POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico di Torino

Master's Degree Thesis

Reliability of Application Execution on A Processor with/without OS

Supervisors

Prof. Luca Sterpone

Prof. Sarah Azimi

Ph.D. Corrado De Sio

Candidate

Roozbeh Nikzadi

THE ACADEMIC YEAR 2022/2023

Abstract

In safety-critical applications, such as aerospace, automotive, medical, and industrial control systems, the occurrence of faults can have serious consequences, including loss of human life, property damage, and environmental damage. For this purpose, fault occurrence is a critical concern in these applications.

It is crucial to prevent defects from arising in the first place, but it is equally necessary to detect and reduce the faults when they do arise. This is accomplished by using a combination of fault-tolerant architecture, redundancy, and error detection and correction systems in both hardware and software designs.

Typically, measures like the probability of failure on demand (PFOD), the probability of dangerous failure per hour (PFH), or the safety integrity level are used to quantify fault occurrence in safety-critical systems (SIL). These metrics offer a numerical assessment of the level of safety that the system is capable of achieving and are applied to direct the design and testing procedures.

This thesis aimed to figure out the reliability of application execution on a processor with and without an operating system and choose which one is more reliable for safety-critical applications.

To this end, In order to examine the impact of radiations hitting the hardware while an application is being run, a system-on-chip architecture has been examined. Furthermore, two platforms have been produced to cover the aim of using a processor with the operating system and without. Then a fault injection platform has been developed with the goal of injecting faults in the main memory of RAM. The idea is to replicate radiation-related errors while an application is running. SEU (single event upset) has been chosen as the fault model for this experiment. Two programs from the MiBench benchmark suite and two other programs from a random source have been used as test input for the radiation impacts of an investigation. For the injection result, SDC, Halt, and Empty files have been chosen as the output classifications.

It was possible to see that the fault rate in some programs running on processors with operating systems is greater than the error rate in the same applications running on processors without operating systems by comparing the results given by the fault injection platform. Therefore, it may be concluded from this conclusion that reliability could alter in the same program operating on a different system. So, in order to decrease the probability that an error would occur, we must choose an application carefully and supply it with a proper system.

Acknowledgments

I want to express thanks to my family for their continual backing over the years, from the time I was in elementary school to the present, and for allowing me to create my path for the future. My mother deserves a special mention because of all the work, support, and motherly care she provided for me.

I appreciate Corrado's support and assistance, as well as his knowledge and wise counsel throughout the process. Sarah, thank you for being a strong leader and guiding this thesis in the best way possible. Thank you Luca for including me in your team as a thesis student, and thank you to all of my friends and colleagues who have supported me over the past several months.

Many thanks to my wife for being supportive and helping me during my difficult times. It would not have been able to reach this point without you.

Finally, thank me for believing in me, for being resistive, for all of my efforts throughout these years, and for never giving up.

Contents

List of Figures

List of Tables

Abbreviations

SoC	System On Chip
RAM	Random Access Memory
SDIO	Secure Digital Input Output
PS	Processing System
PL	Programmable Logic
MP SoC	Multi Processor System On Chip
FPGA	Field-Programmable Gate Array
DSP	Digital Signal Processing or Processor
DMA	Direct Memory Access
USB	Universal Serial Bus
HDL	Hardware Description Language
VHDL	Very High-Speed Integration Circuit
AXI	Advanced eXtensible Interface
AMBA	Advanced Microcontroller Bus Architecture
Pmod	Peripheral Module
SDC	Silent Data Corruption
ELF	Executable and Linkable Format
MBU	Multiple Bit Upset
SET	Single Event Transient
SEL	Single Event Latch-up
XSCT	Xilinx Software Command-Line Tool
TCL	Transaction Control Language

Chapter 1

Introduction

The NASA Mars surveyor project aimed to study Mars with a number of robotic missions, including the Mars climate orbiter mission. The Mars climate orbiter was created to examine the seasonal fluctuations, dust storms, and water cycle of the planet as well as its climate and weather patterns.

The spaceship was sent into orbit on December 11, 1998, and on September 23, 1999, it began orbiting Mars. Nevertheless, a software malfunction led to a navigation error as the spacecraft approached Mars. Based on the spacecraft's speed and position, the engines were set up to fire for a particular amount of time. Even so, some of the software utilized the metric system, while other applications used the imperial units, to calculate these figures.

On September 23, 1999, the spacecraft hit the Martian atmosphere and was finally destroyed. Until a NASA Mars Climate Orbiter Mishap investigation board study, the precise reason for the spaceship's failure remained unknown. The committee eventually decided that a software flaw caused the issue.

The mistake happened as the spacecraft was approaching Mars and was around 60 miles (100 km) above the surface of the planet. When the engine fired, the spacecraft was traveling at a speed of around 1.5 miles per second (2.4 km/s), however, the wrong duration made it slow down far more than anticipated. Because of this, the spacecraft reached an improper orbit and approached the Martian atmosphere too closely, where it burnt up and was destroyed.

The lack to adapt to accepted software design and verification standards was found to be the primary contributor to the software issue. In particular, one team created the navigation system for the spaceship using metric units, while another team integrated the program into the spaceship using imperial units. The navigation issue was caused by the unit's incompatibility, which went undetected throughout testing and verification.

The incident served as a reminder of the significance of rigorous software testing and verification procedures as well as efficient communication and interaction across several teams involved in the preparation and execution of a space mission. It also led to adjustments to NASA's software validation and development processes, such

as the adoption of standardized software design approaches and the establishment of a different review panel to oversee software testing and verification.

In addition, the automotive industry is one in which fault tolerance is crucial to guaranteeing the safe and dependable running of automobiles. Modern cars are becoming more complicated and linked, thus it is crucial to develop and deploy systems that can manage mistakes, malfunctions, and breakdowns without affecting the car's general usefulness and safety. Electronic systems and control units are one area in the automobile industry where fault tolerance is essential. To operate a variety of tasks, including engine management, brake systems, entertainment systems, and advanced driver assistance systems (ADAS), modern automobiles primarily rely on electronic parts and software. These systems combine a large number of sensors, actuators, and sophisticated software algorithms to create a seamless and secure driving experience. Electronic devices, nonetheless are sometimes subject to faults or mistakes because of a variety of reasons, including component failures, electromagnetic interference, software problems, or even cyberattacks. Redundancy systems and fault-tolerant designs are used by vehicle manufacturers to lessen the effects of these errors. For instance, backup sensors or actuators that can take over in the event of a breakdown may be present in essential systems. To detect problems and fix them before they have a substantial impact or endanger safety, control units are frequently fitted with error detection and correction devices.

A system’s reliability is determined by how consistently it can carry out its planned function over time without failing or degrading. To ensure that a system can operate continuously and reliably, it must be designed a constructed using the proper techniques and processes, such as fault tolerance, redundancy, and maintenance which will explain in the following.

-**Fault Tolerance:** A system's fault tolerance is its capacity to continue operating effectively in the face of flaws, errors, or failures. A fault-tolerant system is created to identify and resolve such problems with a minimum negative effect on the system as a whole. Redundancy is frequently used to create fault tolerance, which entails replicating crucial components or adding backup systems that can seamlessly take over in the event of a breakdown. Additionally, techniques for error detection and correction are used to spot and minimize any faults or inconsistencies that could occur while the system is in use. Fault-tolerant systems enable continuous and dependable operation by integrating redundancy and error detection/correction mechanisms, which lowers the possibility of failures and their potential effects.

-**Redundancy:** Particularly in critical areas where errors can have serious repercussions, redundancy is essential to guaranteeing the reliability and accessibility of systems. Redundancy offers backup or alternate functioning in the case of a breakdown or failure by including additional pieces or processes. This strategy increases the system's overall resilience and lessens the effects of failures. Creating

multiple communication links, using backup power sources, or duplicating essential components are just a few examples of how redundancy may be done. Redundancy plays a crucial role in fault-tolerant systems to guarantee that the system can keep running even if a component or subsystem fails, minimizing downtime and preserving continuous operation. Organizations may increase reliability and resilience by using redundancy and eventually, provide a useful service that is more reliable.

-**Maintenance:** Regular inspections, fixes, and replacements are crucial for ensuring a system's long-term reliability. While reactive maintenance is carried out when necessary to address detected problems, scheduled maintenance can be arranged at certain periods. By resolving possible failure points and maintaining the system in top operating shape, maintenance aims to maintain the dependability of the system. Replace worn-out parts, updating software to fix bugs or vulnerabilities, and cleaning system components to avoid performance deterioration are just a few examples of chores that may be involved.

Incorporating suitable redundancy and backup systems is necessary to ensure reliability in addition to maintenance activities. In order to lessen the impact on system operation, redundancy offers backup components or processes that can take over in the case of a breakdown. Backup solutions, such as redundant data storage or backup power sources, add to system reliability by guaranteeing ongoing functioning even in the face of unforeseen circumstances.

Regular testing and maintenance are essential to preserving system reliability over time. Potential problems can be proactively handled by following a regular maintenance plan and carrying out exhaustive testing, enabling the early identification and correction of flaws. This proactive strategy lowers the likelihood of unanticipated failures and guarantees the system's dependability, minimizing downtime and increasing overall operational effectiveness. Therefore, to assuring the long-term dependability and performance of a system, a complete strategy that incorporates appropriate redundancy, trustworthy error detection and repair techniques, and regular maintenance is essential.

1.1 Motivation

The primary goal of this thesis, which falls under the category of reliability analysis, is to analyze and contrast the reliability of two platforms, one of which is a processor using an operating system and the other which is a processor without an operating system, so-called bare metal, especially in the context of safety-critical applications.

A complete experimental setup has been developed to look at the two systems' reliability further. The experiment involves the creation of the operating system and the two platforms, Baremetal. In addition, four programs have been chosen at random

to run on each platform. These programs will save their output so that it may be analyzed later. The experiment's next phase introduces a platform for injecting faults. By purposefully inserting a defect into the RAM where the application code is stored, this platform may mimic errors. Multiple fault injections are performed to ensure a wide variety of failure circumstances. The output of the apps as a result of each fault injection is noted and preserved. A separate platform for comparison is created in order to assess the effect of defects on application execution and compare the reliability of the platforms. This platform contrasts the output of the application execution, referred to as the Faulty-Outputs (obtained after fault injection), with the output of the application execution known as the Golden Outputs (obtained from the fault-free execution). It is feasible to comprehend the effectiveness of errors on each program and acquire insights into the general reliability of application execution on both platforms by studying and contrasting these outputs.

Chapter 2

State of the art

An integrated circuit's ability to function reliably and consistently over time and under diverse operating conditions is evaluated during reliability analysis. The reliability, efficiency, and lifetime of the circuit depend on this study. Evaluating probable breakdown mechanisms, such as electromigration[1], heat stress, voltage spikes, or material deterioration, is usually the first step in the procedure. These failure modes may result in decreased performance, sporadic malfunctions, or total circuit failure.

Identification and evaluation of potential failure modes that can have an impact on the performance of the circuit constitute the first stage in reliability analysis. For instance, the term "electromigration" describes the movement of atoms in the metallic interconnects of the integrated circuit (IC) as a result of the passage of high current densities, which can result in wire thinning and possible open circuits. Heat stress can lead to structural damage, mechanical stress, thermal expansion, and even thermal failure in the IC components. Voltage spikes, whether brief or prolonged, can cause electrical stress that can result in malfunctions or irreparable damage if they exceed the circuit's tolerance thresholds. The performance and overall dependability of the IC might be jeopardized by material degradation caused by aging effects or chemical interactions.

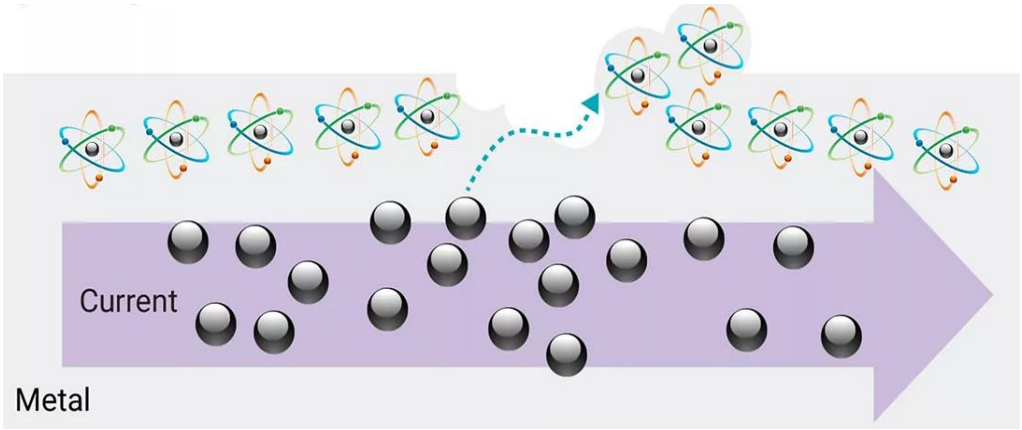

Figure 2. 1 Definition of the Electromigration[2]

Several methods are used to assess reliability. By submitting the circuit to rapid stress conditions, accelerated life testing (ALT) is frequently used to simulate and forecast the lifetime of the circuit. This aids in locating weak spots and determining the circuit's dependability when used normally.

As it seeks to identify the primary reason why circuit failures occur, failure analysis is a vital part of reliability analysis. To avoid future occurrences of failures and raise the circuit's general reliability, it is crucial to comprehend the underlying causes of each one. Failure analysis uses a variety of methodologies and procedures to identify the root cause and particular parts of a failure.[3]

Visual inspection of the failing circuit and its parts constitutes physical inspection. In order to find any obvious indicators of damage, such as burned or cracked components, solder connection flaws, or physical stress on the circuit board, this inspection may involve the use of microscope or other magnification equipment. To evaluate the electrical behavior of the failing circuit, electrical testing is carried out. Signal analysis, voltage measurements, current profiling, and other methods are used to spot anomalies, short circuits, open circuits, and performance deviations from expectations.

Another crucial stage of failure analysis is fault localization, which aims to identify the precise element or area at fault. Fault localization techniques include thermal imaging, X-ray imaging, and infrared analysis, which can assist locate hotspots, soldering flaws, and short circuits. In order to check the circuit at the microscopic level and find any tiny faults or damage, advanced methods like electron microscopy or scanning probe microscopy may also be used.[4]

There are numerous methods that may be used to check if a device meets the required radiation sensitivity requirements. The radiation test is the easiest to understand. In particular testing facilities, integrated circuits are exposed to controlled radiation dosages during radiation testing. These facilities consist of neutron generators, gamma irradiators, and particle accelerators. The gadgets are then observed and assessed for any modifications in functionality, performance, or reliability. Here is one example of an experiment using radiation testing in[5]. In this practical experiment, results gained from radiation test used to Programming the connectivity module, which is constructed on programmable circuitry, especially targets the configuration memory section. For a variety of applications and mitigation strategies, including hardware-accelerated designs and dynamic partial reconfiguration, this connector module is an essential component.

The biggest disadvantage of radiation testing is that it requires a lot of time and financial resources to complete. For it to be successfully completed, a very particular technological facility and months of planning are needed. These factors make a number of other strategies potential excellent compromises, avoiding the main

downsides of the radiation test. The radiation test may also be appropriately prepared using these techniques, increasing its usefulness and efficacy.

A useful technique in reliability analysis, simulation testing provides a cheap and accurate way to assess system behavior under diverse circumstances. Radiation testing is one area where simulation testing is frequently used. Electronic systems are particularly vulnerable to radiation dangers in the aerospace, nuclear, and space exploration industries. Engineers may evaluate the system's reaction to various radiation levels and kinds by simulating radiation impacts in a controlled environment without the requirement for costly and time-consuming real-world testing. An example of simulation could be find in [6].

Emulation, which focuses on modeling different defects that may arise as a result of environmental conditions particular to the operational sectors, is another testing approach used in reliability analysis in addition to simulation. Emulation goes beyond imitating a system's predicted characteristics and aims to mimic real-world fault events, such as those forced on by extremely high or low temperatures or other environmental stresses [7]. Emulation testing can simulate these circumstances in a controlled laboratory setting, for instance, in automotive applications where electronic components are subjected to temperature extremes and difficult working conditions. Engineers can evaluate the performance, reliability, and potential failure modes that may develop under such intense thermal stress by submitting the system to increased temperatures or thermal cycling. They can use this to create reliable heat management systems, pick appropriate materials, or arrange components optimally to increase the system's reliability and lifetime.

When preforming radiation testing, one the most probable faults which could occur is (SEE) and in this category of fault, the most probable one is (SEU), so-called as bitfilp, which usually preform on SoC memory. In[8], [9] and [10] you can see two examples of SEE.[11] is concentrated on creating a fault injection environment that is capable of analyzing the effects of errors on an ARM microprocessor integrated inside a Zynq-7000 AP-SoC, taking into account various fault models impacting the embedded processor's system memory and register assets. In addition, there is a fault injection platform which allows for the injection of a SEU into a randomly selected variable while a test program is running[12]. An alternative strategy has been used in the PROPANE injection tool[13], which is explained. When a specific breakpoint is reached, PROPANE's fault injector module seeks to halt the execution. The execution is then restarted when a part of the executable code that is fault-free is replaced with a defective one. Last but not least, preset variables and their contents are used to examine the execution results and the propagation of the error. Another example can be seen in [14], which is a time-based platform. In [15] A method for testing the resilience of neural networks utilizing hybrid platforms with programmable hardware. For the purpose of simulating the target hardware platform and carrying out the fault injection procedure, the technique depends on reconfigurable hardware.

Regarding the impacts of the SEE on embedded systems, the study given in this thesis follows all the research previously mentioned. It specifically examines the effects of the SEU injected into the main memory of the CPU while the application under study runs twice, once on a system running an operating system (FreeRTOS), and again on a system not running an operating system(Baremetal). The platform under consideration then permits as many injections as the user requests. Additionally, this platform gives us a classification of errors that occur as a result of the injection, together with a tracking of where they happened in the memory, at the very end. A reliability analysis could be performed by examining the error rate at the conclusion of the injection procedure on both systems.

Chapter 3

Background

3.1 Embedded Systems

Embedded systems are specialized systems for computing created to carry out particular functions and communicate with other machines or gadgets. They can be as simple as discrete devices, like a thermostat that regulates a room's temperature, or as complicated as modern autos' multifaceted management and control systems. Consumer electronics, automobile systems, medical gadgets, industrial machinery, and many other applications incorporate embedded systems. These systems frequently operate in resource-constrained contexts with unique needs, therefore they are typically tuned for efficiency, reliability, and real-time performance. Embedded systems enable automation, enhance functionality, and boost overall system performance in a variety of disciplines and industries by smoothly integrating with other hardware and software parts.[16]

Microcontrollers are essential parts of embedded systems because they offer small, low-power processing capabilities that may be customized for a variety of applications. They act as specialized hardware that gives embedded systems the ability to efficiently carry out particular tasks. Input/Output (I/O) ports, integrated memory, and other essential elements are frequently found in microcontrollers, allowing the system to communicate with the outside world and serve its intended function. These devices provide a balance between processing power, energy efficiency, and cost while being customized for the unique needs of the embedded system they are built into. The software that runs on microcontrollers can range from basic, specialized programs to very complicated and convoluted software structures, depending on the complexity of the work and the needs of the system. The software gives the microcontroller the ability to carry out the required operations, process data, and interact with other components, enabling the embedded system to perform successfully in the targeted application domain.

3.1.1 Key Features

The capability of embedded systems to function with little to no human input is one of their fundamental characteristics. Several embedded systems are created to run

autonomously with little to no human interaction. An industrial control system may be configured to automatically regulate the temperature and humidity levels in a plant, while a home automation system can be designed to turn on the lights at a specified time each day.

Furthermore, Real-Time performance is another capability of these systems. Many of these systems are designed to respond to real-time inputs, which implies changes fast and predictably. Real-time operating systems (RTOS), which are created to manage the real-time limitations of the system and guarantee that it functions successfully and efficiently, are used by many embedded devices to do this.

3.1.2 Pros & Cons

High performance, cheap cost, tiny size, low power consumption, and real-time processing are only a few benefits of embedded systems. These benefits are especially helpful in situations where real-time computing, low power consumption, and high reliability are requirements. High performance is one of the main advantages of these systems. As they are optimized for the unique purposes for which they were created, hence they are far more effective than general-purpose computers. Embedded systems frequently use specialized hardware and software that can swiftly and correctly carry out complicated computations.

Embedded systems, however, have a few disadvantages as well. Their restricted functioning is a serious drawback. They aren't normally intended to be general-purpose computers; instead, they are made to carry out specialized functions. In addition, It might take a lot of time and effort to develop embedded systems. It may be necessary to have particular knowledge and abilities to integrate hardware and software components, cope with time restrictions, and optimize the system for resource efficiency.

Moreover, developing embedded systems might be expensive upfront. So, compared to general-purpose computers, their usefulness is constrained.

3.1.3 Safety-Critical Applications

Applications that are considered to be "safety-critical" are those whose malfunction or failure might have severe repercussions, such as endangering human life, seriously harming the environment, or significantly increasing financial losses. These applications are often found in sectors including aviation, healthcare, automobiles, nuclear power, transportation, and defense where safety is of the utmost importance.

Safety-critical application design, development, and operation entail adherence to stringent safety standards, the use of redundancy, fault tolerance, and error detection methods, as well as thorough testing and verification to reduce the risk of failures or

errors. Additionally, the continued safety and dependability of these applications must be guaranteed by regular maintenance, monitoring, and mechanisms for continuous improvement.

In safety-critical applications, embedded systems must meet strict reliability, safety, and performance requirements. These systems must be designed and tested to ensure that they can operate correctly under a wide range of conditions and that they can handle unexpected inputs and events

3.2 Radiation

The emission or transfer of energy as electromagnetic waves or particles across a material is referred to as radiation. This energy can take the form of gamma rays, X-rays, ultraviolet radiation, visible light, infrared radiation, microwaves, or radio waves.

The phenomenon of radiation is one that occurs naturally all around us and has many different sources. Cosmic rays from space, solar radiation, and radioactive substances in the Earth's crust are a few examples of natural sources of radiation. X-ray devices, nuclear power plants, and some industrial operations are examples of man-made sources of radiation.[17]

Ionizing radiation is made up of particles with enough energy to deplete electrons from atoms or molecules, causing them to become ionized. As a result, ions and other charged particles like free radicals may be produced, which may harm biological tissues and other materials. Gamma rays, X-rays, and highly charged protons, neutrons, and alpha particles are a few examples of ionizing radiation. Although these kinds of radiation are frequently utilized in medical imaging and cancer therapy, they can also be created naturally by cosmic rays from space and radioactive elements in the earth's crust. On the other hand, non-ionizing radiation lacks the energy to ionize atoms or molecules. As an alternative, it communicates with matter via different methods including absorption, reflection, and transmission.[18]

3.2.1 Radiation Effects

Highly charged particles may be present when an electronic circuit is working in the hostile environments of space. An energetic ion may permanently or temporarily change the physical makeup of a section of an integrated circuit (IC), which might have an impact on how well it functions.

There is a study predicting the development of radiation –induced faults caused by earthly cosmic rays in the years to come. Contrary to popular belief, dangerous radiation environments are actually present here on the planet.[19]

Depending on the kind, intensity, and amount of radiation as well as the particular electronic component or system subjected to it, radiation impacts can appear in the electronic world in a variety of ways. The following are some typical traits of radiation impacts in electronics:

Total Ionizing Dose (TID): is the sum of all ionizing radiation doses that a system or device has ever received. TID can result in physical modifications to the components of the device, such as the development of traps and flaws, which might impair its functionality or result in long-term harm.[20]

Radiation-Induced Degradation (RID): RID stands for radiation-induced degradation, which is the deterioration of electronic systems or parts over time as a result of radiation exposure. Both ionizing and non-ionizing radiation can produce RID, which might show up as a rise in failure rate or a steady decline in performance.[21]

Radiation Hardness Assurance (RHA): Designing, evaluating, and certifying electronic systems and components to assure their functionality and dependability in radiation conditions is known as radiation hardness assurance (RHA). RHA entails assessing the radiation impacts on electronic systems and components and putting mitigation measures in place to lessen their influence.[22]

3.3 Fault Description

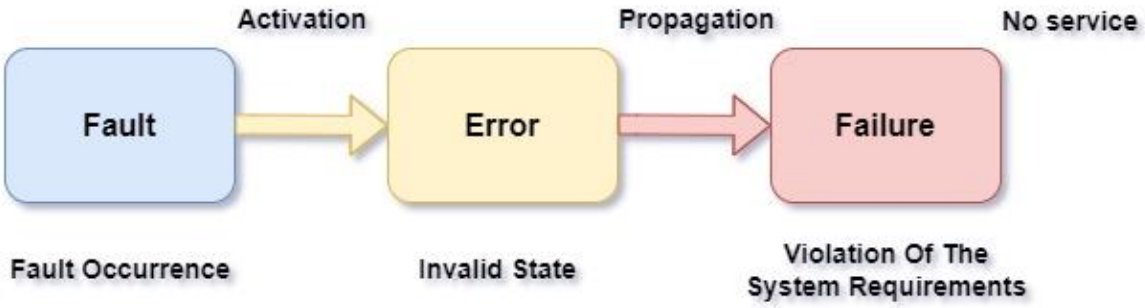

Figure 3. 1 Fault and its Relation with Error and Failure

A fault is a circumstance that arises abnormally or flawlessly that stops the system from working properly in the context of electronics and embedded systems. Physical harm, electrical component failure, software bugs, or environmental conditions like radiation or temperature are just a few of the causes of faults that might occur.

It is important to remember that faults can range from tiny bugs to serious system failures and are not always catastrophic disasters. The maintenance of system dependability, safety, and performance depends on the capacity to recognize and diagnose issues. Testing, monitoring, redundant systems, and error handling systems

are examples of fault detection and correction procedures that are frequently used to reduce the effects of failures and maintain system resilience.

Engineers may reduce the occurrence of faults and improve the overall resilience of electrical and embedded systems by comprehending the many sources of defects and putting into practice suitable design methods, assuring their optimal performance even under difficult operating situations.

3.3.1 Single Even Effect (SEE)

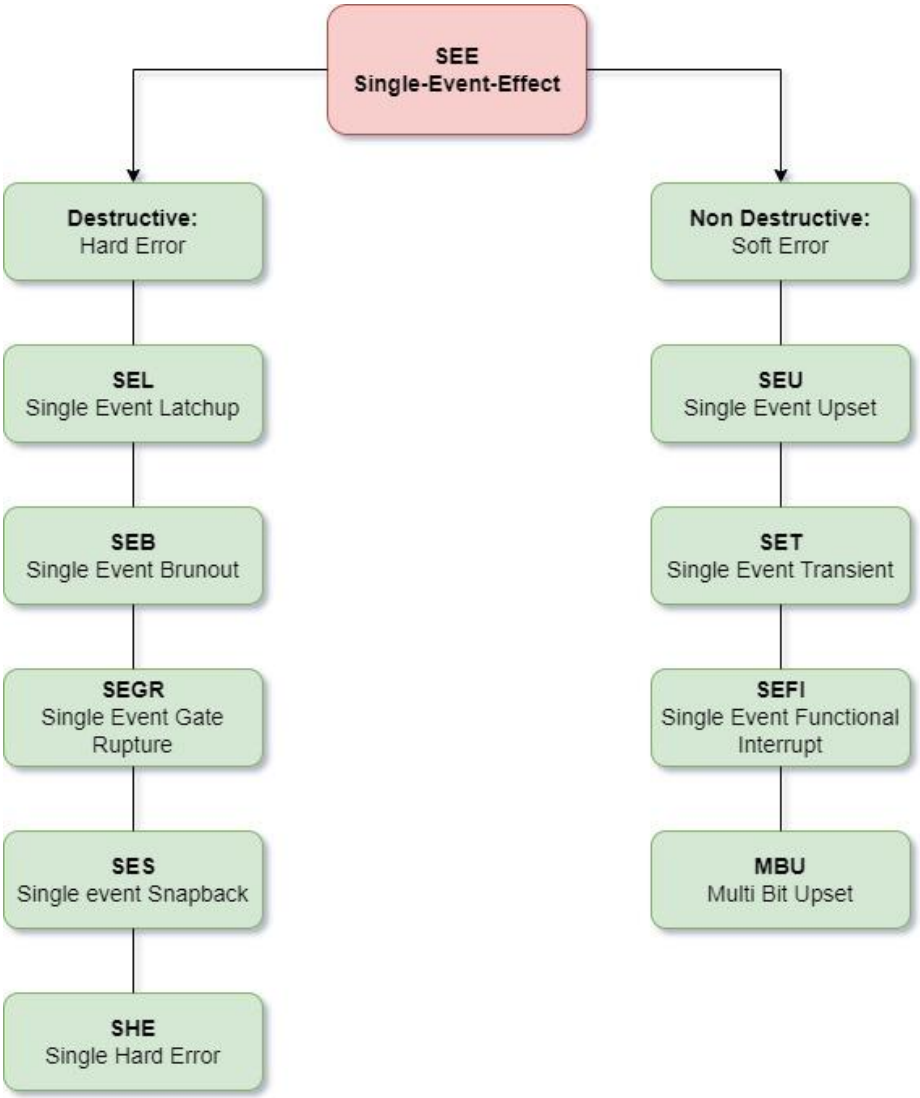

Figure 3. 2 Single Event Effect with its classification

Since electronic components are continuously getting smaller, high-performance microprocessors have been created that are even suited for safety-critical applications

where radiation-induced mistakes, such as the single event effect, are one of the most significant reliability challenges.

There are two basic classifications of fault models, which will be discussed in the following: Destructive faults and Non-Destructive faults. Although non-destructive faults, also known as soft errors, have undesirable consequences that vanish after a set amount of time or after the key cycle or the relevant components, destructive faults, also known as hard errors, irreversibly destroy the device itself. In the following, some types of single even effects will be discussed.

Single Event Upset (SEU)

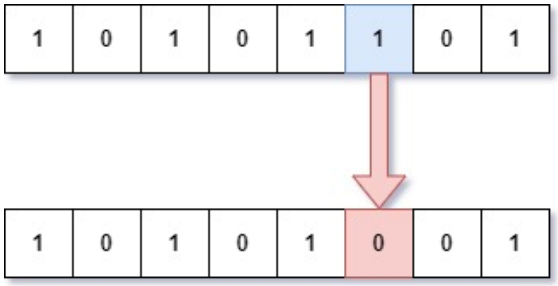

Figure 3. 3 SEU in the memory

A modification in the state brought on by a single ionizing particle (electrons, photons, …) impacts a susceptible node in devices such as a microprocessor, or memories. The free charge generated by ionization at or near an essential node of a logic element causes the state change (Bit-Flip, change 0 to 1 or 1 to 0). An SEU which is also a soft error is a strike-related fault that affects the output or functionality of the related device.[23][24]

Single Event Functional Interrupt (SEFI)

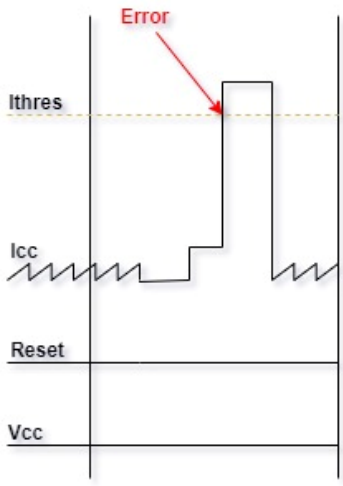

Figure 3. 4 SEFI indication using hardware reset

The sensitive areas of a digital circuit can be affected by high-energy ionizing particles, which can deposit charge and produce electron-hole pairs. A brief or momentary functional stoppage may result from this charge deposition interfering with the circuit's regular operation. Inaccurate data processing, software crashes, or transient system failures are only a few faults or unanticipated behaviors that the interruption may result in. [25][26]

Single Event Latch-Up (SEL)

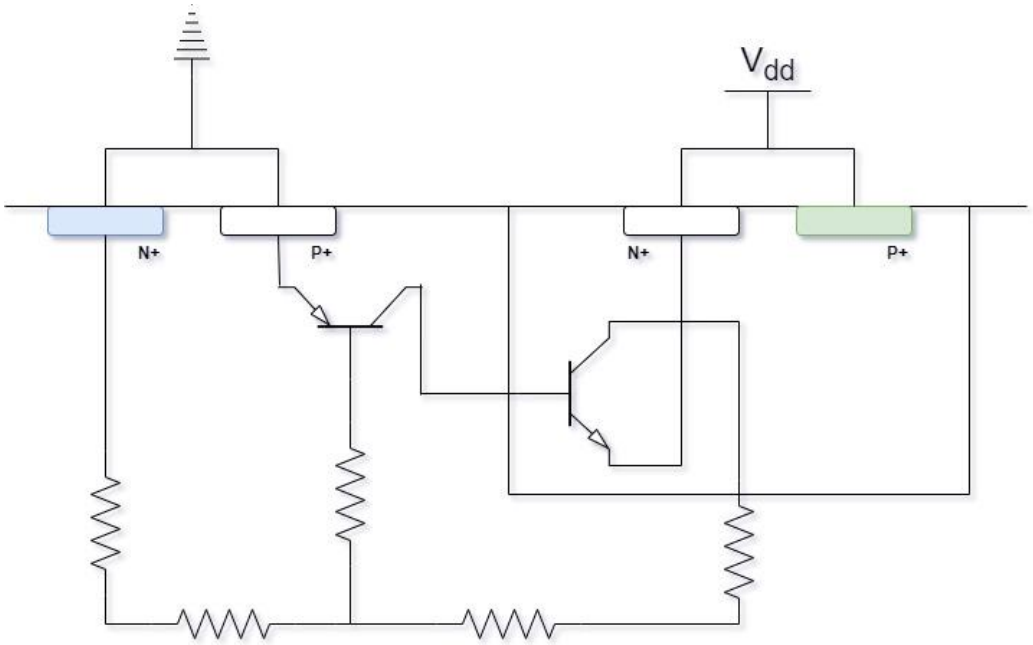

Figure 3. 5 Generation of single event latch-up effect

A malfunctioning device condition is characterized by an abnormally high current induced by the passage of a single energetic particle across vulnerable areas of the device's structure, which could harm the device permanently.

It is occurring under specific circumstances, primarily due to environmental radiation, which causes one of the many silicon PNPN structures to flip from their blocking state to a latched state, which causes the circuitry to misbehave and frequently causes a power supply or ground link to fuse.[27]

Single Event Gate Rupture (SEGR)

Failure of power MOSFETs in space may result from single-event gate rupturing. When a heavy ion hits the device's neck area, the SEGR process begins. The region among the surface p-body diffusions is known as the neck zone. An electron-Hole

pair filament is produced by the impact of the ions. A positive drain bias creates an electric field that causes the produced holes and electrons in an n-channel power MOSFET to move toward the interface and the drain contact, respectively [28]. Engineers may improve the performance and reliability of power MOSFETs in space applications by comprehending the processes involved in SEGR and putting the right safeguards in place, assuring the resilience of electronic systems working in challenging conditions.

Single Event Transient (SET)

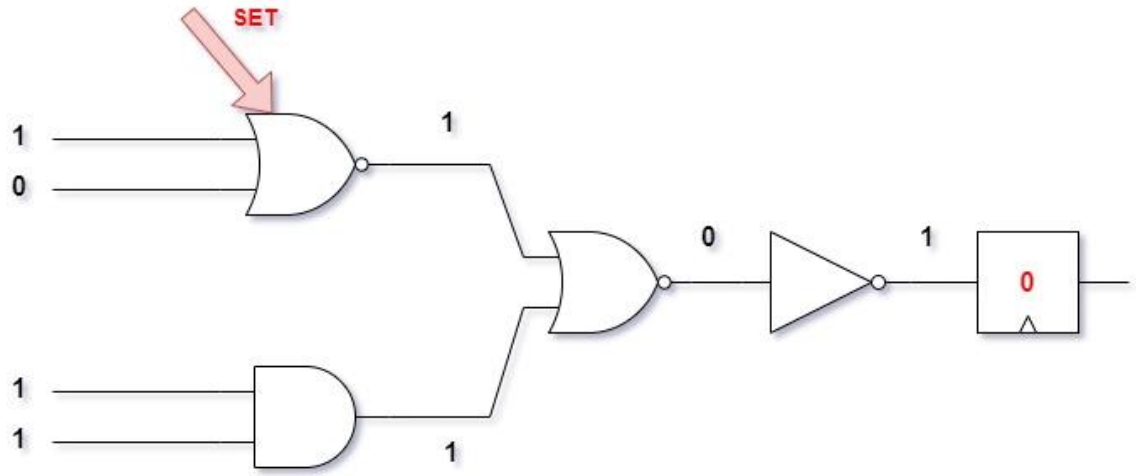

Figure 3. 6 Single Event Transient impact on a logic circuit

When an energetic subatomic particle comes into contact with a component, single-event transients take place. The particle's charge results in a brief voltage disturbance that can latch onto a storage element and trigger a single event upset. The likelihood that an SEU will result from a set is significantly influenced by the logic design approach, storage element behavior, and system timing constraints. Circuit modeling and heavy ion testing of prototype devices are used to investigate these effects.[29]

Single Event Burnout (SEB)

A substantial quantity of charge is deposited in the silicon-based power device's sensitive areas, such as the gate oxide, when a high-energy ionizing particle, such as an energetic proton or alpha particle, impacts the component. Single Event Burnout, which is caused by a high current flowing through the device as a result of this charge deposition, can result in a rapid, catastrophic collapse of the device. The silicon lattice's ionization, which produces electron-hole pairs and causes an electron-hole plasma to develop, is what causes the breakdown. A conductive channel is then created by the plasma, allowing a significant current to pass through the apparatus. If

the device's high current is not adequately regulated, it might quickly become hot and eventually fail.

Different design strategies may be used to reduce the risk of SEB, including radiation-hardened components, ionizing radiation shielding for the equipment, and redundancy and fault-tolerant designs. To analyze the vulnerability of power devices to SEB and gauge their performance in radiation-rich situations, modeling, and testing methods are also employed.[30]

3.4 Hardware Background

Electronic gadgets have become more complicated and advanced thanks to modern technology, which necessitates hardware with great performance and low power consumption. Electronic device design must take into account power consumption, and optimizing the hardware is crucial to achieving maximum power effectiveness.

In the past, discrete components were used to build electronic systems. These parts were different, independent entities that carried out particular tasks for the system. Individual chips or components created for particular purposes, such as amplification, signal processing, logic operations, or memory storage, were used to accomplish a variety of electronic duties. On a printed circuit board (PCB), these discrete components were physically joined and interconnected using a variety of techniques, such as soldering or wire bonding. With traces or conductive pathways etched onto the board's surface to simplify the passage of electrical signals between the components, the PCB served as a platform for mounting and arranging the components. Different components might be chosen and combined to form electronic systems suited to certain applications, allowing for flexibility and customisation. However, compared to the integrated approach provided by contemporary integrated circuits and microcontrollers, this strategy frequently led to bigger system sizes, greater power consumption, and complicated interconnections, making it less effective.

The system-on-chip (SoC) architecture is one approach that has been developed to fulfill this demand for streamlined hardware. Several hardware elements, including microprocessors, memory, communication interfaces, and other peripherals, are combined onto a single chip in an SoC, a form of integrated circuit. The device’s overall power consumption is decreased because of the integration of the components, which allows for more effective use of space and power [31].although all the benefits of SoC, there are also some drawbacks. In[32], a new method to improve the reliability of SoC could be seen.

Several methods, like power gating, clock gating, dynamic voltage and frequency scaling, and low-power circuit design, can be utilized to reduce the power consumption of SoCs. By using these methods, SoCs may run at lower power levels

without compromising performance. Software optimization is just as important for SoCs' energy-efficient functioning as hardware optimization is. To achieve maximum power efficiency, it is essential to build software that is optimized for low power consumption and effective utilization of hardware resources.

It takes knowledge of several different fields, including digital and analog circuit design, system architecture, software development, and verification/validation methods, to build and construct a SoC. SoC design often entails the integration and customization of pre-designed intellectual property (IP) blocks with custom-designed components to satisfy application-specific needs.

3.4.1 ZYNQ-7000

A high-performance processing system (PS) and programmable logic (PL) fabric are combined on a single chip by the Zynq-7000 series of Xilinx System on Chip (SoC) devices. The ARM Cortex-A9-based Zynq-7000 series of processors come in a variety of configurations with varying degrees of processing power and programmable logic resources.

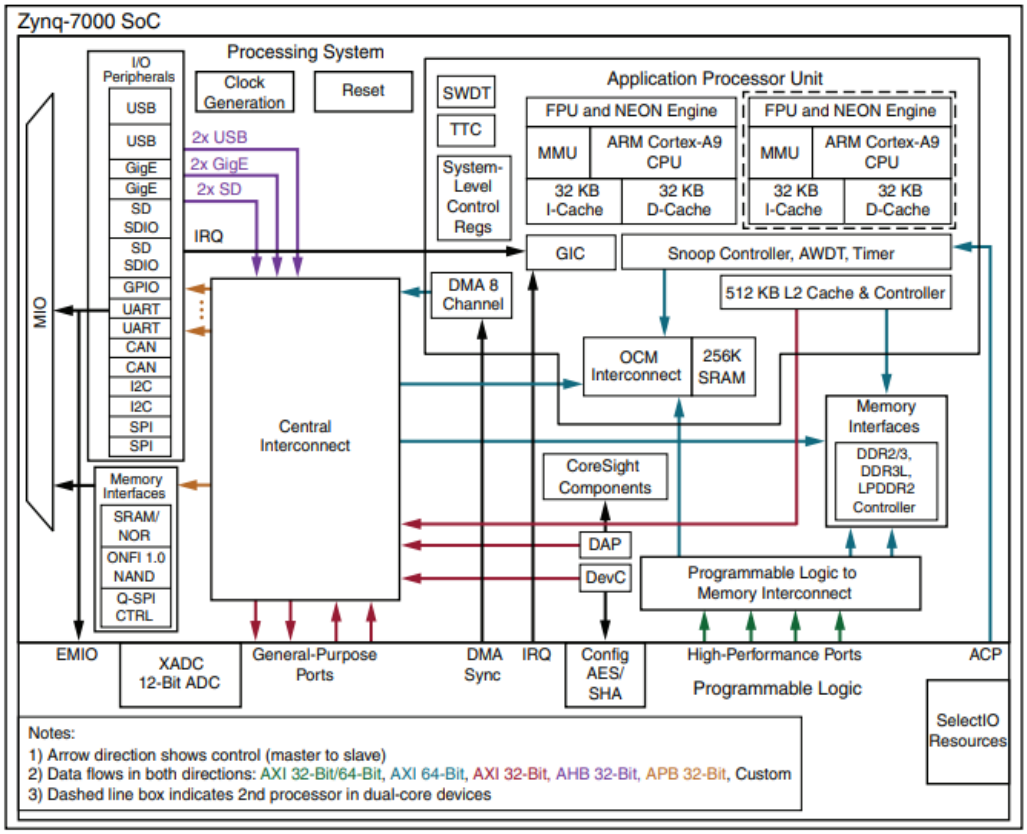

Figure 3. 7 Architecture of ZYNQ-7000 [33]

The two primary series of the Zynq-7000 family are the Zynq-7000 AP SoC and the Zynq-7000 MP SoC. The AP SoC is targeted for applications that demand high levels of programmable logic resources and I/O capabilities, whereas the MP SoC is optimized for applications that require high levels of real-time processing and high-speed processing.

A dual-core ARM Cortex-A9 processor with a maximum clock speed of 1 GHz, on-chip memory with a Level 1 cache and a Level 2 cache, on-chip peripherals like DMA, timers, and interrupt controllers, as well as a variety of industry-standard interfaces like USB, Ethernet, and SDIO make up the PS component in the Zynq-7000. Several operating systems, including Linux, FreeRTOS, and bare-metal firmware, can be used with the PS.[34]

A field-programmable gate array (FPGA) fabric known as the PL component of the Zynq-7000 enables designers to construct unique logic circuits, interfaces, and accelerators. The PL is a programmable logic array made up of DSP blocks, customizable I/O blocks, and programmable logic cells that may be modified to implement specific hardware capabilities. High-level design languages like Verilog or VHDL, as well as high-level synthesis tools like Vivado HLS, can be used to program the PL.

On the Zynq-7000 SoCs, the CPU is surrounded by blocks that include a 512KB level 2 cache and a 256KB On-Chip Memory. This final point might be especially helpful when translating logical addresses. Additionally, there are two level 1 32KB cache memories for data and instructions inside the CPU.

Other integrated peripherals included on the Zynq-7000 SoC include Gigabit Ethernet MAC, USB, UART, CAN, SPI, and I2C interfaces. The Advanced eXtensible Interface (AXI), a high-bandwidth connection included in the Zynq-7000, links the PS and PL components and enables effective data transmission and communication between them. Moreover, it is a component of the ARM AMBA standard and has an on-chip communication interface with a high-speed multiple-master multiple-slave architecture.[35]

The Zynq-7000 SoC's ARM Cortex-A9 CPU has an MMU that supports, safeguards and caches virtual memory. It permits the parallel operation of numerous processes, safeguards memory from illegal access, and provides for effective memory usage and caching.[36]

This SoC is frequently utilized in industries. Examples include robots and space applications, as well as industrial automation, vision in computers, medical devices, and high-resolution media processing.

3.4.2 PYNQ-Z2 Board

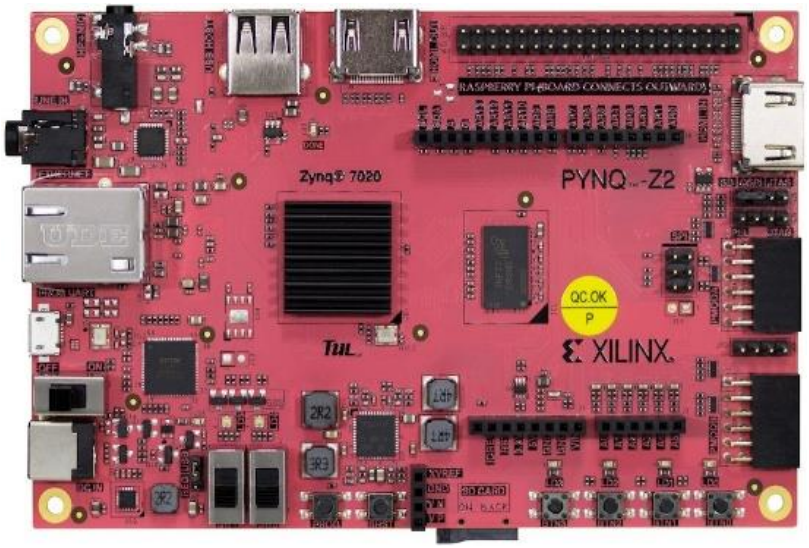

Figure 3. 8 ZYNQ-Z2 Board [37]

The PYNQ-Z2 board, created in partnership between Xilinx and Digilent, aims to give developers an accessible and cheap platform for building embedded systems. It is based on the Zynq-7000 SoC, which integrates an FPGA fabric with an ARM Cortex-A9 dual-core CPU. This CPU and FPGA combo enables the development of both software and hardware components on a unified platform.

The device has a 1.3 million logic cell, 53,200 slices, and 120 DSP slice Xilinx XC7Z020-1CLG400C FPGA. The Vivado Design Suite from Xilinx, a potent and adaptable design environment that supports a variety of development processes, may be used to program the FPGA. The PYNQ-Z2 board contains an FPGA, 512MB DDR3 memory, 16MB quad SPI Flash, and a microSD card connector for extra storage in addition to the FPGA. A variety of connectivity choices are also available on the board, including Gigabit Ethernet, USB 2.0, HDMI input and output, and Pmod connectors. This allows interacting with other gadgets and sensors simple.[38][37]

Support for the PYNQ (Python Productivity for Zynq) framework is one of the PYNQ-Z2 board's standout characteristics. An open-source project called PYNQ offers a Python environment for use with the Zynq SoC. As a result, software writers find it simple to design programs that communicate with the board's physical components. Moreover, PYNQ offers several pre-built overlays, which seem to be hardware accelerators that may be utilized to accelerate particular calculations.

Chapter 4

Fault Injection Platform

This thesis's goal, as we've already established, is to compare the reliability of application execution on a processor with and without an operating system. A fault injection platform, which is discussed in this chapter, will be created with that goal in mind. It needs an executable program SW as input data. The SEU fault model, sometimes referred to as bit flip, on the processor system's SRAM has been chosen for this investigation. The intended outcome will be utilized as an investigation for the reliability study. This fault model will be used on both systems, the one with the operating system (FreeRTOS) and the one without an operating system (Baremetal). SEU will be injected into PS's main memory while a program is running. The findings that are presented will be categorized at the conclusion of the procedure.

4.1 SEU In Memory

As previously mentioned, the PYNQ-Z2 board is the device utilized for this project.

The Pynq-Z2 board's main memory is prone to single event upsets (SEUs), much like any other electrical equipment. When ionized particles from radiant energy or other sources impact the sensitive parts of electronic equipment, they can produce SEUs, which can result in short-term or long-term behavioral changes in the affected electronics.

An adaptable and strong platform featuring a dual-core ARM Cortex-A9 CPU and programmable logic fabric is the Xilinx Zynq-7000 System on Chip (SoC). The flexibility of software and the power of hardware's parallel processing are combined through this integration to enable a wide range of applications.

The main system memory utilized to run programs is one of the most important parts of the Zynq-7000 SoC, and consequently, the Pynq-Z2 board. Here, DDR3 RAM (Double Data Rate 3 Random Access Memory) serves as the primary system memory. DDR3 RAM is a frequently utilized form of memory because of its very high bandwidth and capacity.A bit flip in the data stored in the DDR3 RAM might result from an SEU in the Pynq-Z2 board's main memory and create problems or flaws in the system's functionality. For instance, it could modify a few numbers or stop the application from running.

4.2 Injection Platform

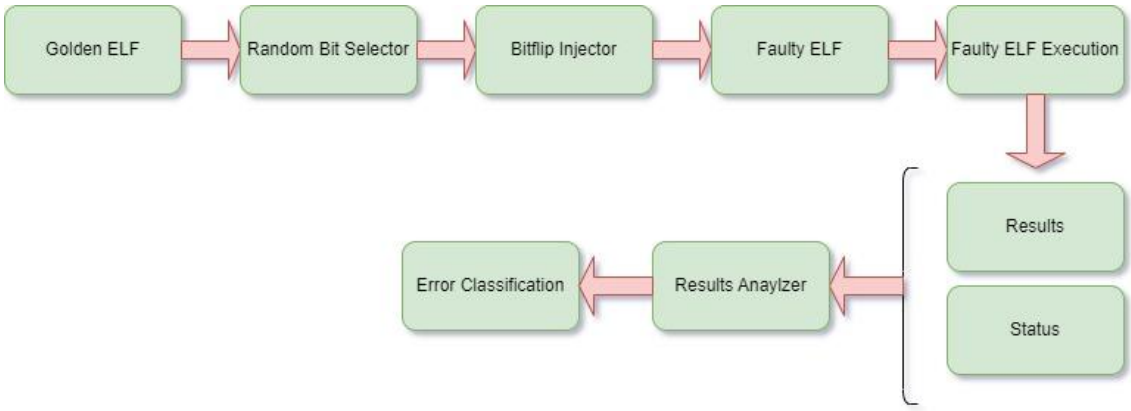

Figure 4. 1 Fault Injection Platform Diagram

The fault injection platform's objective is to provide a platform that automatically injects an SEU during program execution and gathers the results for the final comparison and categorization. It is intended to have a platform that runs the program normally initially before saving the output as error-free output (golden output). The next step is to run the program again, but this time set breakpoints at certain addresses obtained by reading the program counter step by step, and use these addresses as the time intervals at which the breakpoints should be enabled. In this manner, every time an application's execution reaches one of these breakpoints, it is stopped. The value is then read into the required address, and the ELF file is made flawed by injecting a bit flip into a random bit of its binary. Upon injection, the application's execution will continue, and the output will be stored as erroneous output at the conclusion. The output will be saved in a particular text file as many times as this process needs to be repeated. All of these flawed outputs will ultimately be compared against the golden output, and the results will be classified. Keep in mind that Python 3 and its connection to the ZYNQ board through the serial communication protocol embedded into the ZYNQ board—in this example, UART—perform all these actions and updates automatically. The following sections go into further depth about each phase.

4.2.1 Fault Injection Technique

A number of methods, such as single event upset (SEU), single event Latchup (SEL), and other models, can create errors in memory. SEU served as the fault model for injection to both platforms in this project. In this case, both systems will be examined by SEU separately.

4.2.2 Determine Target Memory

The choice of where to inject the fault is crucial. On-chip memory or off-chip memory might be the injection target. The location to inject the fault should be decided once the memory has been specified. That implies that I should select a specific address or region in the memory where I wish to inject the fault. The reason for this aim is that a use injecting addresses with application code already present in them is more efficient. In other words, if injection occurs at empty memory locations, the outcome could not be affected. The main memory on the PYNQ-Z2 board is DDR3 RAM which has all of the executable files put into it. In Figure 4.1, addresses of each part of memory that the program binary will load into are shown (note that these addresses are different for each application).

I included a random address generator in the injection platform, which randomly selects a memory address from among all the addresses in the SRAM memory and uses that address to inject SEU into it. As the application code was loaded between 0x00000000 and 0x0000f36f, the random address generator chose a random address within this range to inject into.

```
section, .text:        0x00000000 - 0x000088df
section, .init:        0x000088e0 - 0x000088eb
section, .fini:        0x000088ec - 0x000088f7
section, .rodata:      0x000088f8 - 0x00008d56
section, .data:        0x00008d58 - 0x00009743
section, .eh_frame:    0x00009744 - 0x00009747
section, .mmu_tbl:     0x0000c000 - 0x0000ffff
section, .init_array: 0x00010000 - 0x00010003
section, .fini_array: 0x00010004 - 0x00010007
section, .bss:         0x00010008 - 0x0001006f
section, .heap:        0x00010070 - 0x0001206f
section, .stack:       0x00012070 - 0x0001586f
```

Figure 4. 2 memory sections with the desired addresses of each part

4.2.3 Random Bit Generator and Injection

It is now time to begin the injection operation after choosing the memory and memory address for the injection. Of course, the address location selected for injection has value. I developed a random bit generator that reads the value of the selected address and randomly selects one of the 32 bits of that value. The selected bit will be the bit for the SEU injection. So its value will change from 0 to 1 or 1 to 0.

4.2.3 Monitor The Result

When the injection operation is complete, the program will continue to run while the desired output is saved into a text file. The procedure will run for as long as the user specifies. In the end, all the faulty outputs produced by the platform will be compared with the golden ones.

4.2.4 Code Explanation

As has been explained before, the fault injection platform has been provided by python3 which runs all the injection processes automatically and saves the result for further classification.

To control the Xilinx board by Python, first, we have to access the XSCT which stands for Xilinx Software Command-Line Tool. This could be done by using a Python library called Pyserial which setup a serial connection for several types of devices. In Xsct, commands are written in TCL. We can create TCL commands and call them from a python script. There are some steps to run the application with the Xsct usingTCL command, which will be discussed in the following.

The first step is to configure the hardware programming part. To do this, select the path where the application will be executed and connect the Xilinx server program with the JTAG port. This could be done with the "Connect" command. Then using the "Targets" command all the devices connected to the JTAG port will be listed. Then FPGA bit stream which was provided by Vivado before, has to be downloaded.

Once hardware programming is finished, the software part is started. The first thing that has to be done is to target a processor core to run the software, In our case, ARM Cortex-A9. Then we have to initialize different parts of PS. This is done using the "Source ps7_Init.tcl" command followed by "ps7_init" and "ps7_post_config". The last command is used to provide hardware with the clock. After initialization, it's time to download the ELF file. The command to download the ELF is "dow <elf name>.elf". after this command, the ELF file is downloaded but the processor does not start to execute it and stops and the processor will be in a suspended state. To run the processor, the "con" command has to be issued and it will start to execute from the address it suspended.

After starting the execution, the idea is to stop the execution and inject the fault. To do this, some breakpoints will be added to the program on some addresses, which have been chosen randomly. "bpadd -addr " is the TCL command to put a breakpoint at a certain address. These breakpoints will be enabled using "bpenable" command. Then there is a random address generator which every time execution stopped at the breakpoint, chooses a random address from memory. Bitflip injection starts after

choosing the random address. The value in the random address will be read, 1 bit of this value will be chosen randomly, and SEU will be injected into this random bit which changes it either to 1 or 0. Then the breakpoint will be removed using "bpremove" command and the processor will continue to executant with "con" command. This algorithm will be repeated a number of times as requested by the user. At each time of execution, the result will be saved on a text file.

4.3 Output Classification

The project's results will be examined in this section of the thesis. I created a platform that, at the end of the test operating period, gathers all the findings generated by the fault injection platform, compared these results with the non-faulty ones (one-time execution of application without injecting bit flip), the so-called golden output, and then categorizes and classifies the results gained after comparison. In order to do this, many categories will be mentioned and explained in the sections that follow.

4.3.1 Silent Data Corruption (SDC)

An anomaly in data transfer or storage that goes unnoticed by the system or application and may cause data loss or corruption is referred to as silent data corruption (SDC), where the computation result differs from what was anticipated. As the system might not notice the fault or notify the user of it, the user could not be aware of the issue until it poses a major problem, which is why it is dubbed a "silent" error.[39]

SDC may manifest itself in a variety of ways, including bit flips, memory problems, and network issues. When a single bit in a binary file is switched from 1 to 0 or vice versa, this is known as a "bit flip," because it may alter the data's meaning. When a bit of memory flips, memory errors can happen, resulting in improper data read or write operations. When data is transported through a network but becomes damaged in the process, network faults can happen.

Golden matrix:

$$\begin{bmatrix}1 & 1\\ 2 & 2\end{bmatrix}\times\begin{bmatrix}1 & 1\\ 2 & 2\end{bmatrix}=\begin{bmatrix}3 & 3\\ 6 & 6\end{bmatrix}$$

SDC occurrence after fault injection:

$$\begin{bmatrix}1 & 1\\ 2 & 2\end{bmatrix}\times\begin{bmatrix}-5546967 & -1704372\\ -1360752 & 2138963\end{bmatrix}=\begin{bmatrix}-6907719 & 1968526\\ -1381543 & -3579159\end{bmatrix}$$

You can see an example of SDC occurrence, created as the result of bit flip injection in the memory. The matrices above indicate a multiplication of two identical matrices, which should have the same elements. After fault injection, matrix two is affected by the fault, and its value changes.

4.3.2 Halt

Stop in application execution describes a circumstance in which an application freezes or stops operating, frequently as a result of an unforeseen error or problem. The user may need to force stop or restart the application once it halts since it may become unresponsive to user interaction.

A halt in the running of a program can be brought on by a number of things, including hardware issues, resource conflicts, or code flaws. A program could stall or crash, for instance, if it attempts to access a resource that is no longer accessible, such as a file or network connection. Similarly to this, an application may cease or stop responding if it takes up all of the memory or processing resources that are available.

4.3.3 Empty Files

This occurs when the impact of an error prevents sending the necessary data to the output, leaving the file with no output and the classification platform returning empty files.

In addition to these categories, I offer another classification of memory to determine precisely which parts of memory will have more faults. In another word, to observe the fraction of the fault's propagation over several memory parts. Each memory sector has a unique address, as seen in Figure 4.2, which shows memory sections. We can categorize the occurrence of faults in each sector of memory with ease if we know the start and end addresses of each section. We can specifically track errors and see how frequently they occurred in each part of memory.

Chapter 5

Experimental Analysis

This section includes the Application selection for the fault injection process and the results gained by the injection platform on both Baremetal and FreeRTOS. By comparing and analyzing the experimental results given in this chapter, we can reach a conclusion for the reliability analysis in both platforms.

In addition, there are also some other experimental results, which are the comparison of application execution time, SoC temperature, and the Vcc in the selected applications to observe how these elements change during execution in both bare-metal and FreeRTOS platforms.

5.1 Application Selection

I selected four applications for the injection process in this experiment. Two of them were chosen from the MiBench benchmark, automotive subcategory (basicmath and qsort). The other two applications are Dhrystone and Matrix multiplier. MiBench benchmark chosen because of its importance in safety-critical applications as it has automotive benchmarks inside it. The other applications were chosen as they have mathematical operations to test the system's reliability over hard calculations. Each of these applications will be explained in the paragraphs that follow.

MiBench

MiBench benchmark suite is a popular benchmarking tool used to assess the effectiveness of embedded systems. It is made up of a collection of small to medium-sized applications that mimic typical embedded system workloads. These C-coded applications have a variety of application areas, such as those in the automobile, telecommunications, consumer electronics, and network security industries.[40]

Basicmath

Basicmath is one of the applications that exist in the MiBench benchmark suite with a focus on the measurement of the efficiency of mathematical operations on embedded systems. It is made up of a collection of algorithms that operate on arrays of set-point integers to add, subtract, multiply, divide, and take the square root of numbers.

Qsort

Qsort is another application in the MiBench benchmark suite and is widely used as a sorting algorithm on embedded devices. It is an accurate representation of real-world applications, which utilize for sorting huge quantities of data. This application is frequently employed to evaluate how well sorting algorithms perform across many platforms because it is made to be modular and operates on various embedded devices.

Dhrystone

This benchmark counts the instances of a group of C expressions that represent common operations in practical applications. These operations could be integer arithmetic and memory access. It returns DMIPS as the result, which is Dhrystones per second and is a measurement of the number of actions that the machine can carry out in a single second. One of the cons that this benchmark has is, it could not be used for contemporary computer workloads, since it excludes Floating-point operations. Nevertheless, it is one of the important benchmarks for assessing performance. [41]

Matrix Multiplier

As it is obvious from the name of this benchmark, the matrix multiplier benchmark consists of the multiplication of matrixes. This benchmark has the possibility to do different matrixes with different sizes, and it has a variety of algorithms to perform different types of measurements (real, complex, and so on).

5.2 Experiment Result

A reliability examination of two systems is what this thesis's goal is. This section conducts an in-depth, experimental investigation of each platform. A thorough discussion on how reliable each platform will be possible thanks to the reliability analysis, which will be built on the gathered data from the experiment. A thorough evaluation of the platforms' reliability performance and suitableness for safety-critical applications may be made by evaluating and contrasting the experimental results. This will provide important insights into how well the platforms perform in terms of reliability.

As we saw in section 5.1, four applications are considered, as applications under test, and for the fault model, I chose SEU. In this way, there will be four experimental results for each platform. In other words, once we have an experiment on each of the four applications in the Baremetal Platform, there will be the same experiment on these applications in the FreeRTOS platform. Then as I share with you in the following, there are some tables to show the result of these experiments.

5.2.1 Baremetal platform

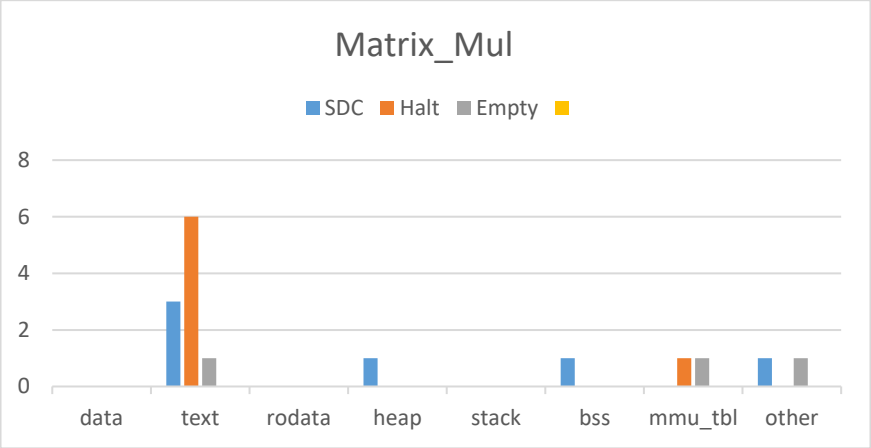

Figure 5. 1 Matrix_Mul experimental results (Baremetal)

The figure provides a view of the application's tendency to experience execution halts, showing that these halts most frequently occur in the text area of memory. This suggests that the majority of execution halts may be caused by problems with code execution, such as crashes, infinite loops, or exceptions. The figure also shows that SDC errors are the second most common type of error seen.

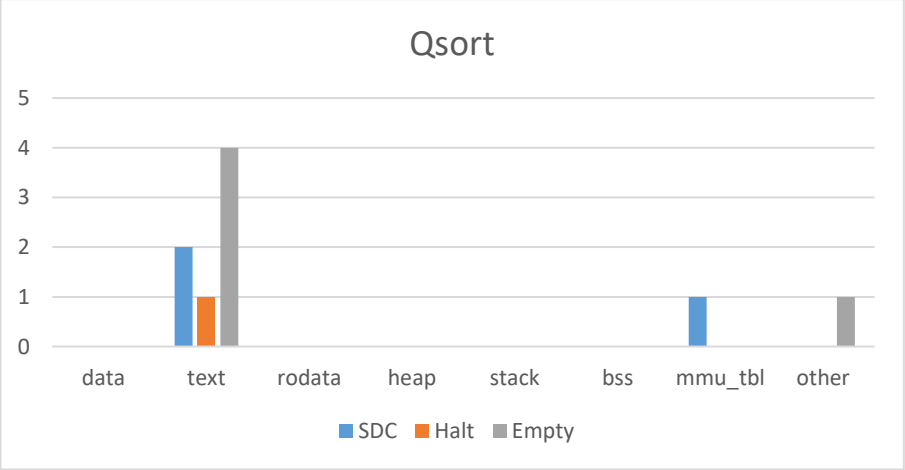

Figure 5. 2 Qsort experimental results (Baremetal)

The most frequent errors seen when using the Qsort application are empty files. According to the figure provided, these mistakes are primarily found in the text section of the memory. The presence of empty files indicates possible problems with the application's input/output or file-handling functions.

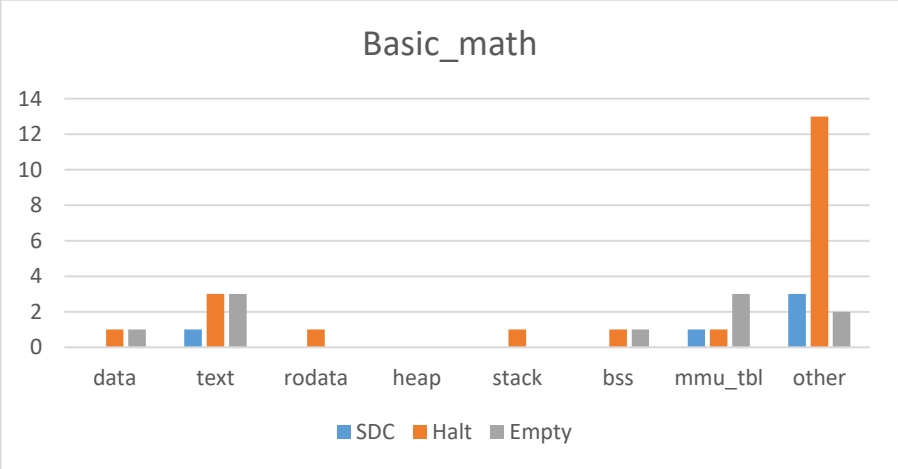

Figure 5. 3 Basic_math experimental results (Baremetal)

Based on the presented graphic, it is clear that the majority of halts in the basic_math application take place in a particular area that is outside the addresses stored in the SRAM. This shows that causes other than the SRAM itself, such as external dependencies, input/output processes, or computational logic situated outside the SRAM module, may be the cause of these halts.

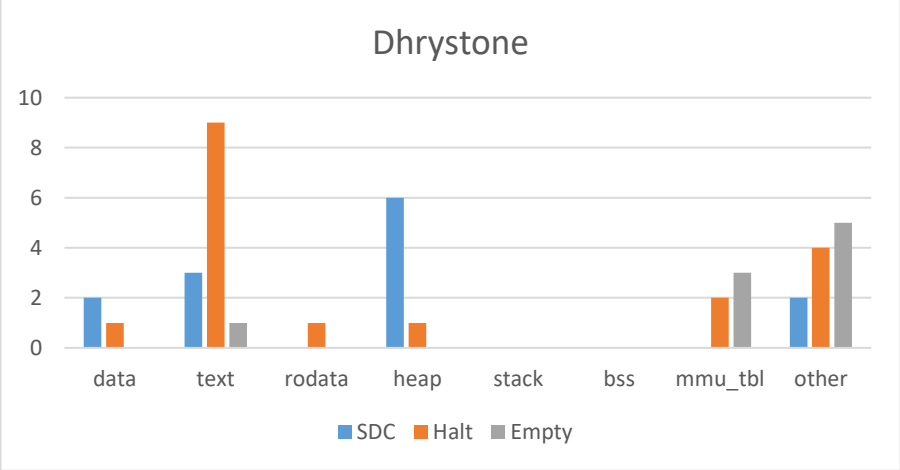

Figure 5. 4 Dhrystone experimental results (Baremetal)

The Dhrystone application displayed a higher rate of SDC (Setup and Hold Violation) errors than other Baremetal platform programs. According to the provided figure, the heap and text sections of memory held an excessive quantity of these errors. This implies that the SDC errors seen in the Dhrystone application may have been caused by the memory management and code execution within these sections.

5.2.2 FreeRTOS Platform

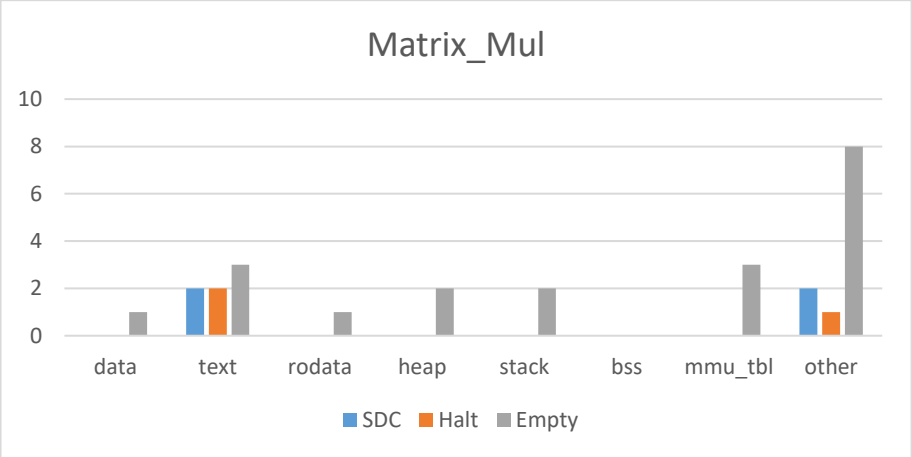

Figure 5. 5 Matrix_Mul experimental results (FreeRTOS)

In the FreeRTOS platform, the results are different with respect to the Baremetal one. Here as we see, an empty file is listed as the most error which is occurred and it mostly happens in the other section.

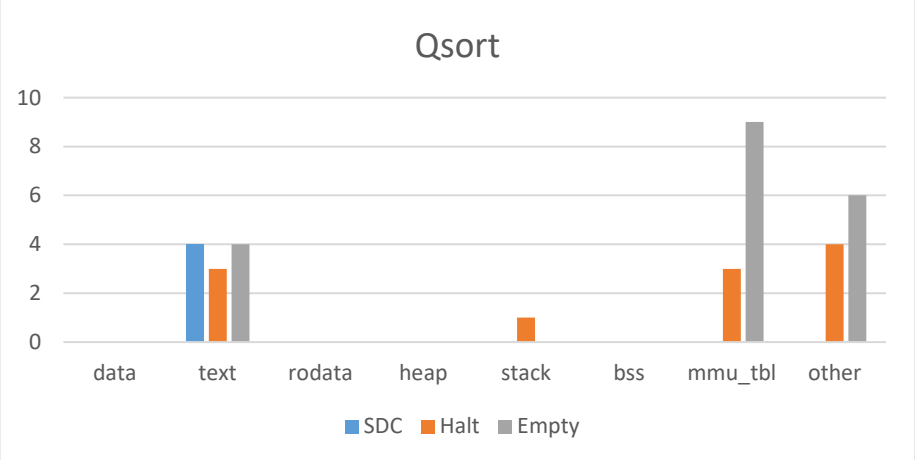

Figure 5. 6 Qsort experimental results (FreeRTOS)

For Qsort application, there is a sharp rise in facing with errors in mmu_tbl and other sections of memory with respect to the Matrix_mul, While SDC is only happened in the text part.

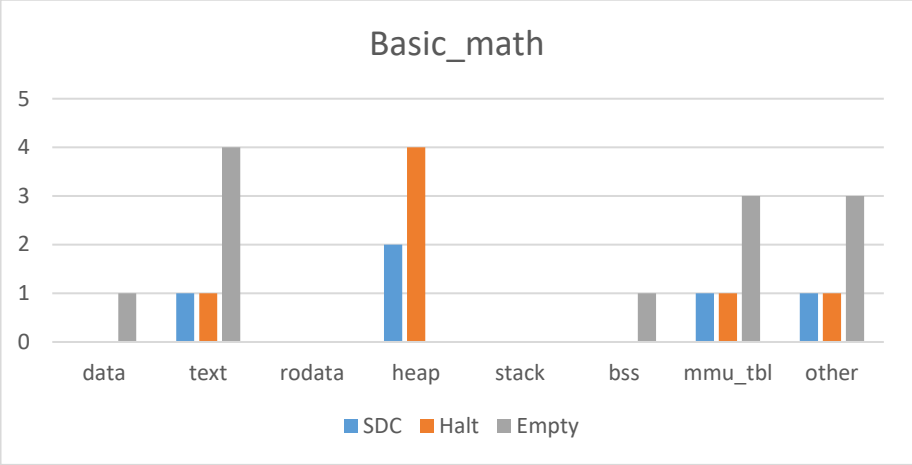

Figure 5. 7 Basic_math experimental results (FreeRTOS)

In this application, as it is obvious, the errors appeared in almost all sections of memory except stack and rodata sections, and we see that in heap, halts and SDC errors counted as the most probable faults, while in other sections, empty is.

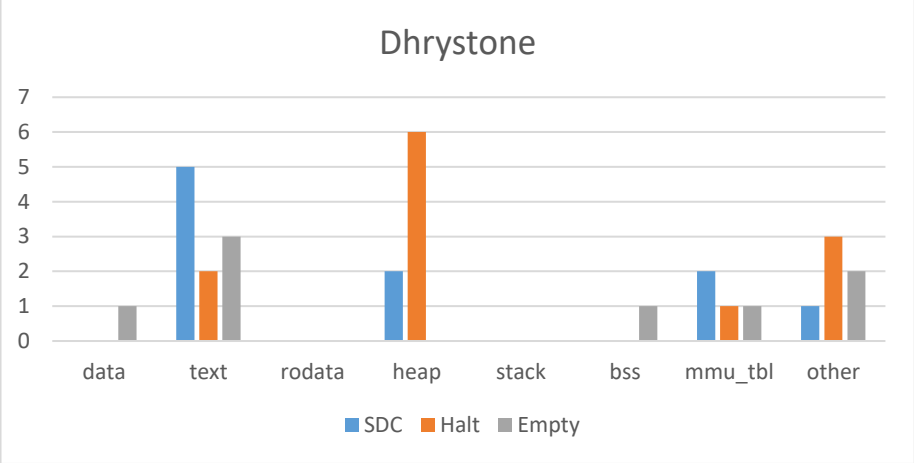

Figure 5. 8 Dhrystone experimental results (FreeRTOS)

Results gained in the Dhrystone application in the FreeRTOS platform are different from the same application in Baremetal, figure5.8. Here, halts mainly happened in the heap and SDC happened more times in the test than in other sections.

In the figures above, results contain three types of categories, SDC, Halt, Empty, and for sure, there is a category for the correct results, which I did not mention in the chart. These three categories appear when there is a fault in a system or when the comparison of the golden result does not match with the faulty ones. Hence, the figures indicate the type of diversity, with the indication of these fault appearances in each memory section.

5.2.3 Temperature and Execution Time

The temperature of an SoC is a key factor in guaranteeing reliable and efficient operation in the fields of semiconductor design, system integration, and performance enhancement. Thermally-induced problems including timing errors, increased power consumption, and lower reliability can be brought on by excessive heat. The temperature of the SoC is managed and controlled using efficient thermal management techniques, including as thermal design, and power distribution optimization.

For SoCs, temperature is a crucial factor since high temperatures can lead to instability and shorten chip life. High temperatures can also have an impact on the functionality and dependability of other system parts, including memory and power supply circuits. To maintain the temperature within acceptable ranges, it is essential to design SoCs with effective cooling technologies and to carefully monitor the power consumption.

Another important consideration is application execution speed, especially in contexts like high-performance computing, data centers, and mobile devices. Increased productivity, less latency, and enhanced user experience are all benefits of faster execution times. As a result, there is a big emphasis on software and hardware design, including the use of specialized hardware accelerators, parallel processing methods, and software optimization, to optimize application performance.

This experiment was carried out to determine which platform is more effective for our purposes and is used in sectors like aerospace or automotive by measuring the temperature and application execution time on both platforms.

Result of table 5.1 shows, temperature and execution time, which are almost the same on both platforms. Hence, we can take into account that, these two factors are not depended on the platform in our case of study.

Benchmarks	**Baremetal**		**Operating system**	
Rad2Deg	Exe_time	Temp(C°)	Exe_time	Temp(C°)
	0.070	41.98	0.070	41.02
Matrix_mul	0.004687	41.04	0.004666	40.02
Qsort	0.003034	42.41	0.004601	42.01
Dhrystone	0.138793	39-42	0.138793	40-42

Table 5. 1 Experimental result of execution time and Temperature

5.2.4 Error Rate Analysis

A crucial procedure used to evaluate and interpret the error rate of experimental data once an experiment is complete is called error rate evaluation. By dividing the number of inaccurate or false results by the total number of observations or predictions made during the experiment, the error rate can be calculated. This estimate, which is typically expressed as a percentage, offers a numerical assessment of the precision and reliability of the results of the study. Researchers can learn more about the performance and efficacy of their experimental methods, identify potential error sources, and make well-informed choices on the validity of their findings and the quality of the data. In [42] a novel estimate method that provides an accurate assessment of the possibility of Application Error Rate by taking radiation effects on the configuration memory and logic layer of FPGAs into account.

In this experiment, errors appear when system finish the process of the injection and the output is present, but the output is not the expected one and is not equal with the golden output. Hence, error rate is the sum of all the errors appear, as the result of the comparison of golden outputs and faulty outputs and it is in percentage.

Results of the fault injection experiment are shown in tables 5.2 and 5.3, which are the results on both platforms. With comparison of the error rate appearance, seen in tables, in Matrix mul and Qsort, in baremetal system, error rate is less with respect to the operating system, while for the Basic math and Dhrystone, FreeRTOS has less error rate. The results of this experiment demonstrate that an application's reliability is impacted by its underlying platform in addition to the application itself. Different platforms may display varied degrees of reliability, which can affect how frequently mistakes or failures occur within the same application. As a result, while choosing an application, it is essential to provide it with a suitable platform that is renowned for its reliability. The probability of an error occurring may be reduced by assuring a compatible and reliable platform, thereby improving the system's overall performance and reliability. The key to developing reliable and error-resistant apps is careful platform selection.

Applications	**Error Rate In [%]**
Matrix Mul	1.6
Qsort	0.9
Basic Math	3.8
Dhrystone	4

Table 5. 2 Error Rate of Memory Fault Injection in Baremetal platform

Applications	Error Rate In [%]
Matrix Mul	2.7
Qsort	3.4
Basic Math	2.4
Dhrystone	3.5

Table 5. 3 Error Rate of Memory Fault Injection in FreeRTOS platform

Chapter 6

Conclusion

In this thesis, reliability analysis on two platforms has been examined. This analysis can show reliability on two platforms, including a platform with the operating system and one without the operating system, the so-called "bare metal". These two platforms created and examined by SEU faults on the memory of resource of Zynq-7020 in which the software is executing on the ARM microprocessor integrated into the Zynq. Be aware that we can add different sorts of SEE with a small adjustment to the fault injection environment. Due to the necessity to verify the reliability of a system using different fault models and different benchmarks, tool development is still ongoing and will continue along with the relevant work that is already scheduled for the future.

A fault categorization campaign begins after fault injection is finished. The tool's primary goal is to identify impacts of injected fault to the main memory, and classify and display the occurrence of different sorts of faults that constitute misbehavior in each area of the main memory. This is done by first identifying the mistakes by comparing the golden outputs with the outputs generated by the fault injection platform, and then the fault categorization tools begin to process and categorize the faults.

6.1 Future Work

As I complete this thesis on reliability analysis of application execution on a processor, take into account potential future research and development opportunities in this field. The goal of this section is to suggest potential directions for future study that would build on the findings and implications discussed in this work. By identifying these areas, I plan to produce processor reliability analysis, which will inspire and guide future academics in their pursuit of knowledge.

In this thesis bitflips, or Single Event Upsets (SEUs), were the major fault model for reliability analysis in this work. Other fault models, including MBU, SET, and SEL, can be performed for upcoming work, and this fault model expansion could uncover a wide range of probable failures, including soft errors and timing issues. This addition would provide an improved understanding of the weaknesses and reliability difficulties that contemporary CPUs deal with.

In addition to performing other fault models for this analysis, adding other applications and benchmarks also could be a good idea to improve the vulnerability of the system. It would be beneficial to select the benchmark applications from those that will be employed in an operational environment in accordance with the system being tested and its purposes.

Appendix

Fault Injection Platform

```
from pylinx import XsctServer,Xsct
import serial
import random
import time
import os
import chardet
random.seed(42)

SKIP = 0
DONT_SKIP = []
pc_address =
"C:\Xilinx\Vitis\XSCT_auto\matrix_mul_addr_nxt_sram.txt"

ser=serial.Serial('COM7',baudrate=115200, timeout=1)
win_xsct_executable =
r'C:\Xilinx\Vitis\2022.1\bin\xsct.bat'
xsct_server = XsctServer( win_xsct_executable, port=99,
verbose=False)
xsct= Xsct('localhost', 99)

start = 0x0
end =   0x30000
xsct.do('cd C:/Xilinx/Vitis/XSCT_auto/os')
time.sleep(1)

xsct.do('source matrix_mul_exe.txt')
pc_address_file = open(pc_address,"r")
data=pc_address_file.readlines()
file_number = 1
j=0
for i in range(1000):
  print(f'Doing file #{file_number}')
  exit_loops = 0
   breakpoint
  random_data_time = random.choice(data)
  print(random_data_time)
```

```
  j+=1
  if j==150:
    print("its time")
  if i!=0:
   xsct.do('rst')
   if not SKIP or file_number in DONT_SKIP:
    time.sleep(1)
   if not SKIP or file_number in DONT_SKIP :
      xsct.do('ps7_init')
      xsct.do('ps7_post_config')
      xsct.do('dow matrix_mul.elf')
  address, n = random_data_time[0:8],
random_data_time[9:].strip()
  if not n:
    n=0
  else:
    n=int(n)
  if not SKIP:
    print(xsct.do('bpenable -all'))
    print(xsct.do('bpadd -addr' + ' 0x' + address))
    print(xsct.do('bpenable -all'))
    time.sleep(1)
  while n !=0:
    n-=1
    time.sleep(.00001)
  def generate_random_address(start, end):
    return random.randint(start, end)
  random_address = hex(generate_random_address(start,
end))
  print(random_address +"    random addr")
  value_in_rnd_addr = xsct.do('mrd '+random_address)
  value_in_rnd_addr = value_in_rnd_addr[12:20]
  print(value_in_rnd_addr + "    value in random addr")

  def bitflip(value_in_rnd_addr):
    hex_number = int(value_in_rnd_addr,16)
    binary_representation =
bin(hex_number)[2:].zfill(32)
      # Choose a random bit to flip
    random_bit = random.randint(0,
len(binary_representation) - 1)
      # Flip the chosen bit
    new_binary_representation =
binary_representation[:random_bit] + str(1 -
int(binary_representation[random_bit])) +
binary_representation[random_bit + 1:]
```

```
      # Convert the binary representation back to
hexadecimal
    new_hex_number = hex(int(new_binary_representation,
2))
    return new_hex_number,random_bit
    print(new_hex_number +"   random bit bitflip")
  new_hex, rnd_bit = bitflip(value_in_rnd_addr)
  if SKIP and file_number not in DONT_SKIP:
     SKIP-=1
     file_number += 1
     continue
  xsct.do('mwr'+' '+ random_address + ' ' +
str(new_hex))
  value_after_bitflip = xsct.do('mrd'+'
'+random_address)
  value_after_bitflip = "0x"+ value_after_bitflip[12:20]
  print(value_after_bitflip +"    value bitflip")
  xsct.do('bpremove -all')
  xsct.do('con')

  path = r'C:\pythonProject\final_output\output'
  output=''
  start_time = time.time()
  start_application_time = start_time
  stop_reading = False
  while not stop_reading:
    if time.time() -start_application_time > 30: #stop
if is runnning for more than 30 sec
        print("This took too long")
        break
    output_tmp = ser.readline().decode("utf-8",
errors="replace")
    output+=output_tmp
    if output_tmp:
        if "stop" in output_tmp:
          stop_reading=True
          print("Stop word found")
        start_time = time.time()
    else:
        if time.time() - start_time > 10:
          stop_reading = True
          print("TIMEOUT")
  random_data_time = random_data_time.strip()
  file_path = os.path.join(path, str(file_number)+"_"+
str(random_address)+"_"+
str(rnd_bit)+"_"+str(random_data_time)+".txt")
```

```
  print("Writing to file...", end ='')
  final_output_file = open(file_path, 'w',
encoding='utf-8')
  final_output_file.write(output)
  final_output_file.close()
  print("Wrote")
  file_number += 1
```

Classification Platform

```
import os
import filecmp

dic_classify = {}
def Mem_section_classification (output_file):
    parts = output_file.split("_")
    if len(parts) >= 2 :
        section = int(parts[1],16)
         # Check if the second part is within the
specified range
        if 0x0000 <= section <= 0x1a8b:
            return "text"
        elif 0x1a8c <= section <= 0x1a97:#init
            return "init"
        elif 0x1a98 <= section <= 0x1aa3:#fini
            return "fini"
        elif 0x1aa4 <= section <= 0x1c84:#rodata
            return "rodata"
        elif 0x1c88 <= section <= 0x20f7:#data
            return "data"
        elif 0x20f8 <= section <= 0x20fb:#eh_frame
            return "eh_frame"
        elif 0x4000 <= section <= 0x7fff:#mmu_tbl
            return "mmu_tbl"
        elif 0x8000 <= section <= 0x8003:#init_array
           return "init_array"
        elif 0x8004 <= section <= 0x8007:#fini_array
           return "fini_array"
        elif 0x8008 <= section <= 0x9b6b:#bss
           return "bss"
        elif 0x9b6c <= section <= 0xbb6f:#heap
           return "heap"
```

```
        elif 0xbb70 <= section <= 0xf36f:#stack
            return "stack"
        else:
            return "other"
# Set the path to the file you want to compare to
golden_file_path =
"C:/pythonProject/golden_output/matrix_mul_golden.txt"
golden_file = open(golden_file_path,"r")
length_golden = len(golden_file.read())

# Set the path to the directory containing the files
you want to compare
final_output_file_path =
"C:/pythonProject/final_output/matrix_mul_output"

# Loop through each file in the directory and compare
its content with output
with open(golden_file_path,'r') as golden:
    golden_content = golden.read()
for output_file in os.listdir(final_output_file_path):
    full_path = os.path.join(final_output_file_path,
output_file)
    with open(full_path,'r' , encoding='utf-8') as
output:
        output_content = output.read()
    if golden_content==output_content:
        print(f"{golden_file_path} and {output_file}
same.")
        if 'Correct' not in dic_classify:
            dic_classify['Correct']={'total':0}
        dic_classify['Correct']['total']+=1
        result = Mem_section_classification
(output_file)
        if result not in dic_classify['Correct']:
            dic_classify['Correct'][result] = 0
        dic_classify['Correct'][result]+=1
    else:
        print(f"{golden_file_path} and {output_file}
different.")
        file_path =
os.path.join(final_output_file_path, output_file)
        if os.path.getsize(file_path) != 0:
            length_content = len(output_content)
            if length_golden <= length_content:
               if 'SDC' not in dic_classify:
                    dic_classify['SDC']={'total':0}
```

```
                dic_classify['SDC']['total']+=1
                result = Mem_section_classification
(output_file)
                if result not in dic_classify['SDC']:
                    dic_classify['SDC'][result] = 0
                dic_classify['SDC'][result]+=1
            elif length_golden > length_content:
                if 'Halt' not in dic_classify:
                    dic_classify['Halt']={'total':0}
                dic_classify['Halt']['total']+=1
                result = Mem_section_classification
(output_file)
                if result not in dic_classify['Halt']:
                    dic_classify['Halt'][result] = 0
                dic_classify['Halt'][result]+=1
        else:
                if 'Empty' not in dic_classify:
                    dic_classify['Empty']={'total':0}
                dic_classify['Empty']['total']+=1
                result = Mem_section_classification
(output_file)
                if result not in
dic_classify['Empty']:
                    dic_classify['Empty'][result] = 0
                dic_classify['Empty'][result]+=1
for key, value in dic_classify.items():
    print(key,value)
```

References

[1] L. A. Cardona, A. Ullah, L. Sterpone, and C. Ferrer, “A novel tool-flow for zero-overhead cross-domain error resilient partially reconfigurable X-TMR for SRAM-based FPGAs,” *Journal of Systems Architecture*, vol. 81, pp. 112–120, 2017, doi: https://doi.org/10.1016/j.sysarc.2017.10.009.

[2] “electromigration”, Accessed: Jun. 26, 2023. [Online]. Available: https://www.synopsys.com/glossary/what-is-electromigration.html

[3] M. Rausand and K. ØIen, “The basic concepts of failure analysis,” *Reliab Eng Syst Saf*, vol. 53, no. 1, pp. 73–83, Jul. 1996, doi: 10.1016/0951-8320(96)00010-5.

[4] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A Survey on Software Fault Localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016, doi: 10.1109/TSE.2016.2521368.

[5] C. De Sio, S. Azimi, and L. Sterpone, “On the analysis of radiation-induced failures in the AXI interconnect module,” *Microelectronics Reliability*, vol. 114, p. 113733, Nov. 2020, doi: 10.1016/J.MICROREL.2020.113733.

[6] L. Sterpone, F. Luoni, S. Azimi, and B. Du, “A 3D Simulation-based Approach to Analyze Heavy Ions-induced SET on Digital Circuits,” *IEEE Trans Nucl Sci*, vol. PP, p. 1, Jun. 2020, doi: 10.1109/TNS.2020.3006997.

[7] C. De Sio, S. Azimi, and L. Sterpone, “An Emulation Platform for Evaluating the Reliability of Deep Neural Networks,” in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2020, pp. 1–4. doi: 10.1109/DFT50435.2020.9250872.

[8] F. Rosa, F. Kastensmidt, R. Reis, and L. Ost, “A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability,” in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, 2015, pp. 211–214. doi: 10.1109/DFT.2015.7315164.

[9] S. Azimi, B. Du, and L. Sterpone, “Evaluation of transient errors in GPGPUs for safety critical applications: An effective simulation-based fault injection environment,” *Journal of Systems Architecture*, vol. 75, pp. 95–106, Apr. 2017, doi: 10.1016/J.SYSARC.2017.01.009.

[10] S. Azimi, C. De Sio, A. Portaluri, D. Rizzieri, and L. Sterpone, “A comparative radiation analysis of reconfigurable memory technologies: FinFET versus bulk

CMOS," *Microelectronics Reliability*, vol. 138, p. 114733, Jun. 2022, doi: 10.1016/j.microrel.2022.114733.

[11] S. Azimi, C. De Sio, D. Rizzieri, and L. Sterpone, "Analysis of single event effects on embedded processor," *Electronics (Switzerland)*, vol. 10, no. 24, Dec. 2021, doi: 10.3390/electronics10243160.

[12] D. Oliveira, V. Frattin, P. Navaux, I. Koren, and P. Rech, "CAROL-FI: An Efficient Fault-Injection Tool for Vulnerability Evaluation of Modern HPC Parallel Accelerators," in *Proceedings of the Computing Frontiers Conference*, in CF'17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 295–298. doi: 10.1145/3075564.3075598.

[13] M. Hiller, A. Jhumka, and N. Suri, "PROPANE: An Environment for Examining the Propagation of Errors in Software," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, in ISSTA '02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 81–85. doi: 10.1145/566172.566184.

[14] S. Azimi, B. Du, and L. Sterpone, "A Zero-Timing Overhead SET Mitigation Approach for Flash-based FPGAs," in *2018 18th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2018, pp. 1–4. doi: 10.1109/RADECS45761.2018.9328665.

[15] C. De Sio, S. Azimi, and L. Sterpone, "FireNN: Neural Networks Reliability Evaluation on Hybrid Platforms," *IEEE Trans Emerg Top Comput*, vol. 10, no. 2, pp. 549–563, 2022, doi: 10.1109/TETC.2022.3152668.

[16] B. J. Mealy, "Work in progress - embedded system-based introductory programming course for computer and electrical engineering students," in *2008 38th Annual Frontiers in Education Conference*, 2008, pp. F3E-17-F3E-18. doi: 10.1109/FIE.2008.4720620.

[17] C. De Sio, S. Azimi, L. Bozzoli, B. Du, and L. Sterpone, "Radiation-induced Single Event Transient effects during the reconfiguration process of SRAM-based FPGAs," *Microelectronics Reliability*, vol. 100–101, p. 113342, Sep. 2019, doi: 10.1016/J.MICROREL.2019.06.034.

[18] C. De Sio, S. Azimi, L. Sterpone, and B. Du, "Analyzing Radiation-Induced Transient Errors on SRAM-Based FPGAs by Propagation of Broadening Effect," *IEEE Access*, vol. 7, pp. 140182–140189, 2019, doi: 10.1109/ACCESS.2019.2915136.

[19] J. T. Wallmark and S. M. Marcus, "Minimum Size and Maximum Packing Density of Nonredundant Semiconductor Devices," *Proceedings of the IRE*, vol. 50, no. 3, pp. 286–298, 1962, doi: 10.1109/JRPROC.1962.288321.

[20] L. Sterpone, B. Du, and S. Azimi, “Radiation-induced single event transients modeling and testing on nanometric flash-based technologies,” *Microelectronics Reliability*, vol. 55, no. 9–10, pp. 2087–2091, Aug. 2015, doi: 10.1016/J.MICROREL.2015.07.035.

[21] L. Sterpone and S. Azimi, “Radiation-induced SET on Flash-based FPGAs: Analysis and Filtering Methods,” in *ARCS 2017; 30th International Conference on Architecture of Computing Systems*, 2017, pp. 1–6.

[22] D. Rizzieri *et al.*, “Programmable SEL Test Monitoring System for Radiation Hardness Assurance,” in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2023, pp. 1–7.

[23] C. De Sio, S. Azimi, A. Portaluri, and L. Sterpone, “SEU Evaluation of Hardened-by-Replication Software in RISC- V Soft Processor,” in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021, pp. 1–6. doi: 10.1109/DFT52944.2021.9568342.

[24] G. C. Cardarilli, F. Kaddour, A. Leandri, M. Ottavi, S. Pontarelli, and R. Velazco, “Bit flip injection in processor-based architectures: a case study,” in *Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002)*, 2002, pp. 117–127. doi: 10.1109/OLT.2002.1030194.

[25] C. Sio, S. Azimi, L. Sterpone, and D. Codinachs, “Analysis of Proton-induced Single Event Effect in the On-Chip Memory of Embedded Process,” in *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2022, pp. 1–6. doi: 10.1109/DFT56152.2022.9962341.

[26] S. Azimi, C. De Sio, and L. Sterpone, “Analysis of radiation-induced transient errors on 7 nm FinFET technology,” *Microelectronics Reliability*, vol. 126, p. 114319, Jun. 2021, doi: 10.1016/j.microrel.2021.114319.

[27] S. Azimi and L. Sterpone, “Micro Latch-Up Analysis on Ultra-Nanometer VLSI Technologies: A New Monte Carlo Approach,” in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017, pp. 338–343. doi: 10.1109/ISVLSI.2017.66.

[28] M. Allenspach *et al.*, “Single-event gate-rupture in power MOSFETs: prediction of breakdown biases and evaluation of oxide thickness dependence,” *IEEE Trans Nucl Sci*, vol. 42, no. 6, pp. 1922–1927, 1995, doi: 10.1109/23.489234.

[29] S. Azimi, L. Sterpone, B. Du, and L. Boragno, “On the analysis of radiation-induced Single Event Transients on SRAM-based FPGAs,” *Microelectronics*

Reliability, vol. 88–90, pp. 936–940, Sep. 2018, doi: 10.1016/J.MICROREL.2018.07.135.

[30] S. Kuboyama, S. Matsuda, T. Kanno, and T. Ishii, "Mechanism for single-event burnout of power MOSFETs and its characterization technique," *IEEE Trans Nucl Sci*, vol. 39, no. 6, pp. 1698–1703, 1992, doi: 10.1109/23.211356.

[31] K. Mori, H. Yamada, and S. Takizawa, "System on Chip Age," in *1993 International Symposium on VLSI Technology, Systems, and Applications Proceedings of Technical Papers*, 1993, pp. K15–K20. doi: 10.1109/VTSA.1993.263614.

[32] A. Portaluri, C. De Sio, S. Azimi, and L. Sterpone, "A New Domains-based Isolation Design Flow for Reconfigurable SoCs," in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2021, pp. 1–7. doi: 10.1109/IOLTS52814.2021.9486687.

[33] "Zynq-7000 SoC Data Sheet:overview,", Accessed: Jun. 22, 2023. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview

[34] "Zynq-7000 SoC Technical Reference Manual." https://www.xilinx.com/content/dam/xilinx/support/documents/user_guides/ug585-Zynq-7000-TRM.pdf (accessed Jun. 22, 2023).

[35] "AXI Documentation", Accessed: Jun. 22, 2023. [Online]. Available: https://developer.arm.com/documentation/ihi0022/e/?lang=en

[36] "ARM Reference Manual", Accessed: Jun. 22, 2023. [Online]. Available: https://developer.arm.com/documentation/ddi0406/latest/

[37] "PYNQ-Z2 reference manual", Accessed: Jun. 22, 2023. [Online]. Available: https://dpoauwgwqsy2x.cloudfront.net/Download/PYNQ_Z2_User_Manual_v1.1.pdf

[38] "PYNQ-Z2 Specifications", Accessed: Jun. 22, 2023. [Online]. Available: https://www.tul.com.tw/images/PYNQ-Z2_PA_v2_pp_20201209_STD.pdf

[39] W. Yang, B. Du, C. He, and L. Sterpone, "Reliability assessment on 16 nm ultrascale+ MPSoC using fault injection and fault tree analysis," *Microelectronics Reliability*, vol. 120, p. 114122, 2021, doi: https://doi.org/10.1016/j.microrel.2021.114122.

[40] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14. doi: 10.1109/WWC.2001.990739.

[41] “Dhrystone Benchmarking for ARM Cortex Processors”, Accessed: Jun. 23, 2023. [Online]. Available: https://developer.arm.com/documentation/105958/latest

[42] L. Sterpone *et al.*, “A Novel Error Rate Estimation Approach forUltraScale+ SRAM-based FPGAs,” in *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2018, pp. 120–126. doi: 10.1109/AHS.2018.8541474.