulation with the OpenPiton processor, two steps need to be done: Building a simulation model followed by running the application(test) on a simulation model A simulation Model consists of a set of design under test (DUT) Verilog files, a set of top-level Verilog files that create a testbench environment, a list of Verilog file lists (Flists) that specify the DUT Verilog files as well as the top-level testbench Verilog files,

and a list of Verilog simulator (e.g. Mentor Modelsim) command line arguments. OpenPiton has two different simulation models: the assembly/c test simulation model known as (The Manycore Simulation Model) and The Unit Test Simulation Model [28]. The simulation type used in our work is the manycore simulation model which can be configurable. Both the number of tiles and all cache sizes are configurable. The number of tiles can be configured depending on the 2D mesh that one targets in the design, it can be done as shown:

-x_tiles=X_TILES -y_tiles=Y_TILES

All caches can be configurable such as L1 data and instruction caches, L1.5 cache, and L2 cache as mentioned before in 2.3.1. This can be done as follows:

-config l1i size and -config l1i associativity

-config l1d size and -config l1d associativity

-config l15 size and -config l15 associativity

-config l2 size and -config l2 associativity

In our case study, we set them to default.

A simulation model is built by using the **sims** tool, in addition to the **msm** build argument that tells the manycore simulation model to build the simulation using Mentor ModelSim. building the simulation can be done like this:

sims -sys=manycore -x_tiles=X_TILES -y_tiles=Y_TILES -msm_build -build_id=NAME

Following the building of the simulation is running the test on the simulation model. It can be done as follows :

sims -sys=manycore -x_tiles=X_TILES -y_tiles=Y_TILES -msm_run <assembly_test_file>

Since our tests use threads, mapping between the software and hardware threads is an essential thing. In OpenPiton, each core has two hardware threads, thus each core can handle up to two threads. Software threads can be done as follows :

-midas_args=-DTHREAD_COUNT=thread_count

Where thread_count is the number of threads that will handle your program. Thread mapping, by default, initiates with the first core and continues sequentially in an incremental order. The stride number specifies how many hardware thread units are skipped between consecutive threads. By default, it is set to 1, meaning no threads are skipped. For example, if there are 4 threads and the stride number is set to 2, those 4 threads will be mapped to the first hardware thread unit of the first core, the first hardware thread unit of the second core, the first hardware thread unit of the third core, and the first hardware thread unit of the fourth core as shown in the figure 4.1

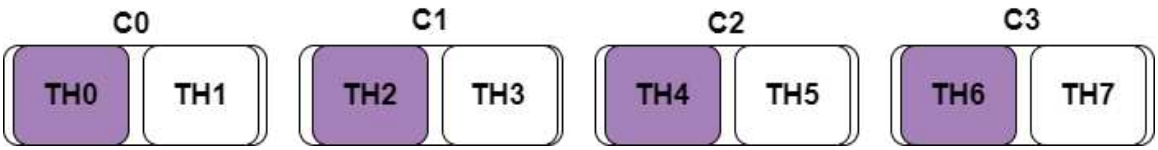

Figure 4.1: Thread Mapping

It can be added in the run command as follows :

-midas_args=-DTHREAD_STRIDE=thread_stride

Where thread_stride is the stride number that the user can configure. following these two arguments, another last argument must be set which is

-finish_mask=mask_vector

This argument is used to indicate which hardware thread units should signal a successful completion for the test to be deemed passing. The mask_vector is specified as a bit vector in hexadecimal format.

Finally, the whole run command for simulation can be something like this :

sims -sys=manycore -x_tiles=4 -y_tiles=1 -msm_run FS.s build_id=fs_build

-finish_mask=1111 -midas_args=-DTHREAD_COUNT=4

-midas_args=-DTHREAD_STRIDE=2

To enable the automation of the framework through additional scripts, we utilize the transcript generated from the simulation run. We then modify the ***vsim*** command line by appending the argument ***-do "script_name"***.

4.2 Experiment 1: TMR Implementation on 1-core, 2-core and 3-core system Implementing Fibonacci Series Benchmark

In this experiment, we studied the effect of integrating TMR to OpenPiton on system robustness against SEU when the Fibonacci Series Benchmark was used. As explained in 4.1, 1C,2C, and 3C Random experiments were done using the Fibonacci series Benchmark.

In our benchmark application, we set the Fibonacci series to get the 10 term of the Fibonacci series expecting the last term to be 34 in decimal or $1a$ in hexadecimal. For every case, reliability was evaluated across 1000 iterations, and correct percentages were estimated by matching the majority voter results with error-free outputs from the golden-free model.

The assembly code that is used to implement the Fibonacci series is as follows:

Listing 4.1: the Fibonacci series assembly code

```
1:   set fb_result_1, %g6
2:         ! build address of the L2 Control Reg
3:
4:     mov 0, %l0          ! Initialize term 0
5:     mov 1, %l1          ! Initialize term 1
```

```
 6:      mov 8, %o0          ! Set the number of terms to generate
 7: loop0:
 8:          cmp %o0,0
 9:          ble endloop0
10:          nop
11:      add %l0, %l1, %l2   ! term2 = term0 + term1
12:      mov %l1, %l0        ! term0 = term1
13:      mov %l2, %l1        ! term1 = term2
14:
15:    stx    %l1,[%g6+0]
16:
17:    sub %o0 ,1, %o0
18:          ba loop0
19:          nop
20: endloop0:
21:          nop
22: nop
23:      ta       0
```

In the first section of the experiment, fault injection is done randomly in the actual data of a general-purpose register (RD)

4.2.1 Experimental Results of Fibonacci Experiment without TMR

In this section, a framework similar to the one described in 3.3 is executed **without integrating TMR** into the system. Only a single core is instantiated in OpenPiton, and fault injection is randomly applied to one bit in one register. The reliability metric ”correct” was found to be 82.6%, indicating that out of the 1,000 iterations performed, 826 iterations produced correct results.

The following table 4.2 indicates some of this experiment’s output samples:

Table 4.2: Experimental Results of Fibonacci Experiment without TMR

iteration_no	tile	reg	pos	val	time	t0_out	fb_result	category
0	0	17	09	1	15	0	0xh18385000	WRONG
1	0	20	61	1	7	1	0xh0022	CORRECT
2	0	18	34	1	16	1	0xh0022	CORRECT
3	0	10	52	1	21	0	0xh00	WRONG
..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..
996	0	24	40	1	16	1	0xh0022	CORRECT
997	0	11	36	1	47	1	0xh0022	CORRECT
998	0	20	54	1	38	1	0xh0022	CORRECT
999	0	21	16	1	19	0	0xh10022	WRONG

4.2.2 Experimental Results of Fibonacci RD Experiment with TMR

For the 1C scenario, where faults are injected into only one core out of the three instantiated OpenPiton cores after TMR implementation

The experimental results showed improvement in the reliability of the system against the injected fault framework, where **correct** metric turned out to be 96.1%, meaning out of 1000 iterations, 961 iterations where the outputs were correct.

Thus **The TMR Mitigation Effect** was calculated to be 13.5%. While this implementation does introduce additional time and power overhead, the mitigation benefits outweigh these costs. Importantly, no area overhead is incurred, as the approach leverages the otherwise unused resources available in multi-core processors. Table 4.7 shows a sample of Experiment 1 output.

In the 2C scenario, faults are injected into two of the three instantiated OpenPiton cores after TMR implementation. Given the increased severity of this fault injection, a decrease in the **correct** metric is anticipated. However, in our case study, the decrease was minimal (0.4%), demonstrating that TMR continues to provide robust immunity to the system. The overall **TMR mitigations effect** was calculated to be 13.1%. The table 4.4 shows Experiment 2C sample outputs.

In the 3C scenario, as detailed in Table 4.1, the experiment involves injecting faults into

Table 4.3: Experimental Results of 1C Fibonacci Experiment without TMR

iteration_no	tile	reg	pos	val	time	t0_out	tmr_result	category
0	0	21	13	1	140	1	0xh0022	CORRECT
1	1	27	17	1	53	1	0xh0022	CORRECT
2	0	08	45	1	72	1	0xh0022	CORRECT
3	2	17	49	1	10	0	0xh4488800	WRONG
..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..
996	1	29	42	1	47	1	0xh0022	CORRECT
997	0	18	47	1	183	1	0xh0022	CORRECT
998	2	07	26	1	112	1	0xh0022	CORRECT
999	0	09	63	1	52	1	0xh0022	CORRECT

Table 4.4: Experimental Results of 2C Fibonacci Experiment with TMR

iteration_no	tile_1	tile_2	reg_1	reg_2	pos_1	pos_2	val_1	val_2	time	tile0_out	tmr_result	category
0	1	2	19	12	57	33	1	1	58	1	0xh22	CORRECT
1	2	0	21	12	57	38	1	1	286	1	0xh22	CORRECT
2	0	2	27	1	27	6	1	1	52	1	0xh22	CORRECT
3	2	1	3	31	54	52	1	1	66	1	0xh22	CORRECT
..	..	..	..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..	..	..	..
996	0	2	5	30	57	37	1	1	312	1	0xh22	CORRECT
997	1	2	5	19	51	21	1	1	157	1	0xh22	CORRECT
998	0	1	5	1	48	62	1	1	183	1	0xh22	CORRECT
999	0	2	30	12	34	47	1	1	27	0	0x00	FALSE

one, two, or all three of the instantiated cores. This setup is designed to simulate the overall impact of faults on the system with TMR integration. Although it might be presumed that injecting faults into all three cores would inevitably lead to erroneous outputs, this is not necessarily the case. The actual impact depends significantly on the timing and location of the fault injection. If faults are introduced into non-critical registers not utilized by the program, or if the faults are injected before the system writes to the relevant registers, the output may remain unaffected.

This experiment also yielded favorable results, the **correct** metric achieved a value of 95.1%, indicating that out of 1000 iterations, 951 iterations produced correct outputs. Consequently, the overall **TMR mitigation effect** was determined to be 12.5%, underscoring the enhanced robustness of the system against faults achieved through the implementation

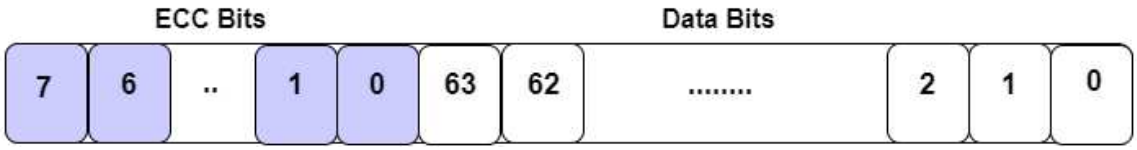

Figure 4.2: OpenPiton Data Register

of TMR.

The table 4.5 shows Experiment 3C sample outputs.

Table 4.5: Experimental Results of 2C Fibonacci Experiment with TMR

iteration_no	tile_combination	reg_1	reg_2	reg_3	pos_1	pos_2	pos_3	val_1	val_2	val_3	time	tile0_out	tmr_result	category
0	0	16	19	25	55	12	50	x	x	x	222	1	0xh022	CORRECT
1	0	15	22	20	47	14	22	x	x	x	86	1	0xh022	CORRECT
2	10	30	21	2	61	21	58	1	x	x	105	1	0xh022	CORRECT
3	101	20	17	25	23	15	38	1	x	1	69	1	0xh022	CORRECT
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
996	0	4	2	11	37	32	43	x	x	x	189	1	0xh022	CORRECT
997	101	16	24	6	43	33	54	1	x	1	249	1	0xh022	CORRECT
998	11	4	2	13	62	49	43	1	1	x	93	1	0xh022	CORRECT
999	110	7	28	10	47	41	14	1	1	x	321	1	0xh022	CORRECT

4.2.3 Experimental Results of Fibonacci ECC Experiment without TMR

In the second section of the experiment, fault injection is done randomly in the Error Correction Code (ECC) of a general-purpose register. ECC is a mitigation technique employed to detect and correct data corruption. However, the inclusion of ECC can impact the system's performance, particularly when subjected to a radiation environment. In this segment of the experiment, bit-flips are introduced into the ECC of general-purpose registers.

The OpenPiton data packet includes the last 8 bits designated for Error Correction Code (ECC), as highlighted in blue in 4.2. Therefore, the fault injection will target these ECC bits to assess the impact of errors within this specific section of the data packet.

Similarly to 4.2.1, a fault injection framework was applied to a single core instantiated in OpenPiton, with the key difference being that faults were injected into the ECC bits. As a result, an increase in the **correct** percentages was anticipated due to the restriction of fault

injections to specific positions inside of the registers. The **correct** was found to be 88.2%, indicating that out of 1,000 iterations, 882 produced correct outputs.

The following table 4.6 indicates some of this experiment's output samples. Note that **pos** column has values equal to or greater than 64.

Table 4.6: Experimental Results of Fibonacci Series ECC Experiment without TMR

iteration_no	tile	reg	pos	val	time	t0_out	fb_result	category
0	0	31	64	1	38	1	0xh0022	CORRECT
1	0	09	66	1	8	0	0xh18385000	WRONG
2	0	29	69	1	44	1	0xh0022	CORRECT
3	0	07	71	1	43	1	0xh0022	CORRECT
..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..
996	0	31	64	1	27	1	0xh0022	CORRECT
997	0	06	70	0	17	1	0xh0022	CORRECT
998	0	22	69	1	43	1	0xh0022	CORRECT
999	0	08	65	1	21	1	0xh0022	CORRECT

4.2.4 Experimental Results of Fibonacci ECC Experiment with TMR

For the 1C scenario, similarly to 4.2.2 faults are injected into the ECC code of the data packet of only one core out of the three instantiated OpenPiton cores after TMR implementation. The yielded results showed improvement in system robustness against faults, the **correct** metric turned out to be 97.7%, meaning out of 1000 iterations, 977 iterations yielded correct results. Thus **The TMR Mitigation Effect** was calculated to be 9.5%. We can observe that the TMR Mitigation Effect is lower when faults are injected into the ECC than when injected into data because ECC's inherent complexity and its focus on detection over correction make it more challenging for TMR to effectively mitigate errors. This contrasts with data, where TMR directly corrects errors, leading to a higher mitigation effect.

In the 2C scenario, faults are injected into the ECC bits of two out of the three instantiated OpenPiton cores following the implementation of TMR, similar to the Fibonacci RD experiment discussed in 4.2.2. As expected, a decrease in the **correct** metric was observed,

Table 4.7: Experimental Results of 1C Fibonacci ECC Experiment with TMR

iteration_no	tile	reg	pos	val	time	t0_out	tmr_result	category
0	0	15	70	1	216	1	0xh0022	CORRECT
1	2	02	68	0	67	1	0xh0022	CORRECT
2	1	21	70	1	87	1	0xh0022	CORRECT
3	1	11	67	1	32	0	0xh00	WRONG
..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..
996	1	11	67	0	128	1	0xh0022	CORRECT
997	2	31	68	1	169	1	0xh0022	CORRECT
998	1	10	67	1	166	1	0xh0022	CORRECT
999	2	17	71	1	157	1	0xh0022	CORRECT

resulting in a value of 96.1%. Consequently, the overall **TMR Mitigation Effect** was calculated to be 7.9%. This value is lower than that observed in the Fibonacci RD experiment, for the reasons previously mentioned.

The table 4.8 shows Experiment 2C sample outputs. Note that the values of both **pos_1** and **pos_2** are equal or greater than 64.

Table 4.8: Experimental Results of 2C ECC Fibonacci Experiment with TMR

iteration_no	tile_1	tile_2	reg_1	reg_2	pos_1	pos_2	val_1	val_2	time	tile0_out	tmr_result	category
0	2	1	23	13	64	66	0	0	207	1	0xh22	CORRECT
1	1	0	13	11	69	70	0	1	112	1	0xh22	CORRECT
2	0	2	31	03	66	70	1	1	256	1	0xh22	CORRECT
3	2	1	11	12	65	71	1	1	161	1	0xh22	CORRECT
..	..	..	..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..	..	..	..
996	2	1	29	12	70	68	1	1	136	1	0xh22	CORRECT
997	0	2	15	18	66	67	1	1	82	1	0xh22	CORRECT
998	2	0	12	24	64	70	1	1	18	0	0x00	FALSE
999	2	1	11	28	68	65	1	1	93	1	0xh22	CORRECT

In the 3C scenario, similarly to 4.2.2 , but here the experiment involves injecting faults into the **ECC bits** of one, two, or all three of the instantiated cores. This experiment yielded favorable results, demonstrating percentages that were higher than those observed in the 2C scenario and lower than those in the 1C scenario. An increase in the **correct** metric is found relative to that achieved from the 2C FS ECC Experiment, achieving 97.5%. the overall

TMR mitigation effect turned out to be 9.3%.

The table 4.9 shows Experiment 3C sample outputs. Note that the values of both **pos_1**,**pos_2** and **pos_3** are equal or greater than 64.

Table 4.9: Experimental Results of 3C Fibonacci Experiment with TMR

iteration_no	tile_combination	reg_1	reg_2	reg_3	pos_1	pos_2	pos_3	val_1	val_2	val_3	time	tile0_out	tmr_result	category
0	110	26	25	16	67	66	70	1	1	x	202	1	0xh022	CORRECT
1	100	23	03	05	71	67	64	0	x	x	126	1	0xh022	CORRECT
2	10	03	12	19	70	64	66	1	x	x	176	1	0xh022	CORRECT
3	110	02	06	09	67	64	68	1	0	x	69	1	0xh022	CORRECT
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
996	0	07	01	11	70	64	69	x	x	x	16	1	0xh022	CORRECT
997	101	15	23	31	64	70	70	1	x	1	236	1	0xh022	CORRECT
998	11	08	13	22	68	70	69	1	1	x	164	1	0xh022	CORRECT
999	10	18	16	19	65	64	67	0	x	x	98	1	0xh022	CORRECT

To assess the overhead introduced by TMR, we measured the execution time of the FS benchmark. Before implementing TMR, the test duration was 17 seconds, while after integrating TMR, it increased to 19 seconds. Consequently, the time overhead is calculated to be 11%. This is remarkable, as the time overhead is minimal, and there is no additional area overhead due to the effective utilization of the vast resources (i.e., unused threads) available in multi-core processors.

4.3 Experiment 2: TMR Implementation on (1C, 2C, or 3C random) fault injection in OpenPiton Implementing Matrix Multiplication Benchmark

In this experiment, we studied the effect of integrating TMR to OpenPiton on system robustness against SEU when the Matrix Multiplication Benchmark was used. As explained in 4.1, 1C, 2C, and 3C Random experiments were done using the Matrix Multiplication Benchmark.

We have implemented a matrix multiplication between matrix A 2×3 and matrix B 3×1 resulting in a matrix C 3×1.

Matrix A (2×3)is as shown :

$$\begin{bmatrix} 1 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

Matrix B (3×1) is as shown:

$$\begin{bmatrix} 5 & 1 & 2 \end{bmatrix}$$

The order of the matrices can be modified through the variables *n,m,p* where **Matrix A** order is given as *(n*m)* and **Matrix B**order is given as *(m*p)* resulting in **Matrix C** with order *(n*p)*.

The assembly code that is used to implement Matrix Multiplication is as follows:

Listing 4.2: Matrix Multiplication assembly code

```
!  A (2*3) we want to define n =2n*m
! B (3*1) we want to deine p=1 m*p
!and we will define m =3    c should be n*p  (2*1)
#define A_no_of_columns 3
#define B_no_of_columns 1
#define c_no_of_columns 1
#define n 2
#define p 1
#define m 3
/*******************************************************/
#include "boot.s"
#include "piton_def.h"

.global main
main:
```

```
    set matrix_A, %g1              ! Loop starting address
    set matrix_B, %g2              ! Loop starting address
    set result_matrix, %g3         ! Loop starting address
    add %g0, 0x0, %i0                ! Loop counter_a_n

loop_n:

     cmp %i0,n
         be done_loop_n
         nop
         add %g0, 0x0, %i1                ! Loop counter_b
      add %g0, %g0, %l6                 ! Sum variable

loop_p:
        cmp %i1, p
        be  done_loop_p
        nop
   add %g0, 0x0, %i2                 ! Loop counter_a_b_m -k counter

loop_m:

       cmp %i2,m
        be  done_loop_m
        nop

        !load A ij    i0 and i1
      mulx %i0 ,A_no_of_columns, %i3 !(i*no of columns*8)
         mulx %i3,0x8,%l0
      mulx %i2,0x8,%l1   !(j*8)
         add %l0,%l1,%l1
      add %g1, %l1, %l1
```

```
        ldx [%l1], %l3 ! we here have A


       !load B ij    i0 and i1
        !getting the address of B
        mulx %i2 ,B_no_of_columns, %i3
        sll %i3, 0x3, %l0
        sll %i1, 0x3, %l1
        add %l0,%l1,%l1
        add %g2, %l1, %l1
        ldx [%l1], %l4 ! we here have  B
        mulx %l4,%l3,%l5
        add %l6,%l5,%l6

        !storing  Cij
        !getting the address of c
        mulx %i0 ,c_no_of_columns, %i3
        sll %i3, 0x3, %l0
        sll %i1, 0x3, %l1
        add %l0,%l1,%l1
        add %g3, %l1, %l1
        stx %l6,[%l1+0]
        add %i2,0x1,%i2
        ba loop_m
        nop


done_loop_m:
        add %i1,0x1,%i1
                ba loop_p
                nop
```

```
done_loop_p:
       add %i0,0x1,%i0
        ba loop_n
                nop

done_loop_n:
      add %g3, 0x0, %l1
          ldx [%l1], %l5 ! we here have  c1
      add %g3, 0x8, %l1
          ldx [%l1], %l6 ! we here have  c2
          ta T_GOOD_TRAP

          nop
          nop
!==========================
.data


!m1 (2*3)

! 1,1,2
! 3,4,5
.global matrix_A
matrix_A:
!first row 1,1,2 (1*3)
 .word 0x0,0x1
 .word 0x00000000, 0x00000001
 .word 0x00000000, 0x00000002
 .word 0x00000000, 0x00000003
 .word 0x00000000, 0x00000004
 .word 0x00000000, 0x00000005
```

```
.global matrix_B
matrix_B: !(3*1)
!first row
 .word 0x0,0x5
 .word 0x0,0x1
 .word 0x0, 0x02
.global result_matrix
result_matrix:
    .skip n*p*8 ! Allocate memory for 2x3 elements (2*1*8 bytes)
.end
```

For every case, reliability was evaluated across 1000 iterations, and correct percentages were estimated by matching the majority voter results with error-free outputs from the golden-free model.

In the first section of the experiment, fault injection is done randomly in the actual data of a general-purpose register (RD)

4.3.1 Experimental Results of MxM RD Experiment without TMR

In this section, Fault injection is done to a single core instantiated by OpenPiton. the **Correct** was 74.6%, meaning that out of 1000 iterations, only 746 iterations were correct.

This percentage is significantly lower than that obtained from the FS experiments, which can be attributed to the complex nature of the Matrix Multiplication (MxM) benchmark. Unlike the FS benchmark, MxM involves loops, conditional statements, and a greater number of instructions.

The following table 4.10 indicates some of this experiment's output samples:

Table 4.10: Experimental Results of MxM RD Experiment without TMR

iteration_no	tile	reg	pos	val	time	t0_out	MM_result_1	MM_result_2	category
0	0	19	36	1	18	0	0xh00	0xh00	FALSE
1	0	15	15	1	12	1	0xh01d	0xh00a	CORRECT
2	0	18	13	1	18	1	0xh01d	0xh00a	CORRECT
3	0	07	64	1	5	1	0xh01d	0xh00a	CORRECT
..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..
996	0	06	28	1	44	1	0xh01d	0xh00a	CORRECT
997	0	23	56	1	45	1	0xh01d	0xh00a	CORRECT
998	0	14	20	1	15	1	0xh01d	0xh00a	CORRECT
999	0	13	36	1	47	1	0xh01d	0xh0a	CORRECT

4.3.2 Experimental Results of MxM RD Experiment with TMR

For the 1C scenario, faults are injected into only one core out of the three instantiated OpenPiton cores after TMR implementation.

The experimental results showed improvement in the reliability of the system against the injected fault framework, where **correct** metric turned out to be 95%, meaning out of 1000 iterations, 950 iterations where the outputs were correct. This means that **The TMR Mitigation Effect** was calculated to be 20.4%. This mitigation percentage is higher than that obtained in FS benchmark Experiments which was 13.5%. This is because The matrix multiplication process inherently involves a large number of complex instructions such as loops, which can be more effectively protected by TMR. The distributed nature of errors across many operations makes it easier for TMR to identify and mask the errors. The following table 4.11 indicates some of this experiment's output samples:

In the 2C scenario, faults are injected into two of the three instantiated OpenPiton cores after TMR implementation. This means an expected decrease in the **correct** metric due to the severity of the fault injections. The **correct** metric was 92.1% and The overall **TMR mitigations effect** was calculated to be 17.5%.

The table 4.12 shows Experiment 2C sample outputs.

In the 3C scenario, as detailed in Table 4.1, the experiment involves injecting faults into

Table 4.11: Experimental Results of MxM RD Experiment with TMR

iteration_no	tile	reg	pos	val	time	t0_out	MM_result_1	MM_result_2	category
0	0	25	13	1	215	1	0xh1d	0xh0a	CORRECT
1	2	15	11	1	149	1	0xh01d	0xh00a	CORRECT
2	0	23	62	1	151	1	0xh01d	0xh00a	CORRECT
3	2	17	15	1	110	1	0xh01d	0xh00a	CORRECT
..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..
996	1	18	18	1	265	1	0xh01d	0xh00a	CORRECT
997	0	16	55	1	235	1	0xh01d	0xh00a	CORRECT
998	2	05	34	1	38	1	0xh01d	0xh00a	CORRECT
999	1	21	09	1	104	1	0xh01d	0xh0a	CORRECT

Table 4.12: Experimental Results of MxM 2C RD Experiment with TMR

iteration_no	tile_1	tile_2	reg_1	reg_2	pos_1	pos_2	val_1	val_2	time	t0_out	tmr_result_1	tmr_result_2	category
0	2	0	16	7	5	9	1	1	125	1	0xh01d	0xh00a	CORRECT
1	0	1	1	8	13	62	1	1	124	1	0xh01d	0xh00a	CORRECT
2	1	2	1	17	6	18	1	1	262	1	0xh01d	0xh00a	CORRECT
3	0	1	22	27	46	29	1	1	109	1	0xh01d	0xh00a	CORRECT
..	..	..	..	..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..	..	..	..	..
996	1	2	1	24	26	32	1	1	120	1	0xh01d	0xh00a	CORRECT
997	0	1	18	26	19	36	1	1	54	1	0xh01d	0xh00a	CORRECT
998	1	0	18	19	30	60	1	1	242	1	0xh01d	0xh00a	CORRECT
999	0	2	19	5	62	32	1	1	89	1	0xh01d	0xh00a	CORRECT

one, two, or all three of the instantiated cores. This setup is designed to simulate the overall impact of faults on the system with TMR integration.

Similarly to 4.2.2's 3C scenario, a higher **correct** metric is expected in comparison to that of MxM RD 2C experiment, achieving a value of 91.5% and the overall **TMR mitigation effect** was determined to be 16.9%,

The table 4.13 shows Experiment 3C sample outputs.

Table 4.13: Experimental Results of 3C MxM RD Experiment with TMR

iteration_no	tile_combination	reg_1	reg_2	reg_3	pos_1	pos_2	pos_3	val_1	val_2	val_3	time	tile0_out	tmr_result_1	tmr_result_2	category
0	11	23	07	17	8	18	22	1	1	x	48	1	0xh0a	0xh01d	CORRECT
1	111	09	07	12	48	24	14	1	1	1	248	1	0xh0a	0xh01d	CORRECT
2	110	27	12	31	15	43	28	1	1	x	17	0	0xh00	0xh00	FALSE
3	10	14	19	10	22	28	8	1	x	x	75	1	0xh0a	0xh01d	CORRECT
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
996	0	27	25	8	40	7	27	x	x	x	212	1	0xh0a	0xh01d	CORRECT
997	0	07	23	10	49	16	39	x	x	x	303	1	0xh0a	0xh01d	CORRECT
998	1	05	22	12	14	51	46	1	x	x	231	1	0xh0a	0xh01d	CORRECT
999	0	22	12	09	41	30	5	x	x	x	38	1	0xh0a	0xh01d	CORRECT

In the second section of the experiment, fault injection is done randomly in the Error

Correction Code (ECC) of a general-purpose register.

4.3.3 Experimental Results of MxM ECC Experiment without TMR

Similarly to 4.2.3, a fault injection framework was applied to a single core instantiated in OpenPiton, with the key difference being that faults were injected into the ECC bits. Similarly, an expected increase **correct** metric is found, as the fault injections are done only to the 8 bits of the ECC part of the OpenPiton data packet. The **correct** was found to be 79.5%, indicating that out of 1,000 iterations, 795 produced correct outputs.

The following table 4.14 indicates some of this experiment's output samples. Note that **pos** column has values equal to or greater than 64.

Table 4.14: Experimental Results of MxM ECC Experiment without TMR

iteration_no	tile	reg	pos	val	time	t0_out	MM_result_1	MM_result_2	category
0	0	16	69	1	2	1	0xh01d	0xh00a	CORRECT
1	0	02	71	1	2	1	0xh01d	0xh00a	CORRECT
2	0	21	65	1	10	0	0xh00	0xh05	FALSE
3	0	07	64	1	5	1	0xh01d	0xh00a	CORRECT
..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..
996	0	05	69	0	48	1	0xh01d	0xh00a	CORRECT
997	0	09	71	0	44	1	0xh01d	0xh00a	CORRECT
998	0	29	67	1	56	1	0xh01d	0xh00a	CORRECT
999	0	26	68	1	20	0	0xh00	0xh00	FALSE

4.3.4 Experimental Results of MxM ECC Experiment

For the 1C scenario, similarly to 4.2.4 faults are injected into the ECC code of the data packet of only one core out of the three instantiated OpenPiton cores after TMR implementation. The yielded results showed improvement in system robustness against faults, the **correct** metric turned out to be 96.9%, meaning out of 1000 iterations, 969 iterations yielded correct results. Thus, **The TMR Mitigation Effect** was calculated to be 17.4%.

As expected and previously explained, a decrease in the TMR mitigation percentage is expected in comparison to that of RD Experiment (20.4%).

The following table 4.15 indicates some of this experiment's output samples. Note that **pos** column has values equal to or greater than 64

Table 4.15: Experimental Results of MxM ECC Experiment with TMR

iteration_no	tile	reg	pos	val	time	t0_out	MM_result_1	MM_result_2	category
0	2	06	64	0	200	1	0xh1d	0xh0a	CORRECT
1	0	30	71	1	163	1	0xh01d	0xh00a	CORRECT
2	2	07	70	1	94	1	0xh01d	0xh00a	CORRECT
3	2	02	69	0	27	1	0xh01d	0xh00a	CORRECT
..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..
996	0	08	69	1	113	1	0xh01d	0xh00a	CORRECT
997	1	11	67	0	250	1	0xh01d	0xh00a	CORRECT
998	2	29	66	1	239	1	0xh01d	0xh00a	CORRECT
999	1	25	69	1	51	1	0xh01d	0xh0a	CORRECT

In the 2C scenario, similarly to 4.2.2, faults are injected into the ECC bits of two out of the three instantiated OpenPiton cores. As expected, a decrease in the **correct** metric was observed, resulting in a value of 93.1%, and the **TMR Mitigation Effect** was calculated to be 13.6%.

The table 4.16 shows Experiment 2C sample outputs. Note that the values of both **pos_1** and **pos_2** are equal or greater than 64

Table 4.16: Experimental Results of MxM 2C ECC Experiment with TMR

iteration_no	tile_1	tile_2	reg_1	reg_2	pos_1	pos_2	val_1	val_2	time	t0_out	tmr_result_1	tmr_result_2	category
0	2	0	27	09	66	70	1	1	138	1	0xh0a	0xh01d	CORRECT
1	1	0	21	19	65	69	1	1	15	0	000	00	FALSE
2	0	1	24	28	71	66	1	1	12	1	0x000	0x00	FALSE
3	1	0	30	27	69	71	1	0	198	1	0xh000a	0xh01d	right
..	..	..	..	..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..	..	..	..	..
996	2	1	30	30	65	68	1	1	90	1	0xh0a	0xh01d	right
997	2	1	06	16	66	70	0	1	133	1	0xh0a	0xh01d	right
998	1	2	11	24	70	65	1	1	238	1	0xh0a	0xh01d	right
999	1	0	20	16	69	66	1	0	97	1	0xh0a	0xh01d	right

In the 3C scenario, similarly to 4.2.2 but the experiment involves injecting faults into the **ECC bits** of one, two, or all three of the instantiated cores. An increase in the **correct**

metric is found relative to that achieved from the 2C FS ECC Experiment, achieving 95.9%. the overall **TMR mitigation effect** turned out to be 16.4%.

The table 4.17 shows Experiment 3C sample outputs. Note that the values of both **pos_1**,**pos_2** and **pos_3** are equal or greater than 64.

Table 4.17: Experimental Results of 3C MxM ECC Experiment with TMR

iteration_no	tile_combination	reg_1	reg_2	reg_3	pos_1	pos_2	pos_3	val_1	val_2	val_3	time	tile0_out	tmr_result_1	tmr_result_2	category
0	111	14	25	27	70	71	69	1	0	1	150	1	0xh0a	0xh01d	CORRECT
1	110	30	19	13	69	64	66	1	0	x	83	1	0xh0a	0xh01d	CORRECT
2	111	21	06	04	64	68	66	1	1	0	263	1	0xh0a	0xh1d	FCORRECT
3	10	29	04	18	66	66	71	1	x	x	291	1	0xh0a	0xh01d	CORRECT
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
996	1	06	27	7	68	65	71	1	x	x	220	1	0xh0a	0xh01d	CORRECT
997	111	01	22	24	68	71	70	1	1	1	36	0	0xh0	0xh0	FALSE
998	100	11	19	24	69	65	68	1	x	x	40	1	0xh0a	0xh01d	CORRECT
999	0	23	27	16	65	69	67	1	x	x	39	1	0xh0a	0xh01d	CORRECT

To assess the overhead introduced by TMR, we measured the execution time of the MxM benchmark. Before implementing TMR, the test duration was 20 seconds. After integrating TMR, the execution time increased to 21 seconds. Consequently, the time overhead is calculated to be 5%. This result is favorable, as there was no significant increase in time overhead after TMR integration.

4.4 Conclusions

Multiple observations have been made based on the various experiments that have been conducted:

- **Impact of Fault Severity on the correct Metric in the 2C Scenario**:

  As the severity of fault injection increases in the 2C scenario, a corresponding decrease in the **correct** metric is observed. This outcome aligns with expectations, as injecting faults into two cores introduces a higher likelihood of errors, thereby reducing the system's ability to produce correct results.

- **Comparative Analysis of the 3C Scenario**:

In the 3C scenario, the **correct** metric is higher than in the 2C scenario. This can be attributed to the random combination of fault injections across one, two, or three cores. Importantly, the presence of faults in all three cores does not necessarily lead to incorrect results. The outcome depends on both the timing and the location of the fault injections, indicating that certain faults may not impact the system's critical operations.

- **Enhanced Correct Metric in ECC Experiments**:

  In the ECC experiments, an increase in the **correct** metric is observed, which is expected due to the restriction of fault injections to the 8 ECC bits as The data packet comprises 64 data bits and 8 ECC bits. Since ECC bits are fewer and primarily used for error correction, faults confined to these 8 bits have a more localized impact. On the other hand, faults in the 64 data bits directly affect the actual information being processed, making it more difficult for the system to recover or produce correct outputs. However, The TMR improvement percentage is lower in ECC case as ECC's inherent complexity and its focus on detection over correction make it more challenging for TMR to effectively mitigate errors. This contrasts with data, where TMR directly corrects errors, leading to a higher mitigation effect.

- **Complexity-Driven Susceptibility in MxM Benchmarks**:

  The **correct** metric for Matrix Multiplication (MxM) benchmarks is lower compared to that of Fibonacci Sequence (FS) benchmarks due to the MxM 's higher complexity, greater instruction density, and increased sensitivity to faults. These factors make MxM benchmarks more vulnerable to faults compared to the simpler and more linear FS benchmarks.

- **Higher TMR Mitigation Percentages in MxM Benchmarks**:

Despite the increased susceptibility to errors, the overall **TMR Mitigation** percentage for MxM benchmarks is higher than that of FS benchmarks. The complexity of MxM, which includes loops, comparisons, and more instructions, provides more opportunities for TMR to mitigate the errors that arise, thereby enhancing the system's robustness.

Based on the yielded results, TMR has shown huge improvements in system robustness against soft errors. Enhancements of 13.5% and 20.4% in the Fibonacci series (FS) and matrix multiplication (MxM), respectively, were achieved with the integration of TMR in the data registers fault injection in a single core. Similarly, when the fault injection is done in two cores simultaneously, the results showed substantial enhancement of 13.1% and 17.5% in the FS and MxM respectively. Moreover, integrating TMR added a time overload for FS and MxM benchmarks with an increase of 11% and 5% respectively. This result is remarkable, as the time overhead remains minimal, and there is no additional area overhead, thanks to the effective utilization of the extensive resources (i.e., unused threads) available in multi-core processors.

Table 4.18 presents a comprehensive summary of the experimental results obtained throughout this thesis.

4.5 Summary

This chapter primarily addresses the experiments conducted to evaluate the impact of integrating Triple Modular Redundancy (TMR) on the robustness of the many-core processor OpenPiton against Single-Event Upsets (SEUs). Two key experiments were performed involving fault injections into two selected benchmarks—Fibonacci Sequence (FS) and Matrix Multiplication (MxM)—without the application of the proposed multi-threaded TMR. The benchmarks and the algorithms employed were described, followed by

Table 4.18: Experimental Results of RD and ECC Implementations

Experiment	FS Results (%)		MxM Results (%)	
	Result	Improvement	Result	Improvement
RD				
No_TMR	82.6	–	74.6	–
1C	96.1	13.5	95	20.4
2C	95.5	13.1	92.1	17.5
3C	95.1	12.5	91.5	16.9
ECC				
No_TMR	88.2	–	79.5	–
1C	97.7	9.5	96.9	17.4
2C	96.1	7.9	93.1	13.6
3C	97.5	9.3	95.9	16.4

an overview of the experimental environment with multithreaded TMR integration.

The first set of experiments involved implementing TMR on 1-core, 2-core, and 3-core systems using the FS benchmark, targeting general-purpose data. Similarly, the second set of experiments applied the same setup to the MxM benchmark. Both experiments demonstrated promising results, showing an average improvement of 13.5% in FS and 20.4% in MxM, respectively.

Additionally, the experiments investigated fault generation using a random binary combination approach, where faults were injected into 1-core, 2-core, or 3-core systems. The improvement percentages achieved were consistent with those observed in the previous experiments, further validating the effectiveness of TMR integration in enhancing robustness against SEUs.

Observations and conclusions are detailed, providing insights into each case study discussed in the thesis.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Recent developments in the avionics domain have demonstrated an increasing interest in transitioning to commercial off-the-shelf (COTS) many-core processors, owing to their promising performance and moderate power and cost efficiency. However, the scalability of core numbers is constrained by the absence of reliability certification for many-core COTS processors in radiation environments. Our research discussed implementing TMR to Many-core processors OpenPiton to increase robustness against SEUs. The thesis begins by explaining the motivation behind the industry's transition towards commercial off-the-shelf (COTS) many-core processors. It then provides a comprehensive overview of critical topics, including Single Event Upsets (SEUs) and reliability assessment techniques such as radiation ground testing, fault injection methods, and formal verification approaches.

Next, the OpenPiton framework is introduced, highlighting its extensive resources and configurability features. This is followed by a discussion of the benchmarks employed, such as the Fibonacci series (FS) and matrix multiplication (MxM), along with a detailed explanation of the algorithms used to implement them. The thesis also explores various soft error mitigation techniques, categorizing them into software-based, hardware-based, and

hybrid approaches. One particular focus is on the ”Multi-threaded TMR” technique, where its implementation and the challenges encountered, such as race conditions, are thoroughly analyzed, along with strategies for addressing these issues.

Chapter 4 goes deeper into the experimental setup, starting with the OpenPiton simulation environment and progressing to the two primary case studies. The first case study involves running simulations using the FS benchmark for fault injection across one, two, and three cores. In this experiment, fault injections are conducted in two distinct areas: first, in the actual data of the general-purpose registers (RD), and second, in the ECC bits of the general-purpose registers. The second case study follows a similar approach but uses the MxM benchmark instead.

The results of these experiments are presented with detailed observations for each. The findings demonstrate that Triple Modular Redundancy (TMR) significantly improved the robustness of OpenPiton against SEUs, with average improvements of 14.1% for the Fibonacci series and 17.3% for matrix multiplication. However, despite these gains in reliability, the use of TMR introduced a notable time overhead, calculated as 133% for FS and 150% for MxM.

We observed an increase in fault severity as the number of cores subjected to fault injections grew. Additionally, while the correct output metric was higher in experiments involving ECC, the TMR mitigation percentages in these experiments were lower compared to those in the RD experiments. This can be attributed to the complexity of ECC, which makes it more challenging for TMR to effectively mask its errors.

Another noteworthy observation is that although matrix multiplication (MxM) exhibits lower correct output percentages, likely due to its greater complexity relative to the Fibonacci series (FS), it still shows higher TMR mitigation percentages. This is because the increased complexity of MxM provides TMR with more opportunities to mask errors effectively.

5.2 Future Work

The TMR implemented in the context of this thesis presents a base for a complete framework that can be further enhanced in the following future steps:

- Automate the framework to be able to take the c code of the application and automatically change it to its assembly code where the TMR is implemented.
- Optimize the memory usage and the simulation time while using TMR.
- Using TMR for only the critical part of the code, thus a criteria for choosing the critical variables and code snippets needs to be created.
- Extend the research to emulate these experiments using Field Programmable Gate Arrays (FPGA).

Appendix A

The 3C fault injection script is given below:

```
#!/usr/bin/tclsh8.5
proc randomlist {list} {

        lindex $list [expr {int(rand()*[llength $list])}]

}
set loop_max 1000
for {set i 0} {$i < $loop_max} {incr i} {
puts "loop begin"
set fb1 [open "faults_deposit_TMR_1_4.txt" r]
set fb2 [open "faults_deposit_TMR_2_4.txt" r]
set fb3 [open "faults_deposit_TMR_3_4.txt" r]

set fault_list_1 [read $fb1]
set fault_list_2 [read $fb2]
set fault_list_3 [read $fb3]

close $fb1
```

```
close $fb2
close $fb3

while 1 {

set combination [expr {int(rand() * 8)}]

set binary_combination [format %b $combination]

set tile1 [string index $binary_combination 0]
set tile2 [string index $binary_combination 1]
set tile3 [string index $binary_combination 2]

#000 N0 injection                -----0- NO
#001 injection in core0           ----1---single
#010  injection in core1         -----2---single
#011  injection in core 0 and 1 ----3----------double
#100  injection in core2           ---4 ---single
#101   injection in core 0 and 2 ---5----------double
#110   injection in core 2 and 1----6----------double
#111   injection in core 0,1,2    ---7----------------triple


# if {$combination == 0} {
# set injection "1"


# } elseif {
# set pos2_f "${pos2}"
# }
# else {
# }
```

```
#set tile [randomlist {01 02 10 12 20 21}]
#set tile1 [string index $tile 0]
#set tile2 [string index $tile 1]

puts "$tile1"
puts "$tile2"
puts "$tile3"



set reg1 [randomlist { 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
    17 18 19 20 21 22 23 24 25 26 27 28 29 30 31}]
set reg2 [randomlist { 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
    17 18 19 20 21 22 23 24 25 26 27 28 29 30 31}]
set reg3 [randomlist { 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
    17 18 19 20 21 22 23 24 25 26 27 28 29 30 31}]

set pos1 [expr {int(rand() * 64)}]
if {$pos1 < 10} {
set pos1_f  "0${pos1}"
} else {
set pos1_f "${pos1}"
}
set pos2 [expr {int(rand() * 64)}]
if {$pos2 < 10} {
set pos2_f  "0${pos2}"
} else {
set pos2_f "${pos2}"
}
```

```
set pos3 [expr {int(rand() * 64)}]
if {$pos3 < 10} {
set pos3_f  "0${pos3}"
} else {
set pos3_f "${pos3}"
}

set time [expr {int(rand() * 330+ 8)}]
set uniq_list_1 [lsearch $fault_list_1 "${tile1}${reg1}${pos1_f}${time
    }"]
set uniq_list_2 [lsearch $fault_list_2 "${tile2}${reg2}${pos2_f}${time
    }"]
set uniq_list_3 [lsearch $fault_list_3 "${tile3}${reg3}${pos3_f}${time
    }"]
if {($uniq_list_1 == -1) || ($uniq_list_2 == -1) ||($uniq_list_3 ==
    -1) } {
set fault_list_1 [linsert $fault_list_1 [llength $fault_list_1] "${
    tile1}${reg1}${pos1_f}${time}"]
set fault_list_2 [linsert $fault_list_2 [llength $fault_list_2] "${
    tile2}${reg2}${pos2_f}${time}"]
set fault_list_3 [linsert $fault_list_3 [llength $fault_list_3] "${
    tile3}${reg3}${pos3_f}${time}"]
break
}
}
puts "parameter choose val excluded"
set remove_1 {rm -f /nfs/private/a/ait/fvg/openpiton/build/faults_
    deposit_TMR_1_4.txt}
exec bash -c $remove_1
set cmd_file_1 [list echo ${fault_list_1} > /nfs/private/a/ait/fvg/
    openpiton/build/faults_deposit_TMR_1_4.txt]
exec {*}$cmd_file_1
```

```
set remove_2 {rm -f /nfs/private/a/ait/fvg/openpiton/build/faults_
    deposit_TMR_2_4.txt}
exec bash -c $remove_2
set cmd_file_2 [list echo ${fault_list_2} > /nfs/private/a/ait/fvg/
    openpiton/build/faults_deposit_TMR_2_4.txt]
exec {*}$cmd_file_2
set remove_3 {rm -f /nfs/private/a/ait/fvg/openpiton/build/faults_
    deposit_TMR_3_4.txt}
exec bash -c $remove_3
set cmd_file_3 [list echo ${fault_list_3} > /nfs/private/a/ait/fvg/
    openpiton/build/faults_deposit_TMR_3_4.txt]
exec {*}$cmd_file_3
puts "element added to faults list & just befor run time"
run $time us
puts " after run time"

set val1 "x"
set val2 "x"
set val3 "x"



if {$tile1 == 1} {

set force_dir_1 "sim:/cmp_top/system/chip/tile0/g_sparc_core/core/
    sparc0/exu/exu/irf/irf/bw_r_irf_core/rd_data${reg1}[${pos1}]"
if {[examine -binary $force_dir_1]=="1'b0"} {
set val1 "1"
} else {
set val1 "0"
}
```

```
force -deposit sim:/cmp_top/system/chip/tile0/g_sparc_core/core/sparc
    0/exu/exu/irf/irf/bw_r_irf_core/rd_data${reg1}[${pos1}] 'b${val1}
force
}

if { $tile2 == 1} {

set force_dir_2 "sim:/cmp_top/system/chip/tile2/g_sparc_core/core/
    sparc0/exu/exu/irf/irf/bw_r_irf_core/rd_data${reg2}[${pos2}]"
if {[examine -binary $force_dir_2]=="1'b0"} {
set val2 "1"
} else {
set val2 "0"
}
force -deposit sim:/cmp_top/system/chip/tile2/g_sparc_core/core/sparc
    0/exu/exu/irf/irf/bw_r_irf_core/rd_data${reg2}[${pos2}] 'b${val2}
force
}

if {$tile3 == 1} {

set force_dir_3 "sim:/cmp_top/system/chip/tile3/g_sparc_core/core/
    sparc0/exu/exu/irf/irf/bw_r_irf_core/rd_data${reg3}[${pos3}]"
if {[examine -binary $force_dir_3]=="1'b0"} {
set val3 "1"
} else {
set val3 "0"
}
force -deposit sim:/cmp_top/system/chip/tile3/g_sparc_core/core/sparc
    0/exu/exu/irf/irf/bw_r_irf_core/rd_data${reg3}[${pos3}] 'b${val3}
force
}
```

```
puts "force action"
run 200 us
puts "after run 118 us"
set tr [open "transcript" r]
set data [read -nonewline $tr]
close $tr
# Get the last line
set uniq_tr [lsearch -glob [lrange [split $data \n] end-1 end] "#
    Stopped*"]
if {$uniq_tr == -1} {
set state "Hang"
} else {
set state "no_hang"
}
set error_state [lsearch -glob [lrange [split $data \n] end-50 end] "*
    ERROR:*"]
if {$error_state == -1} {
set state "no_error"
} else {
set state "ERROR"
}
set dir_name "tile_${binary_combination}_reg${reg1}_${reg2}_${reg3}_
    pos${pos1}_${pos2}_${pos3}_val${val1}_${val2}_${val3}_time${time}"
set dir "/nfs/private/a/ait/fvg/openpiton/build/TMR_3/reg_file_faults_
    data_deposit/${dir_name}"
set dir_mv "/nfs/private/a/ait/openpiton/build/TMR_3/reg_file_faults_
    data_deposit/${dir_name}/"
set rep_name "report_${dir_name}.txt"
set cmd_echo [list echo Hang > /nfs/private/a/ait/fvg/openpiton/build/
    TMR_3/reports_regdata_deposit/$rep_name]
set cmd_error_msg {less transcript | grep 'ERROR:'> report_error.log}
```

```
set cmd_error_name [list mv /nfs/private/a/ait/fvg/openpiton/build/
    report_error.log /nfs/private/a/ait/fvg/openpiton/build/$rep_name]
set cmd_error [list mv /nfs/private/a/ait/fvg/openpiton/build/$rep_
    name /nfs/private/a/ait/fvg/openpiton/build/TMR_3/reports_regdata_
    deposit]
if {$state=="Hang"} {
exec {*}$cmd_echo
} elseif {$state == "ERROR"} {
exec bash -c $cmd_error_msg
exec {*}$cmd_error_name
exec {*}$cmd_error
} else {
# Save the register file and the L2 cache of the tiles at the end of
    execution
mem save -outfile tile0_reg_file.mem /cmp_top/system/chip/tile0/g_
    sparc_core/core/sparc0/exu/exu/irf/irf/bw_r_irf_core/rfg -
    wordsperline 1
mem save -outfile tile1_reg_file.mem /cmp_top/system/chip/tile1/g_
    sparc_core/core/sparc0/exu/exu/irf/irf/bw_r_irf_core/rfg -
    wordsperline 1
mem save -outfile tile2_reg_file.mem /cmp_top/system/chip/tile2/g_
    sparc_core/core/sparc0/exu/exu/irf/irf/bw_r_irf_core/rfg -
    wordsperline 1
mem save -outfile tile3_reg_file.mem /cmp_top/system/chip/tile3/g_
    sparc_core/core/sparc0/exu/exu/irf/irf/bw_r_irf_core/rfg -
    wordsperline 1


mem save -outfile tile0_l2.mem /cmp_top/system/chip/tile0/l2/data_wrap
    /l2_data/l2_data_array/sram_l2_data/ram
mem save -outfile tile1_l2.mem /cmp_top/system/chip/tile1/l2/data_wrap
    /l2_data/l2_data_array/sram_l2_data/ram
```

```
mem save -outfile tile2_12.mem /cmp_top/system/chip/tile2/12/data_wrap
    /12_data/12_data_array/sram_12_data/ram
mem save -outfile tile3_12.mem /cmp_top/system/chip/tile2/12/data_wrap
    /12_data/12_data_array/sram_12_data/ram


puts "mem save"

#Treatment after the simulation terminated
#uniq comm23and is used to generate the instruction file and the reg01
     dump without repetitive values
puts "before treatment script"


set treatmentScript {

less tile3_reg_file.mem | grep '17:' > t3_rf

cat   t3_rf  > report_sim.log
}


exec bash -c $treatmentScript


puts " treatment script executed"



set cmd_rep [list mv /nfs/private/a/ait/fvg/openpiton/build/report_sim
    .log  /nfs/private/a/ait/fvg/openpiton/build/TMR_3/reports_regdata_
    deposit/$rep_name]
```

```
exec {*}$cmd_rep



}
transcript file ""
transcript file /nfs/private/a/ait/fvg/openpiton/build/transcript
puts "before restart"
restart -force
puts "after restart"
puts "end of loop iteration number $i"}
puts "end of loop"
exit
```

Appendix B

The fault analyzer python code is given below:

```
import os
import pandas as pd
all_files = os.listdir("/nfs/private/a/ait/fvg/openpiton/build/TMR_3/
    reports_regdata_deposit/")
print(len(all_files))
fault_list = []
fault_list_t0 =[]

def slice_string (file_name,n):
    print("--------------START OF NEW LOOP------------")
    print (file_name)
    print(n)
   # report_tile_100_reg05_22_16_pos54_8_37_val1_x_x_time144.txt
    file_name = file_name.split('_')


    tile_combination=file_name[2]
    reg_1 = file_name[3]
    reg_1 = reg_1 [3:]
```

```
reg_2 = file_name[4]
reg_3 = file_name[5]

pos_1 = file_name[6]

pos_1 = pos_1[3:]
pos_2=file_name[7]
pos_3=file_name[8]

val_1 = file_name[9]
val_1 = val_1[3]

val_2=file_name[10]
val_3=file_name[11]

time = file_name[12]
time =time [4:-4]
dir =['/nfs/private/a/ait/fvg/openpiton/build/TMR_3/reports_
    regdata_deposit/',all_files[n]]
name=''
name= name.join(dir)
f = open('%s' % name ,'r')
report_list =f.readlines()



tmr_result=report_list[0].split('  ')
tmr_result=tmr_result[1]
tmr_result=tmr_result[-15:]
print (tmr_result)
```

```
    if ((int(tmr_result,16) ==34)):
        tile0_out=1
    else:
        tile0_out=0



    if (tile0_out ==1):

        categorie= 'CORRECT'


    else:
        categorie= 'FALSE'


    fault_spec_list = [tile_combination, reg_1,reg_2,reg_3,pos_1, pos
        _2,pos_3, val_1,val_2,val_3,time,tile0_out,tmr_result,categorie
        ]
    return fault_spec_list



for i in range (len(all_files)):
    fault_list.append(slice_string(all_files[i],i))

df_data_deposit = pd.DataFrame(fault_list,columns =['tile_combination
    ','reg_1','reg_2','reg_3','pos_1','pos_2','pos_3','val_1','val_2','
    val_3','time','tile0_out','tmr_result','categorie'])
file_data_deposit = 'data_deposit_faults_TMR_core.xlsx'
```

```
df_data_deposit.to_excel(file_data_deposit)

for j in range (len(fault_list)):
    fault_list_t0.append(fault_list[j][-1])




t0_right =0
t0_wrong = 0




    if (fault_list_t0[k]=='CORRECT'):
          t0_right += 1
    elif (fault_list_t0[k]=='FALSE'):
         t0_wrong += 1

fault_list_t0_total= len(fault_list_t0)




t0_right= 100 * float(t0_right)/float(fault_list_t0_total)
t0_wrong= 100 * float(t0_wrong)/float(fault_list_t0_total)


print ('tile3')

print ("CORRECT:                ",t0_right)
print ("FALSE:                  ",t0_wrong)
```

Bibliography

[1] Chifa Dammak, Otmane Ait Mohamed, and Mounir Boukadoum. Seu reliability assessment framework for cots many-core processors. In *2022 International Conference on Microelectronics (ICM)*, pages 42–45, 2022.

[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*, PP:1–1, 01 2019.

[4] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Morgan kaufmann, 2017.

[5] Bryan Schauer. Multicore processors–a necessity. *ProQuest discovery guides*, 59, 2008.

[6] TSMC Leading Semiconductor Manufactor logic technology. https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_2nm. Last Accessed: August 7, 2024.

[7] Raoul Velazco, Pascal Fouillat, and Ricardo Reis. *Radiation effects on embedded systems*. Springer Science & Business Media, 2007.

[8] Robert C Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3):305–316, 2005.

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

[10] Muhammad Sadi, Md Khan, Md Uddin, and Jan Jürjens. An efficient approach towards mitigating soft errors risks. *Signal Image Processing : An International Journal*, 2, 10 2011.

[11] The ibm power4. https://www.ibm.com/history/power. Last Accessed: August 11, 2024.

[12] Balaji Venu. Multi-core processors-an overview. *arXiv preprint arXiv:1110.3535*, 2011.

[13] Jeffrey S George. An overview of radiation effects in electronics. In *25th International Conference on the Application of Accelerators in Research and Industry*, volume 2160, page 060002, 2019.

[14] Jeffrey S. George. An overview of radiation effects in electronics. *AIP Conference Proceedings*, 2160(1):060002, 10 2019.

[15] 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 manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 217–232, New York, NY, USA, 2016. Association for Computing Machinery.

[16] 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.

[17] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004.

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

[19] 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.

[20] 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.

[21] Antonio Martínez-Álvarez, Felipe Restrepo-Calle, Sergio Cuenca-Asensi, Leonardo M. Reyneri, Almudena Lindoso, and Luis Entrena. A hardware-software approach for on-line soft error mitigation in interrupt-driven applications. *IEEE Transactions on Dependable and Secure Computing*, 13(4):502–508, 2016.

[22] Heather Quinn, Zachary Baker, Tom Fairbanks, Justin L. Tripp, and George Duran. Robust duplication with comparison methods in microcontrollers. *IEEE Transactions on Nuclear Science*, 64(1):338–345, 2017.

[23] Benjamin James and Jeffrey Goeders. Automated software compiler techniques to provide fault tolerance for real-time operating systems. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1452–1455, 2021.

[24] A. Serrano-Cases, A. Martínez-Álvarez, R. Possamai Bastos, and S. Cuenca-Asensi. Bare-metal redundant multi-threading on multicore socs under neutron irradiation. *IEEE Transactions on Nuclear Science*, 70(8):1643–1651, 2023.

[25] Gennaro S. Rodrigues, Felipe Rosa, Fernanda L. Kastensmidt, Ricardo Reis, and Luciano Ost. Investigating parallel tmr approaches and thread disposability in linux. In *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 393–396, 2017.

[26] Joshua Monson, Mike Wirthlin, and Brad Hutchings. Fault injection results of linux operating on an fpga embedded platform. pages 37 – 42, 01 2011.

[27] Andrew Milluzzi, Alan George, and Alan George. Exploration of tmr fault masking with persistent threads on tegra gpu socs. In *2017 IEEE Aerospace Conference*, pages 1–7, 2017.

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