UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

GENNARO SEVERINO RODRIGUES

Approximate Computing Strategies for Low-Overhead Fault Tolerance in Safety-Critical Applications

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Microeletronics

Advisor: Prof. Dr. Fernanda Kastensmidt
Coadvisor: Prof. Dr. Alberto Bosio

Porto Alegre
October 2019

CIP — CATALOGING-IN-PUBLICATION

Severino Rodrigues, Gennaro

Approximate Computing Strategies for Low-Overhead Fault Tolerance in Safety-Critical Applications / Gennaro Severino Rodrigues. – Porto Alegre: PGMICRO da UFRGS, 2019.

137 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2019. Advisor: Fernanda Kastensmidt; Coadvisor: Alberto Bosio.

1. Approximate Circuits. 2. Approximate-TMR. 3. Fault Tolerance. 4. Critical Systems. I. Kastensmidt, Fernanda. II. Bosio, Alberto. III. Título.

ABSTRACT

This work studies the reliability of embedded systems with approximate computing on software and hardware designs. It presents approximate computing methods and proposes approximate fault tolerance techniques applied to programmable hardware and embedded software to provide reliability at low computational costs. The objective of this thesis is the development of fault tolerance techniques based on approximate computing and proving that approximate computing can be applied to most safety-critical systems.

It starts with an experimental analysis of the reliability of embedded systems used at safety-critical projects. Results show that the reliability of single-core systems, and types of errors they are sensitive to, differ from multicore processing systems. The usage of an operating system and two different parallel programming APIs are also evaluated. Fault injection experiment results show that embedded Linux has a critical impact on the system's reliability and the types of errors to which it is most sensitive. Traditional fault tolerance techniques and parallel variants of them are evaluated for their fault-masking capability on multicore systems. The work shows that parallel fault tolerance can indeed not only improve execution time but also fault-masking. Lastly, an approximate parallel fault tolerance technique is proposed, where the system abandons faulty execution tasks. This first approximate computing approach to fault tolerance in parallel processing systems was able to improve the reliability and the fault-masking capability of the techniques, significantly reducing errors that would cause system crashes.

Inspired by the conflict between the improvements provided by approximate computing and the safety-critical systems requirements, this work presents an analysis of the applicability of approximate computing techniques on critical systems. The proposed techniques are tested under simulation, emulation, and laser fault injection experiments. Results show that approximate computing algorithms do have a particular behavior, different from traditional algorithms. The approximation techniques presented and proposed in this work are also used to develop fault tolerance techniques. Results show that those new approximate fault tolerance techniques are less costly than traditional ones and able to achieve almost the same level of error masking.

Keywords: Approximate Circuits. Approximate-TMR. Fault Tolerance. Critical Systems.

RESUMO

Este trabalho estuda a confiabilidade de sistemas embarcados com computação aproximada em software e projetos de hardware. Ele apresenta métodos de computação aproximada e técnicas aproximadas para tolerância a falhas em hardware programável e software embarcado que provêem alta confiabilidade a baixos custos computacionais. O objetivo desta tese é o desenvolvimento de técnicas de tolerância a falhas baseadas em computação aproximada e provar que este paradigma pode ser usado em sistemas críticos.
O texto começa com uma análise da confiabilidade de sistemas embarcados usados em sistemas de tolerância crítica. Os resultados mostram que a resiliência de sistemas *single-core*, e os tipos de erros aos quais eles são mais sensíveis, é diferente dos *multi-core*. O uso de sistemas operacionais também é analisado, assim como duas APIs de programação paralela. Experimentos de injeção de falhas mostram que o uso de Linux embarcado tem um forte impacto na confiabilidade do sistema. Técnicas tradicionais de tolerância a falhas e variações paralelas das mesmas são avaliadas. O trabalho mostra que técnicas de tolerância a falhas paralelas podem de fato melhorar não apenas o tempo de execução da aplicação, mas também seu mascaramento de erros. Por fim, uma técnica de tolerância a falhas paralela aproximada é proposta, onde o sistema abandona instâncias de execuções que apresentam falhas. Esta primeira experiência com computação aproximada foi capaz de melhorar a confiabilidade das técnicas previamente apresentadas, reduzindo significativamente a ocorrência de erros que provocam um *crash* total do sistema.
Inspirado pelo conflito entre as melhorias trazidas pela computação aproximada e os requisitos dos sistemas críticos, este trabalho apresenta uma análise da aplicabilidade de computação aproximada nestes sistemas. As técnicas propostas são testadas sob experimentos de injeção de falhas por simulação, emulação e laser. Os resultados destes experimentos mostram que algoritmos aproximados possuem um comportamento particular que lhes é inerente, diferente dos tradicionais. As técnicas de aproximação apresentadas e propostas no trabalho são também utilizadas para o desenvolvimento de técnicas de tolerância a falhas aproximadas. Estas novas técnicas possuem um custo menor que as tradicionais e são capazes de atingir o mesmo nível de mascaramento de erros.

Palavras-chave: Computação Aproximada, TMR aproximado, tolerância a falhas, sistemas críticos.

LIST OF ABBREVIATIONS AND ACRONYMS

ABFT	Application-Based Fault Tolerance
AMBA	ARM Advanced Microcontroller Bus Architecture
API	Application Programming Interface
ATMR	Approximate Triple Modular Redundancy
CDMR	Conditional Double Modular Redundancy
CMP	Chip Multiprocessor
COTS	Commercial Off-The-Shelf
CRC	Cyclic Redundancy Check
DD	Displacement Damage
DUT	Design Under Test
DVFS	Dinamic Voltage and Frequency Scaling
DWC	Duplication With Comparison
ECC	Error Correction Code
EDDI	Error Detection by Duplicated Instructions
FFT	Fast Fourier Transform
FI	Functional Interruption
FIM	Fault Injection Module
FIT	Failure in Time
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
HLS	High-Level Synthesis
HPC	High-Performance Computing
ICAP	Internal Configuration Access Port
IC	Integrated Circuit

IP	Intellectual Property
LET	Linear Energy Transfer
MBU	Multibit Upset
MCU	Multicell Upaset
MSE	Mean Square Error
MWTF	Mean Work To Fail
NIEL	Non-Ionizing Energy Loss
NVP	N-Version Programming
OCM	On-Chip Memory
OpenMP	Open Multi-Processing
OS	Operating System
OVPSim	OVP Simulator
PAED	Parallel Approximate Error Detection
PL	Programmable Logic
PS	Processing System
PSNR	Peak Signal-to-Noise Ratio
ROI	Region of Interest
RTCA	Radio Technical Comission for Aeronautics
SDC	Silent Data Corruption
SEB	Single Event Burnout
SEE	Single Event Effect
SEFI	Single Event Functional Interrupt
SEGR	Single Event Gate Rupture
SEL	Single Event Latch-up
SER	Soft Error Rate
SET	Single Event Transient

SEU	Single Event Upset
SHE	Single Hard Error
SIHFT	Software-Implemented Hardware Fault Tolerance
SoC	System-on-a-chip
TID	Total Ionizinf Dose
TMR	Triple Modular Redundancy
TPA	Two-Photon Absorption
UT	Unexpected Termination
μSEL	Micro Single Event Latch-Up

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Safety-critical systems must be extremely reliable. Those systems often deal with human lives or costly apparel; therefore, an error can be catastrophic. In recent years the industry has turned to commercial off-the-shelf (COTS) devices for their projects because of their relatively low cost and high performance. COTS, however, usually do not provide reliability, thus needing the implementation of fault tolerance techniques in order to be safely used for safety-critical systems development. Approximate computing has emerged as a computing paradigm capable of achieving good performances on execution time and energy consumption, as well as inherent reliability. However, it pays the price for it in precision loss and has to consider "good enough" results as acceptable (i.e., near the expected traditional computation output). Using approximate computing on safety-critical systems could improve their performances while also making them inherently more reliable. However, it can be conflicting with some of the safety-critical systems requirements, such as accuracy.

Factors such as power efficiency and execution performance are of great importance for embedded systems, and can be improved through approximate computing (HAN; ORSHANSKY, 2013). The approximate computing paradigm works with the idea that most applications are able to tolerate some flexibility in the computed result, based on a quality threshold specified by the application requirements. Indeed, several algorithms can present a *good enough* result even when executing inexact computations. An image processing algorithm, for example, might be able to tolerate some variations in the output quality, given the fact that the human eye is not able to perceive small differences between images. Such an algorithm might therefore skip some computation in order to execute faster and have a lower memory footprint, causing an acceptable degradation on the output image. In that context, approximate computing has been proposed as a means to provide computational resources savings, alongside with execution time and energy consumption improvements, with controlled quality degradation. Approximate computing techniques can be applied on every level of the computational stack, from circuit and hardware to embedded software. Those techniques (VENKATARAMANI et al., 2015) have been used in many scenarios, from big data to scientific applications (NAIR, 2014).

The literature presents a plethora of approximation strategies, both for software and hardware. The loop-perforation technique is an excellent software approximation example, being able to achieve useful outputs while not executing all the iteration of an iterative code (SIDIROGLOU-DOUSKOS et al., 2011). Indeed, the authors claim this approach typically delivers performance increases of over a factor of two while introducing an output deviation of less than 10%. Another approximation technique for software applications consists of reducing the bit-width used for data representation (RUBIO-GONZALEZ et al., 2013), also achieving a better execution speed than their non-approximate counterparts. Hardware-based approximation techniques usually make use of alternative speculative implementations of arithmetic operators. An example of this approach is the implementation of variable approximation modes on operators (SHAFIQUE et al., 2015). Hardware approximation is also present in the image processing domain in the form of approximate compressors (MOMENI et al., 2015).

The quality degradation inherent to approximate computing is, however, not to be forgotten. Although some quality degradation is acceptable for image processing algorithms, as exemplified before, it might not be acceptable for high dependability systems. A 10% quality degradation on an image might pass by unperceived, but an error of 10% in a banking system that computes the profits and taxes from a conglomerate will indeed be a severe problem. Even the perfect example of acceptable quality degradation (image processing) calls for an in-depth analysis on the acceptance lever for that degradation: surely no graphics processing units manufacturer wants to be known as the one which provides low-quality graphics in a very fast frame rate.

Safety-critical applications are an excellent example of a category on which approximate computing can indeed bring good fruits but are of delicate implementation. Applications defined as safety-critical deal with human lives and high-cost equipment, and therefore call for high dependability, i.e., low error rates. Safety-critical systems such as aerospace and avionics applications are often exposed to space radiation. Indeed, even systems that operate at ground level can be subject to space radiation (BAUMANN, 2005), and some of those are also categorized as safety-critical systems (e.g., self-driven cars and their collision avoidance algorithms).

Radiation effects in semiconductor devices vary from data disruptions to permanent damage. The state of a memory cell, register, latch, or any part of the circuit that holds data can be changed by a radiation event. Single radiation events might cause *soft* or *hard* errors. Soft errors are the primary concern for commercial applications (POIVEY et al., 2003) and occur when this radiation event is strong enough to change the data state without permanently damaging the system (TYLKA et al., 1996), manifesting as many types of errors. In software applications, those errors can be categorized into two major groups: SDCs (silent data corruption) and FIs (functional interruption) (HSUEH; TSAI; IYER, 1997). An SDC occurs when the application finishes properly, but its final memory state differs from the expected gold state. FIs are considered when the application hangs or terminates unexpectedly. Hard errors are permanent damages to the system and are often related to dose-rate radiation effects (i.e., associated with the accumulation of radiation and its impacts on the behavior of the transistor).

The new transistor technologies' reduction of the dimensions and operation thresholds have improved their energy consumption and performance. Their sensitivity to radiation, however, is often not a concern for the industry that focuses their efforts on higher transistor density and functionality at low cost. Indeed, the reduction of the transistor sizes on new technologies can now lead to radiation-induced faults, that would otherwise occur on space environments, to occur at ground level (TAUSCH et al., 2007). Although those fault-induction-related issues are not a significant concern for the traditional consumer, which can accept sparse little errors, they are indeed a severe concern for safety-critical systems.

The traditional hardware manufacturers are not motivated to develop new radiation-hardened technologies because of their high development cost and, consequently, low-profit margin due to limited application (BAUMANN, 2005). On the other side, the safety-critical industry is also often not interested in radiation-hardened hardware, which is expansive and does not provide the same performance as the state-of-the-art hardware devices. The industry has turned to COTS (commercial off-the-shelf) embedded processors and systems-on-a-chip (SoC) combined with fault tolerance techniques (PIGNOL, 2010). COTS are typically low cost, very flexible, and consume little power. They do not provide, however, inherent fault tolerance (apart from traditionally used methods such as memory error correction codes, which alone does not provide all the reliability required for safety-critical systems), and therefore call for hardware- and software-based fault tolerance techniques to provide reliability. The Zynq$^{\text{TM}}$-7000 All Programmable SoC (KADI et al., 2013), for instance, is an example of COTS system composed of two ARM processor cores and a field programmable gate array (FPGA) that is capable of serving a wide range of safety-critical applications, such as avionics communications, missile control, and smart ammunition control systems. COTS devices provide a myriad of system configuration parameters, which may directly affect fault tolerance.

Fault tolerance can be applied at the hardware level by duplicating or triplicating an entire component and adding voters and checkers that verify the consistency of the processed data. Those techniques, however, introduce a prohibitive area and power overhead. Software-based fault tolerance does not need extra hardware and is widely presented and discussed in the literature (SAHA, 2006; OSINSKI; LANGER; MOTTOK, 2017). In that case, redundancy is applied at the task level and executed in single or multiple processing cores. Although software-based techniques may present no hardware area overhead, it pays the cost on execution time and memory footprint, as well as energy consumption (that derive from those). One example of a fault tolerance mechanism that can be applied to both hardware and software is duplication with comparison (DWC), which duplicates the application and implements a checker to compare any discrepancy between the data generated by the two independent executions. DWC is capable of finding errors, but not masking them. A third execution would be needed to mask the error, making a vote for the correct data possible.

Concerning fault tolerance, approximate computing can mask a higher number of errors by relaxing data precision requirements. On systems that do not need high accuracy or quality, the approximation can be used because the small errors it introduces are not big enough to be considered a problem. Besides, the execution time reduction attained by approximate computing can improve an application reliability by reducing its exposure time: it is evident that an application that executes faster will be subject to less radiation, and therefore less to radiation-induced faults. SoCs arise as perfect implementation platforms for approximate computing. Industry-leading companies offer SoC presenting both an FPGA logic layer (PL) and an embedded processor as a processing system (PS), such as the aforementioned Zynq$^{\mathrm{TM}}$-7000 All Programmable SoC. Approximate computing projects can profit from the hardware-software co-design made available from COTS systems to implement any level of approximation, or as means of co-processing. Indeed, as will be further detailed, many approximation techniques consist of executing on programmable hardware approximate versions of standard functions that would otherwise be executed on hardcore microprocessors.

This work investigates the use of approximate computing on safety-critical systems. The approximate computing paradigm can be used to achieve several fundamental requirements of embedded safety-critical systems, such as low power consumption and high performance. Those, however, are achieved at the cost of precision and accuracy, which are serious concerns regarding critical applications. Another significant point of interest is reliability: approximation methods shall be able to tolerate errors or at least support traditional fault tolerance techniques. Therefore, it is essential to study not only the improvements approximate computing can bring to a project, but also its costs, and how it affects the dependability of those projects.

Approximation is presented in this work applied at the hardware and software level. On hardware projects, the techniques are implemented in hardware description language (HDL) with and without the aid of high-level synthesis. In a first analysis, the implementation cost and precision loss of approximation methods are assessed. Then, they are subject to fault tolerance analysis by fault injection experiments. Those experiments are intended to evaluate both the approximation fault tolerance by itself and the efficacy of traditional fault tolerance mechanisms when applied to an approximate application. Alternatively, novel approximate fault tolerance techniques, based on the traditional ones, are proposed and evaluated. The proposed techniques intend to provide fault coverage close to the traditional ones but at lower costs.

The fault injections are performed in four different methodologies: fault injection emulation, fault injection simulation, and laser and heavy-ion radiation experiments. Each one of those methodologies serves better for a specific evaluation purpose. The fault injection emulation on programmable hardware, for example, can be used to evaluate the behavior of the design under a situation of cumulative faults, in an effort to find out on which point (given the number of accumulated injected faults) the design begins to present errors. It can also be used to perform exhaustive studies on programmable hardware, to find out which bits of the bitstream used to program the FPGA are critical (i.e., a bit-flip on this bit will provoke errors). On the other hand, fault injection simulation can be used to inject faults on the register file of the processor to analyze which are the most critical registers and how faults affecting the register file propagate to become errors in a given context.

The work will start by presenting an overview and the state-of-the-art of the primary subject of the thesis: approximate computing. The first chapter is a survey with discussions over the approximation techniques most present at the literature, their impacts on the system that implement them, and how they can be applied to all the computation stacks. Following, the text presents an overview of radiation effects on electronic devices: the types of faults that the devices are subject to, their source, and how they affect the system becoming errors. The work then follows by a chapter that covers the methodologies that are used in this work and their peculiarities.

As an introductory discussion on the findings and to develop the justification of the proposals of this work, this work analyses the reliability of embedded systems, especially embedded microprocessors. This chapter will develop a discussion on fault tolerance techniques and discuss the reliability of applications running bare-metal and on top of both Linux and FreeRTOS operating systems. It will also evaluate the reliability of parallel algorithms running on top of multicore processors, with different parallelization APIs. All that is essential to the development of the rest of the thesis, because it defines the basis of our theory: before evaluating the impact of approximation on reliability, it is fundamental to understand the need for fault tolerance of the systems that will be approximated, and the fact that they have characteristics independent from approximation.

The following chapters then present the approximation methods proposed by this work, as well as the developed approximate fault tolerance techniques and how they corelate. The results are then presented, categorized by every different methodology used to evaluate each technique. The work concludes by presenting the final conclusions and discussing how the present research will be expanded in future works.

The document is organized in the following chapters:

Chapter 1, Introduction: this chapter, presenting the introduction of the works.

Chapter 2, Approximate Computing: presents the state-of-the-art of the approximate computing, as well as how it motivates the work.

Chapter 3, Radiation Effects on Electronic Devices: presents the radiation effects on electronic devices, their sources and effects on the system.

Chapter 4, Analysis Methodologies: details the methodologies used to evaluate the proposed approximate computing and approximate fault tolerance techniques.

Chapter 5, Embedded Systems Fault Tolerance: discusses state-of-the-art fault tolerance techniques and how they relate and motive this work. Also presents a practical introduction on the reliability of embedded systems.

Chapter 6, Proposed Work: proposes approximation methods and approximate fault tolerance techniques.

Chapter 7, Experimental Results and Discussion: presents experimental results of the proposed techniques and discusses them.

Chapter 8, Conclusion: presents the conclusions and the ideas for expanding the research. Also presents the list of publications that originated from this work.

2 APPROXIMATE COMPUTING

Approximate computing has been proposed as an approach for developing energy-efficient systems (HAN; ORSHANSKY, 2013), lowering hardware resources usage and presenting better execution times, and has been used in many scenarios, from big data to scientific applications (NAIR, 2014). It can be achieved from a multitude of ways, ranging from transistor-level design to software implementations. Many systems do not take precision and accuracy as an essential asset. Those are the ones that can profit from this computational paradigm (XU; MYTKOWICZ; KIM, 2016). An example of this type of application is real-time systems, which have very strict deadlines and require strong performance. The approximation can also be unavoidable. Floating-point operations, for instance, have frequent rounding of values, making it inherently approximate. Numerical algorithms are also frequently of approximate nature: the calculation of an integral, for example, is not natural for a computer, and consists of an iterative calculation of a finite sum of terms (and not an infinite one, as the mathematical theory defines it).

Even on systems where quality and accuracy are essential, the mere definition of a good quality result can be malleable. On image processing, for example, the final output is evaluated by a human perspective (the quality of the image). This perspective is subjective: some people are more capable of analyzing the quality of an image than others, and the definition of a "good enough" quality is even more debatable. Typical error-resilient image processing algorithms can indeed accept errors of up to 10% (RAHIMI et al., 2015), which would be unacceptable for a military system calculating the trajectory of a ballistic projectile, for example. This margin of error acceptance can be exploited to improve energy consumption and execution performance.

The weak definition of "error acceptance" can also be used by approximate computing for quality configuration. Given that different systems have different quality requirements, a designer might make use of just the necessary energy, hardware area, or execution time, to meet the needs of his project. An excellent example of how a circuit can be configured in that manner is by using different refresh rates for memories (CHO et al., 2014), or different precision for data storage and representations (TIAN et al., 2015). The image processing domain is particularly interesting because it is an example of how approximation can be implemented on different levels. A minor loss in precision can be accepted by applying approximation via hardware, by reducing the refresh rates of eDRAM/DRAM and the SRAM supply voltage, which reduces the energy consumption (MITTAL, 2014). On a higher level of implementation, the approximation can be used by loop-perforation (finishing a loop execution earlier than expected) (SIDIROGLOU-DOUSKOS et al., 2011) or by executing specific functions on neural accelerators (MOREAU et al., 2015).

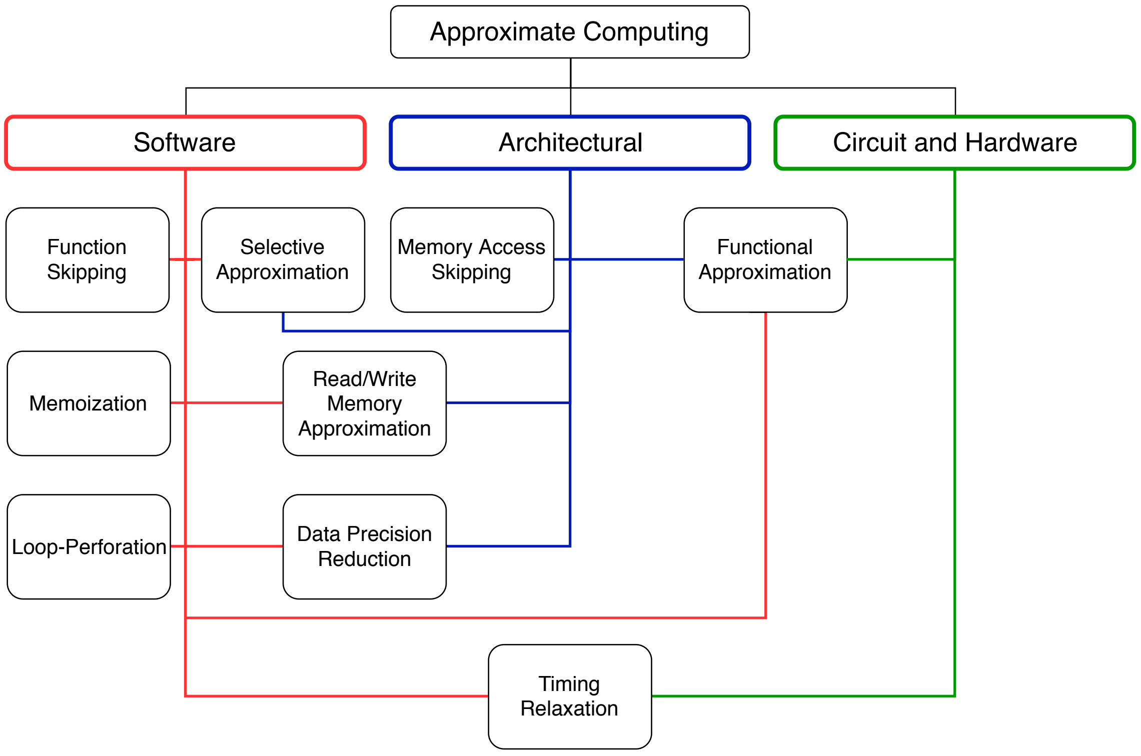
2.1 Quality Metrics and Approximation Methods

Given the plethora of approximation methods and systems that make use of them, the literature also presents an extensive list of error metrics definitions. Some examples of how precision loss is measured on approximate systems are:

- **Pixel Difference:** consists of a full comparison of two images pixel-by-pixel, where every pixel is represented by a value. Normally used to compare gray-scale images, where the pixel value defines the grey intensity (the higher value being complete darkness).
- **Peak signal-to-noise ratio (PSNR):** it is calculated using the mean square error (MSE) between the two images (the original and the approximate one), and indicates the ratio of the maximum pixel intensity to the distortion. It is calculated by the formula $\text{PSNR} = 10log_{10}(\text{MAX}^2/\text{MSE})$, where MAX stands for the maximum value of a pixel in the images.
- **Hamming Distance:** when comparing data bit-wise, the hamming distance consists of the number of positions where the bit values are different between binary strings.
- **Ranking Accuracy:** when approximate computing is applied to ranking algorithms, such as the ones used by search engines, it can generate different results depending on the ranking definitions and the algorithms used. A research result from Bing and Google search engines, for example, will likely be different. The accuracy is defined based on pre-established parameters.
- **Error Probability:** consists of the error rate of all the possible outputs, comparing the result of an approximate function and its non-approximate counterpart. This metric gives the probability for an approximation to present an error but does not evaluate the criticality or impact of that error.
- **Relative Difference:** presents the error in relation to the standard non-approximate output. This metric is capable of evaluating the criticality of an error.

The presented quality evaluation metrics are not mutually exclusive. One applica-

Figure 2.1: Approximate computing classification.



Source: Author.

tion might use several different parameters to evaluate its quality loss. Both PSNR and pixel difference are used as image quality metrics, for example.

Approximation techniques can be applied to all the computation stack levels. Figure 2.1 divides approximation techniques into three groups that define their implementation level: software, architectural, and hardware. As Figure 2.1 shows, some approximation methods can even be implemented in more than one level. Load value approximation, for example, can be both implemented by purely software approaches or at memory control units. Figure 2.1 presents only some of the most used approximation methods and the most discussed in the literature. However, there are uncountable ways of approximating an application, and the very definition of what is to be considered an approximation or not is debatable.

The techniques presented at Figure 2.1 are defined below:

Functional Approximation: Consists of implementing approximate versions of an algorithm. The literature presents a multitude of functional approximation techniques for circuits and architecture levels that implement approximate adders. One approach is to remove the carry chain from the circuit to reduce delay and energy consumption. This can be done by altering the subadders of a standard adder cell of n bits (KAHNG; KANG, 2012). In (KULKARNI; GUPTA; ERCEGOVAC, 2011), the authors presented an approximate 2×2 multiplier design that gives correct outputs for 15 of the 16 possible input combinations and uses half of the area of a standard non-approximate multiplier. Functional approximation can be implemented at software and architectural level using neural networks. A neural network can learn how a standard function implementation behaves in relation to different inputs via machine learning. In a complex system, the neural network can be used to implement approximate functions via software-hardware co-design. Traditional approximable codes can be transformed into equivalent neural networks that present a lower output accuracy but better execution time performance (AMANT et al., 2014).

Selective Approximation: some systems are made of parts that don't need to provide accuracy as much as others. The idea is to take advantage of the fact that even inside an algorithm, some parts affect more the final accuracy than others. Those parts can be approximated to provide energy consumption reduction and improve execution time performance (VASSILIADIS et al., 2015). On the architectural level, selective approximation can appear as alternative speculative implementations of arithmetic operators. An example of this approach is the implementation of variable approximation modes on operators (SHAFIQUE et al., 2015). When applied to software applications, approximate computing usually consist of inexact computations, which provide results with lower accuracy than usual (VENKATARAMANI et al., 2015). Most approximate computing techniques for software consist of modifying the algorithm so that it executes approximately, providing a final result more rapidly.

Function Skipping: in a system composed of tasks that complement each other in the sense of providing a final result, some of the tasks can be skipped while maintaining a level of accuracy and error resiliency defined by the user (GOIRI et al., 2015).

Memoization: traditionally, memoization consists of saving outputs of functions for given inputs to be reused later. Given that some input data are frequently reused, their calculated outputs can be stored and used without the need for the re-execution of the function. Memoization can also be used to approximate applications: if *similar* inputs are to provide *similar* outputs from a given function, it means that the already-calculated function output can approximately cover a range of inputs. In (KERAMIDAS; KOKKALA; STAMOULIS, 2015), the authors propose approximate value reuse for memoization, providing very low accuracy loss.

Loop-Perforation: In loop-based algorithms, loop-perforation can be used to highly reduce the execution time. An excellent example of this type of application is numerical algorithms. The calculation of an integral using the trapezoidal method, for example, consists of calculating the area of a high number of trapezoids under the curve of a function, providing an approximation of the area beneath it. Reducing the number of calculated trapezoids, the final value will be less accurate, but the program will finish earlier. The literature also presents techniques to apply this approximation method on general-use algorithms, filtering out the loops that cannot be approximated and using loop-perforation on those that can (SIDIROGLOU-DOUSKOS et al., 2011). The authors claim this approach typically delivers performance increases of over a factor of two while introducing an output deviation of less than 10%. Loop-perforation is an algorithm-based approximate technique, as it is only applicable to loop-based code, which limits its coverage.

Data Precision Reduction: Data precision reduction is one of the techniques that can be implemented both at software and architectural level. Reducing the data precision of an application (i.e., the number of bits used to represent the data) is a straightforward technique to reduce memory footprint. Reducing memory usage also reduces energy consumption at the cost of accuracy. In (HSIAO; CHU; CHEN, 2013), the authors show that reducing floating point precision on mobile GPUs can bring energy consumption reduction with image quality degradation. This degradation, however, can be acceptable and even unperceivable for the human eye. Lower memory utilization is suitable for safety-critical systems because it reduces the essential and critical bit count, making them less susceptible to faults. Reducing the bit-width used for data representation is also a popular approximation method (RUBIO-GONZALEZ et al., 2013).

Timing Relaxation: the operating voltage can be scaled at the circuit level, impacting the effort expended on the computation of processing blocks inside the clock period. It affects the accuracy of the final result and also the energy consumption (CHIPPA et al., 2014). In (CHIPPA et al., 2014), the authors propose the voltage overscaling of individual computation blocks, assuring that the accuracy of the results will "gracefully" scale with it. Voltage scaling can be implemented in hardware dynamically. Dynamic voltage and frequency scaling (DVFS), for example, is a power management technique used to improve power efficiency, reducing the clock frequency and the supply voltage of the processor (SUEUR; HEISER, 2010).

DVFS can cause data cells to be stuck with a specific value because it diminishes the threshold between a logical one and zero. This type of approximation impacts the integrity of the hardware and the precision of the data. Timing relaxation can also be implemented in software. On parallel programs, it is achieved by relaxing the synchronization between execution tasks (MISAILOVIC; KIM; RINARD, 2013).

Read/Write Memory Approximation: Consists of approximating data that is loaded from or written in the memory, or the read/write operations themselves. This is primarily used on video and image applications, for example, where accuracy and quality can often be relaxed, to reduce memory operations (RANJAN et al., 2015; FANG; LI; LI, 2012). In (TIAN et al., 2015), the authors propose a technique that uses dynamic bit-width based on the application accuracy requirements, where a control system determines the precision of data accesses and loads. The authors claim that it can be implemented to a general-processor architecture without the need for hardware modifications by communicating with off-chip memory via a software-based memory management unit. Approximation can also be applied to the cache memory. In the event of a load data cache miss, the processor must fetch the data from the following cache level, or at the main memory. This can be a very time-consuming task. Load value approximation can be used to estimate an approximate value instead of fetching the real one from memory. In (SUTHERLAND; MIGUEL; JERGER, 2015), the authors present a technique that uses the GPU texture fetch units to generate approximate values. This approximation causes an error of less than 0.5% in the final image output while reducing the kernel execution time in 12%. In (SAMPAIO et al., 2015), the authors propose an approximation technique for multi-level cell STT-RAM memory technologies by lowering its reliability up to a user-defined accuracy loss acceptance. This memory technology has a considerable reliability overhead, which can be reduced. They selectively approximate the storage data of the application and reduce the error-protection hardware minimizing error consumption.

Memory Access Skipping: using a combination of the memoization and function skipping techniques, it is likewise possible to skip memory accesses. Uncritical data can be omitted, as long as it will not heavily damage the output accuracy. Approximate neural networks can skip reading entire rows of their weight matrices as long as those neurons are not critical, reducing energy consumption and memory access, and improving performance (ZHANG et al., 2015).

The presented approximation techniques can be implemented in a multitude of ways, on various device levels with different impacts on the system behavior. Approximate computing at the software level is less presented in the literature than it is at the architecture level. This is probably due to the origins of approximation being on energy consumption reduction and neural network applications.

2.2 Technological Implementations

Approximation techniques are applied to all the computational levels, as showed Figure 2.1. Likewise, they can be implemented in many abstraction levels. Selective approximation is an example of a technique that can be applied at the software and architectural computation stacks, and implemented via software code modification, programmable hardware, and even circuit level. Loop-perforation can also be achieved via code modification for embedded software and programmable hardware (using HLS), or directly with HDL project modifications. The way the approximation techniques are technologically implemented also has an important impact on their performance.

One of the problems with selective and functional approximation is it introduces error on the system output that is sometimes too big to be acceptable. The works at architectural level of approximate operators (SHAFIQUE et al., 2015), for example do not present a significant hardware implementation area reduction when compared to a traditional operator. Indeed, some of the approximate operators presented by (SHAFIQUE et al., 2015), not only have no hardware area gains, but take more area than traditional operators. The size of the used area on programmable hardware devices has a direct impact on system reliability (WIRTHLIN, 2015). Therefore, the quality loss (in this case manifested as errors in some operation results) introduced by the approximation would only be acceptable by safety-critical systems if it sharply reduced its area.

Developing alternate approximate versions of an algorithm is a very time-consuming and intellectually demanding work. To deal with this issue, some works propose frameworks that identify approximable portions of code. At (ROY et al., 2014), the authors present a framework to discover what are the data that can be approximated without significantly interfering with the output of the system. They do so by injecting faults in the variables and analyzing how they affect the quality of the output. Another method is to identify parts of an application code that can be executed on approximated hardware, saving resources and energy (ESMAEILZADEH et al., 2012). The type of approximation to be applied to the approximable parts of the application would depend on the application in question and the project requirements. Although those frameworks are presented as general-use, the question remains if they really can be applied to every type of algorithm. Because they base their methodology on simulation, it is hard to believe that they are able to cover every possible kind of fault that can affect every system.

One of the techniques with the most straightforward implementation is data precision reduction. The way it can be used to approximate software and FPGA applications is obvious: it is a matter of code modification. In software, the precision of floating-point units can be easily modified with the use of dedicated libraries, or even by merely changing the type of the variable. The same can be done at VHDL/Verilog projects: a design can be adapted to process smaller vectors of data. Data precision reduction can bring good improvements in area and energy costs for hardware projects, but frequently do not present high costs reduction on software. Fixed-point arithmetic, for example, can be used to approximate mathematical functions, such as logarithm, on FPGA implementations providing low area usage (PANDEY et al., 2013). On software, however, it can increase the execution time of the application because all the operations and data handling routines are implemented at the software level.

Similarly, the loop-perforation technique can be implemented both at software and programmable hardware code. The difference is that, on programmable hardware, a loop might be implemented either as many circuits executing in parallel (one being each iteration of the loop) or one circuit that is re-executed in a timeline. Therefore the impact of loop-perforation on software and hardware implementations can be very different. On software, it will mainly impact the execution time of the application, while in an FPGA implementation, it could also affect the energy and area consumption.

The timing relaxation through voltage scaling technique can be applied at both the processor architecture level and programmable hardware. At the architecture level, voltage scaling is implemented during the design of the circuit. Most FPGA manufacturers make the voltage scaling of the device possible through easy-to-use design tools. Even though it will impact the performance of a software application, it is not part of the software approximation group because its implementation has no direct connection with software development.

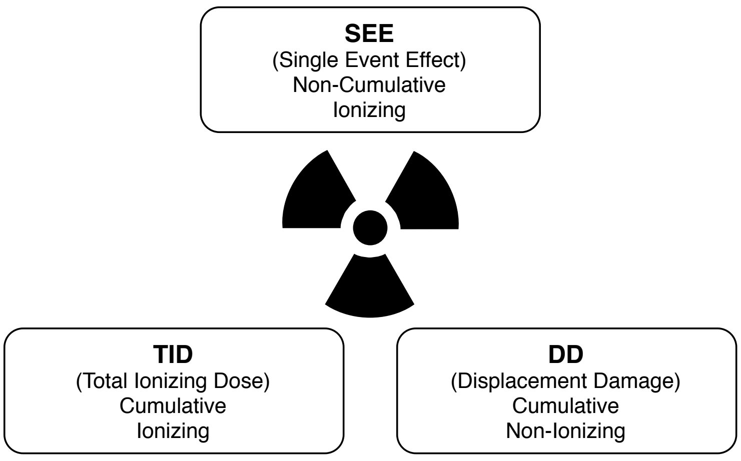
3 RADIATION EFFECTS ON ELECTRONIC DEVICES

Radiation can affect electronics devices in multiple ways, as expressed in Figure 3.1. Single event effects (SEE) are non-cumulative and caused by single events that trigger transient upsets. Total ionizing dose (TID) and displacement damage (DD) are cumulative, which means their effects get worse over time as the system is exposed to radiation. Notice that not all radiation effects are ionizing. As will be further detailed, DD is caused by the kinetic energy of particles. For DD, another physical measurement for energy transfer is used in place of LET, but with similar modeling purposes.

The rate at which soft errors occur in a system is called soft error rate (SER). SER is caused in semiconductor devices mainly because of three sources of radiation: alpha particles, high-energy cosmic rays, and low-energy cosmic rays (BAUMANN, 2005). An ion traveling through a silicon substrate loses energy, generating one electron-hole pair for each $3.6eV$ lost. The linear energy transfer (LET) of an ion defines how much it can interfere with the proper device operation. It depends not only on the mass and energy of the particle but also on the material it is traveling in (represented in units of $MeVcm^2/mg$).

Alpha particles are an important source of ionizing radiation and derive from the naturally present impurities in device materials. It is one of the radiations that can be emitted when the nucleus of unstable isotopes decay to a state of lower energy. Uranium and thorium are the most active radioactive isotopes, and therefore the dominant source of alpha particles in materials alongside their daughter products. Their decay can produce alpha and beta particles, but the latter is not critical for SER because they do not

Figure 3.1: Radiation effects in electronic devices.



Source: Author.

emit enough energy to cause ionization and provoke a soft error. Alpha particles induce electron-holes pairs in their awake and traveling in the silicon. In a packaged semiconductor product, the main source of alpha particles is the package materials, not the materials of the semiconductor device (BAUMANN, 2001).

Neutrons from cosmic radiation with high energy can react with the silicon nuclei and produce secondary ions that induce soft errors. Indeed, cosmic radiation is one of the leading sources of errors in DRAM (NORMAND, 1996), and neutrons are the most likely cosmic radiation to cause soft errors in devices at ground level. Cosmic rays interact with the earth's atmosphere and produce cascades of secondary particles, which continue deeper, creating tertiary particles and so on. Less than 1% of the original flux reaches sea-level altitudes, and most of the particles that arrive consist of muons, pions, protons, and neutrons (ZIEGLER; LANFORD, 1980). Muons and pions have a short life, and protons and electrons are attenuated by Coulombic interactions with the planet atmosphere. Neutrons, however, have a higher flux and stability. Neutrons do not generate ionization in silicon alone. They interact with the chip materials breaking excited nuclei into lighter fragments.

Low-energy cosmic rays induce radiation with the interaction of their neutrons with boron, producing ionizing particles. Very low energy neutrons ($\ll 1MeV$) react with the nucleus of ^{10}B, which breaks releasing energy in the form of a ^{7}Li recoil nucleus and an alpha particle. The alpha particle and the lithium nucleus generated from the absorption of the neutron by the ^{10}B are launched in opposite directions in order to conserve momentum. They are both capable of inducing soft errors, especially in new technologies of lower voltage.

The faults induced by radiation can become errors that might evolve into failures. By definition, a fault is the event itself, manifested as a bit-flip on a memory component, for example. The error, on the other hand, is the effect of the fault on the system. It can pass by unperceived, or be masked by a fault tolerance mechanism. When the system misbehaves, and this is noticed by the user or propagated to another part of the system that, in its turn, shows a problematic external behavior, we say that a failure happened. Taking the example of a fault affecting the memory circuit, the definition of the events would be the following: the bit-flip on the memory data is a fault, the error is the impact it has in the data being stored in the word where the fault was raised, and the failure would be the malfunction of the software that could, for example, use this data as a control variable of a loop, causing the application to never finish its execution. Notice that the fault could have happened in an unused word of memory, causing no errors. Similarly, the error could have been overwritten by a store instruction shortly before happening, and never turn into a failure.

3.1 Single Event Effects (SEE)

When radiation particles transfer enough energy into the silicon of circuits, they generate transient upsets. Upsets are manifested as bit-flips in any part of the circuit that holds data, causing errors (POIVEY et al., 2001; BAUMANN, 2005). In microprocessors, bit-flips can occur in all registers and memories of the processor. Table 3.1 summarizes the different SEE types and their characteristics. The characteristics exposed at Table 3.1 stand for:

- **Non-Destructive:** an SEE that do not cause permanent damage to the system.
- **Destructive:** an SEE that can cause permanent damage to the system.
- **Reset Needed:** the SEE requires a full reset of the system so that it can vanish (i.e., it is a not permanent error).
- **Power Cycle Needed:** A simple reset of the system might not be enough to clean the error. Those errors are often related to physical problems affecting operation of the transistors or logic gates.

Notice that it is possible for an SEE to be both destructive and non-destructive: some of them can be destructive only in some cases, normally related to the intensity or locality of the fault. The SEE types presented at Table 3.1 are defined with more details below:

Single Event Upset (SEU): as soft errors are commonly referred to. Those are non-permanent errors affecting one single bit of one word of data.

Multibit Upset (MBU): occurs when the radiation event has energy high enough to flip multiple bits on a single word. This can be especially problematic for memory circuits that make use of error correction codes, compromising those that cover and mask only one bit (MAIZ et al., 2003).

Multicell Upset (MCU): occurs when the radiation event has energy high enough to affect multiple bits on different localities. The difference between and MBU and an MCU is that the latter consists of bit-flips affecting various parts of a system (e.g., different memory words), while the former consists of multiple bit-flips in a single word (IBE et al., 2006).

Single Event Transient (SET): considered when transient upsets occur in the combinational logic part of the circuit. If propagated and latched into memory elements, those can lead to soft errors (BENEDETTO et al., 2004).

Single Event Functional Interrupt (SEFI): when a soft error occurs in a critical control circuitry (e.g., branch prediction or jump address), it can cause the processing to misbehave to the point where its proper execution is compromised (KOGA et al., 1997). SEFIs lead to a direct malfunction that is easily noticed by the user (e.g., when the application ceases to respond), instead of soft memory errors that may pass unperceived.

Single Event Latch-Up (SEL): SEEs can induce a latch-up by turning on CMOS parasitic bipolar transistors between the well and the substrate (BRUGUIER; PALAU, 1996). SELs are debilitating because a reset (powering it off and back on again) is necessary to remove it. They can also cause permanent damages.

Micro Single Event Latch-Up (μSEL): this type of latch-up is related to the reduction of the transistor size and operating voltages. One of the major differences between μSEL and SEL is that the latter usually occurs in the terminals of a logic gate, while the former occurs in different areas and levels of the die, provoking different effects (AZIMI; STERPONE, 2017). Also, μSEL can occur under ground-level radiation (TAUSCH et al., 2007).

Single Event Burnout (SEB): when a heavy ion passes through a power MOSFET biased in the off state (blocking a high drain-source voltage), transient currents generated by it might turn on a parasitic bipolar-junction transistor that is inherent to this device structure. A permanent short between the source and the drain of the MOSFET is then created due to a regenerative feedback mechanism affecting the new parasitic transistor, which increases collector currents provoking a breakdown (JOHNSON et al., 1996).

Single Event Gate Rupture (SEGR): provoked by a dielectric breakdown of the gate oxide, caused by heavy ions. The heavy ion accumulates charge at the Si-SiO_2 interface in the gate-drain overlap region and results in electric fields in the gate oxide that cause a localized rupture. That rupture causes a permanent short between the gate and the drain of the transistor (JOHNSON et al., 1996).

Table 3.1: SEE classification and key characteristics.

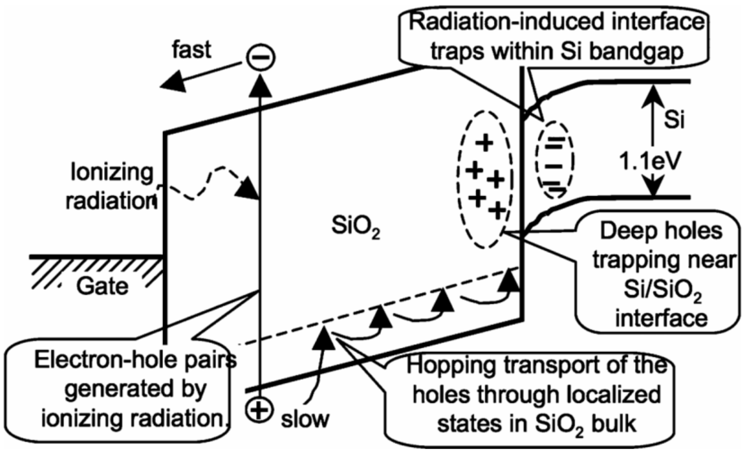
SEE	Meaning	Characteristics			
		Non-Destructive	Destructive	Reset Needed	Power Cycle Needed
MBU	Multibit Upset	X			
MCU	Multicell Upset	X			
μSEL	Micro Single Event Latch-Up	X			X
SEL	Single Event Latch-Up	X			X
			X		
SEB	Single Event Burnout		X		
SEGR	Single Event Gate Rupture		X		
SHE	Single Hard Error	X			X
			X		
SEFI	Single Event Functional Interrupt	X		X	X
SET	Single Event Transient	X			
SEU	Single Event Upset	X			

Single Hard Error (SHE): a sufficiently energetic heavy ion that strikes a MOS transistor gate can locally transfer enough ionizing dose to affect its electrical parameters permanently (DUFOUR et al., 1992). It consists of a total ionizing dose error from a single ion, that can affect SRAM memories (POIVEY et al., 1994)

3.2 Total Ionizing Dose (TID)

Apart from non-cumulative SEE, radiation can also provoke cumulative effects. This type of effect accumulates over time, altering the regular operation of the devices, to the point where they start to become a severe problem. Such is the case of total ionizing dose (TID), which is caused by the same physical event that can cause SEEs: the generation of electron-hole pairs. The difference between SEE and TID is that TID is accumulated over time in the device and provoke gradual and permanent changes in the

Figure 3.2: TID in a CMOS transistor.


Radiation-induced interface traps within Si bandgap
fast
Si
1.1eV
Ionizing radiation
SiO$_2$
Gate
Deep holes trapping near Si/SiO$_2$ interface
Electron-hole pairs generated by ionizing radiation
slow
Hopping transport of the holes through localized states in SiO$_2$ bulk

Source: (ANDREOU et al., 2015).

device behavior.

TID accumulates in the device when the electron-hole pair caused by radiation effects is separated by the electric field concentrated in the transistor gate or field oxide. The electric field prevents the electron-hole pair recombination, and because the now free electrons have high mobility, they are swept from the oxide, leaving holes behind (with low mobility). The holes get trapped in the oxide bulk and at the Si-SiO$_2$ interface, as shown in Figure 3.2. Those trapped holes modify the threshold voltage of the transistor by attracting electrons in the inversion channel, affecting the drain-source current. Because the number of electron-hole pairs is directly proportional to the total amount of radiation dose which the device is subject to, this effect will increase over time and change the characteristics of the transistors. The number of pairs also depends on the dose rate and the gate-oxide electric field, as well as the thickness of the oxide (ANDREOU et al., 2015).

3.3 Displacement Dammage (DD)

Devices can also be subject to non-ionizing cumulative effects. Non-ionizing dose deposited by radiation can be a source of failures in the form of displacement damage (DD). The kinetic energy of the irradiated particles is transferred to the material and can produce atomic displacements. The rate on which this energy is passed to the material is called non-ionizing energy loss (NIEL). The DD energy deposition per unit of mass of material can be calculated by the product of the NIEL and the particle fluence (Φ_i) (JUN et al., 2003), as expressed in (3.1).

$$DD = NIEL_i \cdot \Phi_i \tag{3.1}$$

The NIEL can be calculated as (3.2) shows, where N is the number of atoms per cubic centimeters of the area being affected, T_M is the maximum transferred energy by the collision of an ion with the atoms of the material, T is the energy of the recoiling atoms, T_d is the threshold energy for atomic displacements ($21eV$ for silicon), $\frac{d\sigma}{dT}$ is the differential cross section for atomic displacements and $L(T)$ stands for the Lindhard partition factor. $L(T)$ takes into account that only some of the energy of the recoil will actually go into producing displacements (MESSENGER et al., 2003).

$$NIEL = N \int_{T_d}^{T_M} T \frac{d\sigma}{dT} L(T) dT \tag{3.2}$$

Displacement damage can be caused by protons, neutrons, alpha particles, and high energy photons. As the equations induce, the displacement damage depends on the type of particle radiation, its energy, the total dose, and radiation flux. Some characteristics of the device also impact the possible displacement damage, such as the operating voltage, frequency, and shielding (both intrinsic and extrinsic).

3.4 Analyzing Radiation Effects

This chapter presented the radiation effects on electronic devices that are discussed in the literature. Although knowing them and their impacts is indispensable, some of them are out of the practical scope of this work. The TID and DD are related to circuit-level hardware malfunctions manifested in the system behavior as errors related to the intensive exposure of the device to a hazardous environment. Much like aging effects on electronic devices, those errors can hardly be deal with by the use of programming methods or design strategies. Because of that, the fault tolerance and approximation techniques presented in this work will not have as their main target the effects by TID nor DD. Instead, the techniques presented in this work will focus on dealing with errors caused by non-destructive SEEs such as SEUs, MBUs, SETs, and SEFIs.

4 ANALYSIS METHODOLOGIES

The reliability of a system to radiation-induced transient faults can be measured in many different ways, depending on the available data and experiments performed. Some of the most used metrics for reliability and fault tolerance of safety-critical systems under radiation are the mean work to failure (MWTF)(REIS et al., 2005), the cross-section, and failure in time (FIT) (BAUMANN, 2005), alongside with the already discussed SER (soft error rate). The cross-section (σ) is defined as the area of the device that is sensitive to radiation, with (4.1). A larger cross-section means that a particle that hits the device is more likely to produce a failure. Thus, a design of smaller area (such as an approximate one) will typically present a smaller cross-section. The FIT is commonly as a means to express SER and is equivalent to one failure in 10^9 hours of device operation. MWTF is particularly interesting for this works discussing because it presents a correlation between performance and the fault tolerance of a technique, and is presented in (4.2).

$$\sigma = \frac{\text{number of errors/failures}}{\text{fluence of particles}} \tag{4.1}$$

$$MWTF = \frac{\text{amount of work completed}}{\text{number of errors encountered}} \tag{4.2}$$

When analyzing data from simulation experiments, the error occurrence is often presented as a simple percentage. In this type of analysis, faults are injected into the system, and it is often possible to trace the types of errors and their origin. Thus, it is easy to calculate the percentage of faults that caused errors (and failures) and their types. When analyzing fault tolerance techniques, especially those implemented on embedded software, metrics like cross-section might not be the most appropriate (in fact, using this type of metric would need an adaptation, because there is no particle fluence in this type of experiment). In those cases, data might be better presented merely as the reduction of the percentage of faults capable of inducing failures.

This work evaluates a multitude of methods and techniques, both for approximation computing and fault tolerance. Given the variety of implementations and their different implications, having one unique evaluation methodology for all of them would be impractical. For that reason, this chapter presents a number of different experimental methodologies. All the here methodologies presented are applied to at least one of the proposed ideas on further chapters. Section 4.1 presents a methodology to inject faults on programmable hardware projects, while Section 4.2 presents an approach to inject faults on embedded software, targeting the register file of the processor. Similarly, Section 4.3 presents a methodology to inject faults on the processor register file, but this time using software simulation. A laser fault injection methodology targeting embedded processors is presented at Section 4.4.

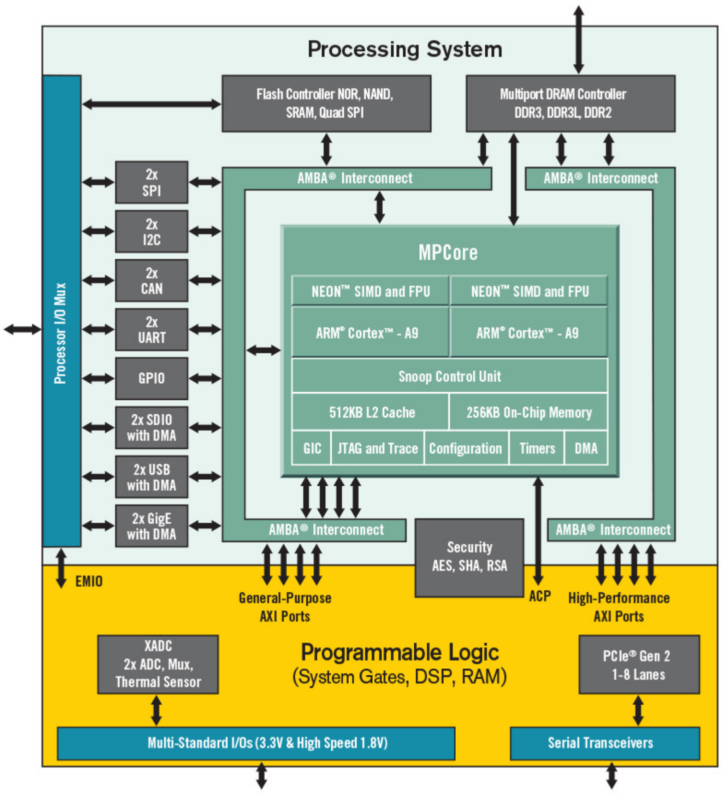
The proposed ideas are implemented and tested in the same hardware. The designs are implemented in a Zynq-7000 APSoC, designed by Xilinx, represented in Figure 4.1. The Zynq board has embedded a high-performance ARM Cortex-A9 processor with two cache levels on the processing system (PS), alongside a PL layer. The PL presents an FPGA based on the Xilinx 7-Series with approximately 27.7Mb configuration logic bits, 4.5Mb block RAM (BRAM), 85K logic cells, and 220 DSP slices, with a frequency of 100MHz. The dual-core 32-bit ARM Cortex-A9 processor runs a maximum of 666MHz and is designed with 28nm technology. It counts with two L1 caches (data and instruction) per core with 32KB each, and one L2 cache with 512KB shared between both cores. A 256KB on-chip SRAM memory (OCM) is shared between the PS and PL parts, and so is the DDR (external memory interface).

4.1 Onboard Fault Injection Emulation on FPGA

Phenomena such as power glitches, electromagnetic interference, and ionizing radiation can cause transient effects on electronic devices. Considering storage elements, such as flip-flops and SRAM memory cells, those effects may cause bit-flips, which are the change of the storage value from a logical zero to a logical one, or vice-versa. Fault injection emulation can be used as a means to assess the reliability of different designs and fault tolerance techniques. SRAM based FPGAs have a massive amount of SRAM memory conveying the configuration information required to program the general-purpose FPGA to a specific function. On Xilinx SRAM based FPGA, such configuration memory is organized in rows, columns, and frames. Each frame includes 101 words of 32 bits, defining the configuration for key FPGA elements such as signal routing switch boxes, multiplexers, and combinational logic truth tables.

The onboard fault injection methodology used in this work to inject faults on programmable hardware is based on the one presented by (TONFAT et al., 2016). It explores

Figure 4.1: The Zynq-7000 APSoC.



Source: (XILINX, 2017).

the use of the Xilinx internal configuration access port (ICAP) hardware module, available on Xilinx's 7 Series FPGAs and APSoCs, to read and write at configuration memory frames. To inject a fault on a specific bit, the frame is read by the fault injection engine using ICAP, the bit content is XOR'ed, and then the frame is written back to the configuration memory. The fault injection engine is implemented on the same FPGA as the DUT but isolated from it with proper floorplanning of the design. It communicates with a host computer that coordinates the injection via a serial port. A campaign planning script running on the computer defines where a bit-flip is to be injected. Two types of fault injection are performed using this method:

- **Random Accumulated Fault Injection:** Bit-flips caused by ionizing radiation are emulated by injecting faults randomly over the area allotted to the DUT. These bit-flips are accumulated over time, as would happen if the system were under ionizing radiation. When an error is detected, the number of faults accumulated until that point is recorded, and the FPGA is reprogrammed, cleaning up all previous bit-flips. This procedure is repeated until a sufficient number of errors is collected, allowing statistical analysis of the design reliability. In that type of injection, it is possible to analyze the fault tolerance of the DUT under an accumulated number of faults and how a different number of faults may impact the output of the system.

- **Exhaustive Fault Injection:** The exhaustive fault injection consists of injecting faults on every bit of the DUT configuration memory. The effect of the fault on the system output is then evaluated. This type of fault injection allows the categorization of the bits as essential and critical bits (TONFAT et al., 2016). Essential bits are those that are used to program the FPGA. A critical bit is an essential bit on which a bit-flip will provoke an error in the system.

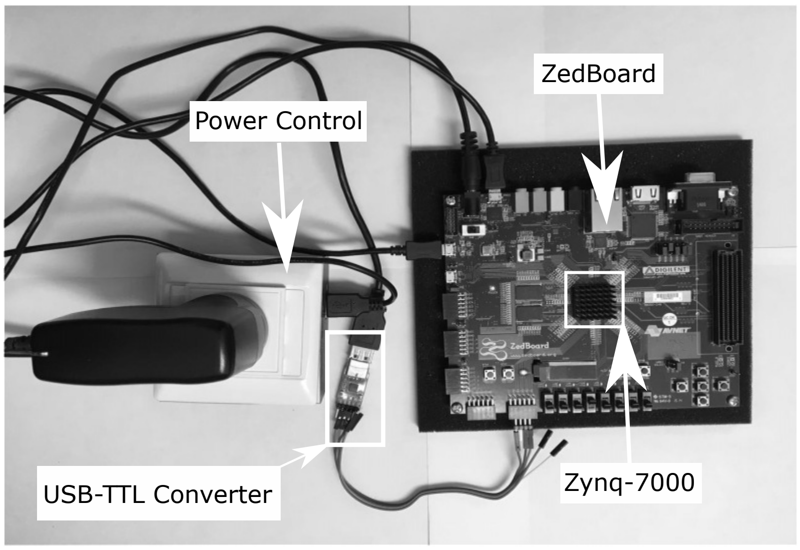
4.2 Onboard Fault Injection Emulation on Embedded Processor

In this work, the ARM processor embedded at the Zynq-7000 APSoC (Figure 4.1) will be used as the target for the software implementation experiments. As Figure 4.1 shows, the board has two embedded ARM Cortex-A9 processor cores. The MPCore (the unit that contains the processor cores, cache memory, OCM, and some configurations and processing units that are out of the scope of this work) communicates with external peripherals through the ARM advanced microcontroller bus architecture (AMBA). AMBA defines the communications standards and defines the AXI interface for communication with the programmable logic layer of the board.

When analyzing embedded software projects, one of the best ways to assess its fault tolerance is by injecting faults in the processor register file. The register file area of the ARM processor is physically minimal. Because of that, physical fault injections experiments (e.g., heavy-ions radiation) targeted to inject specifically in the register file is often impracticable. This work proposes a methodology that uses a register file fault injector implemented at the FPGA layer of the Zynq-7000 APSoC. The injector shall access the register file without compromising the normal system execution in order to perform a reliable register file fault injection. This is assured by the AXI protocol and the fault-injector design, which only accesses the register, which will be targeted by the fault injection at the moment. The adopted methodology follows the scheme presented in (OLIVEIRA; TAMBARA; KASTENSMIDT, 2017). The fault injection emulation system consists of the following modules:

- **Injector Module:** Intellectual property (IP) designed in hardware description language and implemented in the FPGA layer of the Zynq board. It is responsible for

Figure 4.2: Onboard register file fault injection setup view.



Source: Author.

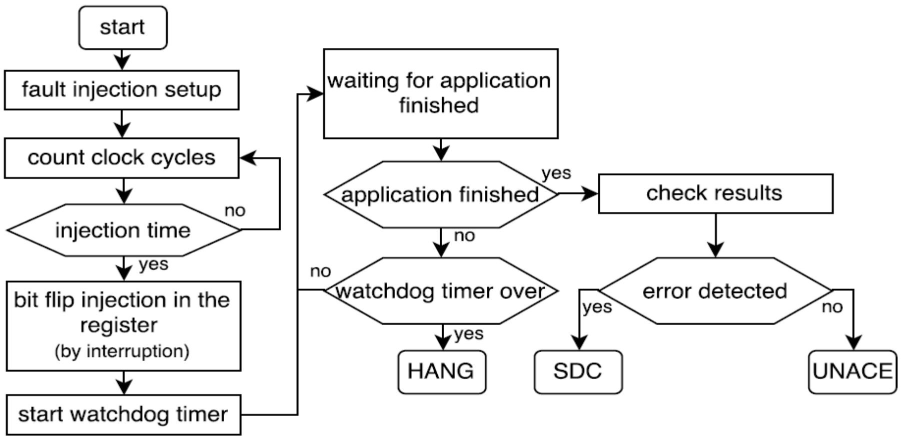
performing the fault injection procedure to be detailed further.

- **Power Control:** Electrical device in charge of powering up the board in each injection cycle.
- **System Controller:** Software application running on a host computer responsible for Power Control management. It also saves the fault injection logs, which are received by serial communication.

Figure 4.2 presents the experiment setup environment. The Zynq board and the power control are connected to a host computer. The host computer is responsible for controlling the system and registering experiment logs. A USB-TTL Converter is responsible for transmitting serial data containing information about the error and is connected to the DUT and the host computer.

The injector randomly injects bit-flips on the processor's register file. The affected ARM registers are the general-purpose ones, from r0 to r12, and the specific ones, which are the stack pointer (sp), link register (lr), and the program counter (pc). The faults are injected using an interrupt mechanism that locks the processor and applies an XOR mask to the target register, provoking a bit-flip. The target register and bit to be flipped are randomly defined. The injection time is also randomly determined, being a random point between the start and the finish of the software execution. It is intended to simulate real scenarios, where the fault can affect the system at any moment. Figure 4.3 presents a procedure flow performed by the injector module. First, the injector is configured with

Figure 4.3: Onboard fault injection emulation procedure flow.



Source: (OLIVEIRA; TAMBARA; KASTENSMIDT, 2017).

all the random injection data defined by the ARM CPU, as generating random numbers in FPGA logic has high complexity. Once configured, the injector counts clock cycles until it reaches the injection time. Next, an interruption is launched to inject the bit-flip in the processor register defined in the configuration. At the end of the application, the module compares the application results with a golden execution output (i.e., without fault injection) to check for errors. This emulation fault injection can be programmed to inject one or more faults per execution of the benchmark algorithm.

Figure 4.3 shows that there are three possible classifications to be made at the end of the fault injection regarding the impact of the fault in the system: hang, SDC, and unace. Those classifications set a standard that is to be used on the other methodologies as well. An unace means that the injected fault caused no errors to the system, i.e., the system did not crash, and the memory is as it was supposed to be (compared to the golden execution). An SDC means that there is at least one silent data corruption, manifested as a difference in the memory data compared to the golden execution output. A hang means that the system crashed, becoming wholly unresponsive or initiating an infinite execution loop.

4.3 Fault Injection Simulation

This work proposes a fault injection via simulation using the OVPSim simulator (IMPERAS, 2017). OVPSim is a full-system simulator used to simulate the execution of code in the target hardware. It uses a just-in-time binary translation, achieving high simulation speeds. That makes OVPSim an instruction-accurate simulator, providing the possibility to analyze the execution of each individual instruction, but not real execution times (e.g., clock cycles). OVPSim and provides public APIs which allow full control of the target simulation and the implementation of the fault injection module. The OVPSim was chosen instead of other popular cycle-accurate simulators (like gem5) because it provides a more reliable processor model. Gem5 targets microarchitecture exploration, which incurs substantial simulation overheads due to the number of modeled aspects. OVPSim also provides better APIs for simulation development than other simulators. Moreover, it has an active development and support team.

The applications execution are simulated on a single-core ARM Cortex-A9 model. This processor was selected due to its presence on COTS devices that are used for safety-critical applications, such as the Xilinx Zync-7000 APSoC, that is extensively used in this work. The model used to simulate the ARM Cortex-A9 was the one developed to be specially used at OVPSim, with ARMv71 architecture. This model is extensively used and validated by embedded software developers, which use the OVPSim simulator to test their projects.

The OVPSim-FIM (OVPSim Fault Injection Module), developed by (ROSA et al., 2015), was slightly modified and configured to be used in this work for fault injection and error evaluation. In this work, the fault injection simulation will only inject faults in the processor register file. A fault is modeled as a bit-flip, to be injected into a register in a certain instruction count (ICOUNT). The ICOUNT holds the number of instructions that were executed so far by OVPSim. Because OVPSim is instruction-accurate and works with just-in-time binary translation, the best way to define the injection moment is with an ICOUNT, which represents an execution point in time.

OVPSim runs the simulation of the benchmarks applications execution on the processor model, finishing the simulation after it reaches a predefined point that is hard-coded in the application code. OVPSim-FIM injects a single fault per simulation execution. All faults are randomly generated; that is, they are random bit-flips, in a random register, in a random ICOUNT. OVPSim-FIM counts with a fault monitor function, which checks the system behavior, dynamically detecting some types of errors, such as hangs (defined at Table 4.1). The results are generated comparing the executions under the effects of fault injections with an error-free run (golden execution) of the system.

The proposed simulation fault injection methodology is divided into five phases according to fault injection activities and results gathering:

1. **Golden Phase:** In this phase, the application is executed with no fault injections.

It is the pure execution, to be taken as a successful run of the application (i.e., the golden run). The applications are compiled using the proper cross-compiler, then executed in the OVPSim simulator. The execution output is saved in a report called "golden report". This report contains vital information, such as the first and final ICOUNT, the memory dump of the application after the golden execution, and the final values of the registers. The first and final ICOUNT define the possible fault injection window and are set by two instructions that are added into the application code. Those instructions are added at the beginning and the end of the code and don't count as possible injection points. They are used solely for fault modeling purposes and do not affect the final application solution or execution.

2. **Fault Creation Phase:** This phase is where faults are created using the information gathered in the previous phase. A single bit flip in the logical registers is provoked by an XNOR operation with a mask. The ICOUNT data of the golden phase is used to determine the possible fault injection execution times. OVPSim-FIM calculates a random injection execution time (i.e., ICOUNT), limited between the first and the final ICOUNT of the golden phase execution. The fault location (that is, the register where the fault is to be injected) is also randomly defined. A fault list is then created. Each fault is represented by a structure containing an ICOUNT, a register, and a mask.
3. **Execution Phase:** At the execution phase, the fault injection module monitors the current ICOUNT while the application is executed in OVPSim, and injects the fault at the fault's ICOUNT, defined at the fault creation phase. The targeted register is then accessed, the mask pattern applied, the resultant value written in the register, and the application execution resumes. OVPSim-FIM system keeps track of the current ICOUNT even after the fault injection, to deal with possible hangs. If an application executes at least 50% more instructions than the golden execution total ICOUNT, it is considered to have a hang error, and its execution is halted (otherwise, the simulation could execute indefinitely). The OVPSim-FIM also watches for possible OS exceptions, such as segmentation faults. When this type of error happens, it halts the simulation and reports the problem.
4. **Error Detection Phase:** In this phase, a comparison between the golden phase execution (golden execution, fault-free) and the execution phase (faulty execution) executions is made. Errors are classified in four different groups, defined in Table 4.1.

Table 4.1: Error type classifications for simulated fault injections using OVPSim-FIM.

Group	**Error Definiton**
Hang	Causes the application to be stuck in a certain point.
SDC	Difference in the final memory from the golden phase execution and the fault-injected test executions.
Unace	Injected fault caused no error.
Exception	Error captured by the Operating System.

5. **Final Phase:** At this phase, the results from all fault injections and errors found are grouped in a single report, which is the one used to generate the final data.

The OVPsim-FIM test reports present over sixteen types of errors. For practical reasons, those types of errors are re-classified on four different groups: hang, SDC, unace, and exceptions. Those groups are defined in Table 4.1. As will be discussed in Chapter 5, the exception error group can be further categorized to provide a better view of the types of errors.

4.4 Laser Fault Injection

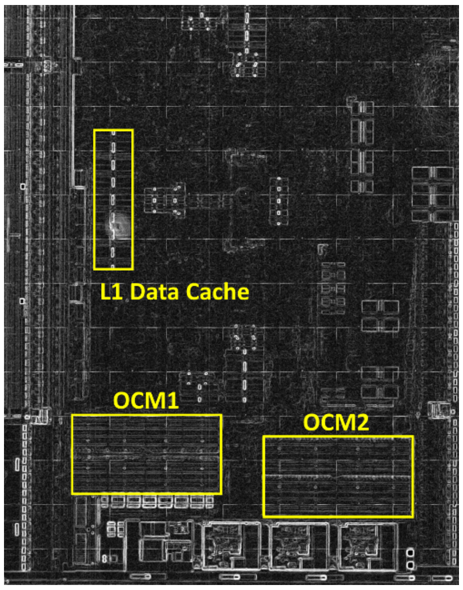
Laser testing is commonly used as an in-lab tool for injecting transient localized perturbations into a device under test by photoelectric stimulation, especially for SEEs investigations (BUCHNER et al., 2013), security evaluation (TRICHINA; KORKIKYAN, 2010), and more generally to evaluate the fault tolerance of an application (POUGET et al., 2008).

The fault injection experiments were performed on the two-photon absorption (TPA) microscope of IES laser facilities, University of Montpellier. The TPA method was preferred to the more classical single-photon approach because it provided a better reproducibility of the fault occurrences in the 28nm DUT (dual-core 32-bit ARM Cortex-A9 processor embedded at the Zynq-7000 APSoC) with a thick substrate (700μm). The laser wavelength is 1064 nm, with a pulse duration of 30ps, and the beam was focused through the backside of the DUT by a 100$\times$ lens. The DUT was scanned under the static beam using motorized stages.

The laser pulse energy is set at 250pJ. This value was found in previous work (POUGET et al., 2017) to induce between 0 and 3 bit flips per pulse in the region of interest of the DUT, depending on the laser position. This energy is slightly above the energy threshold for a single bit flip. Laser pulses are triggered with a frequency of 10Hz or 20Hz (depending on the benchmark under execution) without any synchronization with the DUT clock, while the target application is executed in loop by the DUT. Making use of this asynchronous approach, the methodology assures randomness concerning the arrival time of the laser pulse and the current application execution cycle. This allows the statistical coverage of any vulnerability time window without the need for unreasonable experiment time. Fig. 3 presents a microphotograph of the processor with the fault injections regions highlighted. Two Regions Of Interest (ROI) are defined: one to cover the L1 data caches of both processors ($215 \times 780\mu\text{m}^2$) and other that covers the OCM (two areas of $820 \times 440\mu\text{m}^2$ each). Both the ROIs were scanned repeatedly for each benchmark execution with a step of 2 μm and 3μm, respectively, for the X and Y-axis. Considering the constant pulse triggering rate, the maximum scanning speed along the x-axis was adjusted to have at least one pulse every μm along x. Due to the acceleration and deceleration phases at the extremities of each scan line, this approach leads to smaller steps along the x-axis between consecutive pulses near the edges of the ROI. This approach was preferred to a strict step in order to maintain a constant laser pulse rate, and thus an accurate control of the number of pulses per application execution cycle. Indeed, in this work, we are more interested in time-related statistics than in the accurate spatial localization of the occurrence of the fault. Notice that the DUT has only one OCM of 256KB, as explained before, but this memory is scattered in the two areas that were defined as one single ROI and presented in Figure 4.4.

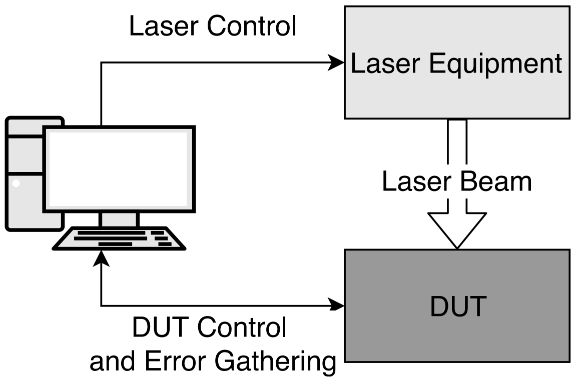
The experiment setup is presented in Figure 4.5. It consists of the DUT, a host computer, and the laser equipment. The host computer is responsible for controlling the laser beam and listening to messages from the DUT. The DUT periodically sends messages to the host computer, to report an error or to confirm it is alive. Error messages are reported when there is a difference between an execution output and the golden output. The golden output is the result of a fault-free execution at the beginning of the experiment, called golden execution. The alive message is essential because some faults will cause the DUT to be irresponsible or hang, needing a reset. A reset consists of re-programming and configuring the DUT and is performed when a timeout occurs while the host computer waits for an alive message from the DUT. This timeout is set to about three minutes but may vary for different experiments with different response times. During a reset, the DUT warns the host computer so that the laser beam is deactivated. It prevents any errors during the system initialization and golden execution. The laser beam is then re-activated

Figure 4.4: Infrared microphotograph of the DUT core under test, showing the scanned areas (L1 Data Cache and OCM).



Source: Author.

Figure 4.5: Laser experiments setup.



Source: Author.

after the host computer receives an alive message from the DUT.

The communication between the host computer and the DUT is rather complex and is highly susceptible to errors because it happens during the fault injection. To avoid errors that are not interesting to our experiment and would make it less efficient, we developed a strategy to reduce this communication to a minimum necessary. During the benchmark execution, the algorithm runs N times, filling in an output vector, which will be then compared with the result from the fault-free execution (golden value). This way, the DUT only has to send messages to the host computer every N runs (and when an error is detected). The value of N may vary for different benchmarks, according to their execution times (those details will be presented at Chapter 7). After each algorithm execution, the output vector is compared with the golden value to check for its correctness. When the output value and the golden value are different, the DUT sends to the host computer

Table 4.2: Error type classifications for laser fault injections.

Type	**Error Definiton**
Hang	Causes the application to be stuck at a certain point.
SDC	Output difference between the golden execution and the one exposed to laser fault injection.
Multi-SDC	Multiple SDC occurrences in the same run, e.g.: multiple positions of the output vector corrupted.

a message containing the details of the error (position on the output vector and the value of the incorrect output). The host computer receives the error messages and saves them into a log to be further analyzed. The errors are classified into three different types: hang, SDC, and multi-SDC. They are defined at Table 4.2. It is important to notice, however, that depending on the objective of the experiment and the system being tested, the studied error types might be different. For instance, a given fault tolerance technique may consider for its coverage analysis Multi-SDCs and SDCs as of equal importance, and present the numbers of both of them simply as SDCs. Table 4.2 merely presents the error types that this fault injection methodology can differentiate.

5 EMBEDDED SYSTEMS FAULT TOLERANCE

Faults caused by radiation on electronic devices can become errors that need to be treated before evolving into failures. The more usual way of doing it on complex systems is with fault tolerance techniques implemented in programmable hardware or embedded software (AVIZIENIS et al., 2004). Fault-tolerant devices are usually expansive, therefore the industry tends to turn to in-house developed fault tolerance techniques. Those techniques shall be able to detect errors and masking (or correcting) them when possible.

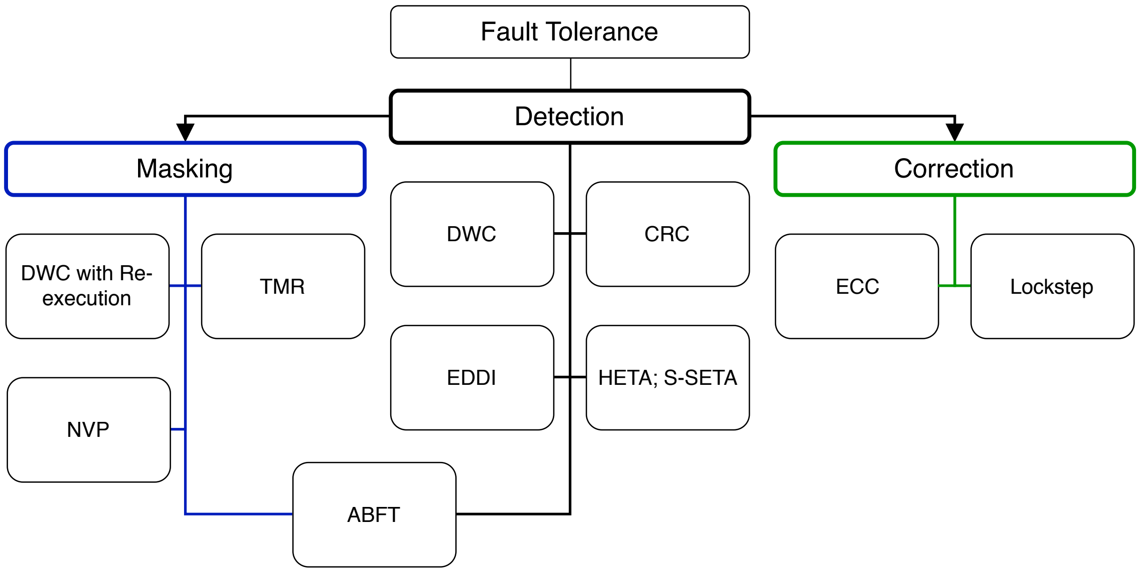
This chapter starts by discussing how fault tolerance is implemented to protect safety-critical systems at Section 5.1. Section 5.2 presents a practical analysis on the reliability of parallel multicore systems. Section 5.3 expands this analysis by evaluating parallel fault tolerance techniques, and Section 5.4 proposes using approximate computing to improve those techniques. Finally, Section 5.5 discusses the general use of approximation on fault tolerance and some of its motivations.

5.1 Fault Tolerance

Figure 5.1 classifies fault tolerance techniques in three major groups concerning their capability. A fault tolerance technique shall be able to detect errors. What it does with this information, however, may vary. As will be further exemplified in this work, for some systems, fault detection is enough. Nevertheless, safety-critical systems often call for error masking or correction. The difference between an error masking and correction is that masking an error consists of keeping the system safe and hiding the error from the end-user (or the rest of a more complex system). An excellent example of this type of fault tolerance technique is triple modular redundancy (TMR) (SANCHEZ-CLEMENTE; ENTRENA; GARCIA-VALDERAS, 2016), which avoids the use of an erroneous data value, outputting a correct one. Correcting an error is a much harder and complex task, and from a system point of view, the impact would be the same as masking it. As an example, the lockstep technique (OLIVEIRA et al., 2018) finds an error and rolls all the system execution back to a safe-state before the error happened, and then resumes the system execution with the hopes that the error has forever vanished.

The literature presents an enormous set of techniques implemented in software to protect applications against hardware errors. Those are called software-implemented hardware fault tolerance (SIHFT) techniques (GOLOUBEVA et al., 2003), and achieve

Figure 5.1: Fault tolerance techniques classification.



Source: Author.

protection with function redundancy and variables replication. An example of this type of technique is EDDI (error detection by duplicated instructions) (OH; MITRA; MC-CLUSKEY, 2002). EDDI detects faults by comparing two different executions of the program, mapping all numbers on the original computation to new values, and applying transformations to the program so that it can be backward comparable with the original calculation. Techniques like HETA (AZAMBUJA et al., 2013) and S-SETA (CHIELLE et al., 2015) detect control-flow faults and put the system in a fail-safe state. The CFT-tool (CHIELLE et al., 2012) is capable of combining these techniques to detect both SDCs and functional interruption (FI) errors. CFT-tool inserts the fault tolerance methods directly on the Assembly level code of the program to be protected (after the compilation). Nevertheless, and it can present some limitations for complex applications and those which are supposed to run on top of operational systems. Techniques called application-based fault tolerance (ABFT) encode the used data, profiting from unique application characteristics (HUANG; ABRAHAM, 1984). ABFT shall be specifically designed for the application under protection. Therefore, it is not scalable to a high range of applications and tends to be costly in design. Both SIHFT and ABFT come with the cost of execution time overhead.

Redundancy methods such as TMR and duplication with comparison (DWC) are employed in a multitude of systems, both to provide error detection and masking. TMR can be implemented bot to protect a hardware module (QUINN et al., 2017) or software code (QUINN et al., 2015). It consists of triplicating the hardware (or software code) and voting the output of the redundancies: if at least two results match, it is considered the correct output. When TMR is used on hardware, it mainly implies an area overhead. When it is applied to software, it mostly provokes execution time overhead. DWC techniques are only capable of detecting errors, but can implement re-execution methods to provide error masking. This way, an error can be detected and mitigated before becoming a failure. DWC techniques have an overhead of two times the execution time of the original application for pure redundancy and three times when applying re-execution for error masking. N-version programming (NVP) (CHEN; AVIZIENIS, 1995) is a programming strategy that consists of developing a number of different (but equivalent) algorithms from the same specification. With this method, designers hope to achieve fault tolerance via code redundancy (voting the results from each one), expecting that two different programmers independently generating code would not produce software that is susceptible to the same errors.

Cyclic redundancy check (CRC) (KOOPMAN; CHAKRAVARTY, 2004) is commonly used on network and storage systems to detect errors affecting the stored data. This error checking method is broadly used on network systems because it is easy to implement on hardware and perfect to detect burst errors, as well as those caused by noises in the transmission. A multitude of CRC designs is proposed in the literature, but it consists of check values based on the calculations of polynomials, that shall be re-calculated to verify if the check value remains the same. If it is not, there is an error in the data. CRC can be used as a first step for error correction. Error correction codes (ECC) (DONG; XIE; ZHANG, 2011) is also presented in the literature in various forms. Hamming ECC, for instance, is extensively used to protect NAND flash memories against errors. This method provides the correction of one error and the detection of up to two errors (with no correction possible in this case).

In (FAYYAZ; VLADIMIROVA, 2014), an approach based on task level migration is proposed as fault tolerance for aerospace FreeRTOS applications on multi-processor systems. The technique consists of migrating tasks from a faulty processor to a fault-free one. The detection of the error is done by middleware blocks assumed to be fault tolerant. The problem with that approach is that a high amount of assumptions must be taken beforehand by the programmer. Decisions (like which tasks will be migrated to which processor nodes) are made in the programming phase. Unfortunately, a programmer can never safely predict which processing nodes will fail.

The authors in (HUANG et al., 2014) proposed a Python-based programming model (PyDac) to improve the reliability of heterogeneous many-core systems. They evaluate this proposal in an architecture composed of six ARMv2a high throughput soft cores, and one embedded PowerPC, which is optimized for single-thread performance. Running on a Linux kernel in the embedded PowerPC, PyDac decomposes the application in redundant parallel subtasks scheduled on the soft cores with Python virtual machines. The system dynamically checks the results of the subtasks to ensure the resilience of the six soft cores. However, the authors assume that the main PowerPC processor is fault-free, which is not a realistic assumption, and did not implement any technique to protect it against soft errors.

In most of the cited works, recovery approaches are proposed to deal with the error. Nevertheless, a plethora of safety-critical applications may not need recovery. As stated in (FREITAS et al., 2007), real-time systems have to deal with *data freshness* requirements, which defines the time interval on which data is considered valid. For instance, an automatic navigation system may have an error during its execution, but because its data freshness has a minuscule time interval, the error will soon disappear as the algorithm keeps its execution generating a new value. Because of that, an error correction procedure is not always necessary. However, the system shall be aware of the error to put itself in a fail-safe mode. Indeed, in some cases, it is better to warn the user about the error and let him decide how to handle it. Such is the case of some errors that might affect an airplane system, for example. Trying to correct an error in an airplane can cause the system to overwrite the pilot's demands and cause a catastrophe. In those cases, it is often better to warn the pilot that certain data is not to be trusted or alert for a malfunction and let him deal with the situation in the most suitable manner. This type of situation calls for an error detection system (without the masking capability). In those cases, the values of the redundant re-computations are only used for comparison and error checking. That is where a designer may profit from approximate computing, as will be proposed in Chapter 6 (Section 6.3).

5.2 Parallel Embedded Software on Multicore Processors Reliability

Safety-critical systems manage the execution of many resource-sharing applications. Especially for avionics applications, the designs require the Radio Technical Commission for Aeronautics (RTCA) certification to operate in most countries. For software, the DO-178B/C certificate is desired, while for hardware the DO-254 certificate is needed

(HILDERMAN; BAGHI, 2007). This certificate's exigencies are highly conflicting with memory sharing software systems, imposing a large number of limitations and safety measures to avoid catastrophic errors.

There are reasons to limit safety-critical applications for single-core architectures (alongside with other hardware limitations). Performance is not the main concern for that area of engineering. Nevertheless, the tendency for the microprocessors industry is to turn to multicore to achieve better performance. The developers tend to focus on the satisfaction of their larger market shares. As safety-critical systems are not the most abundant consumers of the industry, they may have to adapt to make use of what the industry has to offer, which is hardware designed with a focus on performance, not on fault-tolerance.

The reliability of multicore systems is a major concern for safety-critical applications. Multicore processors tend to be more susceptible to SEUs because of their high level of miniaturization and the tendency to have a large number of memory cells. On the other hand, having access to multiple processing cores opens the possibility to achieve fault tolerance via execution redundancy (MUSHTAQ; AL-ARS; BERTELS, 2013). Given the fact that the industry has turned to multicore processors and parallelism as the *modus operandi* to provide a better performance, it is imperative to understand the implications of parallelism on the reliability of the system. This chapter will discuss the implications of using parallelism on safety-critical systems, and present experimental data acquired by fault injection simulation to evaluate the reliability of those systems. The first section is focused on the embedded multicore systems reliability study as a whole, while the second section will present a study on the effects of using parallelism applied to the fault tolerance techniques themselves (i.e., not only in the algorithm that is being protected).

Having access to multiple processing creates an excellent environment to achieve fault tolerance via redundant execution. A myriad of application program interfaces (APIs) for parallelism is available. One of the most used APIs for that purpose is OpenMP (Open Multi-Processing), which supports multi-platform shared memory multiprocessing programming in Fortran and C/C++. OpenMP is supported by most platforms, processor architectures, and operating systems. Another highly used interface is the POSIX Threads API, usually referred to as Pthreads. Pthreads allows a program to create and manage multiple flows of execution, i.e., threads. It is available on most Unix-based operating systems, such as Linux and MAC OS. Those APIs vary in terms of abstraction level and implementation complexity. Consequently, the resultant application will differ when developed using different APIs, even when using the same parallelization strategy. When applying those applications to safety-critical systems, it is imperative to know their critical failure points and susceptibility to errors.

Bare metal applications are suitable for small systems, offering a significant degree of control during their development. However, those applications' complexity increases quickly, reducing the overall system reliability. Additionally, bare metal applications developed for specific projects are more prone to development errors. In contrast, commercial operating systems (OS) are well known and tested by a large community. Thus, most of the development bugs are assumed to be fixed. To guarantee safe management of resources, a commercial OS such as Linux is attractive. Developing a specific OS for radiation-hardened or safety-critical systems is costly. On the other hand, executing bare metal applications on a complex system could induce a waste of resources that would be better managed by an OS. Using an operational system is also a significant concern because the operating system itself may be affected by faults. The Linux OS susceptibility to soft errors has been studied by (MONSON; WIRTHLIN; HUTCHINGS, 2010). The fault tolerance of the μC-Linux embedded OS, commonly deployed in real-time applications such as the automobile industry, is also well-documented (STERPONE; VIOLANTE, 2007).

When making use of a parallel API such as OpenMP and Pthreads, an operating system such as Linux is needed. Therefore, it is not a negligible part of the system, impacting directly on the system fault tolerance, and needs to be studied. This part of the work aims at investigating the parallelization paradigm effects onto application reliability and their combined impact when deployed alongside a complex OS such as Linux. Additionally, traditional fault tolerance techniques applicability is explored on each of those system configurations. For this purpose, OpenMP and Pthreads applications are evaluated with fault tolerance techniques at the software level. Additionally, the usage of redundancy TMR and DWC with re-execution, here named conditional double modular redundancy (CDMR), is investigated at the software level in single and dual-core processors under fault injection simulation.

Fault injection simulation experiments are performed with the OVPSim, as presented at Section 4.3. The faults are injected into registers in single and dual-core versions of an ARM Cortex-A9 processor during the execution of the benchmark applications. The types of errors are the same presented in Table 4.1. The errors from the exceptions group are divided into segmentation faults and unidentified errors. This categorization was made because it was clear that segmentation fault error was the most recurrent type of exception error, which makes it important. Exceptions that are not categorized as segmentation faults will be called unidentified errors. Bare Metal applications present no exception type errors, because of the absence of an operating system to catch those. The bare metal sequential algorithms versions are only tested on single-core execution, while the Linux applications are evaluated both executing on dual-core and single-core. The impact of Linux is studied by comparing the results of fault injections between the same algorithms running as bare metal applications and on top of the Linux OS. As the algorithms are the same, and so is the simulation and fault injections, it is expected that all the difference between the results is provoked because of the usage of Linux OS. To provide statically relevant results, the fault injection simulation experiments were executed up to the point where the error distribution ceased to vary. Therefore, the number of experiments performed shall be enough to provide reliable results.

Three benchmark applications were tested: Bit Count, Matrix Multiplication, and Vector Sum. Bit Count is an ordinary bit count verification that counts how many bits are set in a given word. Matrix Multiplication is a simple matrix multiplication operation, and the Vector Sum is the sum of two vectors. Those are simple codes focused on pure calculations and processing, making no use of any programming strategy in special.

The first analysis regards the sequential versions of the benchmarks. Two versions of each sequential algorithm are presented. The first one is a bare metal application implementation, which is executed on top of no operating system. Because there is no operating system, there is no process dedicated to the management of the two available ARM cores. Bare metal applications presented in this chapter were executed using only one processor core. The same cannot be said about software that is executed on top of Linux. When running on top of an operating system, an application is subjected to this system's scheduling and resource management algorithms. Considering that a comparison between a dual-core and a single-core execution would be unfair, the results from a single-core version of the Linux execution are presented (i.e., single-core ARM Cortex-A9). In this first analysis, a preliminary idea of the impact of using an operating system on the application fault tolerance is studied. Table 5.1 presents the results of those injections.

The OpenMP and Pthreads parallel versions of each benchmark application are also executed and compared. Those versions of the application are always executed on top of Linux OS (on both cores of the dual-core processor). The resources management is

Table 5.1: Fault injection results in percentage of errors, running sequential applications on single-core ARM Cortex-A9.

		Errors [%]				
		Unace	SDC	Hang	Exceptions	
Application	Version				Seg. Fault	Unidentified
Bit Count	Bare metal	45.6	40.9	13.5	-	-
	Linux	40.4	42.1	0.2	17.0	0.3
Matrix Mult.	Bare metal	58.3	31.3	10.4	-	-
	Linux	34.3	46.8	0.4	17.8	0.7
Vector Sum	Bare metal	56.4	32.4	11.2	-	-
	Linux	55.7	25.7	0.4	17.4	0.8

Table 5.2: Fault injection results in percentage of errors, running sequential and parallel applications on dual-core ARM Cortex-A9.

		Errors [%]					**Speedup**
		Unace	SDC	Hang	Exceptions		
App.	Version				Seg. Fault	Unidentified	
Bit Count	Sequential	51.2	30.9	5.9	10.3	1.7	1.00
	OpenMP	47.8	28.6	3.7	19.7	0.2	1.96
	Pthreads	37.5	46.4	0.6	15.2	0.3	1.09
Matrix Mult.	Sequential	45.4	37.0	4.8	12.3	0.5	1.00
	OpenMP	33.4	42.6	4.7	19.1	0.2	2.02
	Pthreads	32.6	45.6	1.1	20.4	0.3	2.34
Vector Sum	Sequential	43.6	40.2	4.9	10.9	0.5	1.00
	OpenMP	29.7	47.4	4.7	18.0	0.3	1.81
	Pthreads	31.6	47.5	1.4	19.1	0.4	2.56

entirely done by the Linux OS. Sequential versions of the algorithms were also executed, but this time on the dual-core processor model. The results from this dual-core sequential execution are used as a reference to evaluate the impact of using the parallelization APIs on the system‘s fault tolerance. Table 5.2 presents the results of those simulations.

From Table 5.1, it is clear that the impact of Linux OS is different depending on the workload. Still, the usage of Linux OS on a single-core processor making no use of parallelization APIs seems to have little influence on the overall fault tolerance. Bare metal applications present no exceptions, because of the absence of an operating system to catch them. However, this information alone is not enough to say that they are less susceptible to errors, for exceptions presented at Linux may manifest themselves as other kinds of errors in bare metal applications. Supporting that theory, the percentage of hangs is lower on Linux. This is probably because the operating system is catching those errors and handling them as exceptions instead of letting them turn into hangs.

It is evident by the results of Table 5.2 that the usage of parallel code increases the occurrence of SDC type errors, being the Pthreads versions more susceptible than the OpenMP ones. Parallel applications are also more susceptible to Segmentation Fault errors. That increase in the percentage of errors in parallel applications justifies the development of special fault tolerance techniques to handle those different applications. The speedups presented were gathered from the real execution on the Zynq-7000 board (i.e., not simulation) and are calculated in relation to the sequential version of the given benchmark application. It is noticeable that the Pthreads applications achieved superlinear performance gains. Although the unaces decrease using Pthreads comparing to sequential, the speedup factor helps to significantly reduce the exposition time of the application under soft errors.

Comparing the executions on Linux and bare metal, the Matrix Multiplication running on Linux has fewer unaces than their bare metal counterparts, which means a higher number of errors. This is also true for the other two benchmarks. Other related works have seen similar behavior, such as (SANTINI et al., 2016).

The results from the execution on top of Linux OS using a single-core processor and on a dual-core processor for a sequential application present exciting data. The dual-core execution was less susceptible to errors than the single-core execution, except for the Vector Sum algorithm. It happens because faults are injected in both processors, so the probability for a fault to hit a vital instruction is higher on single-core execution when executing sequential algorithms. The comparison between the executions of the sequential version of the applications and the parallel ones in the dual-core processor, running on top of Linux OS, clearly shows that parallel executions are more susceptible to errors than their sequential counterparts. Some of the benchmarks presented almost the double of segmentation fault exceptions on the parallel version.

Results show that parallel applications are more susceptible to errors than their sequential counterparts, presenting higher numbers of SDC and Segmentation Fault errors (except for Bit Count OpenMP version). Remember that even when executing sequential algorithms, with Linux in dual-core processors, both cores may be at use (because of the OS scheduler). This is conflicting with some of the radiation results from the experiment from (TAMBARA et al., 2016), where using parallelism implied on a smaller cross-section when compared to the use of a single-core processor. Still, their work differs from ours in many ways, notably in the sense that our system under evaluation is more complex (with an OS and dual-core ARM processor).

After the results from Tables 5.1 and 5.2, it becomes clear that parallel applications executing on multicore architectures need to be protected at least as much as sequential ones. The fact that parallel applications behave differently from their sequential counterparts when exposed to faults raises the question of whether fault tolerance techniques classically used to protect sequential algorithms would have the same efficiency when applied to parallel applications. In an effort to answer that question, this chapter will evaluate two fault tolerance techniques: TMR and a variant that we called conditional double modular redundancy (CDMR), which is a DWC with re-computation if the values mismatch.

- **Triple Modular Redundancy (TMR):** The TMR technique applied to embedded software consists of running the part of the code to be protected three times. Each of those executions saves the computed data in different memory spaces. After the executions, the three values are compared and checked for inconsistencies. It is essentially a software adaptation of the TMR technique used for programmable logic devices. If no difference between the values is found, then no error is to be found by the technique. If one of the values differs from the other two, then it is considered erroneous, and it is masked (by considering the other values as the correct output and discarding the different one). TMR may cause a penalty of more than 300% in the processing of the portion of the code to be protected when applied in sequential applications running in single processors. However, this overhead may be less significant when using parallel applications in multicore processors.
- **Conditional Double Modular Redundancy (CDMR):** In an effort to lower the overhead caused by the TMR technique while maintaining a good error detection and correction, we propose a variant of the TMR technique. The CDMR technique executes the protected code twice. If the results of the two executions differ, then the protected code is executed one more. The result of this third execution is then considered a reliable output. This technique hopes to achieve efficiency close to TMR without having to execute three times the same code unnecessarily.

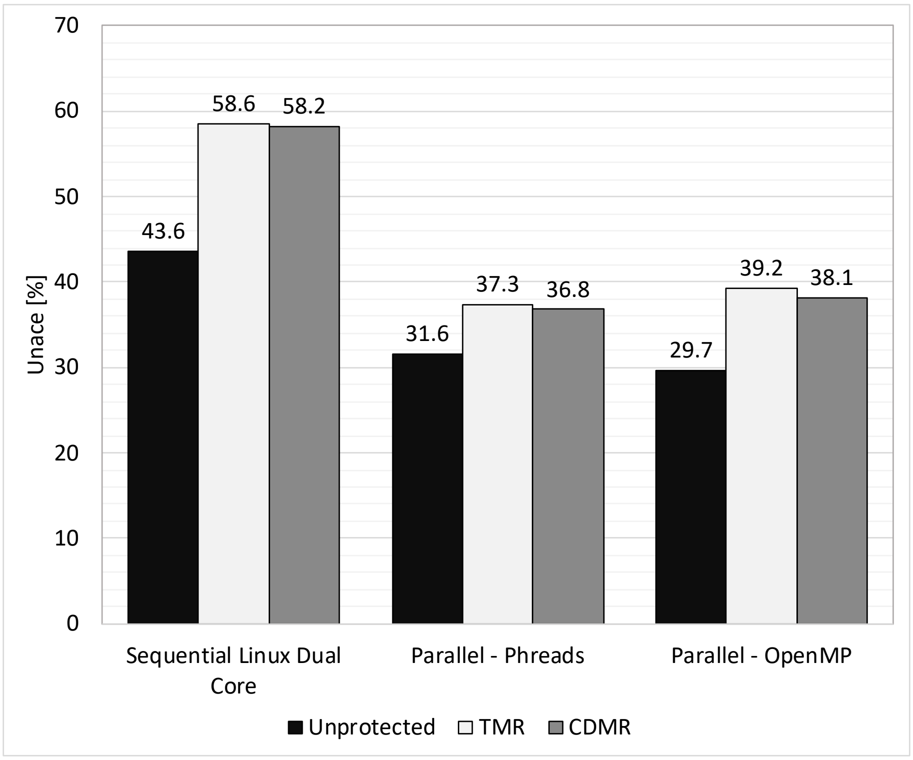
In this section, both TMR and CDMR implementations present time redundancy; that is, the fault tolerance technique itself is not parallel (Section 5.3 will present an study on parallel TMR). In the parallel versions of the benchmarks, the parallelism itself is applied to the application algorithm, not to the fault tolerance technique. The fault tolerance techniques presented were applied to the same previously evaluated benchmarks. They are tested using OVPsim-FIM to inject the faults and analyzing the errors for each execution. In the following graphs, the percentage of unaces for each application version is presented for each fault tolerance technique and for the unprotected version as a mean of comparison of the technique‘s efficiency. Remember that the unaces value represents faults that did not turn into errors, so high unaces numbers mean excellent protection.

Figure 5.2 presents the graphical results for the vector sum application under fault injection and with the TMR and CDMR techniques applied to it. It is easy to notice that the technique is more efficient for sequential applications. We also see that the fault tolerance techniques show limited improvement in fault mitigation, increasing up to 34.4% the unaces. Figure 5.3 presents the results for the matrix nultiplication application. In this case, CDMR was less efficient than TMR for the Pthreads execution. Once again, the fault tolerance techniques were not very effective, presenting a maximum increase of only 29.7% on the unaces occurrence. Figure 5.4 presents the results for the bit count application. Again, CDMR was less effective than TMR for the Pthreads version of the application but presented almost the same results for the OpenMP version. In that case, the fault tolerance techniques were even less effective. Figure 5.5 presents the data from the bare metal applications running with and without fault tolerance. It shows the fault tolerance techniques were more effective when executing on sequential bare metal than on Linux (both sequential and parallel). In the best-case scenario, the usage of the techniques has increased by 80.7% the unaces occurrence.

Notice that both TMR and CDMR techniques are not able to mask all the errors. That is expected when executing an operating system apart from the application, which may present its own errors. Our fault tolerance techniques do not protect Linux OS itself, only the applications. In addition, the studied fault tolerance techniques only protect the code from SDCs. That explains the coverage not being 100% on bare metal. An SDC is considered any mismatch in the final memory map and not only in the application output data memory. Therefore, any SDC error in the memory that does not necessarily affect the application output is considered an error too.

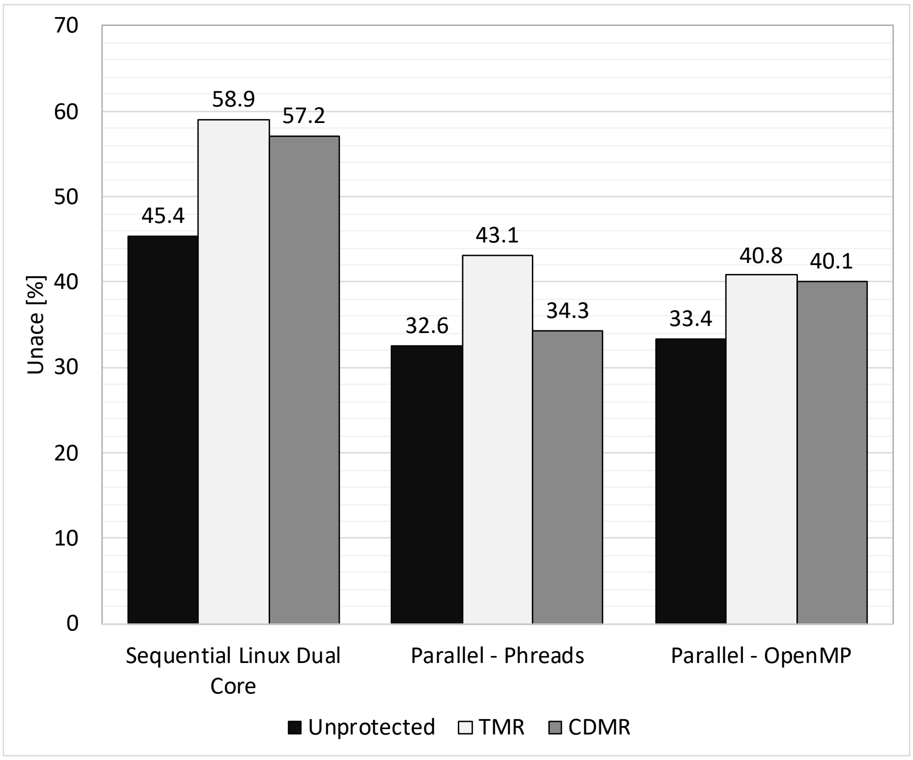
Table 5.3 presents the performance overhead by using TMR and CDMR. Each overhead was calculated with respect to the same algorithm‘s unprotected code execution time (e.g., an overhead of three means an execution time equals the original one multi-

Figure 5.2: Percentage of unaces on Vector Sum application running on top of Linux OS.



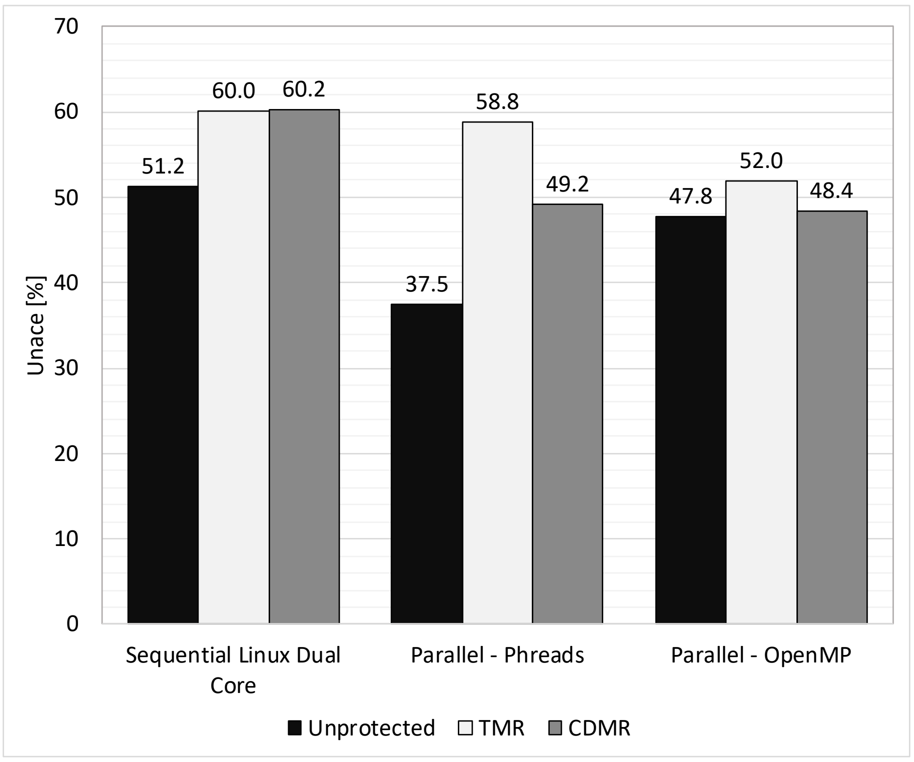
Source: Author.

Figure 5.3: Percentage of unaces on Matrix Multiplication application running on top of Linux OS.



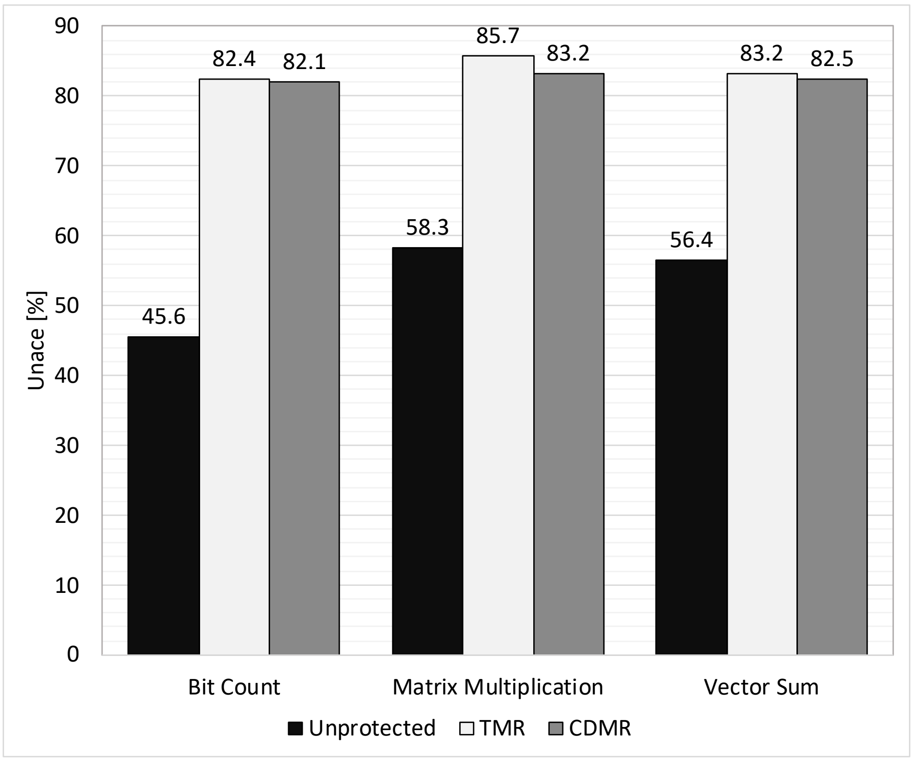
Source: Author.

Figure 5.4: Percentage of unaces on Bit Count application running on top of Linux OS.



Source: Author.

Figure 5.5: Percentage of unaces on all algorithms running bare metal.



Source: Author.

Table 5.3: Performance overhead of fault-tolerance techniques.

Application	Version	TMR	CDMR
Bit Count	Bare metal	3.05	2.01
	Linux Sequential	2.87	1.87
	Linux Parallel	2.87	1.93
Matrix Multiplication	Bare metal	3.00	2.04
	Linux Sequential	2.94	1.97
	Linux Parallel	3.00	1.99
Vector Sum	Bare metal	3.11	2.70
	Linux Sequential	2.95	1.98
	Linux Parallel	2.98	2.01

plied by three). CDMR presents a lower overhead than TMR, as expected. Note that the usage of an operating system does not impact the overhead, nor does the use of single or dual-core. Just like the speedups from Table 5.2, the performance overhead presented in Table IV were gathered from the real execution on the Zynq-7000 board, and not from a simulation.

5.3 Parallel Fault Tolerance

The results from the previous section put in evidence the need for fault tolerance techniques on parallel systems. This section raises the question of the impact of parallelism on fault tolerance techniques, and how it varies according to parallelism intensity. It evaluates different fault-tolerance approaches with various amounts of threads, stimulating the Linux threads scheduler to obtain data regarding its error susceptibility.

To evaluate the parallelization on fault tolerance techniques, four parallel TMR implementations of a matrix multiplication kernel are proposed: one based on sequential execution and three on parallel execution. The matrix multiplication divides the workload between the two processing cores creating threads during the execution time (using Pthreads). Resource management and thread scheduling are in the hands of the Linux OS scheduler. The total number of threads (i.e., including the matrix multiplication and TMR technique) and available processor cores are expected to influence the application fault tolerance. The approaches are presented below, in order of parallelism intensity:

- **Full Sequential TMR (FS-TMR):** This implementation is entirely sequential, and by consequence, the matrix multiplication executes three times voting the result at the end. Notice that the Pthreads API is not used in this implementation.

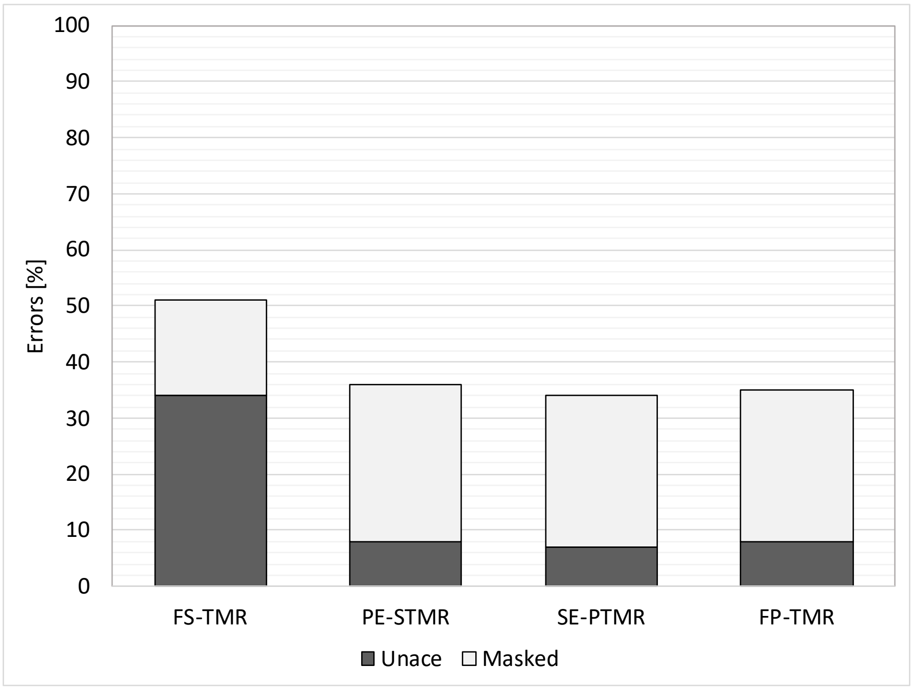
- **Parallel Execution, Sequential TMR (PE-STMR):** In the PE-STMR implementation, the algorithm to be protected is executed in parallel, but the TMR technique is coded sequentially. That means the TMR sequentially executes three times a parallel matrix multiplication algorithm, and then the results are voted.
- **Sequential Execution, Parallel TMR (SE-PTMR):** For this implementation, the TMR technique is parallelized, but the matrix multiplication algorithm is sequential. Therefore, the processor will launch three threads in parallel (TMR redundancies), and a sequential code will be executed on each of these threads.
- **Full Parallel TMR (FP-TMR):** Finally, both the matrix multiplication application and the TMR technique are implemented as parallel code. Three threads are launched in parallel (TMR redundancies), and each of them executes parallel code.

The PE-STMR adopts a sequential TMR approach with a parallel application. Thus, the primary application flow creates the two child threads, waiting for its completion to launch the next two threads. In contrast, the SE-PTMR implements a parallel TMR into a sequential application (i.e., the matrix multiplication is calculated in a single thread). The FP-TMR deploys a total of nine threads (one per redundancy execution plus two for each one of them to parallel the matrix multiplication). Finally, the sequential version (FS-TMR) employs no parallelism, creating no new threads.

Like in the last section, the testing experiments performed on the techniques was the fault injection simulation with the OVPSim presented at Section 4.3, with the same types of errors defined at Table 4.1. However, a new classification is added: the “Masked” data presented at the following figures represent the percentage of errors that were present in the system but were masked by the TMR techniques (they would become SDCs if no TMR was implemented).

Figure 5.6 presents the fault masking performances for the executions under fault injection. It is noticeable that about 35% of the FS-TMR occurrences are unaces, that is, no error at all. Also, it was capable of masking about 50% of the SDCs. Results show that PE-STMR achieved much more fault tolerance. Only about a fourth of the SDCs were not corrected by it. SE-PTMR result data is similar to the results from PE-STMR. The fault tolerance is lower in this implementation, however. FP-TMR results are also very similar to PE-STMR. Comparing the FP-TMR and the SE-PTMR with the PE-STMR approach, the last one achieves better fault tolerance. It indicates that parallelizing the TMR

Figure 5.6: Comparison between all the TMR fault masking performances.



Source: Author.

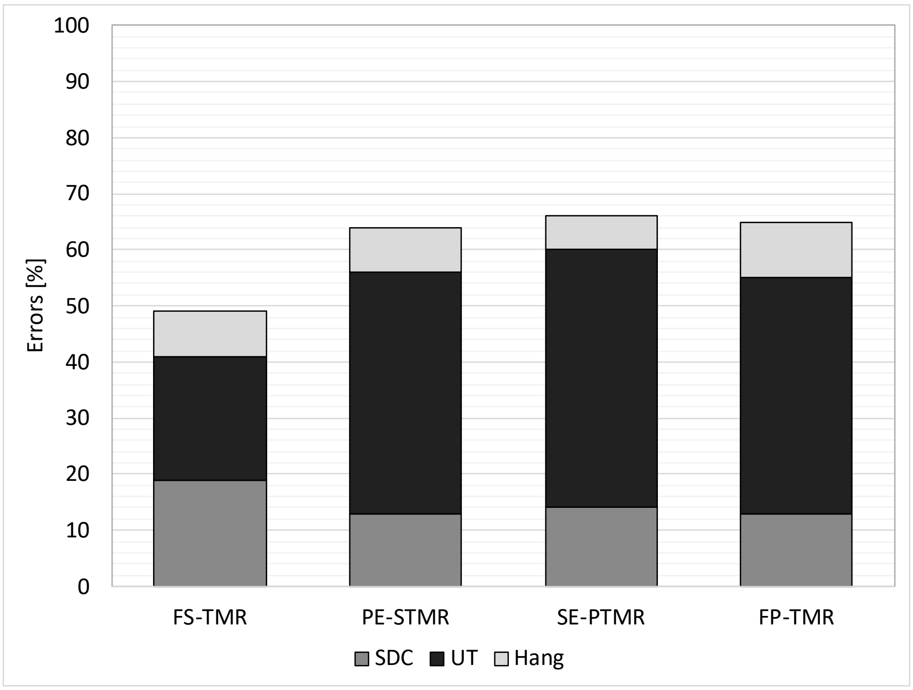
technique can have a negative impact on dual-core processors, augmenting the number of errors. The fact that FP-TMR is the one with the less occurrence of unaces and corrected errors shows that the Pthreads API is terrible for fault tolerance. Nevertheless, the parallel TMR benchmarks showed better performance, indicating that the independence between threads confines the error in a single Pthread context (which is only partially shared).

Figure 5.7 presents a comparison between all the benchmarks error occurrences. An interesting fact is that the full sequential code (FS-TMR) shows about half the percentage of unexpected terminations when compared to the parallel applications. Also, no matter the number of threads, all the parallel applications are equally susceptible to this type of error. PE-STMR is the technique less vulnerable to SDCs, while SE-PTMR is the most susceptible among the parallel ones. This is another indication that parallelizing TMR has a severe impact on fault tolerance. On the other hand, parallelizing the application reduces the susceptibility to SDCs.

5.4 Approximating Parallel Fault Tolerance

So far, TMR has been used as a means to memory fault-masking, protecting the system against possible SDCs. When implementing TMR with Pthreads on a modern operating system running on a multicore architecture, it can be used to tolerate other types

Figure 5.7: Comparison between all the implementations error occurrence.



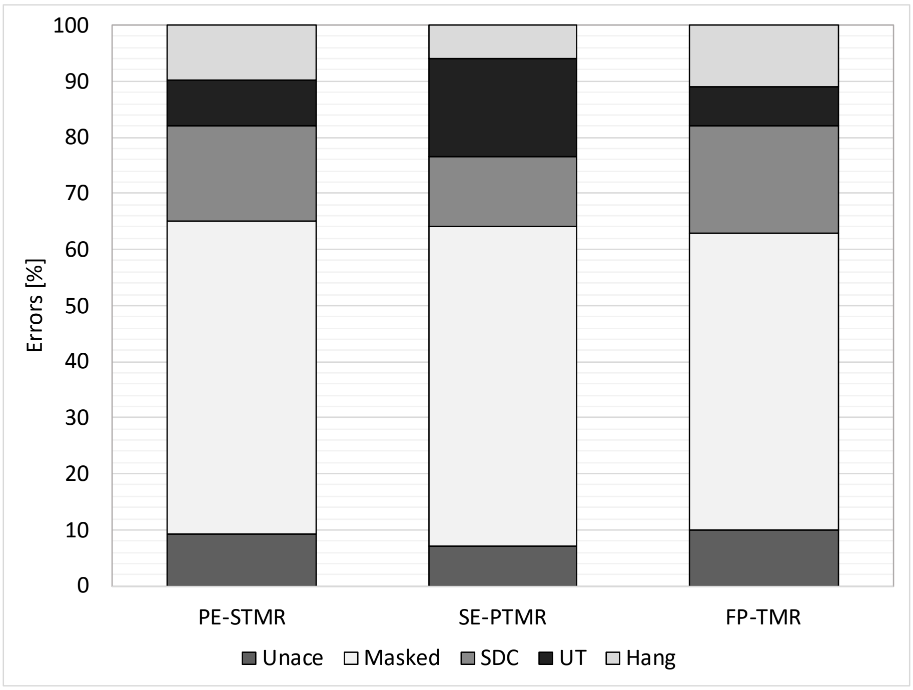
Source: Author.

of errors. Figure 5.7 shows that multicore applications running on Linux are especially subject to unexpected terminations. Those are errors on which the application is unexpectedly terminated by the OS due to an internal problem, such as invalid memory accesses or segmentation faults. Errors as such are also present on bare metal applications but are not manifested in the same way due to the absence of an operating system that could catch them.

As expected, the parallel versions of TMR implied a higher occurrence of UTs (Figure 5.7). In an ordinary Linux system application, the entire application is terminated in the case of a UT on any of its threads. The problem with this approach is that it kills processes that are probably healthy and capable of providing correct data.

To deal with that problem, this section proposes using an approximation strategy based on function skipping to improve the parallel TMR proposals. This approximation strategy provides the possibility to tolerate UTs. Since every redundant TMR thread is executing the same computation, the application can finish execution even if one or two of them terminates unexpectedly. A segmentation fault handler is implemented into the TMR code to catch exceptions that would provoke unexpected terminations and, instead of terminate the application, kill only the faulty thread. We call this method "thread disposability". The implementation of thread disposability on the full sequential benchmark is not possible due to its absence of parallel threads (apart from the main one).

Figure 5.8: Parallel TMR implementations with thread disposability.



Source: Author.

Figure 5.8 presents the results for the parallel TMR implementations with thread disposability. It is noticeable that the occurrence of UTs drops dramatically. In contrast, the presence of other errors remains almost unchanged. It indicates that the thread disposability technique has little influence on the normal executing of ordinary TMR. As expected, the simulation results show that different parallelization approaches imply various susceptibilities to errors. The usage of parallelism highly increases the occurrence of unexpected terminations, that is, errors that affect the normal behavior of the application and are caught by the OS before causing severe problems. Because of that, the use of full sequential TMR appeared to be the best choice for software protection among ordinary alternatives. Using parallel TMR techniques without thread disposability causes more errors than it is capable of masking, making it unsuitable.

Thread disposability proves to be a useful method to deal with unexpected termination errors. Due to its non-intrusive nature, it costs virtually no extra execution overhead. With the use of this approach, the parallel implementations of TMR achieved similar or better results than their sequential counterpart. Thread disposability makes it possible for a full parallel TMR to have better fault tolerance than a full sequential one. It makes the use of parallel TMR techniques viable, achieving both good fault-masking and overhead reduction.

This is an excellent example of how approximate computing can be applied to fault tolerance algorithms to improve them. Thread disposability not only improves a traditional fault tolerance technique by expanding its fault coverage capability but also improves its execution performance by freeing computational resources that were being wasted on useless computation. It is very hard, though, to evaluate its impact on the execution time performance, once the occurrence of errors is not predictable. However, the good results from this first experience of approximate computing on parallel embedded software is a motivation for proposed works presented at Chapter 6. Apart from presenting approximation methods and studying their reliability, Chapter 6 also proposes the usage of approximate computing as a means to improve fault tolerance.

5.5 Approximate Fault Tolerance: Discussion and Motivations

Approximation itself implies the idea of inherent error tolerance. On approximate systems, a specified error tolerance has to be considered, but that is not the same error definition used when discussing radiation effects and safety-critical systems. Approximation errors are caused by the system itself and manifested as quality or accuracy degradation. Also, when dealing with approximation, the decision of whether an error caused a failure or not is a matter of definition related to what would be considered a "correct" application output, which is often hard to be defined. Taking, for instance, the example of image outputs, the correctness of the output is tied to an image quality definition, which is different from one human being to another because of biological reasons. This accuracy relaxation from the approximate system can, however, be used in favor of fault tolerance on safety-critical systems: a system that accepts some accuracy degradation can ignore errors in memory that have a low impact on the data value, for example. Also, the reduction of the complexity, achieved by approximation, can help to reduce the system's susceptibility to faults (e.g., by reducing the critical area of a hardware circuit).

On safety-critical systems, however, the definition of error is related to the occurrence of a fault. In this scope, the approximation can be used in two manners. First, it can be used to improve the application execution time, energy consumption, and even reliability. Secondly, approximate computing can also be used to reduce the costs of fault tolerance techniques. The impact of using approximation on those two levels, however, is different. As already discussed, the approximation of the application directly impacts its accuracy, and therefore reliability. Approximating fault tolerance techniques may, however, be developed in such a way to avoid affecting the accuracy of the application, or affecting it only up to the acceptable level that is defined by its quality (or accuracy) requirements.

TMR is one of the most traditional fault tolerance techniques presented by the literature. Approximate TMR (ATMR) (GOMES et al., 2014) is based on implementing each redundancy task with a different architecture or algorithm to provide the capability of masking multiple errors. When applied to hardware projects, ATMR has been presented as a way to achieve fault coverage almost as good as traditional TMR but avoiding the huge area overhead that it costs (ARIFEEN et al., 2016). Designers might accept a lower fault coverage if the area overhead of the project is to drop significantly. Also, a smaller hardware area implies higher fault tolerance due to the reduction of the critical area. Therefore ATMR might be, in some cases, not only less costly but also more reliable than traditional TMR. In traditional TMR, at least two redundancies need to have the same correct value at a given time so that the correct output can be voted. Using approximations on TMR is not trivial, because of the errors caused by the accuracy loss: even in the absence of a fault, two TMR redundancies of different accuracies will present different outputs. At (GOMES et al., 2015), the authors present an ATMR approach that guarantees that the result of at least two redundancy circuits will always be the same (at the absence of a fault). The idea is using different forms of approximation on each redundancy so that two of them will not be affected by approximation errors at the same time, and the ATMR will be able to mask that error. The authors present their approximation method and prove mathematically that the errors introduced by the approximation will not harm the normal behavior of the ATMR. They also propose a full ATMR (FATMR) approach where all the three circuits are approximations (instead of having one non-approximate circuit and two approximations). This ATMR technique can also be used alongside tools that generate the best possible approximate functions with genetic algorithms (ALBANDES et al., 2018). The evolutionary algorithm is capable of generating the best combination of approximate functions possible for a given system. However, the ATMR and FATMR methodologies are still limited by their mathematical and theoretical constraints.

Most of the approximation techniques presented in the literature are application-specific. Therefore, it is very hard or impossible to apply the same approximation technique to any possible design or code. Knowing all the possible approximation methods and which type of design is a better fit for each of them is barely impossible work. Also, some approximation methods are applicable to multiple types of applications and hardware designs. Therefore the designer should test all of them before deciding for the one with better performance. All that would demand design time that most developers can not afford. This work tries to solve those issues by presenting easy-to-implement approximation methods that can be applied both to programmable hardware and embedded software. Approximate fault tolerance techniques are also proposed by applied those methods to traditional TMR.

6 PROPOSED WORK

This chapter is divided into two sections. Section 6.1 proposes three approximate computing techniques intended for general use. This section presents those techniques and a brief evaluation of their applicability and cost. All those approximation strategies can be applied both to programmable hardware and embedded software. Section 6.2 uses two of the proposed approximate computing techniques to develop approximate versions of a traditional fault tolerance technique: the triple modular redundancy. Section 6.3 proposes an approximate error detection technique developed for multicore real-time systems.

6.1 Approximation Methods

Chapter 2 presented a multitude of approximation methods. This work will cover in practice three of them: data precision reduction, functional approximation, and loop-perforation. Section 6.1.1 starts by evaluating data precision reduction implemented in hardware and how it can be used to reduce the used area. Section 6.1.2 presents a combination of functional approximation and loop-perforation applied. Finally, Section 6.1.3 proposed a numerical-specific approximation that motivates the universal use of approximate computing.

6.1.1 Data Precision Reduction

This section presents a practical analysis on the usage of data precision reduction approximation on programmable hardware. This is done by creating new data types with reduced bit-size. Representing values with a limited number of bits saves hardware resources in detriment of data precision. As will be detailed further, high-level synthesis (HLS) is used to generate hardware from software code; thus, the proposed approximation method is assured to be applicable both to software and hardware projects, although the impact on each would be different.

Variables on software are usually defined by standard types. Those types define how the variables' read, write and arithmetic and boolean operations are executed on the hardware, as well as their bit-size. All those characteristics have a significant impact on the system performance, energy consumption, and also fault tolerance (BARROIS; SENTIEYS; MENARD, 2017). The same is valid for variables on projects that make use of hardware description languages (HDL), specifically when those are generated by HLS.

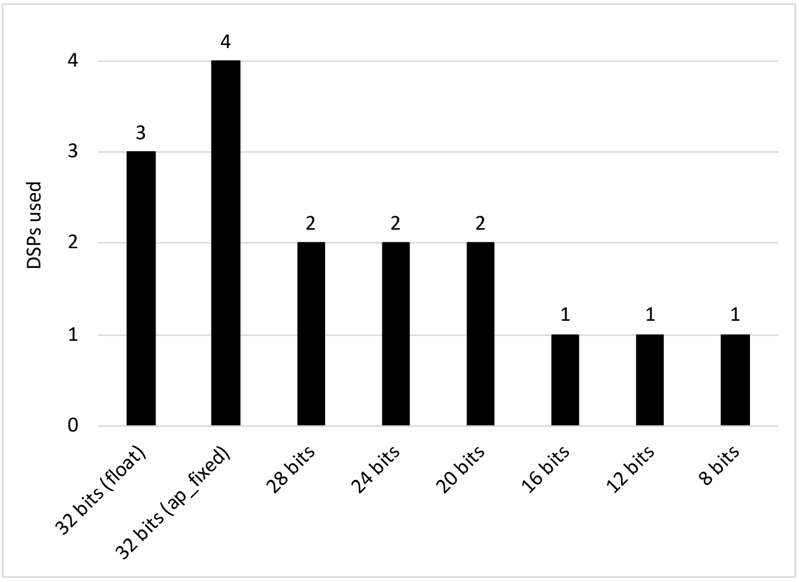
The benchmark designs analyzed in this section are coded in C language. The hardware implementation is then generated by Vivado HLS. Vivado HLS is a tool provided by Vivado Design Suite (Xilinx). It produces hardware description using a C or C++ language code as input. Vivado HLS also counts with a series of hardware optimizations and generation tools, providing considerable control over the final product. The *ap_fixed.h* library provided by Vivado HLS allows arbitrary data type creation, for fixed-point variables, and is used in this work to create approximate data types. In the regular floating-point representation (IEEE-754 standard), 28% of the 32 bits are designated to represent the exponent while the other 72% represents the significand. The approximate fixed-point data types evaluated in this section keep that same share for the representation of the numbers above the decimal point and the value below the decimal point.

Figure 6.1 presents the used area - concerning DSPs - for eight floating-point data type sizes applied to both operands of a simple multiplication between two variables implemented in the FPGA of a Zynq-7000 APSoC by Xilinx (further implementation details are given at Chapter 7). The figure shows that the multiplication between two traditional IEEE-754 standard *float* variables costs one less DSP than using 32-bit variables implemented by the *ap_fixed.h* library. Nevertheless, variables generated by *ap_fixed.h* with less than 32 bits consume less DSPs. This alone is an indication of its capability of saving resources that can then be used to improve performance. In this particular case, the saved DSPs could then be used to improve performance, executing other operations in parallel. Figure 6.1 shows that it is possible to implement two 28-bit multiplications in parallel using the same number of DSPs as one single 32-bit operation. Data precision reduction implemented on embedded software will also be studied in this work. As Section 6.2.2 will discuss, the effects on software are different. While it affects the are the usage of programmable hardware, on embedded software it will have a direct influence on the memory footprint of the application.

6.1.2 Successive Approximation

This proposal consists of a type of functional approximation combined with loop-perforation. This method is an excellent example of how two theoretical ideas for approx-

Figure 6.1: DSP usage for multiplication using different data type configurations to represent floating point values.



Source: Author.

imation can be combined. It consists of using an inherent characteristic of the algorithm, namely its loop-based execution, in favor of approximation. As will be discussed, this study case also shows how some algorithms are inherently approximate, and that some solutions (such as numerical calculations) can only be solved approximately.

Successive approximation algorithms are numerical calculations for which an exact, straightforward solution is not computationally achievable. Such is the case of the calculation of the integral of a function. Those algorithms are iteration-based and get closer to an acceptable result on every iteration. An example of this kind of numerical algorithm is the trapezoidal rule, which is used to calculate the area under a curve approximately (i.e., the integral of $f(x)$) as the sum of trapezoid areas, as defined in (6.1).

$$\int_a^b f(x)dx \approx (b-a)\left(\frac{f(a)+f(b)}{2}\right) \tag{6.1}$$

One can take (6.1) and make a more accurate approximation by breaking up the interval between the points a and b into a number n of smaller intervals. Then, the algorithm consists of computing the approximation for each of those intervals and adding the results afterwards, achieving a better result than the one provided by (6.1) alone. The bigger the size of n, the better the result will approximate the real integral solution. This is called an iterated rule. The iterated calculation for the trapezoidal rule is presented in (6.2). In this case the intervals present the form $[kh,(k+1)h]$, where $h=(b-a)/n$ and

k ranges from 0 to $n-1$.

$$\int_a^b f(x)dx \approx \frac{(b-a)}{n}\left(\frac{f(a)}{2}+\sum_{k=1}^{n-1}\left(f\left(a+k\frac{b-a}{n}\right)\right)+\frac{f(b)}{2}\right) \tag{6.2}$$

Because the value is approximated in each iteration, it is expected that if an error occurs, causing an iteration result that is out of the expected calculation path, it will be corrected in the following iterations, as the algorithm will then get the calculation back on track. This statement is valid when analyzing soft errors, not permanent errors. In the case of permanent errors that compromise the hardware and the right execution of the algorithm, the executions of all future iterations would be compromised, therefore compromising the convergence of the algorithm. Moreover, if an SEU occurs in one of the last iterations (or the very last one) it may be too late for the algorithm to converge back to an acceptable result. Therefore, it is expected that the higher the value of n, less susceptible to errors the algorithm will be. Conversely, having a higher number of n would also increase the execution time of the computation. Past works show that applications executing in a radioactive environment with a higher execution time are more prone to have errors, as they would be exposed to more radiation (REIS et al., 2005) and TID (QUINN, 2014).

Increasing the number of n would, therefore, have a positive impact on the algorithms protection against faults and exactitude (more iterations imply on a better approximation), but a negative impact on performance. This negative impact on performance could then impact the system reliability, even to the point of repealing the positive impact from the successive approximation. Those factors shall be intensely studied by a developer that intends to use numerical methods on safety-critical systems. As stated before, some solutions are only achievable by numerical analysis, so this study is imperative for any complex critical-system development.

Because of their nature, successive approximation algorithms and numerical methods, in general, have their own sources of errors (manifested as inaccuracy):

- **Simplification Errors:** because every numerical method is actually an approximate model of the reality, they can only relate to the mathematical reality to a certain extent.
- **Truncation Errors:** given that the accuracy of floating-point values is limited, exactitude may be lost.

- **Accumulation Errors:** in some numerical algorithms errors may propagate, so that the final result will be less exact than middle-term results. The types of errors above may also contribute to the accumulation of errors on each iteration.

The first two types of errors are also present on ordinary computation methods, but the last one is natural for iterative algorithms, like numerical methods. It is important to keep in mind, however, that this kind of error is different from the one caused by SEUs. The former is a characteristic of the programming paradigm, while the latter is caused by a harmful environment. Programming paradigms are known to affect the fault tolerance of a system even when using numerical methods are not being used.

Another critical factor to take into account when using this type of computational method is convergence. It is possible for some numerical methods to converge faster to the result with the required exactitude, thus requiring a lower number of n iterations. It is also possible for some methods to require a higher number of n. The real problem, however, is when the method diverges. Some methods can even converge for a specific interval and diverge for another. The convergence problem is out of the scope of this work, but it is important to notice that as an issue. There are, fortunately, numerical methods that are guaranteed to always converge to a correct, acceptable result.

6.1.3 Taylor Series Approximation

Given the extensive amount of possible approximate computing methods presented by the literature, finding the most suitable strategy for a given application is a significant challenge. Most approximation methods presented in the literature are specially developed for a single application, being unscalable, and many times even inapplicable for a different purpose or algorithm. Theoretically speaking, one can say that an infinity of applications has still not been approached by approximate computing studies, and never will. For some applications, developing a unique approximate design might be an extreme intellectual work.

One of the proposals of this work is to use Taylor series to numerically approximate functions. Although many numerical techniques for mathematical approximation are available, Taylor series was selected due to its high applicability and simplicity (REN; ZHANG; QIAO, 1999). Another reason to chose it was the fact that its terms can be previously calculated, implying in a performance gain when implemented in software. In particular, Taylor series was chosen instead of Maclaurin series due to its higher usability and generality (FOY, 1976; MOLLER et al., 1997). Maclaurin series is a specific case of Taylor series and has smaller applicability due to its constraints.

Because mathematical functions can represent any algorithm, this approximation approach can be used to generate an approximate version of almost any given software (given some mathematical limitations imposed by the technique, to be further defined). The same strategy can also be used to provide approximate hardware. Both software and hardware implementations are better discussed in Chapter 7. The main contribution of this part of the work is to provide a theoretical basis for the generation of approximate numerical versions of any software or hardware design. This proposal relies on approximation theory and mathematical analysis to provide mathematically valid approximations.

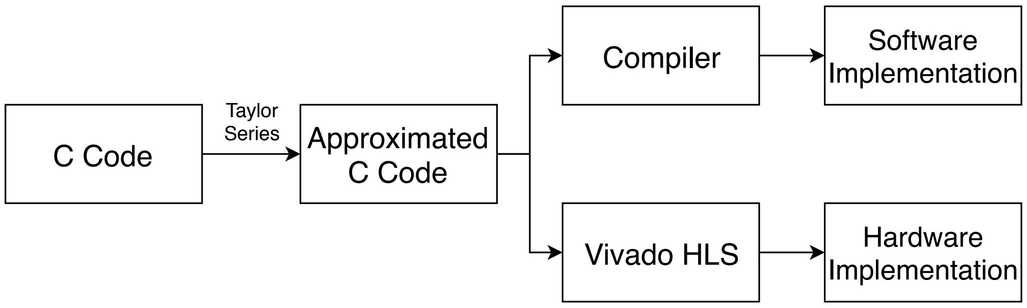
Taylor series are used in mathematics to represent a function as a sum of previously calculated terms. These terms are generated from the values of the function's derivatives at a given point. The more terms are used, the better the representation. This way, functions are approximated using a finite number of terms in a Taylor series. An infinite number of terms would adequately represent the original function. However, calculating infinite terms is computationally impossible. The compact sigma notation for Taylor series is presented in (6.3), where n stands for the current term (from 0 to N), and a stands for the center point (where the derivatives are calculated), being $f^{(n)}(a)$ the nth derivative of the function f at the point a. When $a = 0$, the Taylor series is called a Maclaurin series.

$$\sum_{n=0}^{N} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{6.3}$$

Being an approximation, a Taylor series with finite terms presents a quality degradation when compared with the original function. This degradation is variable depending on the center point used and the number of terms. For some functions, the Taylor series may converge in a given range and diverge when out of its bounds. However, it is possible to estimate this quality loss quantitatively using Taylor's theorem. Functions that contain one or more singularities cannot be represented as Taylor series either. The convergence of a given function using Taylor series approximation needs to be evaluated before its usage. The designer is also accountable for employing a sufficient number of Taylor terms so that the quality loss does not interfere with the generation of a *good enough* result.

Some of the approximation problems presented by Taylor series can be dealt with

Figure 6.2: Taylor series approximations implementation flow.



Source: Author.

by the designer during implementation time. For instance, if the range of the input values of a function is known, the designer can evaluate if an approximation by Taylor series is feasible. Also, even when such an approximation shows up as divergent in a critical range, the designer can achieve a good approximation by merely changing the Taylor series center point a (see Equation 6.3).

As previously discussed at Section 6.1.1, approximate computing can be used to reduce the size of programmable hardware implementations and improve the execution time of software. Both hardware size and execution time are highly related to reliability, therefore using Taylor series to approximate any given function is expected to improve their fault tolerance as well. Thus varying the number of Taylor series terms will impact not only resource usage but also tolerance. For those reasons, this study is fundamental for safety-critical system designers willing to implement numerical algorithms.

An approximation for the exponential function is implemented as benchmark to test the proposed Taylor series approximation. The algorithm is coded in C and implemented both in software and hardware. Hardware implementations are generated by Vivado HLS using the very same code and implemented in the FPGA of a Zynq-7000 APSoC. The software versions are executed in the embedded ARM A9 processor embedded on the same board. Figure 6.2 presents the implementation diagram of the approximations. The mathematical equation for this Taylor series approximation is presented in (6.4).

$$e^x = \sum_{n=0}^{N} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^N}{N!} \tag{6.4}$$

Hardware Implementation: Two parameters are considered during the hardware implementation: data precision type and the usage of a pipeline. The data precision types evaluated are *double* and *float* (i.e., double-precision and single-precision floating-point formats, with 64 and 32 bits, respectively). Pipeline is used to accelerate the Taylor series computation loop and is implemented into the algorithm by merely adding a *pragma* (option from Vivado HLS) in the C code. No particular argument is passed to the HLS pipeline pragma, so by default, it will try to pipeline the loop as much as possible. Each possible combination of those two parameters is tested, resulting in four implementations (i.e., *double* and *float* variables, with and without pipeline).

Table 6.1 presents the resource usage for the four implemented Taylor series approximations. It shows the data from the four variants of the approximation for a variety of numbers of Taylor series terms. The double-precision variant with pipeline presents data from 3 to 13 terms because it is the maximum number of terms implementation that fits in the Zynq-7000 FPGA. For the same reason, the float precision variant with the pipeline is presented from 3 to 34 terms. Resources usage concerning area is divided into four categories: DSPs, FFs, LUTs, and Essential Bits. The last two columns present the data for latency (in clock cycles) and accuracy (in percentage). The accuracy is calculated comparing the output value with the best value obtainable computationally without Taylor series approximation (the exponential function from the *math.h* C library for the given data precision). Vivado HLS implementation reports provided the estimation of the hardware latency and area resource usage presented at the table.

It becomes clear by the analysis of Table 6.1 that the usage of pipeline profoundly affects area resources occupation, while the latency slightly increases with the number of terms. On the other hand, the absence of pipeline implies an almost constant hardware area but provokes an enormous latency increase with a higher number of terms. The essential bits are configuration bits that are really used by the design, on which a bit-flips will possibly cause a errors. This data is important for safety-critical systems design, where it shall be as low as possible. The table also shows that double-precision (*double*) achieves accuracy per number of terms almost at the same rate as single-precision (*float*). Nevertheless, only double-precision was capable of achieving full accuracy. Another interesting fact is that not many terms are needed to provide good accuracy. In fact, 8 terms seem to be enough to provide an accuracy of 99% for any implementation. From that point further, the area and latency costs increase, but the accuracy remains almost the same.

Software Implementation: The same code used at Vivado from the last section was also implemented on Vivado SDK and executed on the ARM processor. The software is bare-metal implemented. However, in this case, only two versions of the algorithm are presented: one for double-precision and another for single-precision. That is because

Table 6.1: Performance and resource usage analysis from HLS hardware implementation of Taylor series approximation.

Precision	Pipe.	Terms	Area				Lat.	Accuracy
			DSPs	FFs	LUTs	Esst. Bits	[c.c.]	[%]
Double	No	5	14	4761	6250	804046	215	89.28417678
		10	14	4765	6254	824732	430	99.97332604
		25	14	4769	6258	810620	1075	100
		50	14	4773	6263	825134	2150	100
		100	14	4777	6267	806337	4300	100
	Yes	3	28	1797	3518	449824	16	54.70235457
		4	42	5987	8967	1126115	41	76.02448002
		5	67	7264	11400	1409144	47	89.28417678
		8	109	19777	27747	3502632	65	99.58761712
		11	162	29396	41078	5183543	83	99.99410196
		13	190	37738	51976	6618716	95	99.99977405
Float	No	5	5	1648	2361	251843	130	89.28417875
		10	5	1652	2365	255535	260	99.97332926
		25	5	1656	2369	250510	650	99.99998919
		50	5	1660	2374	258650	1300	99.99998919
		100	5	1664	2378	256350	2600	99.99998919
	Yes	3	10	837	1454	168274	12	54.70235538
		4	15	2054	3191	342306	24	76.02448475
		5	23	2646	4255	445723	29	89.28417875
		8	38	6276	9498	991407	44	99.58761582
		16	81	15338	22753	2377350	84	99.99998919
		34	177	35882	52737	5494006	174	99.99998919

Table 6.2: Performance analysis from embedded ARM software implementation of Taylor series Approximation.

Terms	Double		Float	
	Exec. Lat.[c.c.]	Accuracy[%]	Exec. Lat.[c.c.]	Accuracy[%]
2	230	28.9872284	238	28.98722979
3	202	54.70235457	164	54.70235531
4	252	76.02448002	204	76.02448465
6	348	95.88087592	284	95.88087586
8	444	99.58761712	364	99.58761569
10	540	99.88980474	444	99.97332912
13	684	99.99977405	576	99.99978126
15	780	99.99999351	644	99.99999681
16	828	99.99999986	696	99.99998933
20	1020	100	844	99.99998933

there is no implementation strategy on embedded software equivalent to the Vivado HLS pipeline.

Table 6.2 presents the data from the embedded software execution performance. The two columns present the data for execution latency (in clock cycles) and accuracy (in percentage). The accuracy is calculated comparing the output value with the best value obtainable for the given data precision (using the function from the *math.h* library). The execution latency of the embedded software is measured by executing the applications on the ARM Cortex-A9 processor embedded Zynq board making use of the *xtime_l.h* C library provided by Xilinx.

As expected, both the execution latency and accuracy increased with the number of Taylor series terms. Table 6.2 shows that the accuracy increases exponentially with the increase of the number of terms. The latency also increases with the number of Taylor series terms, but not as much. Surprisingly, single-precision appears as a better choice when using 10 to 15 terms, providing both better accuracy and execution latency.

6.2 Approximate Triple Modular Redundancy (ATMR)

Given the proposed approximate computing methods, we believe that some of them can be used to improve traditional fault tolerance methods. The most classical fault tolerance method presented in the literature is TMR, as already discussed. Therefore, to evaluate how approximate computing can improve fault tolerance methods, two approximate TMR (ATMR) techniques are proposed, based on two of the approximation techniques presented in this chapter.

Fault tolerance techniques often introduce a high execution time or hardware area overhead. Such is the case of TMR, which costs an overhead of at least 200%. This work proposes an ATMR method to deal with that issue without highly compromising fault tolerance. Differently from (GOMES et al., 2015), (ALBANDES et al., 2018) and (ARIFEEN et al., 2016), the ATMR techniques proposed in this work do not work with approximations limited by a mathematical statement. The ATMRs presented in this work deal with the concept of *approximation intensity*, where a function can be more (or less) accurate, having a direct impact on the method fault coverage, the final answer accuracy and the application execution time.

6.2.1 Hardware ATMR based on Data Precision Approximation

The proposed ATMR benefits from the data precision approximation to generate redundancies that are less accurate than the classical ones, but smaller in area. This ATMR is expected to achieve fault tolerance near to the traditional ones, but with less area overhead. The ATMR is applied to simple codes (two matrix multiplication algorithms). This is intended to evaluate how the studied type of approximation affects data operations its effects on hardware. Using a sophisticated code could mask that information. The fault tolerance of the proposed technique is assessed with fault injection on the FPGA configuration memory.

Results from Figure 6.1 prove that the proposed data precision reduction approximation saves resources. This indicates that the proposed approximation can be used to provide an ATMR design with a lower area overhead. If that turns to be true, it may even be possible to improve general use designs, achieving better performance and resources usage, as well as fault tolerance (given the lower hardware area).

Listing 6.1 presents a pseudo-C code that summarizes the ATMR implementation. Vivado HLS is used to implement hardware-based on them, as explained in Section 6.1.1. Some less important parts of the code are left out for simplification purposes. The ATMR is implemented as three operations, in different functions at the C code, so that Vivado HLS is forced to implement specific hardware for each one of them. Otherwise, it could re-use hardware, which is not desired for the TMR implementation. The voter is implemented as a single independent function and consists of boolean operations that perform a bitwise check between three values.

Listing 6.1: Simplified pseudo-C Code for a an ATMR implementation using 24-bit variables for Vivado HLS.

```
#include <ap_fixed.h>
typedef ap_fixed<32,9> tsize_32;
typedef ap_fixed<24,7> tsize_24;

void main(tsize_32 input_A[2][2], tsize_32 input_B[2][2],
tsize_32 output[2][2])
{
    tsize_24 result1[2][2], result2[2][2], result3[2][2];
    result1 = matrixMult1(matrixA, matrixB);
    result2 = matrixMult2(matrixA, matrixB);
    result3 = matrixMult3(matrixA, matrixB);
    output = bitwiseVoter(result1, result2, result3);
}
```

Between the matrix multipliers and the voter, converters may or may not be needed: depending on the sizes of the data in use. That is because the voter cannot vote values of different bit-sizes. Converters may also be needed inside the matrix multipliers functions implementation, in case that the input matrices are of different sizes from the ones used in the ATMR redundancies. At the Listing 6.1 code, for example, the ATMR uses 24-bit variables. Therefore, additional hardware will be implemented by Vivado HLS to handle the conversion from 32-bit (size of the inputs) to 24-bit variables. Each of the ATMR redundancies can be implemented with different data sizes. The data bit-sizes will affect the final result accuracy and hardware usage. Typically, if a specific data bit-size is applied to two redundancies, it will define the overall accuracy (because of the bitwise voter). However, a designer may choose different approaches to profit from the hardware cost improvement without losing precision (e.g., comparing the values considering an acceptable difference threshold and taking the output from the best accuracy redundancy as the final result). The conversions between different data sizes and types are handled by Vivado HLS. A simple cast from a different data size in the C code is enough. A more complex and probably less costly conversion could be designed, but this type of improvement is not studied. This is also the case of the ATMR voter implementation: it is left for Vivado HLS to transform the code in hardware implementation, and possible improvements are not in the scope of this work.

Six ATMR designs were implemented, varying the data precision of the operations. A non-approximate TMR version is also presented (with the three modules using

32-bit data). The designs are named following the data precision of each redundancy module to simplify the analysis and data presentation. For example, the ATMR design called "32-24-16" is composed of a module with 32-bit, one with a 24-bit and another with 16-bit precision data and operations. Those ATMR designs were applied to two matrix multiplication algorithms, one with matrices of size 3×3 and other of size 2×2.

6.2.1.1 Accuracy Assessment

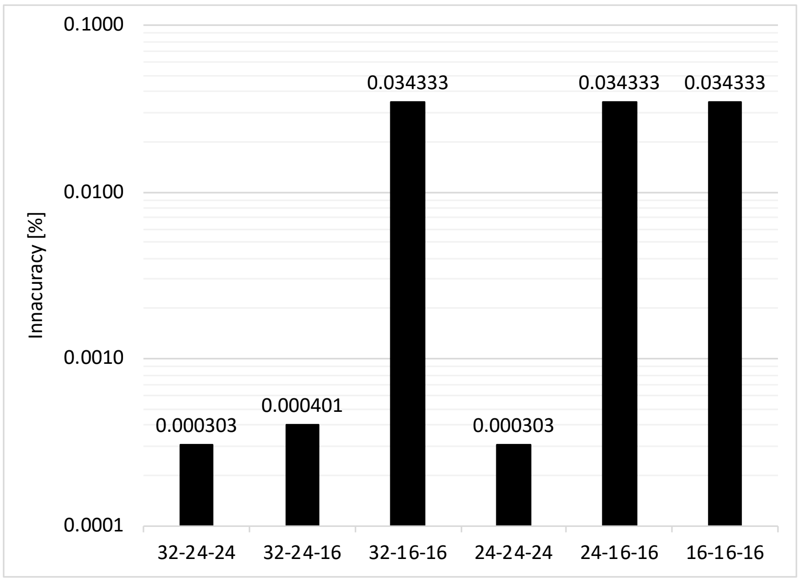
Figure 6.3 presents the inaccuracy generated by the use of approximation for each ATMR applied to the matrix multiplication operation. The data is shown in percentages and log scale. The inaccuracy value is obtained by comparing the output values of the ATMR with the one that gives better accuracy (which is the 32-bit data size multiplication due to its higher bit-size). From Figure 6.3, it is clear that the use of fewer representation bits impacts the accuracy. As expected, if a data bit-size is applied to two different ATMR modules, it determines the inaccuracy. This is due to the ATMR voter applied to the output, which ends up considering the results from this data precision as the final one — because of that behavior, using a 24-24-24 ATMR design results in the same output accuracy as a 32-24-24 one, but with lower area usage. Another interesting outcome is the inaccuracy data for the 32-24-16 ATMR design. In this case, the inaccuracy seems to hover between the ones from the three modules.

The inaccuracy, however, is usually not high. Even in the worst case, the inaccuracy is of less than 0.04%, which means the result is more than 99.96% correct. However, the increase in inaccuracy from one design to another may be a warning for more complex systems. If the inaccuracy for a complex system applying the proposed method would be significant for a 32-24-24 ATMR case, it could be considered unacceptable for the 32-16-16 one (or any situation with two modules employing 16-bit data). The ATMR variants presenting two 16-bit size modules are two orders of magnitude more inaccurate. The inaccuracy of the 3×3 matrix multiplication design follows the same trend observed for the 2×2 matrix multiplication and therefore is not presented.

6.2.1.2 Area Usage Assessment

Table 6.3 presents the FPGA area consumption of each ATMR design for the 2×2 and 3×3 matrix multiplication operation. Data shows that approximation saves DSP usage. This behavior was already expected, given the results from Figure 6.1. Neverthe-

Figure 6.3: Inaccuracy for each ATMR by data precision design applied to a 2×2 matrix multiplication.



Source: Author.

less, the FF occupation was not predicted. The FF usage can be explained by the needed converters between the matrix multiplication operations and the voter function. The LUT area follows almost the same trend of the DSPs, decreasing with the precision reduction. The variation at the LUT usage can also be explained by the needed converters. The DSP usage for the 32-32-32 TMR design was considerably high, taking into consideration that the FPGA used in this work contains 220 DSPs. This fact highlights the importance of the approximation method presented in this work.

The first 3×3 matrix multiplication TMR design is bold to highlight the number of DSPs used. The FPGA used in this work contains 220 DSPs, while the 32-32-32 TMR design for the 3×3 matrix multiplication would require 324. Therefore, this design could not be implemented on this hardware, needing a more expensive one. With the proposed approximation, however, the implementation of an ATMR-protected 3×3 matrix multiplication is now possible. All the 3×3 matrix multiplication ATMR designs fit in the FPGA.

Comparing the data from Figure 6.3 and Table 6.3, it is clear how the data precision reduction method is capable of reducing the area usage of the design with low effect on accuracy. The 32-24-16 ATMR design is capable of reducing the DSP usage to almost half of the 32-32-32 TMR design while introducing an inaccuracy of only 0.0004%. Another excellent example of the proposed approximation method efficiency is the results for the 16-16-16 ATMR design. It was able to reduce the DSP usage to a fourth and the FF usage in half while maintaining an accuracy of more than 99.96% comparing with the 32-32-

Table 6.3: Area usage and performance latency of the ATMR by data reduction designs for 2×2 and 3×3 matrix multiplications.

Benchmarks		Area			Max Latency
TMR Design	Matrices Size	DSP48E	FF	LUT	Target Clock: 10ns
32-32-32	2×2	96	1985	888	9
	3×3	**324(*)**	**7560**	**3541**	**15**
32-24-24	2×2	64	1859	761	9
	3×3	216	6543	2964	14
32-24-16	2×2	56	1763	595	9
	3×3	189	5735	2138	14
32-16-16	2×2	48	1759	945	9
	3×3	162	4576	1368	14
24-24-24	2×2	48	1815	1609	8
	3×3	162	5649	2673	12
24-16-16	2×2	32	1841	1305	6
	3×3	108	3653	1165	11
16-16-16	2×2	24	1032	689	6
	3×3	81	2257	346	9

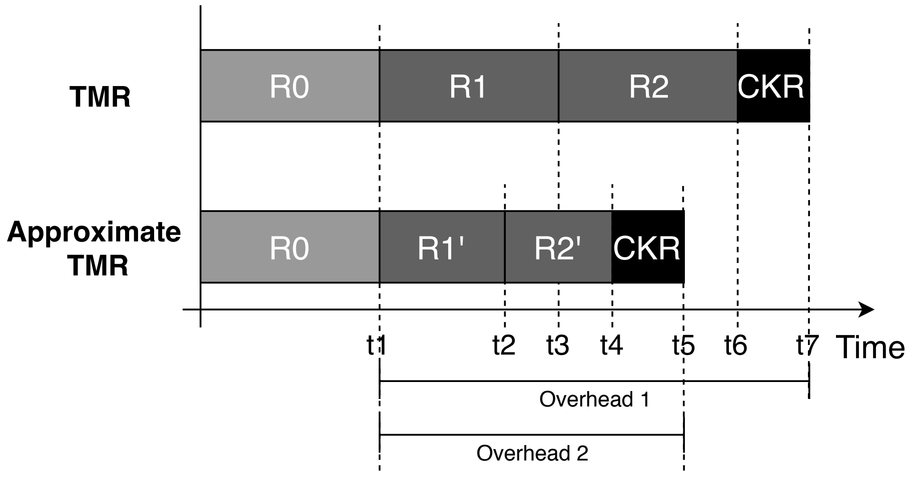
32 design. From Section 6.2.1.1 it is known that the 32-16-16, 24-16-16, and 16-16-16 ATMR designs have all the same accuracy. However, it is clear from Table 6.3 that the 16-16-16 ATMR design is a better choice not only because of the area usage but also due to its lower latency.

6.2.2 Software ATMR based on Successive Approximation

The unique behavior of successive approximation algorithms arises as an opportunity to improve traditional redundancy fault tolerance methods. The number of iterations of a successive approximation algorithm impacts not only the accuracy of the output but also its execution time. When applying a TMR method to a successive approximation algorithm, there is no need to have three tasks with high accuracy. Because only one of the outputs will be taken as the final “correct” one, the others can have a lower accuracy (i.e., fewer iterations). Tasks with lower accuracy and execution time cause less overhead.

Figure 6.4 presents the proposed ATMR method. In the figure, R1’ and R2’ are redundant tasks of R0 with fewer iterations, while R1 and R2 are hard copies of R0. The overhead of a TMR consists of the extra execution time it costs. Unfortunately, the

Figure 6.4: Diagram of the proposed ATMR method.



Source: Author.

overhead of the checker (represented at the figure by the CKR box) is constant. However, reducing the execution time of the tasks, the overhead of the TMR can be lowered. Because R1’ and R2’ execute faster than R1 and R2, the ATMR presents a speedup in relation to the TMR.

Table 6.4 presents five different ATMR configurations applied to the Newton-Raphson algorithm (which will be detailed further in Chapter 7) running in a single ARM Cortex-A9 processor with data cache enabled. This algorithm is an excellent example of successive approximation used to calculate the roots of a function and will be further better explained at Chapter 7. The execution time overhead is presented at the table as a factor and is calculated in relation to a single execution of the Newton-Raphson algorithm with 71 iterations. The execution time on the last column is the total execution time of that ATMR configuration. The benchmarks are named following the number of iterations of each ATMR task (N_0-N_1-N_2, being N_n the number of iterations of the n-th ATMR task). For example, the ATMR configuration called “71-37-14” is composed of one task with 71 iterations, one with 37 iterations and another with 14 iterations. Each ATMR task may have a different number of iterations, but the algorithm remains the same. The number of iterations of each task differs because they start at different starting points and have different stop conditions. As the table shows, the configurations with tasks that contain fewer iterations presented a lower execution time overhead.

The checker plays a critical role in the ATMR method. In a traditional TMR, the checker would make a bit-wise comparison between the three outputs, changing every bit that is different from the other two to the same value. However, with approximate computing, the checker needs to be more complex. The value of the three outputs may

Table 6.4: Execution time overheads of ATMR configurations applied to the Newton-Raphson algorithm.

ATMR Configuration	Execution Time Overhead (factor)	Execution Time [ms]
71-71-71	3.09	963.268
71-71-37	2.48	771.479
71-71-14	2.22	690.381
71-37-37	1.86	579.986
71-37-14	1.60	496.237
71-14-14	1.33	414.201

be different even in the absence of errors, because of the varying accuracy of each ATMR task. To deal with this issue, the ATMR checker is programmed to generate as system output a midterm between the three output values of the redundant tasks. Also, when no error is present, the output from the ATMR task of better accuracy (i.e., more iterations) can be used as output, thus implying in no accuracy loss. Because of this approximate checker, we have to consider a threshold of acceptable difference between the ATMR output value and the expected golden value. If the output value differs from the golden value inside this threshold limit, the ATMR is considered to have masked the error. This acceptance threshold might be different for each application or system and impacts the ATMR error masking performance. Chapter 7 will present, at Section 7.3, the fault masking results for the ATMR for three different thresholds: $\approx 0\%$, 2%, and 5%.

Another way of providing approximate computing in software is through variable data-size reduction. However, using data precision reduction on embedded software is not the same as previously presented at Sections 6.1.1 and 6.2.1. In those sections, C code is used just as a tool to generate hardware implementations with HLS. When working with embedded software, data precision reduction is more limited, as software variables are subject to predefined types. Programming new data types in software is possible, but implies on a large execution overhead, given that all the operations that would otherwise be native to the hardware in use now have to be software-processed. At Section 7.3, where the results from this ATMR are presented and discussed, two versions of the benchmark applications will be presented, making use of *float* (32-bit, single-precision) and *double* (64-bit, double-precision) variables. Because those two types of variables are capable of achieving different accuracies, they are expected to influence the behavior of the successive approximation method. Changing the variable type for a more precise one can, for example, reduce the accuracy difference between more and less precise ATMR tasks, or make the successive approximation algorithm converge faster.

6.3 Parallel Approximate Error Detection (PAED)

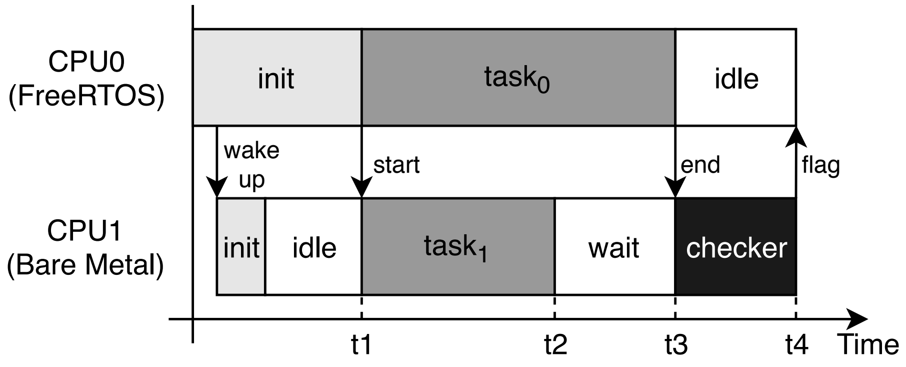
Real-time systems deal with the concept of *data freshness*, which considers that data received by the system has a specific expiration time. For example, an airplane system is continuously receiving data from the sensors. Data received one minute ago may not be valid (or even useful) anymore (e.g., the outside air temperature or a radar system that checks the distance from the ground). Those data are nevertheless critical, and a wrong value may affect the whole system's safety.

The problem when dealing with this type of data is that error correction becomes problematic. First, because of the intrinsic nature of the data: the error is often caused by a malfunction or approximation of a system sensor rather than a faulty code execution. Secondly, because of data freshness: the data shall be valid on its whole time window, which can be very strict. Therefore, data correction procedures require sophisticated implementation. The classical answer for that is the use of sanity checkers. Whenever data is not valid anymore, a flag warns the system that current given data value cannot be trusted. E.g., on avionics systems, a message might alert the pilot that a current sensor data is not assured of representing the real scenario for the next couple of seconds. Such verification is usually provided through redundancy: a copy of the task that manipulates or generates the critical data is executed, and both output values are compared to check for their consistency.

As already discussed in Chapter 5, the trend of the microprocessors industry has been to move to multicore to achieve better execution performance. Safety-critical systems can use extra processing cores to improve reliability. Real-time systems can benefit from parallelization to guarantee the respect of their strict scheduled execution deadlines. The use of redundancy-based error detection techniques such as DWC (CHEYNET et al., 2000) is more attractive on chip multiprocessor (CMP) architectures (GIZOPOULOS et al., 2011). Those techniques impose a high execution time overhead on single-core processors. However, on CMP architectures, they can exploit vacant computing resources to execute redundant tasks without compromising the performance of the system.

This section proposes an error detection technique conceived for multicore real-time systems, that profits from CMP architectures while following the general requirements of most real-time systems. It is designed with aerospace systems in mind and

Figure 6.5: Functional flow of the PAED technique.



Source: Author.

uses approximate computing to reduce the execution time overhead caused by redundant code. The new technique, called parallel approximate error detection (PAED) is adapted for multicore processors. It consists of using a processor core to execute bare metal approximate versions of tasks that are executed on the main system. The main processor runs tasks on top of FreeRTOS. The technique can be applied to any multi- or many-core processing system.

Figure 6.5 shows a graphical representation of the PAED technique. CPU0 executes the critical application to be duplicated, here represented by task$_0$, on top of FreeRTOS. CPU1 is used to execute the redundant task (task$_1$), and a checker function performs the error detection. CPU1 and task$_1$ are highly attached to CPU0 execution through dedicated synchronization primitives. Therefore, there is no need for CPU1 to execute on top of FreeRTOS too. Using bare metal to provide error detection is more reliable, as is further discussed in this work. The purpose of CPU1 is not to correct an eventual error, but to warn the system that a specific data cannot be trusted. In a multicore processing system, this warning could then be used as a flag to start an error-correction method.

As shown in Figure 6.5, CPU1 is wakened up by CPU0 at the system initiation. It then waits until task$_0$ starts its execution on CPU0. The task$_1$ block is a redundant copy of task$_0$. In the figure, task$_1$ has a smaller size than task$_0$ to represent the fact that it is not an identical copy, but rather an approximate version of task$_0$. At the end of task$_0$ and task$_1$ execution (t3), CPU1 checks the integrity of the data comparing the outputs of the tasks. If the data is not valid, a warning flag is raised. All the data management is done by CPU0, including the initialization of the inputs. The CPUs share data memory space regarding the inputs of the tasks, and all the communication flags used for synchronization. Those flags are always either read-only or write-only by each of the CPUs, avoiding memory sharing errors and race conditions. The memory space of task$_0$ is accessed by CPU1 only when the checker is executed, to verify its output value.

As previously discussed, due to the approximation, $task_0$ and $task_1$ will generate different outputs even in the absence of a fault. The accuracy loss due to the approximation has to be taken into account by the error checker, who shall accept a maximum difference threshold between the two values when comparing those. This threshold varies according to the $task_1$ approximation degree. If the values differ inside this difference threshold, the approximate checker cannot know if it was caused by a fault or by the approximation itself. A system designer shall then carefully analyze this threshold before making use of an approximate checker: some systems might not tolerate even the slightest accuracy deviation.

The idle and wait blocks in Figure 6.5 can be used to improve system performance. In a more complex system, executing a multitude of tasks, processors could use those extra time windows to execute other functions. Even a hard real-time system could profit from it, as long as t1, t2 t3, and t4 are well defined. A real-time system designer would know the values of the time intervals defined at the figure, thus knowing with precision how much spare time CPU1 has to execute other tasks. He could then configure the system for optimal task scheduling.

When dealing with real-time systems, execution deadlines have to be taken seriously. Because of that, the error detection system works on a tight synchronization. CPU1 only executes its functions when CPU0 activates the right communication flags. The processors communicate via shared memory resources, highly synchronized. The memory space of each core is well defined. Even though they share the same on-chip memory, the address range accessible to each is different. The only memory shared between the two cores are the flags and the memory that holds the input of the benchmark applications. The checker task, always executed by CPU1, is the only time when CPU1 accesses the CPU0 memory space, only to compare $task_0$ with the one from $task_1$.

7 EXPERIMENTAL RESULTS AND DISCUSSION

This chapter presents results and discussions on experiments realized on the approximate computing techniques and the proposed approximate fault tolerance techniques. Section 7.1 presents discussions on the proposed approximation methods implementations and results regarding their reliability under emulation and laser fault injections. Sections 7.2 and 7.3 present the results for the ATMRs behaviour under onboard and laser fault injections. Finally, Section 7.4 presents the results for the approximate error detection technique under laser fault injection.

The proposals from Chapter 6 are evaluated using the methodologies presented at Chapter 4. However, not all methodologies are applied to all proposals: each of them has its most appropriate evaluation methodology. The Taylor series approximation is the only proposed technique which will be not tested under fault injection experiments. Instead, Section 7.1.1 will analyze its implementation costs and impact on the system accuracy at hardware and software. All other proposed works will be evaluated for their fault tolerance and computational resources cost (e.g., execution time and programmable hardware area). A sufficient number of errors was gathered from each of the experiments to obtain statistically significant results with an error margin of 1% and a confidence level of 95% using the approach presented in (LEVEUGLE et al., 2009).

7.1 Approximation Methods

This section starts by presenting a discussion regarding the implementation of Taylor series approximation on hardware and software, their development, implementation costs, and impact on accuracy in Subsection 7.1.1. Then Subsection 7.1.2 presents a study of the successive approximation under emulation and laser fault injection. Finally, Subsection 7.1.3 presents a study on the behavior of approximate algorithms compared to non-approximate traditional applications, executing bare metal and on top of operating systems (FreeRTOS and Linux).

7.1.1 Taylor Series Approximation

The data from Tables 6.1 and 6.2, presented and discussed in Section 6.1.3, arise a multitude of questions. They show that the hardware and software implementations have each its particularities. A comparison between the area resources usage from HLS and embedded software does not make sense, because the area of the ARM processor is constant and not dependable of the program implementation. However, it is possible to compare some data from hardware and software. As an example, the latency of HLS hardware implementations (particularly the ones without pipeline) is comparable with the execution latency of the embedded software implementation. This section discusses the results obtained from Section 6.1.3 and speculates on their implications.

7.1.1.1 Hardware Implementation Analysis

Section 6.1.3 presented the Vivado HLS implementation details, at Table 6.1. The usage of pipeline arises as a good alternative for projects that need to rely on fast execution, and which are implemented on boards on which area is not a problem. In fact, using pipeline may even not be costly. For instance, comparing the pipelined version of double-precision with its counterpart with no pipeline, Table 6.1 shows that when both achieve an accuracy of more than 99% the pipelined version uses around four times the number of LUTs and FFs but is almost seven times faster. Similar behavior is observed on the *float* variants (also called single-precision, i.e., 32-bit floating-point format).

Hardware area resource usage can be a problem for some systems. As was explained in Section 6.1.3, the maximum numbers of Taylor series terms that fit in the Zynq board FPGA are 13 and 34 for double- and single-precision, respectively, when making use of pipeline. Nevertheless, every benchmark variation shows that the accuracy of the Taylor series approximation largely increases following a small number of terms. Because of that, the need for a high number of terms is improbable. Projects in need of high accuracy can also achieve it while saving area by limiting the size of the pipeline, breaking the computation loop into big chunks. This type of implementation strategy is not analyzed in this work. Table 6.1 also shows that increasing the area of the design increases the number of essential bits. This can be seen as problematic for safety-critical systems since it is highly related to the project susceptibility to errors when exposed to radiation (TONFAT et al., 2016). As referenced in Section 6.1.3, the type of approximation presented in this work might improve the fault tolerance of an application. If that is the purpose of the approximation, a designer may choose not to use the pipeline approach.

The accuracy (defined by the number of Taylor series terms) is determined by different factors for each variant of the benchmark. The versions without pipeline implementations have their accuracy determined by their area size. However, the pipelined versions achieve accuracy by increasing their latency. A designer shall know where he shall pay the cost of accuracy: whether in latency or area. The advantage of using implementations with no pipeline is that, despite taking a long time to output, it is always capable of achieving the best accuracy possible. That is because time is a resource not limited by the hardware, but by project constraints. The same can not be said of pipelined implementations: even though they are faster than the designs that do not use it, they have their maximum accuracy limited by the programmable area of the hardware.

Table 6.1 shows that improving the accuracy of an already accurate version of the algorithm is more costly than enhancing an inaccurate one. As a good example, there is the double-precision pipelined version. In that case, improving the accuracy from 54.7% to 89.28% is less costly in the area than improving it from 99.994% to 99.999%. The same behavior is observed on all versions of the code. It shows that the higher the index of the Taylor series term, the lower is its impact on the final result, thus less critical it is. It also indicates that there is a maximum accuracy attainable by the approximation method, but it is very near 100%. The table proves the importance of a preliminary study to avoid unnecessary or unworthy area usage. For instance, there is no reason to use 100 Taylor terms on the double-precision without pipeline version because 25 is already enough to achieve an accuracy of 100%.

7.1.1.2 Software Implementation Analysis

Section 6.1.3 presented on Table 6.2 the performance details from the Taylor series implementation running as a bare metal application on the ARM A9 processor. The double-precision variant of the algorithm was the only one capable of achieving an accuracy of 100%. Nevertheless, the single-precision met good accuracy with low execution latencies.

Contrary to the hardware HLS approach, the only cost for embedded software to acquire accuracy is execution latency. The memory use of the variants is also different, but the absolute value is so small that it has no cost impact. The output is only one 64- or 32-bit variable, for *double* and *float*, respectively. Unless a project needs accuracy of 100%, there seems to be no reason to use double-precision instead of single-precision.

Nevertheless, it is important to remember that the accuracy of each algorithm variant is calculated taking as parameter its own best result possible (assuming it to be the *math.h* library result). It means that not only double-precision provides accuracy, but also a more precise result. The type of data precision defines how much decimal points the variable can hold. This number may change depending on the target processor architecture or compiler used, but normally double-precision holds thirteen decimal points, while single-precision holds seven. For projects that need high exactitude, the seven decimal points provided by single-precision may not be enough. In those cases, double-precision is a must. However, as Table 6.2 shows, Taylor series can provide accuracy with almost the same computation execution latency as single-precision (*float*).

7.1.1.3 Discussion on the Software and Hardware Implementations

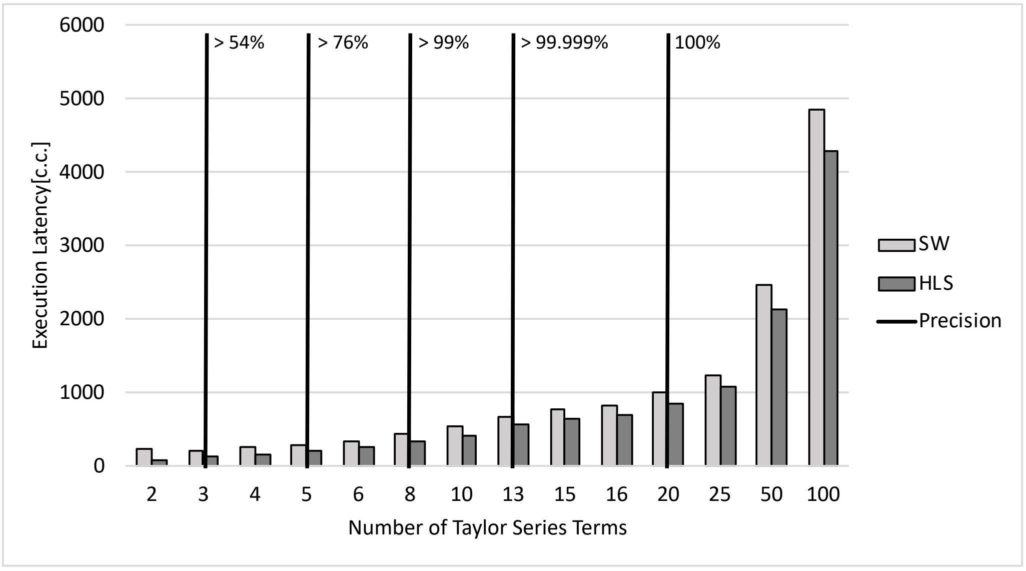
The software implementation achieves accuracy by increasing the number of Taylor series terms, which by its turn increases the execution latency. In the HLS hardware implementations, increasing the number of terms would cause an increase of area and latency (being a higher variation in the area for the pipelined variants and higher latency for the ones without pipeline). Because the pipelined hardware increases area (and almost no latency) to achieve accuracy, it is not fair to compare it with the software implementations. The hardware implementations without pipeline, however, are comparable with the software implementations, as both achieve accuracy at the same price: latency.

Figure 7.1 presents the data from software and hardware execution latency for the double-precision algorithms. The black vertical lines mark some important precision barriers. Following the tendency observed at Tables 6.1 and 6.2, the precision increases faster for the first terms, and slower as it gets near 100%. The unexpected result is that the hardware HLS and software (SW) implementations had virtually the same execution in clock cycles.

The comparison between the HLS hardware without pipeline and software implementations with single-precision regarding execution latency is presented in Figure 7.2. In this case, the software implementation takes more clock cycles to finish than the hardware one. The lines progression shows that the number of required clock cycles rises exponentially, as well as the difference between their absolute values. It indicates that if a higher number of terms were needed, the software approach would take much more clock cycles to finish the execution than the HLS.

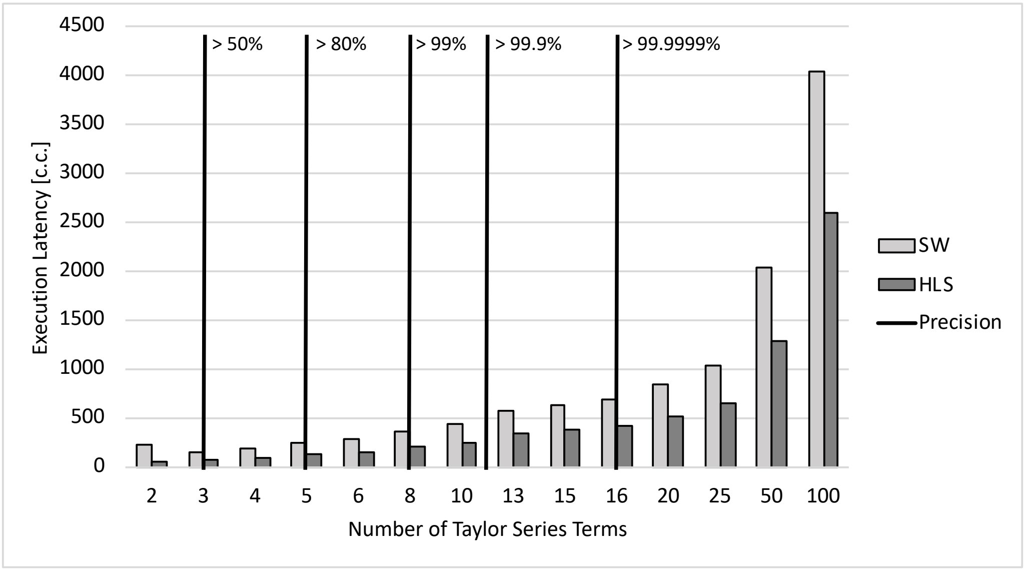
Studying the execution performance only through the number of clock cycles may

Figure 7.1: Execution latency comparison between HLS without pipeline and SW (software) implementations for double-precision.



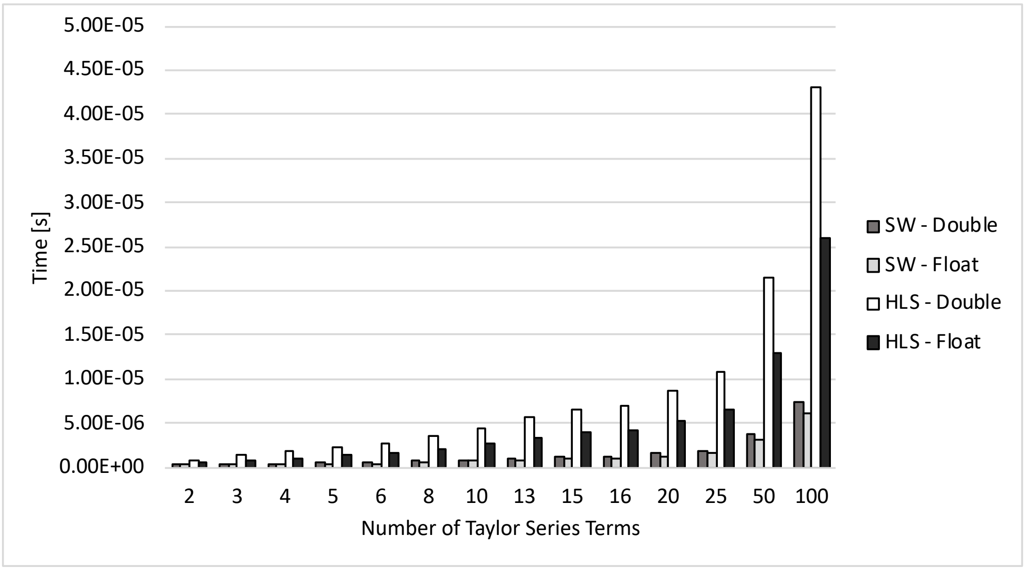
Source: Author.

Figure 7.2: Execution latency comparison between HLS without pipeline and SW (software) implementations for single-precision (*float* type variables).



Source: Author.

Figure 7.3: Time comparison between HLS without pipeline and SW (software) implementations for both data precisions.



Source: Author.

be misleading. That is because the ARM processor and the PL of the Zynq-7000 APSoC have different frequencies. The embedded ARM processor work with on 666MHz while the FPGA on the PL runs with a frequency of 100MHz. It means that a software version of the algorithm may execute faster than an HLS hardware implementation even with a higher clock cycle count. Figure 7.3 presents the execution time in seconds of the two data precision variants from HLS hardware (without pipeline) and software implementations. It shows that software implementations are always faster than hardware with no pipeline. This is an unexpected result, taking into account that hardware implementations tend to be faster than embedded software. In contrast to Figures 7.1 and 7.2, Figure 7.3 shows that a more general and less optimal design (with a higher clock cycles count) can execute faster than an optimal one (with fewer clock cycles). It all depends on the target hardware. It is important to notice, however, that the hardware implementations in Figure 7.3 do not take full profit of the parallelization capacity of the FPGA since they are not implementing pipelines and loop unrolling.

The question of whether to use hardware or embedded software to implement approximation through Taylor series seems to have no definite answer. While software arises as a better alternative than low-area hardware, it is still slower than an optimal pipelined hardware implementation. However, the high area cost of a pipeline for the Taylor series with a high number of terms may prove its implementation unfeasible on smaller FPGAs. A profound evaluation of the alternatives shall be performed before a design decision, since project time and area constraints may vary.

7.1.2 Successive Approximation

Three successive approximation numerical algorithms are presented and used as benchmark applications:

- **Newton-Raphson:** The Newton-Raphson method is an algorithm used to find the roots of a function. It calculates the intersection of the tangent line of the function in an initial guess point x_0 with the x-axis. It is calculated iteratively, as stated in (7.1), until it reaches a sufficient approximation.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{7.1}$$

- **Trapezoid Rule:** The trapezoid rule algorithm is used to calculate the integral of a function. It approximates the area under a curve to some trapezoids and then calculates their areas. Considering N equally spaced trapezoids defined between points a and b of the function, we have each trapezoid k with a base of length $\Delta x_k = \Delta x = \frac{b-a}{N}$. The integral approximation with the trapezoid rule is defined in (7.2).

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{2} \sum_{k=1}^{N} \left(f(x_{k-1}) + f(x_k)\right) \tag{7.2}$$

- **Simpson:** Another way of numerically approximating a function integral is with the Simpson's rule. The difference between the Simpson's rule and the Trapezoid is that it calculates the area of parabolas instead of trapezoids. This way, it usually approximates the result with more exactitude in fewer iterations than the Trapezoid rule. The Simpson approximation for an integral with a step size of $h = (b-a)/2$ is presented in (7.3).

$$\int_a^b f(x)\,dx = \frac{h}{3}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] \tag{7.3}$$

Each one of those benchmarks has three variants. For each variant, the number of iterations is different (and therefore the accuracy), but the algorithm remains the same. Table 7.1 provides some details on each of those variants and how the number of iterations affects the application execution. Some algorithms converge faster to a final acceptable result than others (notice the difference between Trapezoid and Newton-Raphson at Table 7.1), and therefore naturally present a lower number of iterations. The different itera-

Table 7.1: Successive approximation experiments benchmarks details.

App.	Var.	Num. of Iters.	Used Registers	L1 Data Cache Accesses [per ms]	Exec. Time [ms]
Simpson	1	242	r2, r3, r11, pc, sp, lr	178.2k	0.94
	2	423		1648.1k	9.31
	3	3081		2350.4k	18.62
Trapezoid	1	128	r0, r2, r3, r11, pc, sp, lr	6053.2k	202.34
	2	1274		6792.4k	605.29
	3	12746		6763.1k	33540.09
Newton-Raphson	1	14	r2, r3, r11, pc, sp, lr	97.2k	0.44
	2	37		202.4k	1.19
	3	71		682.8k	3.19

tions number of the benchmarks allows a more in-depth assessment of how it impacts the algorithm execution behavior in relation to reliability.

The benchmarks are tested under laser fault injection at the L1 data cache memory and fault injection emulation on the register file. The results are used to assess and discuss how the successive approximation, inherent to those algorithms, affects their fault tolerance. For that purpose, each benchmark variation is compared with each other, so that the number of iterations' impact on the fault tolerance is evaluated. Faults affecting the register file are expected to have a higher probability of vanishing than the ones affecting the cache memory. As Table 7.1 shows, the proposed benchmarks are far from using all registers available. The low use of registers means that the sensitive area of the register file is small. Therefore faults affecting it may touch registers that are not even in use. Injecting faults in the cache memory, however, may lead to unexpected behaviors. Some of the benchmarks have a high number of cache memory accesses. It can cause the fault to be read into the applications and provoke an error or faulty memory space to be overwritten, causing the fault to vanish.

The fault tolerance of the benchmarks is evaluated in two aspects. First, we assess how the number of iterations impacts the error susceptibility, i.e., how each variant presented at Table 7.1 behaves under fault injection. Variating the number of iterations for each benchmark has a significant impact on fault tolerance. This evaluation is made with results from both laser and emulation fault injection. Figures 7.4, 7.5 and 7.6 present the error occurrence (in percentage) for each type of error. Figures 7.7, 7.8 and 7.10 present the error relative probability (per laser pulse) for each benchmark and their variants, as presented at Table 7.2. This probability is calculated by normalizing the error occurrence values with their maximum for each benchmark. The normalization is needed because the error occurrence depends on the execution time and the shots per execution of the benchmark, and those are very different for each application.

Secondly, we evaluate how tolerating small variances on the output value can reduce the number of considered SDC-type errors. For that assessment, we compare the output values with the golden value and check how different they are. So for example, if an application can tolerate an output variation of 2%, an output value will only be considered erroneous if it is less than 98% equal to the golden value. This evaluation was made from the results from the laser fault injection only.

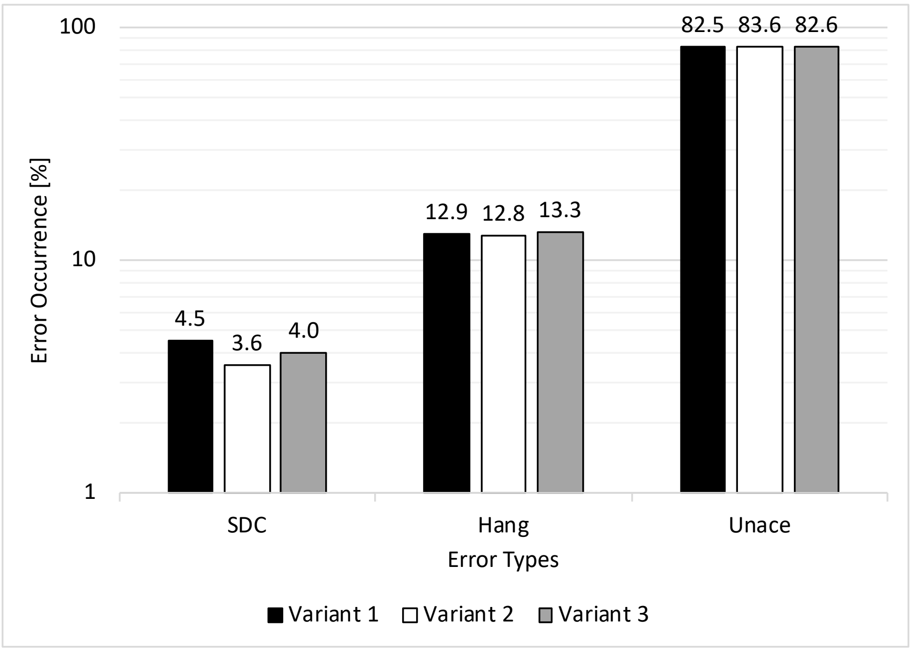
7.1.2.1 Fault Injection Emulation on Register File

Contrary to the laser fault injection, the emulation fault injection is programmed to inject one fault per execution of the algorithm. This would be impossible on laser fault injections due to the frequency of the laser pulse and the delays of the experimental system, and because some of the benchmarks are very fast. The details of the emulation fault injection differ from laser ones not only due to its different characteristics but also because of the different focus of this methodology. All the interesting data on this type of injection was presented in Table 7.1. The most important information about it is the number of registers in use. The execution time is also important because it may define the probability of an error to be corrected (due to a higher amount of iterations), but the emulation fault injection assures that there will be only one fault injected per execution, no matter the execution time.

The data from Table 7.1 is gathered from the implementation at the Zynq-7000 APSoC. This was gathered concerning only the general-purpose registers (from r0 to r12 and the stack pointer, link register, and program counter). The table shows that those benchmarks tend to use few registers. It also shows that the number of accesses to the L1 Data Cache is heavily impacted by the number of iterations of the loop, with the exception of the Trapezoid application. In that case, it is probably because all the Trapezoid variants already have a high data cache access, possibly the highest possible. The fact that Trapezoid is the benchmark with the highest execution time supports that idea.

Figure 7.4 presents the percentage of each error type occurrence from the emulation fault injection at the register file. The "Unace" bars show the percentage of faults that did not cause an error. The y-axis is presented in log scale to facilitate the view of the data,

Figure 7.4: Simpson error occurrence for emulation fault injection.



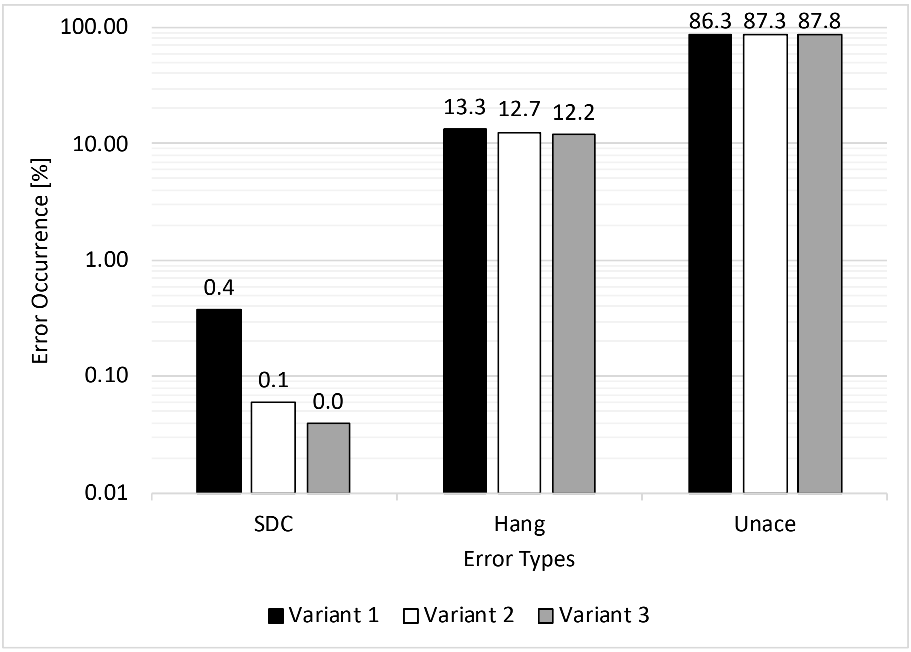
Source: Author.

given that there are significant differences between the occurrences. In that case, increasing the number of iterations of the algorithm has little to no effect on the distribution of errors. The high number of unaces shows the faults tend to vanish. As discussed before, faults are expected to vanish due to the nature of successive approximation. However, it is interesting to see that they did not vanish the same way the ones injected at the cache memory did (as will be presented further at Section 7.1.2.2). In that case, all the variants had the same fault tolerance.

Figure 7.5 presents the percentage of each error type occurrence from the emulation fault injection for the Trapezoid algorithm. For that algorithm, the occurrence of SDCs dropped while increasing the number of iterations. However, the SDC occurrence for variant 1 was already very low. Comparing the results from Figures 7.5 and 7.4, it is clear that the Trapezoid algorithm is much less prone to SDCs than Simpson.

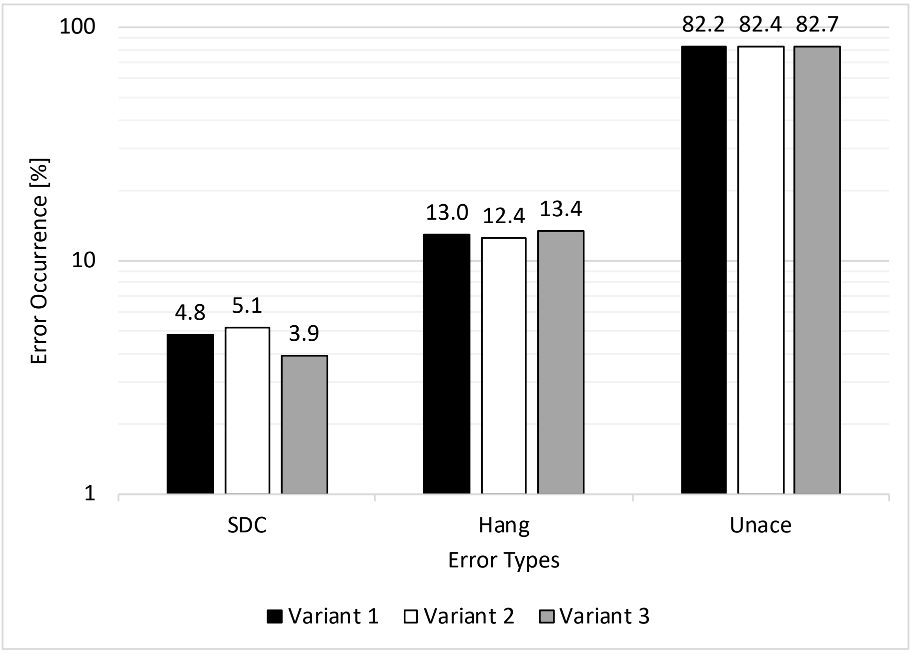
The emulation fault injection results for the Newton-Raphson benchmarks are presented in Figure 7.6. Again, the variation in the number of iterations did not affect the type of error distribution. Hangs are also more frequent than SDC, which is expected given that those are iteration-based algorithms. A significant part of the execution concerns loop management. Therefore it is a definite critical point of failure. Still, most of the faults (around 82%) caused no errors.

Figure 7.5: Trapezoid error occurrence for emulation fault injection.



Source: Author.

Figure 7.6: Newton-Raphson error occurrence for emulation fault injection.



Source: Author.

Table 7.2: Laser fault injection details on successive approximation benchmarks.

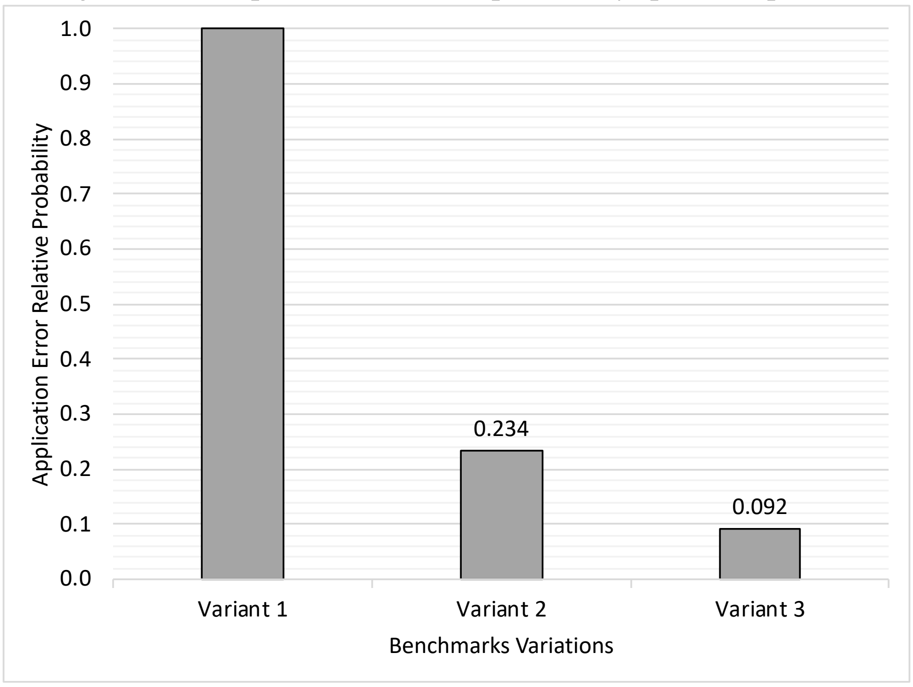
App.	Var.	Num. of Runs (N)	Total Workload [Bytes]	Avg. Shots per Exec.	Exec. Time [ms]
Simpson	1	100	400	0.9669	96.69
	2	100	400	9.4887	948.87
	3	100	400	18.9963	1899.63
Trapezoid	1	150	1200	302.8096	30280.96
	2	70	560	444.4170	44441.70
	3	1	8	546.1586	54615.86
Newton-Raphson	1	100	800	0.417	41.72
	2	100	800	1.302	130.20
	3	100	800	3.379	337.91

7.1.2.2 Laser Fault Injection on Data Cache Memory

The benchmarks execute on loop under the laser fault injection, fulfilling an output vector that is later checked for SDC errors. This was made to extend the execution time of the benchmarks and assure and almost-random fault injection on different points of the algorithm execution. Table 7.2 presents the details of the laser fault injection experiments, applied to each benchmark. The number of runs is the size of the output vector, i.e., the value of N times an algorithm runs per execution. The "Total Workload" represents the size in bytes of the output vector. With that data, it is possible to infer the *workload per run* (the size of the outputs) by simply dividing the number of runs N by the total workload per run. The "Execution Time" is the time of a complete execution (N runs). Finally, the "Average Shots per Execution" column presents the average number of laser shots per execution, which is calculated dividing the execution time and the time between laser shots (i.e., the inverse value of the laser frequency). Notice that the number of runs N is different from the number of iterations presented at Table 7.1. On each run, the benchmarks execute the number of iterations defined by each variant at Table 7.1.

The application error relative probability per laser pulse of the Simpson benchmark is presented in Figure 7.7. As expected, the variants with a larger number of iterations are more fault-tolerant. However, more iterations mean more latency. As Table 7.2 shows, the variant 3 of the Simpson benchmark is almost 20 times slower than the variant 1, but Figure 7.7 shows the error occurrence does not decrease in the same pace. It means that, for this algorithm, increasing the number of iterations improves reliability, but the price is high.

Figure 7.7: Simpson error relative probability (per laser pulse).

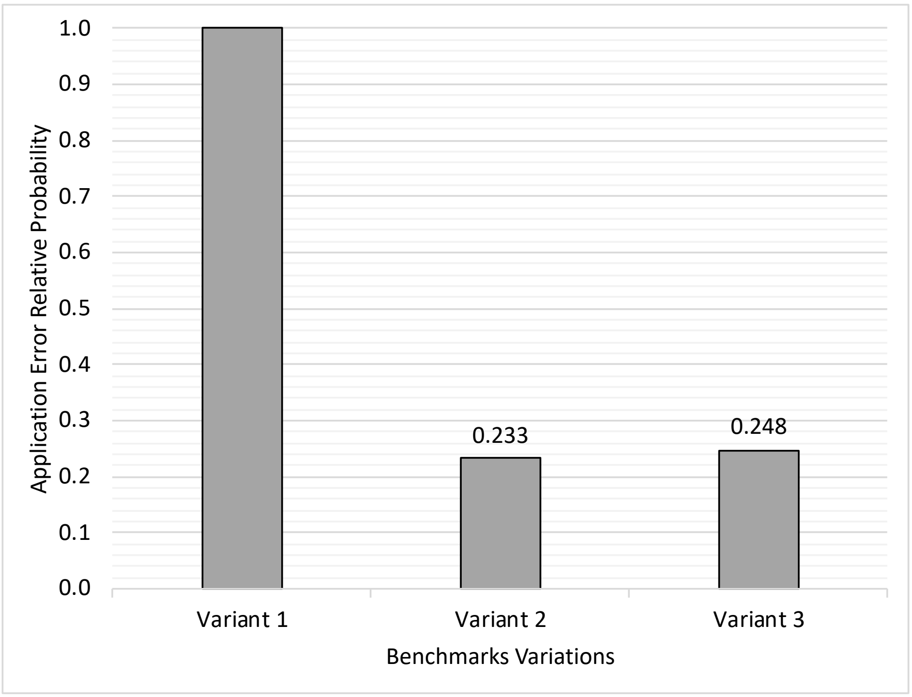


Source: Author.

The Trapezoid rule also shows a significant improvement in reliability for higher numbers of iterations, but it tends to stabilize at a certain point, as Figure 7.8 shows. This is because the Trapezoid rule converges slower than the Simpson method. For that same reason, the number of iterations for each version of this benchmark is higher than the other ones (see Table 7.1). The variants 2 and 3 of the Trapezoid had a very similar result. It indicates that the benchmark might have an optimal point of fault tolerance on around 1200 iterations (according to Table 7.1). Using more iterations than that would add more execution time to the algorithm, but have no impact on fault tolerance. Nevertheless, as Table 7.2 shows, the execution time difference from the three Trapezoid variants are not as big as in other benchmarks.

According to Table 7.2, the execution time of Trapezoid is much longer than Simpson. Given the fact that both algorithms are applied to solve the same problem (calculating an integral), we can draw interesting conclusions by comparing both their results. Figure 7.9 presents the values of the application error relative probability per laser pulse calculated and normalized for Simpson and Trapezoid together. It is noticeable that Trapezoid is much more fault-tolerant than Simpson. This result indicates that having a higher number of iteration is beneficial for fault tolerance, but some applications might pay a high price for that. It is also clear that for this kind of approximate computing algorithm, the drawback for increasing reliability is execution time. When using an approximate com-

Figure 7.8: Trapezoid error relative probability (per laser pulse).



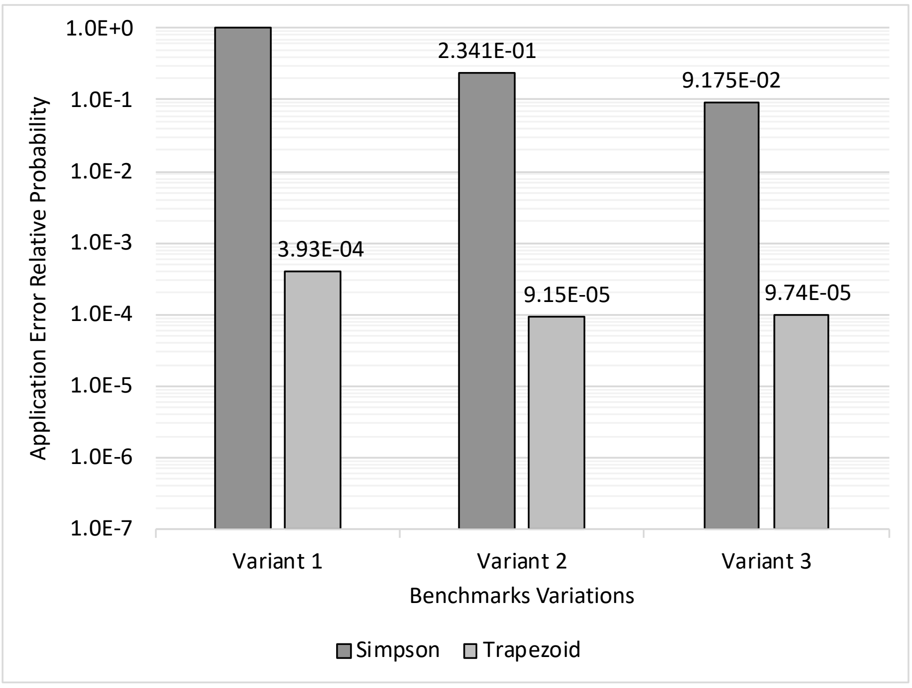
Source: Author.

puting technique to solve a computation problem, different approaches will provide very different fault tolerances, even if they are similar.

Newton-Raphson presents a behavior similar to Simpson. Figure 7.10 indicates that the variant 2 of this benchmark already achieves a considerable fault tolerance improvement, having a relative probability two orders of magnitude smaller than the first variant. It is interesting to notice that this benchmark is the one with the lower number of iterations, as reported in Table 7.1. What it indicates is that the number of iterations alone is not enough to provide a fault tolerance estimation. Different from the other two, this benchmark is not used to calculate an integral, but the roots of a function. It also has a very convergent nature, so a high number of iterations is not necessary.

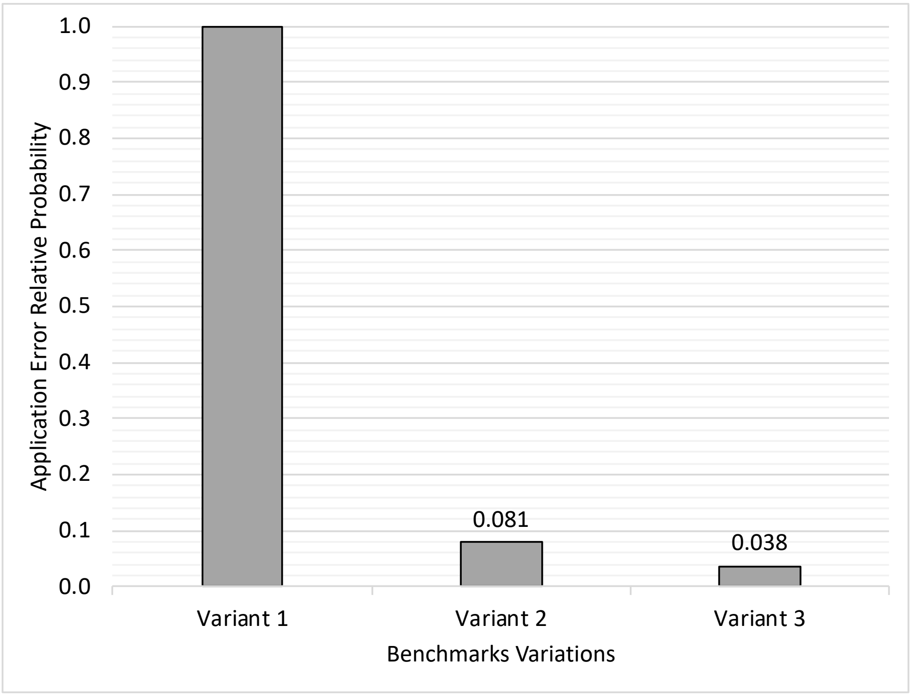
It is interesting to see that the fault effects on the register file and cache memory are very distinct. While the faults injected at the cache tend to vanish for a higher number of iterations, the ones injected at the register file have almost the same effect no matter the loop size (the exception of the Trapezoid benchmark is noticeable, but the difference between the variants SDC error occurrence is still meager). Two facts can explain it: the register usage of the benchmarks is very low, and the data cache memory usage is crucial. As Table 7.1 shows, the benchmarks do not use all the registers. However, the fault injection on the register file is considering all of them. Thus the probability of a fault to affect a register being used is not very high. The way registers are used also affects their

Figure 7.9: Application error relative probability (per laser pulse) calculated for Trapezoid and Simpson together.



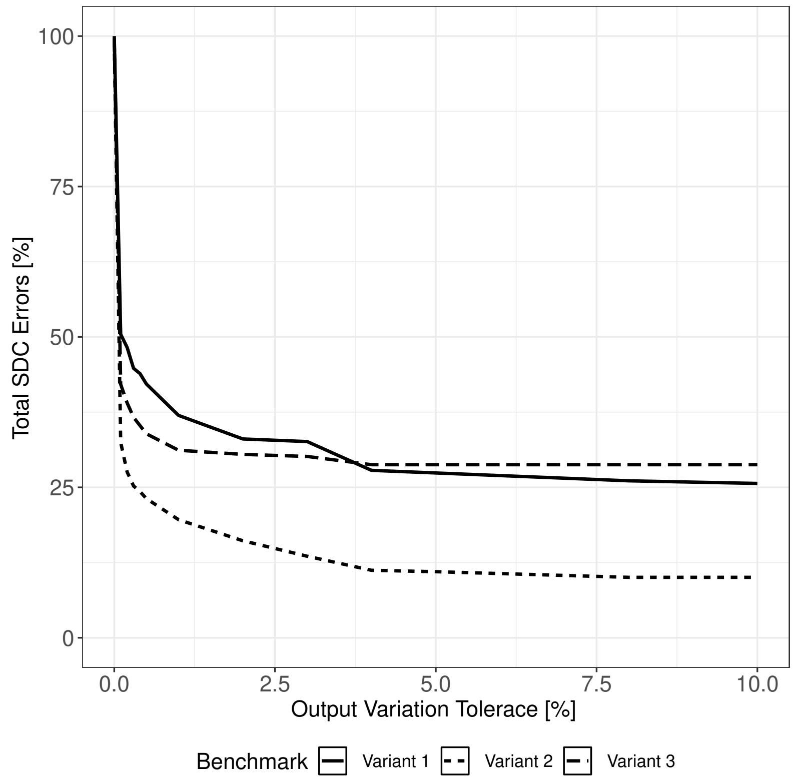
Source: Author.

Figure 7.10: Newton-Raphson error relative probability (per laser pulse).



Source: Author.

Figure 7.11: Error occurrence drop in relation to output variation tolerance for Simpson benchmark.



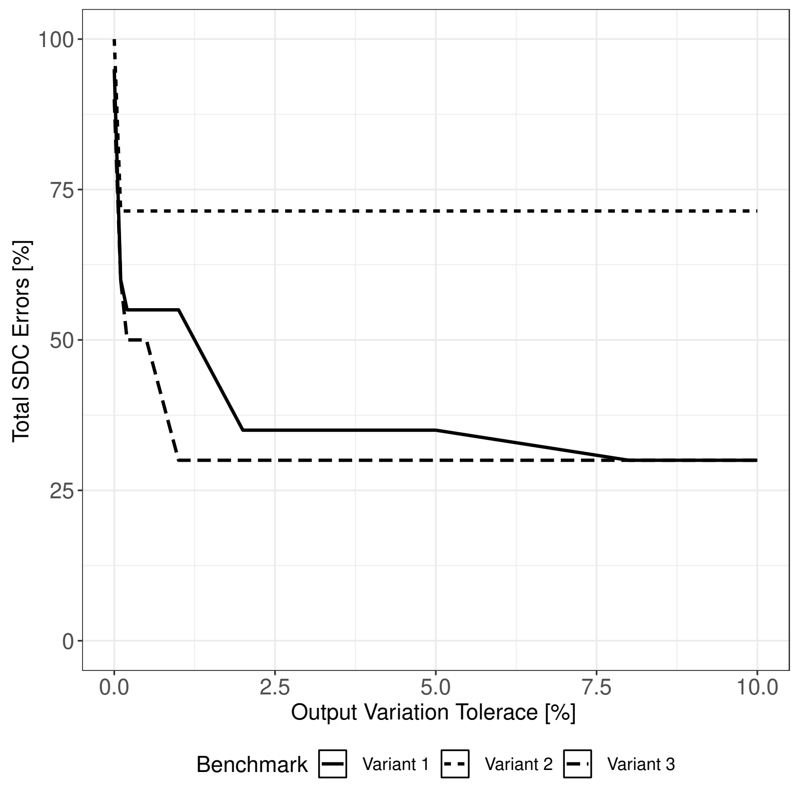
Source: Author.

criticality: they are continually being overwritten, and so are the faults injected into them. The data cache memory has a higher data latency (i.e., data usually stays untouched longer than at registers), and is also where most of the results are stored (while registers are used not to store data, but mainly to process it). Therefore, faults injected in the data cache have a more significant probability of spreading to the final output of the application.

Figure 7.11 presents the results for the error reduction when accepting output variations for the Simpson benchmark. It is clear that having an output variation tolerance of about 2.5% is enough to have a significant reduction of error occurrence. Each variant presented different results on that evaluation, but a general trend is clear. Most of the errors on this application's outputs are small, i.e., the final value does not differ much from the expected. An approximate computing system, which is able to tolerate those small errors, may benefit from this output relaxation to provide reliability.

Figure 7.12 presents the output variation tolerance effect on the number of perceived errors for the Trapezoid algorithm. It has a very different behavior from the other benchmarks. In the worst case, for variant 2, the occurrence of SDC errors drops more than 25%, but it remains even when accepting more significant variations. The other variants presented a drop in total SDC errors of about 70% when accepting up to 2.5% output

Figure 7.12: Error occurrence drop in relation to output variation tolerance for Trapezoid benchmark.



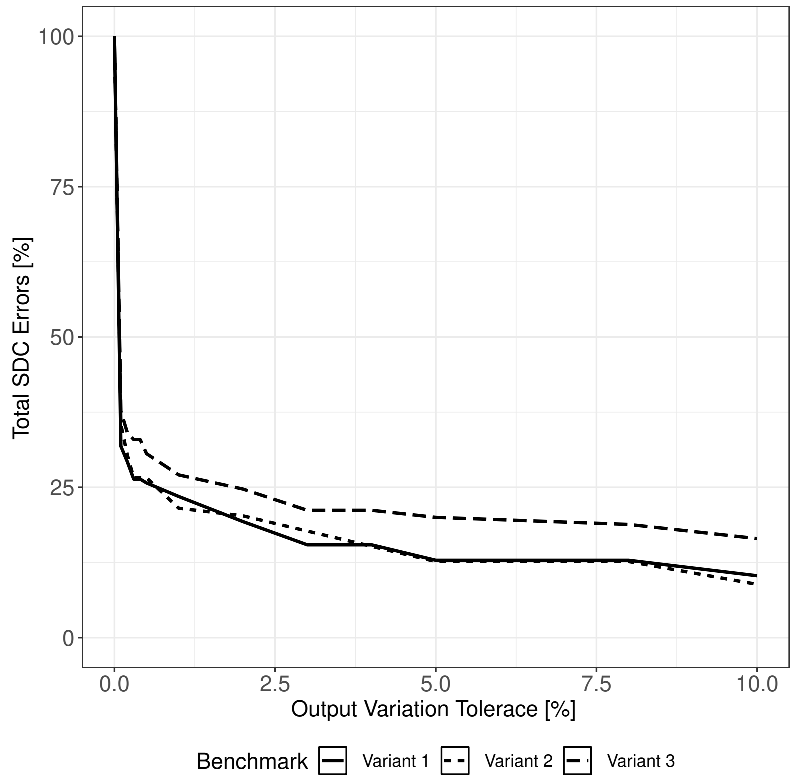
Source: Author.

variation from the golden value. This unexpected Trapezoid behavior can be explained by its already low error occurrence. Because Trapezoid already presented much fewer errors than Simpson, it has fewer errors to tolerate. Thus a more considerable amount of those is significant.

The error drop when variating the output value error tolerance for the Newton-Raphson algorithm is shown in Figure 7.13. It also presents the same trend from the Simpson benchmark. The variant 3 of Newton-Raphson is the one that had a lower drop on error count while increasing tolerance. It means that the errors had a higher difference in relation to the expected output; in other words, they were “more erroneous”. The error drop stagnates after around 4% of output variation tolerance.

All the results from the output tolerance variation show significant error drops. Tolerating small deviances on the expected output of an algorithm is the very definition of approximate computing. Those results indicate that approximate computing no only can be applied to safety-critical systems, but might even improve their fault tolerance. It is important, however, to notice that some systems may not tolerate even minimal output deviations. Those are not good candidates for any approximate computing technique.

Figure 7.13: Error occurrence drop in relation to output variation tolerance for Newton-Raphson benchmark.



Source: Author.

7.1.3 Behaviour and Application Evaluation on Operating Systems

Safety-critical systems may need to manage the execution of many applications, sharing resources. To guarantee the safe management of those resources, the use of an operating system is attractive once running bare metal applications on a system could lead to a waste of resources. As discussed in Chapter 5, it is important to evaluate the possibility of an OS usage and its effects on the system behavior and fault tolerance. Also, knowing that successive approximation has an impact on fault tolerance, it is imperative to evaluate how it co-relates with traditional algorithms (non-approximate), as well as how it behaves when executing on top of a complex operating system.

One can not expect that two different applications will behave similarly, even when executing in the same hardware and the same physical conditions. It is also shown that the system on which an application is executing is a significant factor for the error sensibility evaluation: some of our past works proved that an application's fault tolerance might differ a lot when executing bare metal or on top of a complex operating system such as Linux.

In this section, we present an evaluation of successive approximation algorithms under fault injection simulation, executing both on bare metal systems and on top of FreeRTOS and Linux. We also compare their results with some ordinary computation benchmarks. Given the data from related works, it is expected that a lighter OS such as FreeRTOS will have less impact on the system susceptibility to errors than a more robust, complex ones such as Linux. It is natural for a more complex system to have more critical points of failure. Therefore we expect bare metal applications to be less susceptible to errors.

This part of the work uses the fault injection simulation implemented at OVPSim, and presented at Section 4.3. The fault injection is simulated in this part because the usage of Linux OS makes a physical experiment of fault injection very difficult. Because the operating system has a boot time to initialize itself and takes a big part of the computational resources of the processor, physical fault injection methodologies do not work very well. Using OVPSim fault injection, we were able to inject faults only in the part of the execution that was interesting for this work: the application running on top of the operating system.

The results analysis consists of a comparison between a golden execution of the application (i.e., with no fault injection) and the executions under fault injection. Three successive approximations and three ordinary computing algorithms are presented. Each one of those algorithms will be presented in three different versions. The first one is a bare metal application implementation, that is, executed on top of no OS. The second version runs an application with FreeRTOS operating system. The third version runs on top of Linux OS. The algorithms used as benchmarks are:

- **Successive Approximation Algorithms:** Contains the benchmarks for the Newton-Raphson and Trapezoid methods, both used to approximate the result of the integral of a function, and QSolver, which presents the root computation of quadratic equations.
- **General Purpose Algorithms:**. The Matrix Multiplication, Vector Sum, and Hanoi benchmarks represent ordinary algorithms. Those are normal matrix multiplication, vector sum, and the tower of Hanoi puzzle solver.

Table 7.3 presents the results for every type of error for each application and execution. Note that the Exception errors are divided between segmentation faults and every other type (unidentified). This categorization is made because the majority of exceptions

Table 7.3: Error distribution for simulated fault injections using OVPSim.

Execution		Errors [%]				
		General			Exceptions	
OS	Application	UNACE	SDC	HANG	Seg. Fault	Unidentified
Bare Metal	QSolver	82.1	0.9	17.0	-	-
	Newton-Raphson	77.1	9.6	13.3	-	-
	Trapezoid	87.2	3.4	9.4	-	-
	Matrix Multiplication	70.1	19.5	10.4	-	-
	Vector Sum	65.2	23.6	11.2	-	-
	Hanoi	77.8	13.0	9.2	-	-
FreeRTOS	QSolver	43.0	9.6	47.1	-	0.4
	Newton-Raphson	74.5	8.7	15.9	-	0.9
	Trapezoid	58.3	10.4	31.2	-	0.1
	Matrix Multiplication	42.1	16.6	40.9	-	0.4
	Vector Sum	42.3	14.6	43.0	-	0.1
	Hanoi	24.5	40.8	30.7	-	4.0
Linux	QSolver	53.5	19.4	9.3	6.7	11.1
	Newton-Raphson	53.5	18.8	9.4	6.0	12.2
	Trapezoid	55.7	28.5	0.3	15.3	0.2
	Matrix Multiplication	45.4	37.0	4.8	12.3	0.5
	Vector Sum	43.6	40.2	4.9	10.9	0.5
	Hanoi	58.1	17.5	7.8	8.1	8.5

are usually segmentation faults, therefore it is interesting data.

Analyzing the bare metal results, it is clear that the successive approximation algorithms are much less susceptible to SDC errors than the three other applications. Comparing the worst case scenario for the successive approximation algorithms (Newton-Raphson) with the best case scenario of the other three (Hanoi), we have that the former is about 26% less susceptible to SDC errors than the latter. With the best case scenario for the successive approximation (QSolver) compared with the worst case from the other three (Vector Sum), we find that the former may be up to 96% less susceptible to SDC errors than the later. That means successive approximation algorithms may be from 26% up to 96% more reliable from SDC errors than ordinary calculation algorithms when executing bare metal. On the other hand, those algorithms are more susceptible to HANG errors, according to Table 7.3. Executing bare metal applications have a higher percentage of unace, which means they generated much fewer errors than Linux or FreeRTOS.

On the FreeRTOS cases, comparing the worst case scenario for the successive approximation algorithms (Trapezoid) with the best case scenario of the other three (Vector Sum), we have that the former are about 28% less susceptible to SDC errors than the latter. With the best case scenario for the successive approximation (Newton-Raphson) compared with the worst case from the other three (Hanoi), we find that the former may be up to 78% less susceptible to SDC errors than the later. With that data, it is observable that, on those tests, successive approximation algorithms are from 28% to 78% less susceptible to SDC errors than ordinary calculation algorithms when executing on top of FreeRTOS. FreeRTOS applications are much more susceptible to hangs than their Linux and bare metal counterparts. An application's distribution of errors differs when executing bare metal or on top of an operating system.

When executing on Linux, successive approximation algorithms did not have better fault tolerance than the ordinary computing applications but maintained a better susceptibility to SDC errors on average. It is clear, as already seen at Chapter 5, that the usage of an operating system drastically changes the fault tolerance. That is because the OS itself is a target for faults that may cause errors. In those simulations, inject one fault is injected per execution. Therefore, a big application will have a higher probability of having a fault injected during its execution, while for a small application, the chances are that the fault will be injected during the execution of some OS function. This is, in fact, in pace with the real case scenario, where a short execution time means a lesser probability of having errors, as shown by (REIS et al., 2005). The usage of successive approximation algorithms has its natural fault tolerance masked because of the OS criticality. Linux executions have to deal with segmentation fault errors, which are not present on FreeRTOS and bare metal. Nevertheless, those exceptions represent errors that were caught by the operating system. In the case of the absence of an operating system, those errors could manifest themselves as other types of errors.

7.2 Hardware ATMR

The hardware ATMR presented in this work uses data precision approximation (Section 6.1.1) and is implemented in the FPGA of the programmable logic layer of the Zynq-7000 APSoC. The methodology used to analyze this proposal is the onboard fault injection emulation, presented at Section 4.1. Physical experimental tests, such as radiation and laser fault injection, would also be proper evaluation methods. The DUT for the fault injection campaigns is the matrix multiplication of size 2×2. The ATMR designs presented at Section 6.1.1 are applied to the matrix multiplication designs. The benchmarks are assessed regarding their area consumption and inaccuracy introduced by approximate computing. Their implications are also discussed.

As presented at Section 6.2.1, six ATMR designs are implemented, varying the data precision of the operations. For simplification, the designs are named following the data precision of each redundancy module to simplify the analysis and data presentation (e.g., the ATMR design called "32-24-16" is composed of a module with 32-bit, one with a 24-bit and another with 16-bit precision data and operations).

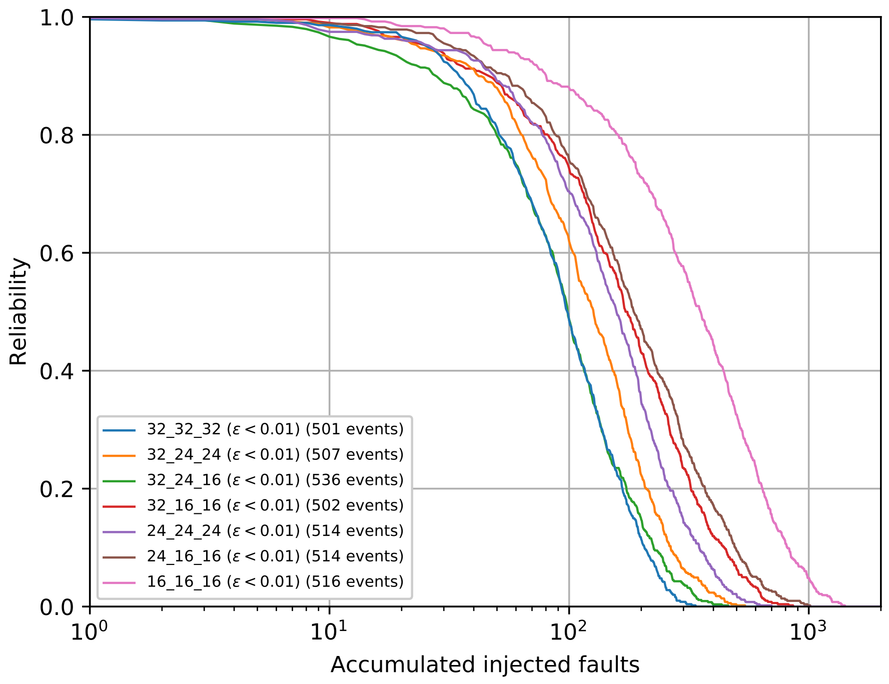
Typically, when checking for errors caused by a fault, the output of the system would be compared with the one from an execution with no faults (called golden execution), and if a difference is found between those two values, we say an SDC error occurred. However, as discussed before, regarding the checker of ATMR methods, the error analysis methodology cannot be the same when dealing with approximate computing. As shown by Figure 6.3, the approximation by data size reduction implies an inevitable accuracy loss. Because of that, there will always be a difference between the output of the full-accuracy golden execution and the approximated versions. In this study case, this difference will always be of at least 0.000303% (the lowest inaccuracy presented in Figure 6.3).

The SDC error occurrence analysis must, therefore, take into account some acceptable differences between the fault-injected system output and the golden output. This acceptable difference between the two values is henceforth called *acceptance threshold* (ε). Section 7.2.1 presents the results for the random accumulated fault injections for two different acceptance thresholds: $\varepsilon = 0.01$ and $\varepsilon = 1$. Section 7.2.2 presents the exhaustive fault injection results for a $\varepsilon = 0.01$, i.e., an SDC is considered only if the difference between the system output and the golden value is equal or higher than 0.01. The value of the acceptance threshold impacts the number of SDCs found by the analysis.

7.2.1 Random Accumulated Fault Injection

Figure 7.14 presents the results for the randomly injected accumulated faults on all the ATMR configurations for an $\varepsilon = 0.01$. The graph presents in the $y-$axis the reliability of the system and, in the $x-$axis, the number of faults accumulated on that point. The reliability is defined as the inverse of the occurrence of errors at a given number

Figure 7.14: Reliability for each ATMR configuration for an acceptance threshold of 0.01.



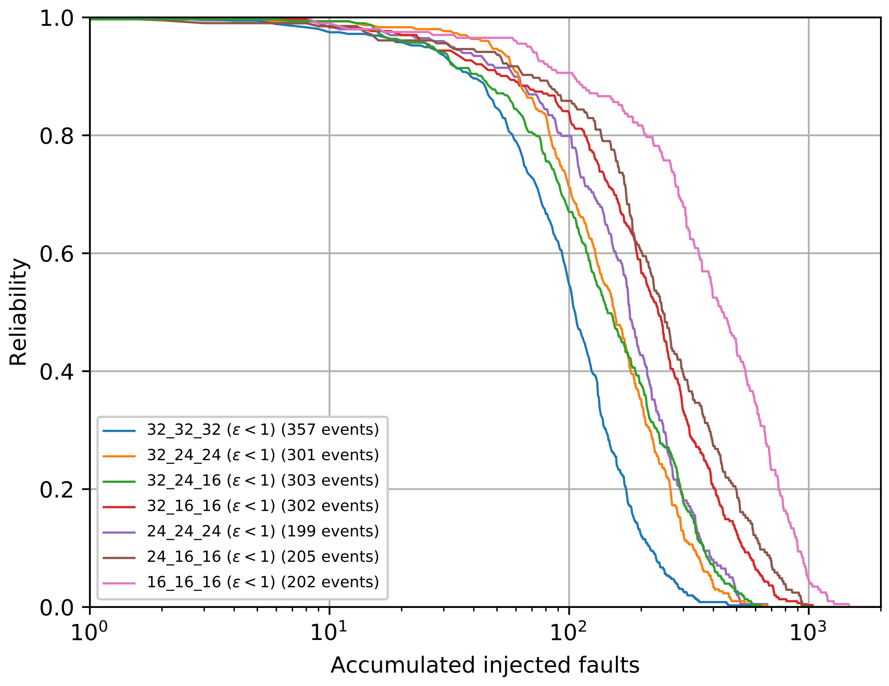
Source: Author.

of accumulated injected faults (e.g., if the reliability at the point is of 0.9 it means that 10% of the observed errors occurred with that number of accumulated injected faults or less). As expected, the ATMR configuration with three redundancies with 16-bit data is the one more reliable. It is clear that its curve is well detached from the other ones. Another expected result is the lower reliability of the full precision ATMR configuration (32-32-32) due to its larger area. However, the 32-32-32 curve is very similar to the 32-24-16 curve.

Figure 7.15 presents the results for the randomly injected accumulated faults on all the ATMR configurations for an $\varepsilon = 1$. That is a very high acceptance threshold, that would only be acceptable on real case scenarios where accuracy is not a strong concern. The ATMR configuration with the highest reliability is again the one with three redundancies with 16-bit data. It is evident the difference between the two extremes of data precision. Nevertheless, the middle-term configurations seem to have similar reliabilities. It is also evident by comparing Figures 7.14 and 7.15 that the behavior of the reliability curve is the same. However, the number of errors (number of events, on each figure legend) has dropped considerably.

The 32-32-32 ATMR configuration arises as to the worst one in terms of reliability

Figure 7.15: Reliability for each ATMR configuration for an acceptance threshold of 1.



Source: Author.

for $\varepsilon = 1$, distancing itself from the other curves. The fact that the 32-24-16 configuration is no more as bad as the 32-32-32 one indicates that this ATMR implementation is terrible when dealing with low-ε errors (Figure 7.14), but is able to handle higher ones (Figure 7.15). This is because this variable of the benchmark has to deal with the low precision of the 16-bit variables and the higher area of the 32- and 24-bit ones. Because the 32-24-16 configuration does not have two redundancies with the same precision, the 16-bit redundancy has a negative effect on accuracy without significant improvement on the fault tolerance with the area reduction.

7.2.2 Exhaustive Fault Injection

Table 7.4 presents the results from the exhaustive fault injections. Because of how the fault injection works, not all the injected faults affect the real DUT area. The fault injector affects a particular “rectangular” area of the FPGA layer, and because of the nature of the FPGA programming, not all of that area will contain the DUT. Therefore, the table presents the number of essential bits (which are the ones used by the design) and critical bits (the ones that caused errors when flipped) of the DUT. The last column of the

Table 7.4: Exhaustive onboard fault injection emulation results for a 2×2 matrix multiplication.

TMR Design	Essential Bits	Critical Bits	Critical Bits Variation (†)
32-32-32	540454	7126	0%
32-24-24	355164	3296	-53.47%
32-24-16	299456	4016	-43.64%
32-16-16	228122	4178	-41.36%
24-24-24	305093	6343	-10.98%
24-16-16	165172	3724	-47.74%
16-16-16	88253	1764	-75.24%

(†) In relation to the 32-32-32 TMR design.

table presents the variation of the number of critical bits in relation to the 32-32-32 TMR configuration.

As expected due to the previous observations, the 16-16-16 ATMR design is the one with the lowest number of critical bits. That is reflected in its high reliability concerning the other configurations. It is interesting to notice, however, that this ATMR configuration has a high percentage of critical bits in relation to essential bits. It indicates that a design of a smaller area tends to be more reliable, even if a higher percentage of this design is critical. This idea is also backed by the fact that the 32-32-32 and 32-24-16 ATMR configurations are the ones with the worst reliability (as presented at Section 7.2.1) and also a high number of critical bits.

The 24-24-24 ATMR configuration is the one with the second highest number of critical bits (being the 32-32-32 the one with the highest). Given this fact, it could be expected that it would also be the one with the second worst reliability. That, however, is not the case. Both Figure 7.14 and Figure 7.15 show that the 24-24-24 configuration is actually between the worse and the best ones, which proves that the precision and accuracy of the design also play a significant role in the system reliability.

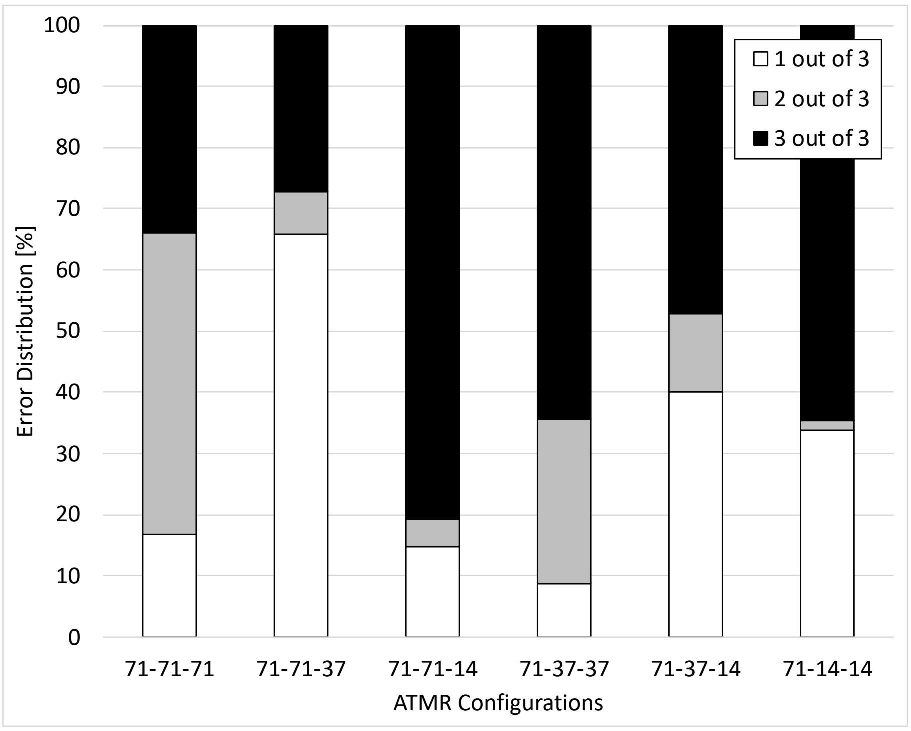
7.3 Software ATMR

As detailed in Section 6.2.2, successive approximation algorithms always produce an approximated output value. As discussed before, on approximate computing, a small deviation from the golden value is reasonable and thus accepted. Therefore, similar to the presentation of the results from Section 7.2, the number of errors and masked errors will always be presented concerning a given acceptable threshold of difference between the ATMR task values (and the golden value). For example, a threshold of 2% means that the error is less than 2% different from the golden value. This is not exactly the same threshold system presented at Section 7.2: this one is based in a percentage of difference, not on absolute values. Results are presented for three different thresholds. The ATMR configurations are the same ones already presented at Section 6.2.2 and are evaluated concerning their fault coverage and execution overhead. However, they are now tested combining two approximation methods: loop-perforation and data precision reduction. Each of the ATMR configurations presents two versions, one implemented with single-precision and the other using double-precision variables. This ATMR technique was tested under laser fault injection on the L1 data cache memory, following the methodology described at Section 4.4.

Figures 7.16, 7.17 and 7.18 present the “Error Distribution" of the ATMR tasks (i.e., the number of ATMR tasks with errors) applied to the single-precision version of the Newton-Raphson algorithm. They respectively present data for $\approx 0\%$, 2% and 5% difference thresholds between the outputs of the tasks and the golden value. The $\approx 0\%$ data presented actually stands for a difference of 0.000013%, which is the difference between the values from the 71- and the 14-iterations executions (without errors). It is written as $\approx 0\%$ for simplification, and because it is the maximum difference that will always be present due to the usage of approximation in this application. Data is presented in percentage and calculated concerning the number of the benchmark executions that had any difference between the task output and the expected golden value. For example, at Figure 7.16 the white bar on the graphs (called “1 of 3") presents the percentage of the executions with errors that contained an error in one of the three ATMR tasks, considering a $\approx 0\%$ difference threshold between the outputs of the tasks and the golden value. To gather this data, the output of each task is compared to the golden value and checked for errors.

Figure 7.16 shows that a considerable amount of errors are not corrected by the ATMR (because most cases presented two or more tasks with errors). This result is expected because of the natural variation of approximate computing algorithms outputs. When using single-precision, the 71-71-14 ATMR is the one with the highest percentage of errors affecting three out of three tasks. Two factors can explain it. First, the 14-iterations task is the one most susceptible to faults. Secondly, the 71-iterations task is the one with higher execution time. A higher execution time means more exposition

Figure 7.16: Number of ATMR tasks with errors for a $\approx 0\%$ difference threshold between the tasks outputs and golden value, on the single-precision version of the Newton-Raphson algorithm.



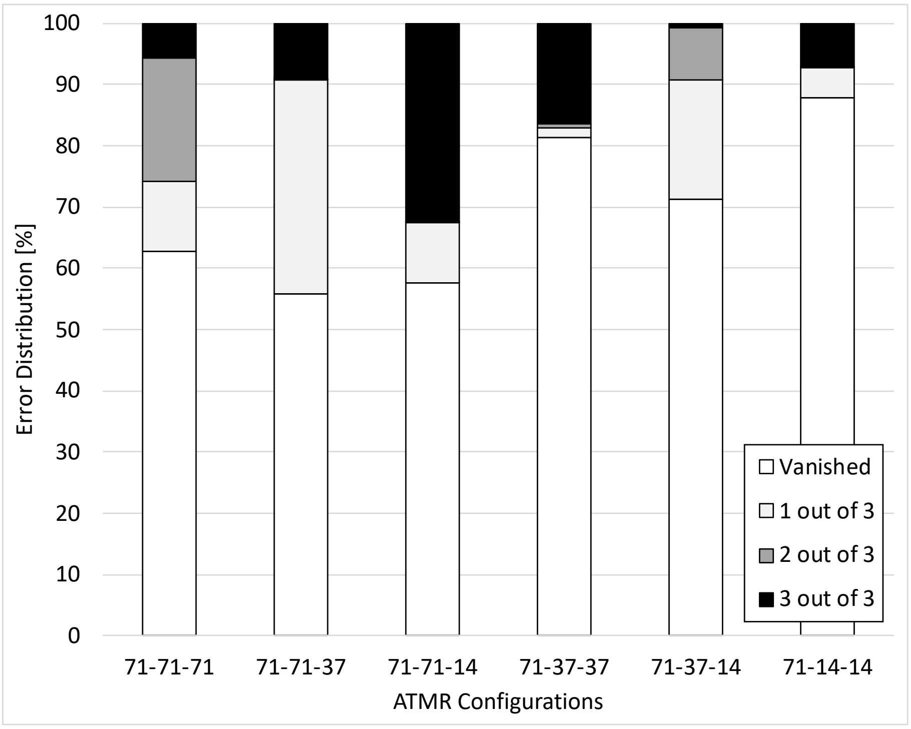
Source: Author.

to faults (because the laser pulse frequency is constant for all benchmarks). Those two factors contribute to a very inefficient ATMR configuration.

At Figures 7.17 and 7.18, the “Vanished" bars represent the amount of errors that are no more present when the difference threshold increased (respectively from $\approx 0\%$ to 2% and from $\approx 0\%$ to 5%). Figure 7.17 shows that increasing the acceptable difference threshold between the outputs and the golden value not only masks some errors but also decreases the number of erroneous tasks. This same behavior is also observed in Figure 7.18, where the difference threshold increased to 5%. Comparing the data from Figures 7.17 and 7.18 it becomes evident that the amount of vanished errors cease to increase at a certain point. It indicates that there may be an optimal difference threshold point, capable of providing good fault tolerance while not compromising too much the output accuracy.

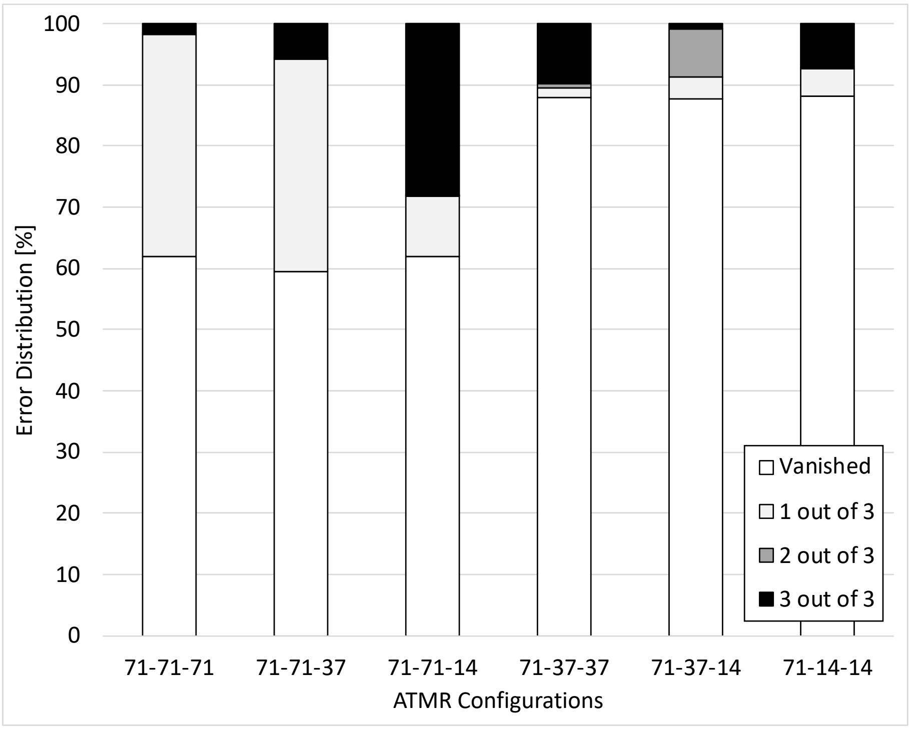
Figures 7.19, 7.20 and 7.21 present the error distribution of the ATMR tasks applied to the double-precision version of the Newton-Raphson algorithm, respectively presenting data for $\approx 0\%$, 2% and 5% difference thresholds between the outputs of the tasks and the golden value. Once again, the “Vanished" bars at Figures 7.20 and 7.21 present the amount of errors that are no more present when the difference threshold increased. Comparing Figures 7.19 and 7.16, using double-precision variables makes a $\approx 0\%$ difference

Figure 7.17: Number of ATMR tasks with errors for a 2% difference threshold between the tasks outputs and golden value, on the single-precision version of the Newton-Raphson algorithm.



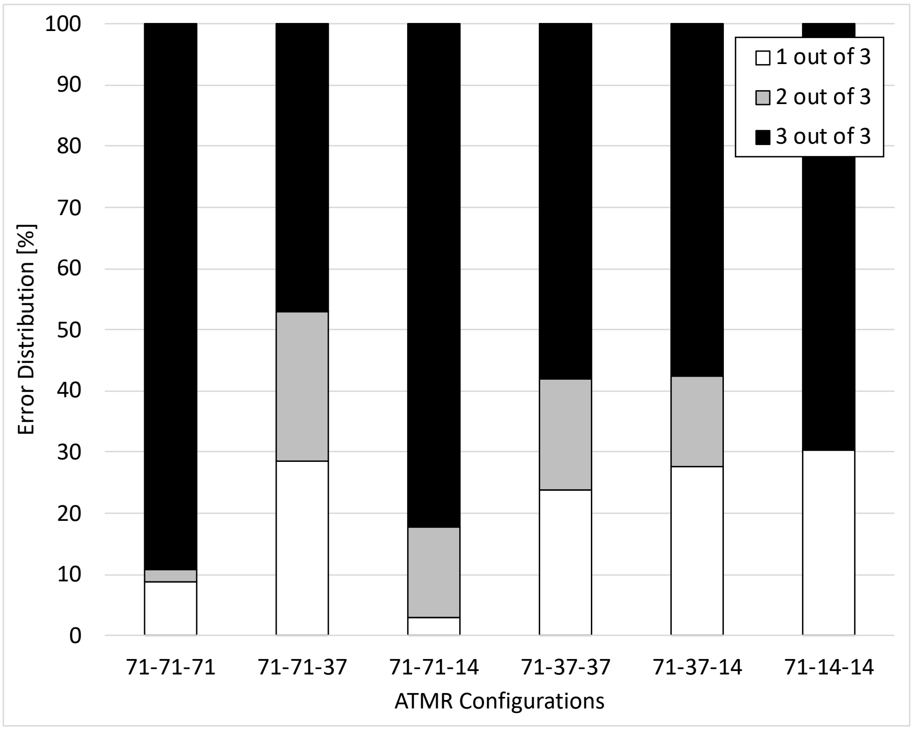
Source: Author.

Figure 7.18: Number of ATMR tasks with errors for a 5% difference threshold between the tasks outputs and golden value, on the single-precision version of the Newton-Raphson algorithm.



Source: Author.

Figure 7.19: Number of ATMR tasks with errors for a $\approx 0\%$ difference threshold between the tasks outputs and golden value, on the double-precision version of the Newton-Raphson algorithm.

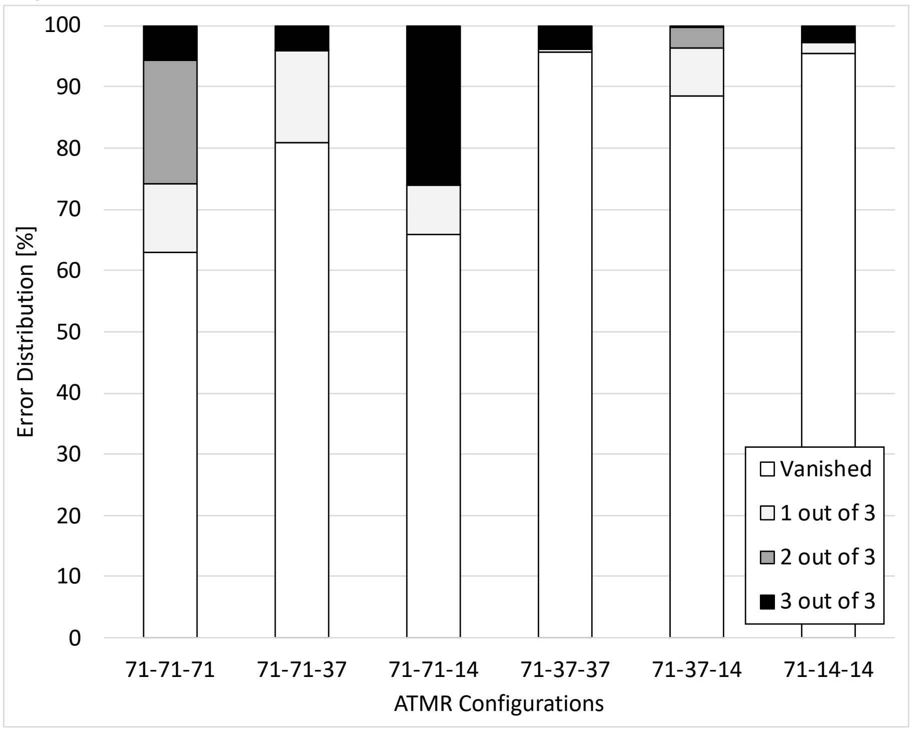


Source: Author.

threshold ATMR even less appropriate. The number of executions with errors affecting two and three tasks is more relevant in that case. However, increasing the difference threshold between the outputs and the golden value highly increases the fault masking capability of the technique. Figure 7.20 shows that a 2% threshold is enough to provide a good fault masking. Figure 7.21 shows that increasing the threshold to 5% does not improve the fault masking performance very much in comparison with a 2% threshold.

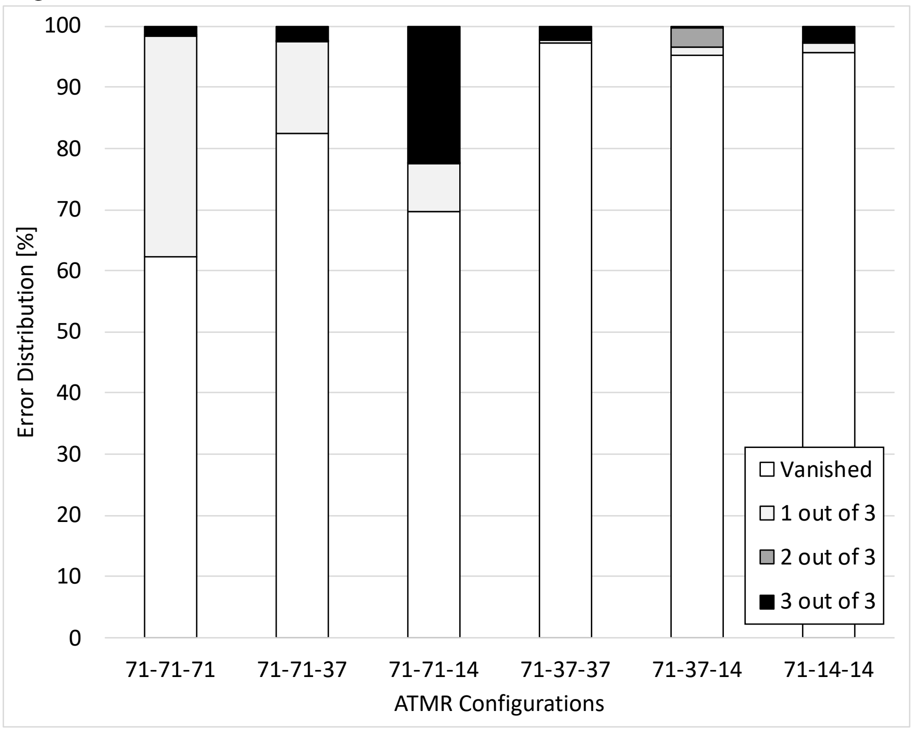
Table 7.5 presents the percentage of masked errors for three thresholds of difference between the ATMR voted values and the golden value. Differently from the data shown at Figures 7.16 to 7.21, this now concerns the value voted by the ATMR, not the outputs from the tasks. As discussed before, the more iterations successive approximation algorithms have, the more fault-tolerant we expect it to be. However, some unexpected results are present. Such is the case of the 71-14-14 ATMR configuration, due to its high performance both for the single and double-precision implementations. Because more iterations usually mean more fault tolerance, this is non-intuitive. Nevertheless, it can be explained by the execution time of this benchmark. It is the one with the lowest overhead (Table 6.4), being subject to fewer fault injections than the others. Literature shows that a high execution time implies in low fault tolerance, once the system is exposed to more

Figure 7.20: Number of ATMR tasks with errors for a 2% difference threshold between the tasks outputs and golden value, on the double-precision version of the Newton-Raphson algorithm.



Source: Author.

Figure 7.21: Number of ATMR tasks with errors for a 5% difference threshold between the tasks outputs and golden value, on the double-precision version of the Newton-Raphson algorithm.



Source: Author.

Table 7.5: Error masking for each ATMR configuration variating thresholds.

ATMR Config.	Single-Precision			Double-Precision		
	≈0% Thres.	2% Thres.	5% Thres.	≈0% Thres.	2% Thres.	5% Thres.
71-71-71	17.37	76.02	97.90	9.30	87.30	87.35
71-71-37	66.80	91.06	94.72	34.68	88.36	88.36
71-71-14	14.84	66.06	66.06	4.29	98.73	98.73
71-37-37	8.82	82.56	89.19	30.09	92.19	94.62
71-37-14	40.03	90.83	91.31	27.67	80.45	80.45
71-14-14	33.82	94.97	94.97	31.88	99.96	99.98

faults, particularly on radioactive environments (REIS et al., 2005) (QUINN, 2014).

Table 7.5 shows that by increasing the threshold, the ATMR was capable of masking many more faults. Even a small difference threshold of 2% is enough to make some configurations mask more than 90% of the errors. The ATMR configuration capable of masking most errors with single-precision with a high threshold is the 71-71-71. However, this configuration performs very poorly for a small threshold. This is probably due to the fact that this is the configuration with the highest execution time and therefore is subject to more faults per execution. In this case, increasing the number of iterations would, instead of improving the fault tolerance (by making the output converge), make it worse (because of the high execution time). The 71-14-14 configuration is the best one at double-precision, and it reaches a good error masking even for a 2% difference threshold. The double-precision implementations have worse performance than the single-precision ones for the $\approx 0\%$ acceptable difference threshold. Nevertheless, increasing the threshold increases the error masking faster than it did on the single-precision cases.

7.4 PAED

The proposed PAED technique is evaluated under laser fault injections, following the same laser configuration detailed on the other experiments of this work. The experiments consisted of injecting faults both at the OCM and L1 data cache memory of the DUT (which is the already detailed dual-core ARM Cortex-A9 processor embedded at the Zynq-7000 SoC).

The benchmarks used at the experiments to evaluate this proposal are the Trapezoid and Newton-Raphson numeric methods that were already presented and discussed

Table 7.6: Details of the benchmarks used to evaluate PAED.

Application		Approximation Method	Exec Time [c.c]	Processed Data [kB]	Approx. Checker Threshold
Cubic	Std.	-	472850	152.625	-
	Apx.	Data Precision	439646	76.312	0.00001
Matrix Multiplication	Std.	-	1253588	56.250	-
	Apx.	Data Precision	1206104	28.125	0.0006498
FFT	Std.	-	1629622	16.000	-
	Apx.	Data Precision	1650040	8.000	0.0065
Trapezoid	Std.	-	4521028	1.028	-
	Apx.	Loop-perforation	3519542	1.028	0.0248
Newton-Raphson	Std.	-	4145720	0.250	-
	Apx.	Loop-perforation	4123290	0.250	0.0248

in Section 7.1.2, and two new ones: cubic and fast Fourier transform (FFT). The cubic benchmark comes from the automotive package of MiBench (GUTHAUS et al., 2001). It consists of calculations of cubic equations solutions, integer square roots, and angle conversions. The code makes use of many trigonometrical functions such as sin and cosine calculations, which are by themselves already approximate. The FFT also comes from the MiBench benchmark suite, and consists of a traditional calculation of an FFT, taking as input vectors representing real and imaginary values. The outputs are stored in the same manner.

Table 7.6 presents the details of the benchmark applications used in this work. The data presented at the table concerns a single execution of the application as a task of the system, thus the low execution time. As discussed in Chapters 2 and 6, there will always exist a small difference between the results of the approximate redundancies and their non-approximate counterparts. The last column of Table 7.6 presents the threshold of the approximate checker implemented at the PAED technique in absolute values. This threshold is intended to accept the highest difference between task$_0$ and task$_1$ (remember Figure 6.5) outputs, caused by the usage of approximation, in the absence of faults. The two approximate computing methods used are loop-perforation and data precision reduction. In the data precision reduction the standard version of the algorithms uses 64-bit floating-point data, while the approximate version uses 32-bit data.

Results for the approximate error detection technique are presented concerning the error detection rate of the technique applied to the benchmarks evaluated. For each benchmark, the technique always executes the standard version at CPU0. The redundancy executed at the CPU1 is either the standard (traditional DWC) or the approximated one (PAED). The y-axis of the figures presented in the following subsections indicates the version of the algorithm being executed by CPU1. Data is presented and analyzed categorizing the experiment results into three types:

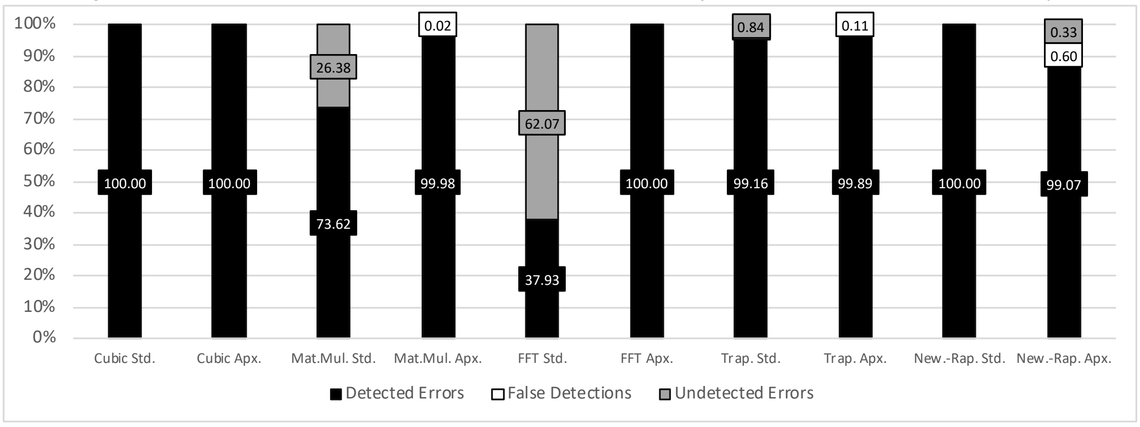
- **Detected Errors:** calculated comparing the total number of erroneous CPU0 calculation outputs (i.e., the value is different from the golden value) and how many of those were found by the error detection technique;
- **False Detection:** presents the percentage of errors detected by the technique that are actually not errors (i.e., the technique issues a warning to the system, but the output from the $task_0$ at CPU0 is actually correct);
- **Undetected Errors:** when there is an error at the $task_0$ output from CPU0, but the technique did not issue a warning to the system, we say we have an undetected error.

Those categories are presented in percentage of total occurrences, being the total occurrences count the sum of events of the three types.

Figure 7.22 presents the results for the error detection rates obtained by the proposed technique of laser fault injections at the OCM. For cubic, trapezoid, and newton-raphson benchmarks, the usage of approximate redundancies had little impact on the error detection when compared to the non-approximate redundancy. In those cases, both PAED and the traditional DWC detected almost all errors. This, however, is not the case of matrix multiplication. The results from the matrix multiplication present poor performance for the standard redundancy. The proposed PAED technique, however, showed a much better error detection than the traditional DWC, detecting almost all errors. The same behavior can be observed when comparing the results from the FFT benchmark, on which the usage of approximate redundancy and the PAED technique highly increased the error detection.

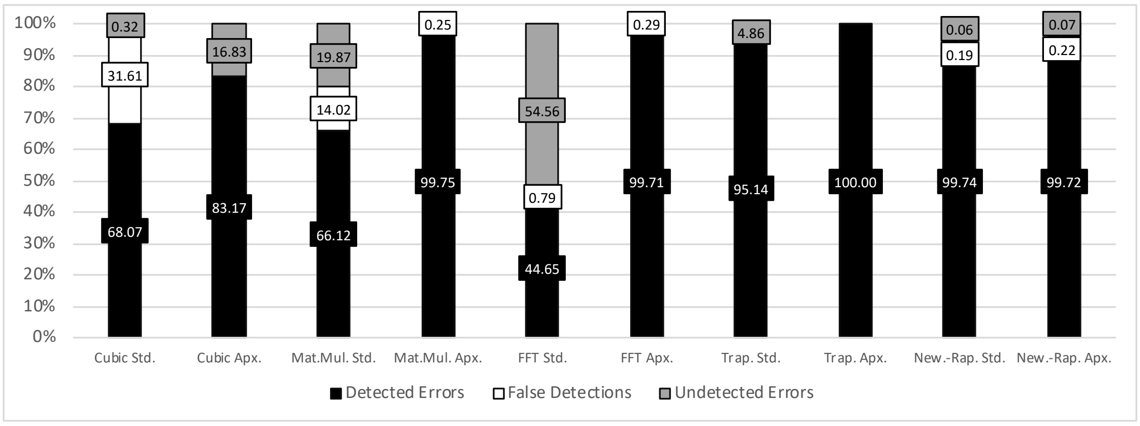
Figure 7.23 presents the error detection rates concerning the laser fault injections at the L1 data cache. In this scenario, the traditional DWC error detection performed poorly for the cubic benchmark. The PAED technique presented a much better error detection but still lower than the ones attained at the other applications. This may be explained by an essential characteristic of the cubic benchmark: it implements approximation by data precision reduction. Because the L1 data cache memory is smaller than the OCM, data size reduction impacts the reliability of the system much less. Matrix multiplication

Figure 7.22: Error detection rates for laser fault injection at the OCM memory.



Source: Author.

Figure 7.23: Error detection rates for laser fault injection at the L1 data cache memory.



Source: Author.

also has data precision reduction as its approximation method. However, it is much more memory-intensive than the cubic benchmark, for it has to constantly read from matrices inputs and write into the output matrix. This line of deduction can also explain why the usage of approximation increased the error detection at the matrix multiplication so much, repeating the observations from Figure 7.22. The L1 data cache memory is a critical area of the system. That is why data precision reduction presents such an important improvement in the reliability of the benchmarks under fault injections in this ROI. This is also proved by the results from the FFT benchmark, where the approximation once again caused a high increase in the error detection rate, and the standard DWC presented terrible results.

It is also important to remember that the OCM memory holds the stack and the heap of the applications, while the L1 data cache holds the more frequently used data. This explains the differences between the results from Figure 7.22 and Figure 7.23. Indeed, the results from 7.23 presented overall a lower error detection. Memory-intensive algorithms such as matrix multiplication are much more prone to errors caused by faults affecting the memory than less intensive ones such as trapezoid. That explains the significant differences between the approximate and standard versions of matrix multiplication and FFT under both laser fault injections. Even though the cubic benchmark uses more memory (Table 7.6), matrix multiplication has more memory accesses and interdependencies, due to the nature of the operation.

8 CONCLUSION

Table 8.1 presents a summary of the results obtained in this work's experiments. Note that it is just a summary, and does not specify the different methodologies used for obtaining the results. This table can be misleading by itself because it only shows the best-case scenarios and because the different experiment methodologies analyze fault tolerance in different ways. It shall be read taking into account all the discussions and details from Chapter 7. For the ATMR on hardware, for instance, the table only presents the results regarding the exhaustive fault injection because the ones from the random accumulated fault injections are too complex to put on a table. One of the points to take into account is, for example, the execution time reduction from PAED. It could be much improved with the evaluation of different approximation intensities. Taylor series error masking only presents data regarding essential bits because no fault injection was performed on it. Some data presented at Chapter 7 had to be left out of the table due to complexity, such as the results from the fault injection on different operating systems and its comparisons.

Successive approximation arises as a promising approach to approximate computing. As the comparison between the Trapezoid and Simpson benchmarks shows, the number of iterations alone is not enough to assure a method will achieve good resilience. The algorithm itself has a significant impact on fault tolerance. Therefore, the study of this kind of approximate computing algorithms is essential before it can be applied to safety-critical systems as reliable software. However, the number of iterations does affect the fault tolerance. Even for a benchmark with a relatively low number of iterations, such as Newton-Raphson, its impact on fault tolerance is noticeable.

Results also show that Taylor series approximation is capable of achieving excellent accuracy with a small number of Taylor series sum terms. The performance and cost of Taylor series approximation depend on the target algorithm accuracy constraints. It proved to be able to provide fast and low-cost approximations for systems with low limitations as well as good approximations (up to 100% accuracy) for those which need it (and can pay for its cost). Hardware implementations appear to be either slower than embedded software or consume too many resources.

The proposed ATMR by successive approximation approach decreases the execution time overhead compared to the classical TMR while keeping an acceptable fault masking rate. Using single and double-precision floating point variable types had impact on the error masking of the method, but the general behavior remained the same. All

Table 8.1: Summary of results.

	Techniques	Cost Reduction			Impact on quality and accuracy	Fault Tolerance	
		HW Area	Exec. Time or Latency	Memory Footprint		Error Detection	Error Masking
Approximation	Data Precision Reduction	-	-	-50%	Loss of 8 decimal digits	-	-
	Successive Approximation	Up to -75% DSP usage	Up to -99.4%	-	User-defined (‡)	-	Up to 87% SDC reduction
	Taylor Series	Up to: -95% DSP -97% FF -97% LUT	Up to -95%	-50% (†)	Up to 71% degradation (in software, very few terms)	-	Up to 97% essential bits reduction
Fault Tolerance	Hardware ATMR	Up to: -75% DSP -70% FF -91% LUT	Up to -40%	-50% (†)	User-defined	-	Up to 75% critical bits reduction
	Software ATMR	-	Up to -58%	-50% (†)	User-defined	-	From 66.8% (≈0% threshold) to 99.9% (5% threshold)
	PAED	-	Up to ≈-30%	-50% (†)	Worst case: second decimal Best case: fifth decimal (‡)	Up to 100% detection	-

(†) When used with data precision reduction.
(‡) Highly dependant on the implementation and system requirements.

the benchmarks showed a trend of having a significant drop in the number o SDC errors for small output variation tolerances. It shows that most of the SDC type errors affecting successive approximation algorithms are not significant. Many applications that use this kind of algorithm may tolerate small variations on the output without a problem. For those applications, successive approximation arises as the perfect method for approximate computing.

The ATMR by data precision reduction proved to be capable of generating implementations with lower area usage while maintaining good accuracy. In the worst case study scenario, the accuracy remained higher than 99.96%. The area reduction provided by this method could be used to provide better performance. A multitude of design strategies could be applied to use the now vacant FPGA area and deliver more parallelism. It is also worth noticing that the proposed ATMR method provided lower latency than the more accurate design option. Results from the fault injection experiments prove that approximating the system impacts its reliability. The smaller, approximated circuits presented higher reliability than the bigger, more accurate ones. The results also indicate that approximate computing improves the system reliability not only by making it smaller but also because of the nature of the approximation.

The results indicate that the proposed parallel approximate error detection (PAED) technique presents a relative better improvement in error detection for memory-intensive tasks than on processor-intensive tasks. This, however, does not imply a limitation of applications for the presented technique. The fact the traditional DWC was capable of a high error detection does not invalidate the PAED technique. On the contrary, PAED was capable of maintaining a proper error detection (even further improving it) while presenting a lower implementation cost on both execution time and memory footprint. The technique is therefore attractive for systems that deal with data freshness, such as real-time systems. In a real case scenario point of view, it means that this type of error detection would be great to alert an aircraft pilot that a particular value is not reliable. Because of the data freshness time window, the pilot would take the best attitude at hand to deal with this problem in the safest manner, until a new refreshed data is generated, this time with a reliable value.

It is nevertheless important to notice that not all systems may be able to make use of approximate computing. Some applications can not afford any inaccuracy, which obviously makes those applications out of the scope of the proposed approach. A further theoretical analysis proved the inherent capability of successive approximation to handle faults before they become errors.

8.1 Future Works

In future works, the author expected to get a bigger picture of how approximate computing impacts system reliability. To achieve that, more fault injection experiments are planned to evaluate the behavior of approximate computing running on top of embedded real-time operating systems, such as FreeRTOS. The works can also be expanded by a deeper study on the reliability of other operating systems that are indeed used by safety-critical systems, such as embedded Linux. Experiments under heavy ions are already taking place and interesting results are emerging. Differently from laser fault injection, the heavy-ions experiments can affect the whole DUT, providing results that make possible the evaluation of the reliability oft he system as a whole, instead of each one of its parts.

One of the problems of approximate computing is that it is often not of easy implementation. Finding the best approximation method for a given algorithm is very consuming work. One of the future work's ideas is the development of a framework that can help software engineers to approximate their codes with minimal efforts. The Taylor series approximation is an example of a method that can be almost universally used for functional approximation.

It is evident by the results presented that combining two or more approximation methods imply a multitude of different effects on system reliability. A designer might then ask himself, which is the optimal configuration between all possible approximation strategies that would achieve the best relation between cost and performance. In future works, evolutionary algorithms could be used to test possible combinations of approximation configurations to find this optimal point between cost, performance, and reliability.

8.2 Publications

The works here presented were published in the following peer-reviewed journals and conferences:

- **G.S. Rodrigues**, Á. B. de Oliveira, F. L. Kastensmidt. *"Analyzing the Use of Taylor Series Approximation in Hardware and Embedded Software for Good Cost-*

Accuracy Tradeoffs". Part of the Lecture Notes in Computer Science book series (LNCS), v. 10824, p. 647-658, 2018.

- **G.S. Rodrigues**, J.S. Fonseca, F.L. Kastensmidt, V. Pouget, A. Bosio, S. Hamdioui. *"Approximate TMR based on successive approximation and loop perforation in microprocessors"*. Microelectronics Reliability, v. 100-101, p. 113385, 2019.
- **G.S. Rodrigues**, Á. B. de Oliveira, F. L. Kastensmidt, V. Pouget, A. Bosio. *"Assessing the Reliability of Successive Approximate Computing Algorithms under Fault Injection"*. Journal of Electronic Testing, v. 35, p. 367–381, 2019.
- **G.S. Rodrigues**, F. Rosa, Á. B. de Oliveira, F. L. Kastensmidt, L. Ost, R. Reis. *"Analyzing the Impact of Fault-Tolerance Methods in ARM Processors Under Soft Errors Running Linux and Parallelization APIs"*. IEEE Transactions on Nuclear Science, v. 64, p. 2196-2203, 2017.
- **G.S. Rodrigues**, A. B. de Oliveira, I. Lopes, V. Pouget, A. Bosio, F.L. Kastensmidt. *"An Approximate Error-Detection Technique for Multi-Core Real-Time Systems"*. 19th European Conference on Radiation and Its Effects on Components and Systems (RADECS), Montpellier, France, 2019.
- **G.S. Rodrigues**, J.S. Fonseca, F. Benevenuti, F. L. Kastensmidt, A. Bosio. *"Exploiting Approximate Computing for Low-Cost Fault Tolerant Architectures"*. 32nd Symposium on Integrated Circuits and Systems Design (SBCCI), Sao Paulo, Brazil 2019.
- **G.S. Rodrigues**, F.L. Kastensmidt, V. Pouget, A. Bosio. *"Approximate TMR Based on Successive Approximation to Protect Against Multiple Bit Upset in Microprocessors"*. 18th European Conference on Radiation and Its Effects on Components and Systems (RADECS), Gothenburg, Sweden, 2018.
- **G.S. Rodrigues**, Á. B.de Oliveira, A. Bosio, F. L. Kastensmidt, E. P. de Freitas. *"ARFT: An Approximative Redundant Technique for Fault Tolerance"*. Conference on Design of Circuits and Integrated Systems (DCIS), Lyon, France, 2018.
- **G.S. Rodrigues**, F. L. Kastensmidt, V. Pouget, A. Bosio. *"Performances VS Reliability: how to exploit Approximate Computing for Safety-Critical applications"*. IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS), Platja d'Aro, Spain, 2018.
- **G.S. Rodrigues**, F. L. Kastensmidt, V. Pouget, A. Bosio. *"Exploring the inherent fault tolerance of successive approximation algorithms under laser fault injection"*.

IEEE 19th Latin-American Test Symposium (LATS), Sao Paulo, Brazil, 2018.

- **G.S. Rodrigues**, F. Rosa, F. L. Kastensmidt, R. Reis, L. Ost. *"Investigating parallel TMR approaches and thread disposability in Linux"*. 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Batumi, Georgia, 2017.
- **G.S. Rodrigues**, F. L. Kastensmidt .*"Evaluating the behavior of successive approximation algorithms under soft errors"*. 18th IEEE Latin American Test Symposium (LATS), Bogota, Colombia, 2017.
- **G.S. Rodrigues**, F. L. Kastensmidt, R. Reis, F. Rosa and L. Ost. *"Analyzing the impact of using pthreads versus OpenMP under fault injection in ARM Cortex-A9 dual-core"*. 16th European Conference on Radiation and Its Effects on Components and Systems (RADECS), Bremen, Germany, 2016.
- **G.S. Rodrigues**, F. L. Kastensmidt. *"Soft error analysis at sequential and parallel applications in ARM Cortex-A9 dual-core"*. 17th Latin-American Test Symposium (LATS), Foz do Iguacu, Brazil, 2016.

The author also worked in collaboration with other researchers in the approximate computing and fault tolerance domains, acting as a co-author in the following publications:

- A. Oliveira, F. Benevenuti, L. Benites, **G.S. Rodrigues**, F. Kastensmidt, N. Added, V. Aguiar, N. Medina, M. Guazzelli, L. Tambara. *"Dynamic heavy ions SEE testing of NanoXplore radiation hardened SRAM-based FPGA: Reliability-performance analysis"*. Microelectronics Reliability, v. 100-101, p. 113437, 2019.
- A. Oliveira, **G.S. Rodrigues**, F. Kastensmidt, N. Added, V. Aguiar, N. Medina, M. Guazzelli. *"Lockstep Dual-Core ARM A9: Implementation and Resilience Analysis Under Heavy Ion-Induced Soft Errors"*. IEEE Transactions on Nuclear Science, v. 65, p. 1783-1790, 2018.
- A. Bosio, I. O'Connor, **G.S. Rodrigues**, F. K. Lima, E. I. Vatajelu, G. Di Natale, L. Anghel, S. Nagarajan, M. C. R. Fieback, S. Hamdioui. *"Rebooting Computing: The Challenges for Test and Reliability"*. IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Noordwijk, Netherlands, 2019.
- V. Fratin, D. Oliveira, C. Lunardi, F. Santos, **G.S. Rodrigues**, P. Rech. *"Code-Dependent and Architecture-Dependent Reliability Behaviors"*. 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Lux-

embourg City, Luxembourg, 2018.

- Á. B. de Oliveira, **G.S. Rodrigues**, F. L. Kastensmidt. *"Analyzing lockstep dual-core ARM cortex-A9 soft error mitigation in FreeRTOS applications"*. 30th Symposium on Integrated Circuits and Systems Design (SBCCI), Fortaleza, Brazil, 2017.

8.2.1 Awards

The work published and presented at the 18th European Conference on Radiation and Its Effects on Components and Systems (RADECS) in Gothenburg, Sweden, entitled "Approximate TMR Based on Successive Approximation to Protect Against Multiple Bit Upset in Microprocessors" won the Jean-Marie Palau award as Best Student Presentation.

REFERENCES

ALBANDES, I. et al. Design of approximate-tmr using approximate library and heuristic approaches. **Microelectronics Reliability**, v. 88-90, p. 898 – 902, 2018. ISSN 0026-2714. 29th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis (ESREF 2018). Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0026271418306875>.

AMANT, R. S. et al. General-purpose code acceleration with limited-precision analog computation. **Proceedings - International Symposium on Computer Architecture**, p. 505–516, 2014. ISSN 10636897.

ANDREOU, C. M. et al. A subthreshold, low-power, rhbd reference circuit, for earth observation and communication satellites. In: **2015 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.: s.n.], 2015. p. 2245–2248.

ARIFEEN, T. et al. Probing approximate tmr in error resilient applications for better design tradeoffs. In: **2016 Euromicro Conference on Digital System Design (DSD)**. [S.l.: s.n.], 2016. p. 637–640.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 1, p. 11–33, Jan 2004. ISSN 2160-9209.

AZAMBUJA, J. R. et al. Heta: Hybrid error-detection technique using assertions. **IEEE Transactions on Nuclear Science**, v. 60, n. 4, p. 2805–2812, Aug 2013. ISSN 0018-9499.

AZIMI, S.; STERPONE, L. Micro latch-up analysis on ultra-nanometer vlsi technologies: A new monte carlo approach. In: **2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2017. p. 338–343.

BARROIS, B.; SENTIEYS, O.; MENARD, D. The hidden cost of functional approximation against careful data sizing — a case study. In: **Design, Automation Test in Europe Conference Exhibition (DATE), 2017**. [S.l.: s.n.], 2017. p. 181–186. ISSN 1558-1101.

BAUMANN, R. C. Soft errors in advanced semiconductor devices-part i: the three radiation sources. **IEEE Transactions on Device and Materials Reliability**, v. 1, n. 1, p. 17–22, March 2001.

BAUMANN, R. C. Radiation-induced soft errors in advanced semiconductor technologies. **IEEE Transactions on Device and Materials Reliability**, v. 5, n. 3, p. 305–316, Sept 2005. ISSN 1530-4388.

BENEDETTO, J. et al. Heavy ion-induced digital single-event transients in deep submicron processes. **IEEE Transactions on Nuclear Science**, v. 51, n. 6, p. 3480–3485, Dec 2004.

BRUGUIER, G.; PALAU, J. M. Single particle-induced latchup. **IEEE Transactions on Nuclear Science**, v. 43, n. 2, p. 522–532, April 1996.

BUCHNER, S. P. et al. Pulsed-laser testing for single-event effects investigations. **IEEE Transactions on Nuclear Science**, v. 60, n. 3, p. 1852–1875, June 2013. ISSN 0018-9499.

CHEN, L.; AVIZIENIS, A. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In: **Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.** [S.l.: s.n.], 1995. p. 113–. ISSN null.

CHEYNET, P. et al. Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. **IEEE Transactions on Nuclear Science**, v. 47, n. 6, p. 2231–2236, Dec 2000. ISSN 1558-1578.

CHIELLE, E. et al. Configurable tool to protect processors against see by software-based detection techniques. In: **2012 13th Latin American Test Workshop (LATW)**. [S.l.: s.n.], 2012. p. 1–6. ISSN 2373-0862.

CHIELLE, E. et al. S-seta: Selective software-only error-detection technique using assertions. **IEEE Transactions on Nuclear Science**, v. 62, n. 6, p. 3088–3095, Dec 2015. ISSN 0018-9499.

CHIPPA, V. K. et al. Scalable effort hardware design. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 22, n. 9, p. 2004–2016, Sep. 2014.

CHO, K. et al. edram-based tiered-reliability memory with applications to low-power frame buffers. In: **2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)**. [S.l.: s.n.], 2014. p. 333–338.

DONG, G.; XIE, N.; ZHANG, T. On the use of soft-decision error-correction codes in nand flash memory. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v. 58, n. 2, p. 429–439, Feb 2011. ISSN 1558-0806.

DUFOUR, C. et al. Heavy ion induced single hard errors on submicronic memories (for space application). **IEEE Transactions on Nuclear Science**, v. 39, n. 6, p. 1693–1697, Dec 1992.

ESMAEILZADEH, H. et al. Architecture support for disciplined approximate programming. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 47, n. 4, p. 301–312, mar. 2012. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/2248487.2151008>.

FANG, Y.; LI, H.; LI, X. Softpcm: Enhancing energy efficiency and lifetime of phase change memory in video applications via approximate write. In: **2012 IEEE 21st Asian Test Symposium**. [S.l.: s.n.], 2012. p. 131–136.

FAYYAZ, M.; VLADIMIROVA, T. Fault-tolerant distributed approach to satellite on-board computer design. In: **2014 IEEE Aerospace Conference**. [S.l.: s.n.], 2014. p. 1–12. ISSN 1095-323X.

FOY, W. H. Position-location solutions by taylor-series estimation. **IEEE Transactions on Aerospace and Electronic Systems**, AES-12, n. 2, p. 187–194, March 1976. ISSN 0018-9251.

FREITAS, E. P. de et al. Deraf: A high-level aspects framework for distributed embedded real-time systems design. In: ____. **Early Aspects: Current Challenges and Future Directions: 10th International Workshop, Vancouver, Canada, March 13, 2007, Revised Selected Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 55–74. ISBN 978-3-540-76811-1. Available from Internet: <http://dx.doi.org/10.1007/978-3-540-76811-1_4>.

GIZOPOULOS, D. et al. Architectures for online error detection and recovery in multicore processors. In: **2011 Design, Automation Test in Europe**. [S.l.: s.n.], 2011. p. 1–6. ISSN 1530-1591.

GOIRI Íñigo et al. Approxhadoop: Bringing approximations to mapreduce frameworks. In: **Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)**. [s.n.], 2015. Available from Internet: <https://www.microsoft.com/en-us/research/publication/approxhadoop-bringing-approximations-to-mapreduce-frameworks/>.

GOLOUBEVA, O. et al. Soft-error detection using control flow assertions. In: **Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems**. [S.l.: s.n.], 2003. p. 581–588. ISSN 1550-5774.

GOMES, I. A. et al. Exploring the use of approximate tmr to mask transient faults in logic with low area overhead. **Microelectronics Reliability**, v. 55, n. 9, p. 2072 – 2076, 2015. ISSN 0026-2714. Proceedings of the 26th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0026271415300676>.

GOMES, I. A. C. et al. Methodology for achieving best trade-off of area and fault masking coverage in atmr. In: **2014 15th Latin American Test Workshop - LATW**. [S.l.: s.n.], 2014. p. 1–6. ISSN 2373-0862.

GUTHAUS, M. R. et al. 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)**. [S.l.: s.n.], 2001. p. 3–14.

HAN, J.; ORSHANSKY, M. Approximate computing: An emerging paradigm for energy-efficient design. In: **2013 18th IEEE European Test Symposium (ETS)**. [S.l.: s.n.], 2013. p. 1–6. ISSN 1530-1877.

HILDERMAN, V.; BAGHI, T. **Avionics certification: a complete guide to DO-178 (software), DO-254 (hardware)**. [S.l.]: Avionics Communications., 2007.

HSIAO, C. C.; CHU, S. L.; CHEN, C. Y. Energy-aware hybrid precision selection framework for mobile GPUs. **Computers and Graphics (Pergamon)**, Elsevier, v. 37, n. 5, p. 431–444, 2013. ISSN 00978493. Available from Internet: <http://dx.doi.org/10.1016/j.cag.2013.03.003>.

HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. **Computer**, IEEE, v. 30, n. 4, p. 75–82, 1997.

HUANG, B. et al. Harnessing unreliable cores in heterogeneous architecture: The pydac programming model and runtime. In: **2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks**. [S.l.: s.n.], 2014. p. 744–749. ISSN 1530-0889.

HUANG, K.-H.; ABRAHAM, J. A. Algorithm-based fault tolerance for matrix operations. **IEEE Transactions on Computers**, C-33, n. 6, p. 518–528, June 1984. ISSN 0018-9340.

IBE, E. et al. Spreading diversity in multi-cell neutron-induced upsets with device scaling. In: **IEEE Custom Integrated Circuits Conference 2006**. [S.l.: s.n.], 2006. p. 437–444.

IMPERAS. **Open Virtual Platforms (OVP)**. 2017. Available from Internet: <http://www.ovpworld.org>.

JOHNSON, G. H. et al. A review of the techniques used for modeling single-event effects in power mosfets. **IEEE Transactions on Nuclear Science**, v. 43, n. 2, p. 546–560, April 1996.

JUN, I. et al. Proton nonionizing energy loss (niel) for device applications. **IEEE Transactions on Nuclear Science**, v. 50, n. 6, p. 1924–1928, Dec 2003.

KADI, M. A. et al. Dynamic and partial reconfiguration of zynq 7000 under linux. In: IEEE. **Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on**. [S.l.], 2013. p. 1–5.

KAHNG, A. B.; KANG, S. Accuracy-configurable adder for approximate arithmetic designs. In: **DAC Design Automation Conference 2012**. [S.l.: s.n.], 2012. p. 820–825.

KERAMIDAS, G.; KOKKALA, C.; STAMOULIS, I. Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders. **1st Workshop On Approximate Computing (WAPCO 2015)**, n. January, p. 6, 2015. Available from Internet: <http://wapco.inf.uth.gr/2015/program.html>.

KOGA, R. et al. Single event functional interrupt (sefi) sensitivity in microcircuits. In: **RADECS 97. Fourth European Conference on Radiation and its Effects on Components and Systems (Cat. No.97TH8294)**. [S.l.: s.n.], 1997. p. 311–318.

KOOPMAN, P.; CHAKRAVARTY, T. Cyclic redundancy code (crc) polynomial selection for embedded networks. In: **International Conference on Dependable Systems and Networks, 2004**. [S.l.: s.n.], 2004. p. 145–154. ISSN null.

KULKARNI, P.; GUPTA, P.; ERCEGOVAC, M. Trading accuracy for power with an underdesigned multiplier architecture. In: **2011 24th Internatioal Conference on VLSI Design**. [S.l.: s.n.], 2011. p. 346–351.

LEVEUGLE, R. et al. Statistical fault injection: Quantified error and confidence. In: **2009 Design, Automation Test in Europe Conference Exhibition**. [S.l.: s.n.], 2009. p. 502–506. ISSN 1558-1101.

MAIZ, J. et al. Characterization of multi-bit soft error events in advanced srams. In: **IEEE International Electron Devices Meeting 2003**. [S.l.: s.n.], 2003. p. 21.4.1–21.4.4.

MESSENGER, S. R. et al. Niel for heavy ions: an analytical approach. **IEEE Transactions on Nuclear Science**, v. 50, n. 6, p. 1919–1923, Dec 2003.

MISAILOVIC, S.; KIM, D.; RINARD, M. Parallelizing sequential programs with statistical accuracy tests. **ACM Trans. Embed. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 12, n. 2s, may 2013. ISSN 1539-9087. Available from Internet: <https://doi.org/10.1145/2465787.2465790>.

MITTAL, S. A survey of architectural techniques for improving cache power efficiency. **Sustainable Computing: Informatics and Systems**, Elsevier, v. 4, n. 1, p. 33–43, 2014.

MOLLER, T. et al. Evaluation and design of filters using a taylor series expansion. **IEEE Transactions on Visualization and Computer Graphics**, v. 3, n. 2, p. 184–199, Apr 1997. ISSN 1077-2626.

MOMENI, A. et al. Design and analysis of approximate compressors for multiplication. **IEEE Transactions on Computers**, v. 64, n. 4, p. 984–994, April 2015. ISSN 0018-9340.

MONSON, J. S.; WIRTHLIN, M.; HUTCHINGS, B. Fault injection results of linux operating on an fpga embedded platform. In: **2010 International Conference on Reconfigurable Computing and FPGAs**. [S.l.: s.n.], 2010. p. 37–42. ISSN 2325-6532.

MOREAU, T. et al. Snnap: Approximate computing on programmable socs via neural acceleration. In: **2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2015. p. 603–614.

MUSHTAQ, H.; AL-ARS, Z.; BERTELS, K. Efficient software-based fault tolerance approach on multicore platforms. In: **2013 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2013. p. 921–926. ISSN 1530-1591.

NAIR, R. Big data needs approximate computing: Technical perspective. **Commun. ACM**, ACM, New York, NY, USA, v. 58, n. 1, p. 104–104, dec. 2014. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/2688072>.

NORMAND, E. Single event upset at ground level. **IEEE Transactions on Nuclear Science**, v. 43, n. 6, p. 2742–2750, Dec 1996.

OH, N.; MITRA, S.; MCCLUSKEY, E. J. Ed4i: error detection by diverse data and duplicated instructions. **IEEE Transactions on Computers**, v. 51, n. 2, p. 180–199, Feb 2002. ISSN 0018-9340.

OLIVEIRA, A. B. de et al. Lockstep dual-core arm a9: Implementation and resilience analysis under heavy ion-induced soft errors. **IEEE Transactions on Nuclear Science**, v. 65, n. 8, p. 1783–1790, Aug 2018. ISSN 1558-1578.

OLIVEIRA, Á. B. de; TAMBARA, L. A.; KASTENSMIDT, F. L. Exploring performance overhead versus soft error detection in lockstep dual-core arm cortex-a9 processor embedded into xilinx zynq apsoc. In: ____. **Applied Reconfigurable Computing: 13th International Symposium, ARC 2017, Delft, The Netherlands, April 3-7, 2017, Proceedings**. Cham: Springer International Publishing, 2017. p. 189–201. ISBN 978-3-319-56258-2. Available from Internet: <http://dx.doi.org/10.1007/978-3-319-56258-2_17>.

OSINSKI, L.; LANGER, T.; MOTTOK, J. A survey of fault tolerance approaches on different architecture levels. In: **ARCS 2017; 30th International Conference on Architecture of Computing Systems**. [S.l.: s.n.], 2017. p. 1–9.

PANDEY, J. G. et al. An fpga-based fixed-point architecture for binary logarithmic computation. In: **2013 IEEE Second International Conference on Image Information Processing (ICIIP-2013)**. [S.l.: s.n.], 2013. p. 383–388.

PIGNOL, M. Cots-based applications in space avionics. In: **2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)**. [S.l.: s.n.], 2010. p. 1213–1219. ISSN 1530-1591.

POIVEY, C. et al. Implications of advanced microelectronics technologies for heavy ion single event effect (see) testing. In: **RADECS 2001. 2001 6th European Conference on Radiation and Its Effects on Components and Systems (Cat. No.01TH8605)**. [S.l.: s.n.], 2001. p. 328–331.

POIVEY, C. et al. In-flight observations of long-term single-event effect (see) performance on orbview-2 solid state recorders (ssr). In: **2003 IEEE Radiation Effects Data Workshop**. [S.l.: s.n.], 2003. p. 102–107.

POIVEY, C. et al. Characterization of single hard errors (she) in 1 m-bit srams from single ion. **IEEE Transactions on Nuclear Science**, v. 41, n. 6, p. 2235–2239, Dec 1994.

POUGET, V. et al. Dynamic testing of an sram-based fpga by time-resolved laser fault injection. In: **2008 14th IEEE International On-Line Testing Symposium**. [S.l.: s.n.], 2008. p. 295–301. ISSN 1942-9398.

POUGET, V. et al. Structural pattern extraction from asynchronous two-photon laser fault injection using spectral analysis. **Microelectronics Reliability**, v. 76-77, n. Supplement C, p. 650 – 654, 2017. ISSN 0026-2714. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0026271417303128>.

QUINN, H. Challenges in testing complex systems. **IEEE Transactions on Nuclear Science**, v. 61, n. 2, p. 766–786, April 2014. ISSN 0018-9499.

QUINN, H. et al. Software resilience and the effectiveness of software mitigation in microcontrollers. **IEEE Transactions on Nuclear Science**, v. 62, n. 6, p. 2532–2538, Dec 2015. ISSN 0018-9499.

QUINN, H. et al. Robust duplication with comparison methods in microcontrollers. **IEEE Transactions on Nuclear Science**, v. 64, n. 1, p. 338–345, Jan 2017. ISSN 0018-9499.

RAHIMI, A. et al. Approximate associative memristive memory for energy-efficient gpus. In: **2015 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2015. p. 1497–1502.

RANJAN, A. et al. Approximate storage for energy efficient spintronic memories. In: **2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2015. p. 1–6.

REIS, G. A. et al. Design and evaluation of hybrid fault-detection systems. In: **32nd International Symposium on Computer Architecture (ISCA'05)**. [S.l.: s.n.], 2005. p. 148–159. ISSN 1063-6897.

REN, Y.; ZHANG, B.; QIAO, H. A simple taylor-series expansion method for a class of second kind integral equations. **Journal of Computational and Applied Mathematics**, v. 110, n. 1, p. 15 – 24, 1999. ISSN 0377-0427. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0377042799001922>.

ROSA, F. et al. 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)**. [S.l.: s.n.], 2015. p. 211–214. ISSN 1550-5774.

ROY, P. et al. Asac: Automatic sensitivity analysis for approximate computing. In: **Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems**. New York, NY, USA: ACM, 2014. (LCTES '14), p. 95–104. ISBN 978-1-4503-2877-7. Available from Internet: <http://doi.acm.org/10.1145/2597809.2597812>.

RUBIO-GONZALEZ, C. et al. Precimonious: Tuning assistant for floating-point precision. In: **Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: ACM, 2013. (SC '13), p. 27:1–27:12. ISBN 978-1-4503-2378-9. Available from Internet: <http://doi.acm.org/10.1145/2503210.2503296>.

SAHA, G. K. Software based fault tolerance: A survey. **Ubiquity**, ACM, New York, NY, USA, v. 2006, n. July, p. 1:1–1:1, jul. 2006. ISSN 1530-2180. Available from Internet: <http://doi.acm.org/10.1145/1147994.1147995>.

SAMPAIO, F. et al. Approximation-aware multi-level cells stt-ram cache architecture. In: **2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)**. [S.l.: s.n.], 2015. p. 79–88.

SANCHEZ-CLEMENTE, A. J.; ENTRENA, L.; GARCIA-VALDERAS, M. Partial tmr in fpgas using approximate logic circuits. **IEEE Transactions on Nuclear Science**, v. 63, n. 4, p. 2233–2240, Aug 2016. ISSN 0018-9499.

SANTINI, T. et al. Reliability analysis of operating systems and software stack for embedded systems. **IEEE Transactions on Nuclear Science**, v. 63, n. 4, p. 2225–2232, Aug 2016. ISSN 1558-1578.

SHAFIQUE, M. et al. A low latency generic accuracy configurable adder. In: **Proceedings of the 52Nd Annual Design Automation Conference**. New York, NY, USA: ACM, 2015. (DAC '15), p. 86:1–86:6. ISBN 978-1-4503-3520-1. Available from Internet: <http://doi.acm.org/10.1145/2744769.2744778>.

SIDIROGLOU-DOUSKOS, S. et al. Managing performance vs. accuracy trade-offs with loop perforation. In: **Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering**. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 124–134. ISBN 978-1-4503-0443-6. Available from Internet: <http://doi.acm.org/10.1145/2025113.2025133>.

STERPONE, L.; VIOLANTE, M. An analysis of seu effects in embedded operating systems for real-time applications. In: IEEE. **Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on**. [S.l.], 2007. p. 3345–3349.

SUEUR, E. L.; HEISER, G. Dynamic voltage and frequency scaling: The laws of diminishing returns. In: **Proceedings of the 2010 International Conference on Power Aware Computing and Systems**. Berkeley, CA, USA: USENIX Association, 2010. (HotPower'10), p. 1–8. Available from Internet: <http://dl.acm.org/citation.cfm?id=1924920.1924921>.

SUTHERLAND, M.; MIGUEL, J. S.; JERGER, N. E. Texture Cache Approximation on GPUs. **Workshop on Approximate Computing**, p. 1–3, 2015.

TAMBARA, L. A. et al. Analyzing the impact of radiation-induced failures in programmable socs. **IEEE Transactions on Nuclear Science**, IEEE, v. 63, n. 4, p. 2217–2224, 2016.

TAUSCH, J. et al. Neutron induced micro sel events in cots sram devices. In: **2007 IEEE Radiation Effects Data Workshop**. [S.l.: s.n.], 2007. p. 185–188.

TIAN, Y. et al. Approxma: Approximate memory access for dynamic precision scaling. In: **Proceedings of the 25th Edition on Great Lakes Symposium on VLSI**. New York, NY, USA: ACM, 2015. (GLSVLSI '15), p. 337–342. ISBN 978-1-4503-3474-7. Available from Internet: <http://doi.acm.org/10.1145/2742060.2743759>.

TONFAT, J. et al. Method to analyze the susceptibility of hls designs in sram-based fpgas under soft errors. In: BONATO, V.; BOUGANIS, C.; GORGON, M. (Ed.). **Applied Reconfigurable Computing**. Cham: Springer International Publishing, 2016. p. 132–143. ISBN 978-3-319-30481-6.

TRICHINA, E.; KORKIKYAN, R. Multi fault laser attacks on protected crt-rsa. In: **2010 Workshop on Fault Diagnosis and Tolerance in Cryptography**. [S.l.: s.n.], 2010. p. 75–86.

TYLKA, A. J. et al. Single event upsets caused by solar energetic heavy ions. **IEEE Transactions on Nuclear Science**, IEEE, v. 43, n. 6, p. 2758–2766, 1996.

VASSILIADIS, V. et al. A programming model and runtime system for significance-aware energy-efficient computing. **Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP**, v. 2015-January, p. 275–276, 2015. ISSN 03621340.

VENKATARAMANI, S. et al. Approximate computing and the quest for computing efficiency. In: **Proceedings of the 52Nd Annual Design Automation Conference**. New York, NY, USA: ACM, 2015. (DAC '15), p. 120:1–120:6. ISBN 978-1-4503-3520-1. Available from Internet: <http://doi.acm.org/10.1145/2744769.2751163>.

WIRTHLIN, M. High-reliability fpga-based systems: Space, high-energy physics, and beyond. **Proceedings of the IEEE**, v. 103, n. 3, p. 379–389, March 2015. ISSN 0018-9219.

XILINX. **Zynq-7000 All Programmable SoC Overview**. 2017. Available from Internet: <http://xilinx.com/>.

XU, Q.; MYTKOWICZ, T.; KIM, N. S. Approximate computing: A survey. **IEEE Design Test**, v. 33, n. 1, p. 8–22, Feb 2016. ISSN 2168-2356.

ZHANG, Q. et al. Approxann: An approximate computing framework for artificial neural network. In: **2015 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2015. p. 701–706.

ZIEGLER, J.; LANFORD, W. The effect of sea level cosmic rays on electronic devices. In: **1980 IEEE International Solid-State Circuits Conference. Digest of Technical Papers**. [S.l.: s.n.], 1980. XXIII, p. 70–71.