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

FILIPE MACIEL LINS

The Effects of the Compiler Optimizations in Embedded Processors Reliability

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

Advisor: Prof. Dr. Paolo Rech

Porto Alegre
September 2017

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Lins, Filipe Maciel

The Effects of the Compiler Optimizations in Embedded Processors Reliability / Filipe Maciel Lins. – Porto Alegre: PGMICRO da UFRGS, 2017.

95 f.: il.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2017. Advisor: Paolo Rech.

1. Fault Injector. 2. Compiler Optimization. 3. Embedded Processors. 4. Soft Errors. 5. COTS. I. Rech, Paolo. II. The Effects of the ComThe Effects of the Compiler Optimizations in Embedded Processors Reliability.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

ACKNOWLEDGMENTS

To UFRGS, Institute of Informatics, PGMICRO, and the Brazilian research agencies CNPq for the financial support and for putting their facilities at my disposal so I could develop my research.

To my advisor, Paolo Rech, for the confidence, lessons, and patience. Thank you for pushing me forward and encouraging me in my academic and personal decisions.

To all my colleagues from laboratories 230 from UFRGS and others that are already in other rooms, cities, or countries.

To all of you, my sincere thanks.

ABSTRACT

The recent advances in the embedded processors increase the compilers complexity, and the usage of heterogeneous resources such as Field Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) integrated with the processors. Additionally, the increase in the usage of Commercial off-the-shelf (COTS) instead of radiation hardened chips in safety critical applications occurs because the COTS can be more flexible, inexpensive, have a fast time-to market and a lower power consumption. However, even with these advantages, it is still necessary to guarantee a high reliability in a system that uses a COTS for safety critical applications because they are susceptible to failures. Additionally, in the case of real time applications, the time requirements also need to be respected. As a case of study, this work uses the Zynq which is a COTS device classified as an All Programmable System-on-Chip (APSOC) and has an ARM Cortex-A9 as the embedded processor. In this research, the impact of faults that affect the register file in the embedded processors reliability was investigated. For that, fault-injection and heavy-ion radiation experiments were performed. Moreover, an evaluation of how the different levels of compiler optimization modify the usage and the failure probability of a processor register file. A set of six representative benchmarks, each one compiled with three different levels of compiler optimization. Exhaustive fault injection campaigns were performed to measure the registers Architectural Vulnerability Factor (AVF) of each code and configuration, identifying the registers that are more likely to generate Silent Data Corruption (SDC) or Single Event Functional Interruption (SEFI). Moreover, the observed reliability variations with register file utilization were correlated. Finally, two of the selected benchmarks, each one compiled with two different levels of optimization were irradiated in the heavy ions experiments. The results show that the best performance, the minor register file usage, or the lowest AVF does not always bring the highest Mean Workload Between Failures (MWBF). As an example, in the Matrix Multiplication (MxM) application, the best performance is achieved in the highest compiler optimization. However, in the fault injection, the higher reliability is obtained in the lower compiler optimization which has, the lower AVFs and the lower register file usage. Results also show that the impact of optimizations is strongly related to the executed algorithm and how the compiler optimizes them.

Keywords: Fault Injector. Optimization. Processor. Reliability. Soft Errors. COTS.

APSoC. MWBF. AVF.

RESUMO

O recente avanço tecnológico dos processadores embarcados aumentou a complexidade dos compiladores e o uso de recursos heterogêneos, como Arranjo de Portas Programáveis em Campo (*Field Programmable Gate Array* - FPGA) e Unidade de Processamento Gráfico (*Graphics Processing Unit* - GPU), integrado aos processadores. Além disso, aumentou-se o uso de componentes de prateleira (*Commercial off-the-shelf* - COTS) em aplicações críticas, ao invés de chips tolerantes a radiação, pois os COTS podem ser mais baratos, flexíveis, terem uma rápida colocação no mercado e um menor consumo de energia. No entanto, mesmo com essas vantagens, os COTS são suscetíveis a falha sendo necessário garantir uma alta confiabilidade nos sistemas utilizados. Assim como, no caso de aplicações em tempo real, também se precisa respeitar os requisitos determinísticos. Como caso de estudo, este trabalho utiliza a Zynq que é um dispositivo COTS do tipo Sistema em Chip Totalmente Programável (*All Programmable System on Chip* - APSoC) no qual possui um processador ARM Cortex-A9 embarcado. Nesta pesquisa, investigou-se o impacto das falhas que afetam o arquivo de registradores na confiabilidade dos processadores embarcados. Para tanto, experimentos de injeção de falhas e de radiação de íons pesados foram realizados. Além do mais, avaliou-se como os diferentes níveis de otimização do compilador modificam o uso e a probabilidade de falha do arquivo de registradores do processador. Selecionou-se seis benchmarks representativos, cada um compilado com três níveis diferentes de otimização. Realizamos campanhas exaustivas de injeção de falhas para medir o Fator de Vulnerabilidade Arquitetural (*Architectural Vulnerability Factor* - AVF) de cada código e configuração, identificando os registradores que são mais propensos a gerar uma corrupção de dados silenciosos (Silent Data Corruption - SDC) ou uma interrupção funcional de evento único (Single Event Functional Interruption - SEFI). Também foram correlacionadas as variações de confiabilidade observadas com a utilização do arquivo de registradores. Finalmente, irradiamos com íons pesados dois dos benchmarks selecionados compilados com dois níveis de otimização. Os resultados mostram que mesmo com o melhor desempenho, o menor uso do arquivo de registradores ou o menor AVF não é garantido que as aplicações irão alcançar a maior Carga de Trabalho Média Entre Falhas (*Mean Workload Between Failure* - MWBF). Por exemplo, os resultados mostram que o melhor desempenho da aplicação Multiplicação

de Matrizes (Matrix Multiplication - MxM) é alcançado no nível de otimização mais alta. No entanto, nos resultados dos experimentos de injeção de falhas, a maior confiabilidade é alcançada no menor nível de otimização que possuem os menores AVFs e o menor uso do arquivo de registradores. Os resultados também mostram que o impacto das otimizações está fortemente relacionado com o algoritmo executado e como o compilador faz esta otimização.

Palavras-chave: Injetor de Falhas, Otimizações, Confiabilidade, Processadores, COTS, APSoC, MWBF, AVF.

LIST OF ABBREVIATIONS AND ACRONYMS

ACP	Accelerator Coherency Port
ADC	Analogue-to-digital Converter
AES	Advanced Encryption Standard
AMBA	ARM Advanced Microcontroller Bus Architecture
APSoC	All Programmable System on Chip
APU	Application Processing Unit
ARM	Advanced Risc Machine
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CAN	Controller Area Network
CISC	Complex Instruction Set Computing
CLB	Configure Logic Block
COLE	Compiler Optimization Level Exploration
COTS	Commercial off-the-shelf
CPU	Central Processing Unit
DAC	Digital-to-analogue Converter
DCE	Dead Code Elimination
DSP	Digital Signal Processor
DUT	Design Under Test
ECC	Error Correcting Coding
EMIO	Extended Multiplexed Input/Output
FF	Flip Flop
FFT	Fast Fourier Transform
FIFO	First In First Out

FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GCC	GNU Compiler Collection
GDB	GNU Debugger
GPIO	General Purpose Input/Output
IOBs	Input/Output Blocks
IPC	Instructions Per Cycle
JPEG	Joint Photographic Experts Group
LUT	Lookup Table
MIO	Multiplexed Input/Output
MMU	Memory Management Unit
MMU	Memory Management Unit
MPE	Media Processing Engine
MWTF	Mean Work Time to Failure
MxM	Matrix Multiplication
OCM	On Chip Memory
PC	Program Counter
PL	Programmable Logic
PS	Processing System
PSoC	Programmable System on Chip
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
SCU	Snoop Control Unit
SIMD	Single Instruction Multiple Data
SoC	System on Chip

SPI	Serial Peripheral Interface
TSMC	Taiwan Semiconductor Manufacturing Company
UART	Universal Asynchronous Receives Transmitter
USB	Universal Serial Bus

LIST OF FIGURES

Figure 2.1 Radiation Sources in Space 20
Figure 2.2 Radiation effects 21
Figure 2.3 SEU and SET in circuits 22

Figure 4.1 An example of APSOC: the Xilinx Zynq-7000 31
Figure 4.2 Single Instruction Multiple Data processing in NEON 32
Figure 4.3 The PL structure of Zynq-7000 34
Figure 4.4 The Configurable Logic Block (CLB) 35

Figure 5.1 Heavy ion experiment setup mounted at the beam line of the LAFN-USP 47
Figure 5.2 View of the surface of a Zynq-7000 device 47
Figure 5.3 Fault Injector First Setup 49
Figure 5.4 Fault Injector flow chart 50
Figure 5.5 Fault Injector Modifications of (OLIVEIRA; TAMBARA; KASTENSMIDT, 2017) 51
Figure 5.6 Fault Injector Final Setup 52

Figure 6.1 Total Architectural Vulnerability Factor SDCs 58
Figure 6.2 Total Architectural Vulnerability Factor SEFIs 58
Figure 6.3 MxM Architectural Vulnerability Factor SDCs 59
Figure 6.4 MxM Architectural Vulnerability Factor SEFIs 59
Figure 6.5 AES Architectural Vulnerability Factor SDCs 60
Figure 6.6 AES Architectural Vulnerability Factor SEFIs 60
Figure 6.7 AVF SDC MxM Double O3 62
Figure 6.8 AVF SDC MxM Double O3 62

Figure 7.1 AVF and Cross Section in MxM 67
Figure 7.2 Radiation and Fault Injection MWBFs in MxM 67
Figure 7.3 AVF x Cross Section in AES 67
Figure 7.4 Radiation and Fault Injector MWBFs in AES 68

LIST OF TABLES

Table 3.1 The main GCC optimizations and the levels at which they are enabled. 26

Table 4.1 The Zynq Logic Fabric Resources 34
Table 4.2 Compiler Optimization effects on code execution, resources utilization.....40

Table 6.1 Compiler Optimization effects on code execution, resources utilization, and reliability 54
Table 6.2 Fault Injection Results.....61
Table 6.3 Heavy Ions Experiment Results 64
Table 6.4 Number of Reads and Writes in the main memory 64

Table A.1 Results of Fault Injection MxM O0.....74
Table A.2 Results of Fault Injection MxM O2.....75
Table A.3 Results of Fault Injection MxM O3.....76
Table A.4 Results of Fault Injection AES O0 77
Table A.5 Results of Fault Injection AES O2 78
Table A.6 Results of Fault Injection AES O3 79
Table A.7 Results of Fault Injection Quicksort O0.....80
Table A.8 Results of Fault Injection Quicksort O2.....81
Table A.9 Results of Fault Injection Quicksort O3.....82
Table A.10 Results of Fault Injection FFT O0.....83
Table A.11 Results of Fault Injection FFT O2.....84
Table A.12 Results of Fault Injection FFT O3.....85
Table A.13 Results of Fault Injection Fibonnaci O0.....86
Table A.14 Results of Fault Injection Fibonnaci O2.....87
Table A.15 Results of Fault Injection Fibonnaci O3.....88
Table A.16 Results of Fault Injection JPEG O0.....89
Table A.17 Results of Fault Injection JPEG O2.....90
Table A.18 Results of Fault Injection JPEG O3.....91
Table A.19 Results of Fault Injection MxM Double O3.....92
Table A.20 Results of Fault Injection MxM Double O3 Continuation.....93
Table A.21 Results of Fault Injection MxM Double O3+ 94
Table A.22 Results of Fault Injection MxM Double O3+ Continuation.....95

CONTENTS

1 INTRODUCTION **15**
1.1 Objectives and Contributions **16**
1.2 Organization **16**
2 BACKGROUND **18**
2.1 RISC Architecture **18**
2.2 Soft Errors Effects in Embedded Processors **20**
3 CODE OPTIMIZATION EFFECTS **24**
3.1 Why optimizations? **24**
3.2 What are the optimizations? **24**
3.3 Optimizations effects in performance and energy consumption **28**
3.4 State of the art **29**
4 CASE STUDY: THE EMBEDDED ARM A9 ..... **30**
4.1 All Programmable System-on-Chip **30**
4.1.1 Processing System 30
4.1.2 Programmable Logic 33
4.1.3 The PS - PL Interfaces 36
4.2 Algorithms **37**
4.3 Setups **38**
5 RELIABILITY EVALUATION METHODOLOGY **43**
5.1 Beam experiment vs fault injection **43**
5.2 Reliability Metrics **44**
5.3 Heavy ion experiment **46**
5.4 Fault injection framework **48**
6 RESULTS **53**
6.1 Fault Injection Results **53**
6.1.1 General purpose Register Setup Results 54
6.1.2 NEON Setups Results 62
6.2 Heavy Ions Experiment Results **63**
6.3 An Analysis of the Dynamic Disassembly Code **64**
7 CONCLUDING REMARKS **66**
7.1 Discussion **66**
7.2 Future Work **69**
REFERENCES **70**
APPENDIX A — TABLES **74**
A.1 MxM Tables **74**
A.2 AES **77**
A.3 Quicksort **80**
A.4 FFT **83**
A.5 Fibonnaci **86**
A.6 JPEG **89**
A.7 MxM Double **92**

1 INTRODUCTION

Commercial-Off-The-Shelf (COTS) systems are becoming attractive for safety-critical applications, like biomedical implantable devices, automotive control systems, aircraft or satellite stabilizers and control circuitry. The main reason for preferring a COTS device to specifically designed rad hard chips is that the latter are typically very expensive as they require unique circuit design and lithography to meet the reliability requirements and the produced volumes are low. On the contrary, COTS components can be low-cost, flexible, have fast time-to-market, low power consumption.

The CHREC Space Processor (CSPv1) is mainly formed around a Xilinx Zynq-7020 Processor which is an example of COTS (RUDOLPH et al., 2014). Unfortunately, COTS systems are very susceptible to radiation effects, particularly Single Event Effects (SEEs). The single impinging particle can generate different types of effects on digital systems. If the particles hit a storage element, such as Static Random Access Memory (SRAM) structures, caches, and registers, it may cause a bit flip. Particles can also modify the result of operations by generating Single Event Transients (SETs) in logic gates.

Reduced Instruction Set Computing (RISC) architectures became popular thanks to the advances in compiler efficiency. In order to succeed in executing complex algorithms using few, simple, and two-inputs instructions, the compiler needs to modify the source code significantly. In recent years, compilers have allowed users to select different levels of optimization to be applied to the code. Optimization is achieved by modifying the number, the usage, and the reference of registers.

Most modern embedded processors are built with RISC architecture which allows only a few basic instructions to be executed, and inputs can come only from the register file. In other words, RISC is a load-store architecture: data in the main memory must be loaded to the register file to be digested, and the data in the register file must be stored in the main memory to be accessible to the user. Hence, the register file is a critical resource for modern computing architectures. As any data to be processed will need to pass through the register file, knowing the probability of an error in a register to propagate to the output may be sufficient to characterize the vulnerability of an application.

Register files have been identified as the most critical resources in modern computing systems (TAN et al., 2011; ISAZA-GONZÁLEZ et al., 2016). Unfortunately, registers cannot be easily protected. Unlikely Dynamic Random Access Memory (DRAM), which can be protected with Error Correcting Codes (ECC), or caches, for which parity

protection is more common than ECC, register files are integrated into the circuitry of processors, increasing the efforts and penalties of adding ECC (ASADI et al., 2005).

For these reasons, the focus of this research is the reliability impact of register file errors in embedded System-on-Chip (SoC) devices. For our study, it was used the Zynq-7000 device, which is equipped with dual-core ARM Cortex-A9 processors integrated with 28nm Artix-7 based programmable logic. Using a homemade fault injection platform, the criticality of each available general purpose register was identified by measuring the probability of an injected fault to generate a SDC or a SEFI. A set of six benchmarks applications were selected: MxM, Advanced Encryption Standard (AES), Quicksort, Fast Fourier Transform (FFT), Fibonacci, and Joint Photographics Experts Group (JPEG).

1.1 Objectives and Contributions

The RISC processors, the increase in the compiler and the optimizations complexity have made it harder to evaluate the fault tolerance of the hardware and their applications. The objective of this work is to understand better how to evaluate the reliability of the hardware and the applications.

In order to achieve this objective, it was necessary to do a radiation experiment and a fault injection in the tested benchmarks. The fault injector accesses the general purpose and NEON registers in the ARM Cortex-A9 in the Zynq-7000 architecture to inject faults. Finally, the APSoC device executing the MxM and AES applications compiled with different compiler optimizations were irradiated with heavy ion particles. Therefore, an evaluation of the effects of compiler optimization in the reliability of the device can be done.

1.2 Organization

This dissertation is organized as follows:

- Chapter 2 - Background knowledge: introduces the RISC Architecture, and the soft errors effects in embedded processors;
- Chapter 3 - Code optimization effects: presents a study about why the optimizations are used, what are the optimizations, the optimization effects in the performance and energy consumption, and the state of the art of the optimization effects in the

reliability;

- Chapter 4 - Case study: The embedded ARM A9: introduces the APSoCs architecture, Xilinx Zynq-7000 as an example of APSoC and the case study device, the algorithms evaluated in this work, and the setups used in the experiments;
- Chapter 5 - Reliability evaluation methodology: presents the differences between the beam experiment and the fault injection, the metrics used to evaluate the reliability, and the methodology used in the heavy ion experiment and fault injection framework;
- Chapter 6 - Results: presents and analyzes the results obtained in the fault injection, and in the heavy ions experiments;
- Chapter 7 - Concluding remarks: presents the concluding remarks of this dissertation, such as its a discussion about the obtained results and future works;

2 BACKGROUND

This chapter introduces the background knowledge for the understanding of the dissertation. It is presented the RISC architecture and the sources of radiation and its effects on circuits and embedded processors.

2.1 RISC Architecture

The Instruction Set Architecture (ISA) is an abstract model of a computer that defines the external Input/Output (I/O) model, the native data types, the instruction set, the memory architecture, the addressing modes, the registers, the interrupt modes, and the exception handling. Based on architectural complexity, ISAs are commonly divided in the Complex Instruction Set Computing (CISC) and the Reduced Instruction Set Computing (RISC).

In the design of the first processors, memory was expensive and had a small capacity (DANDAMUDI, 2005). In order to develop programs which can run using these limited resources, the designers developed the CISC instruction set, which consists of a set of complex instructions so the total size of the program could be reduced and stored in the memory.

The microprogramming in the CISC architectures facilitates the cost-effective implementation of complex instructions by using microcode. The use of small and fast memories is also required to store the microcode to reduce the memory access latency on performance. Another advantage of using these complex instructions is the close in the semantics of the high-level languages and the machine languages.

One of the reasons for first develop the CISC architecture was the fact of the complex instructions were faster than an equivalent using simple instructions. However, the measure of a single instruction is not the only measure of performance. The overall system performance should be considered.

The first RISC processors were designed from IBM, Standford, and UC-Berkeley. The Stanford MIPS, IBM 801, and Berkeley RISC 1 and RISC 2 were all designed with a similar philosophy which utilizes a small, highly optimized set of instructions (ZARGHAM, 1996).

Computing devices based on RISC architecture allow only a limited set of simple two-input operations to be executed. Additionally, data to be digested by the processor

must be loaded into registers. This load of the data in the register guarantees high performance, as simple operations can be executed in few clock cycles as registers are fast. In order to achieve the best performance, data must remain in the register files until they are no longer necessary for any instruction. In fact, loading data from the main memory or even from the cache is considered a time-consuming task that should be executed only when necessary. An anticipated result of this work is that the increasing utilization of the register file increases performance and reduces the code reliability.

RISC architectures became popular thanks to the advances in compiler efficiency. Complex and efficient compilers are the key tool to take full advantage of RISC performance successfully. The compiler rearranges the source code into a sequence of simple instructions. Moreover, the compiler defines when data must be loaded into the registers from the main memory and when data from registers can be stored in the main memory. Several iterations are typically required for the compiler to produce an efficient assembly code from the high-level source code.

The RISC design implements simple instructions that can be executed in one cycle. As a consequence of it, the processor design is simplified. In this design there is no need for microcode and operations can be hardwired.

Most of the CISC instruction set supports the register-to-memory, register-to-register, and the memory-to-memory operations. The RISC architecture allows only special load and store operations to access memory and the other operations work using the register-to-register operations. This difference in the RISC simplifies the instruction set design allowing the execution of one Instruction Per Cycle (IPC). Also, simplifies the design of the control unit. Advances in Integrated Circuits (IC) technology dimensions and the operating voltages has brought an increase in the high density, a decrease in the power, and an increase in the sensitivity of the circuits to single event effects.

The usage of simple addressing modes in the RISC processors allows a fast address computation of operands. Only the load and store instructions need a memory addressing mode. The majority of the instructions use register-to-register operations. A larger register set is needed in the RISC architecture, a consequence of it is the ample opportunities of the compiler to optimize their usage and also, a decrease in the overhead associated with procedure calls and returns.

The RISC architecture uses the fixed simple length instructions which are more efficient than the variable length instructions. As mentioned before the RISC design also use a simple instruction format allowing an efficient decoding and scheduling of instruc-

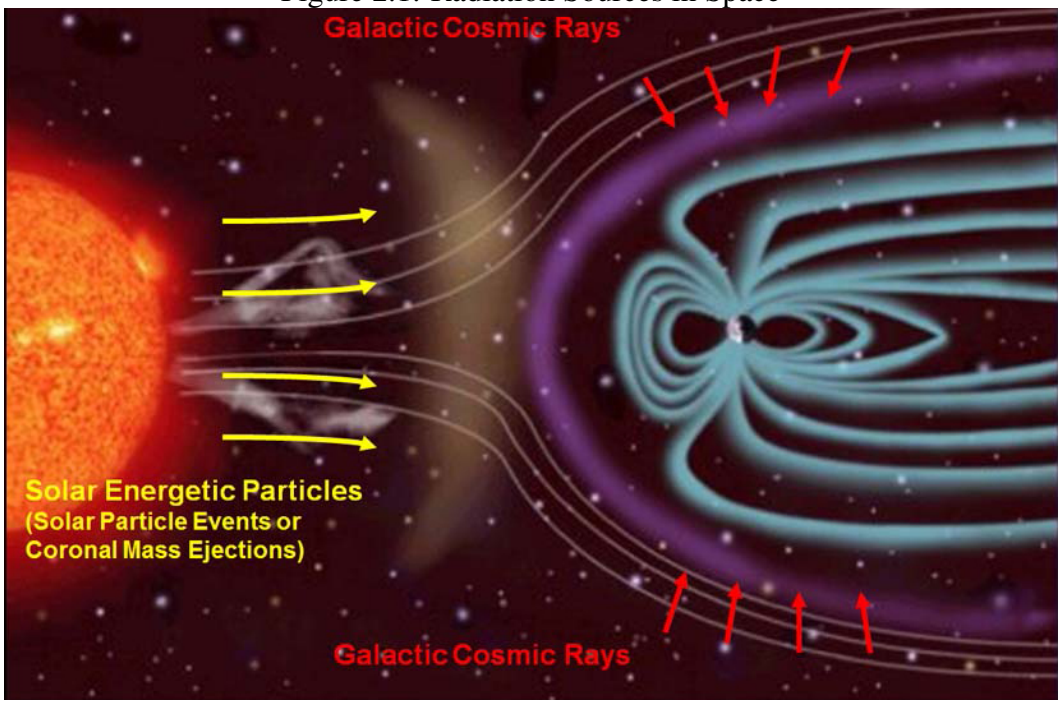
tions.

The first RISC processors had fewer instructions compared to the CISC. The new RISC processors have hundreds of instructions, and some of them are complex such as used in the CISC architecture. These new ones can be called a hybrid of the CISC and RISC.

2.2 Soft Errors Effects in Embedded Processors

Advances in Integrated Circuits (IC) technology dimensions and the operating voltages has brought an increase in high density, a decrease in power, and an increase in radiation sensitivity of the circuits (BAUMANN, 2005).

Figure 2.1: Radiation Sources in Space



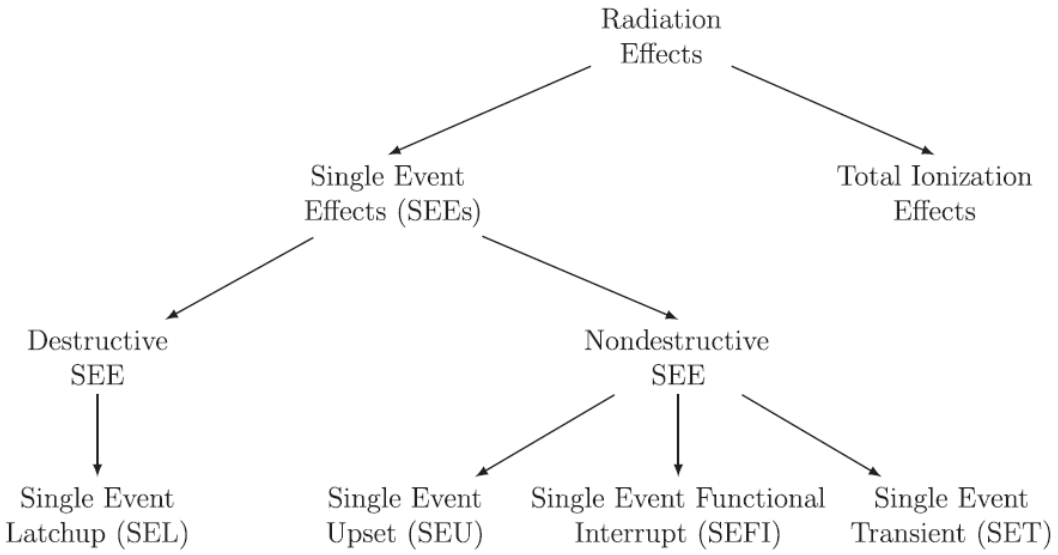
Source: NASA/JPL-Caltech/SwRI

The main sources of ionizing radiation come from solar flares, solar wind and cosmic rays as showed in Figure 2.1. The Van Allen belts are a collection of charged particles, gathered in place by Earth's magnetic field and includes two electron belts and one inner proton belt. The inner belt contains electrons whose energy is less than $5MeV$. The outer belt contains electrons whose energy may reach $7MeV$ (BOUDENOT, 2007). The belts are located in the inner region of the Earth's magnetosphere. Heavy ions may also be trapped in the magnetosphere.

Cosmic rays are high-energy particles arriving from outer space. These particles

are mainly protons, nuclei of hydrogen, nuclei of helium and heavier nuclei. When they arrive on Earth, they collide with the nuclei of atoms in the upper atmosphere producing a shower of secondary particles: x-rays, muons, protons, alpha particles, pion, electrons, and neutrons. Each kind of particle produces different effects on ICs.

Figure 2.2: Radiation effects



Source: (SIEGLE et al., 2015)

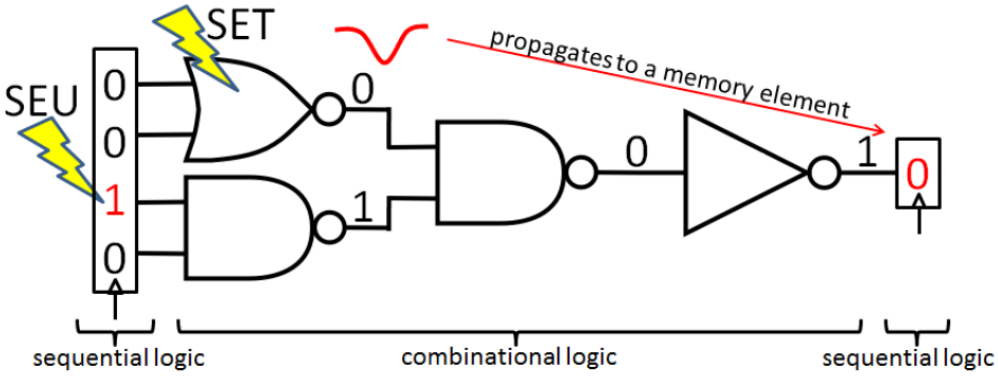
Figure 2.2 shows that the radiation effects can be divided into Total Ionization Dose (TID) Effects, and Single Event Effects (SEEs).

The TID is a cumulative effect of ionizing radiation over the exposure time. In the complementary metal oxide semiconductor transistors the exposure to high-energy ionizing radiation generates electron-hole pairs which causes a buildup of charge within the oxide. This buildup of charge will change the threshold voltage, increase the leakage current, and modify the timing of the CMOS transistor, leading to the parametric degradation and/or functional failure of the electronic device.

SEEs happens when a single ionizing particle deposits sufficient energy to disrupts the normal operation of a circuit. The SEEs can be divided in Destructive and Nondestructive. Destructive SEE is represented by Single Event Latchup (SEL), which triggers parasitic PNPN thyristor structures in a device, which can cause permanent damage or if not permanently a power cycling of the circuit is needed. Nondestructive SEE can be represented by the Single Event Upset (SEU) or Soft Error, the Single Event Functional Interrupt (SEFI), and the Single Event Transient (SET). The SEU happens when an ionizing radiation particle strikes a configuration memory cells, user memory, and regis-

ter changing the logic state of the element. The SEFI happens in complex devices like modern memories, processors, Field-programmable gate array (FPGA) or Application Specific Integrated Circuits (ASICs), and mixed-signal devices and the main effect is a component reset, lock-up, or malfunction. The SET is an electrostatic discharge caused by a single energetic particle strike whose the effect is a soft error.

Figure 2.3: SEU and SET in circuits



Source: (CHIELLE, 2016)

In Figure 2.3 is showed an example of an SEU and a SET. A particle hits a memory element which causes an SEU and changes the stored value from 0 to 1. The SEU in the example is masked by the NAND gate because the other NAND input is 0. On the contrary, if the other NAND input is 1 the output of the NAND would be changed, and the error would propagate.

Another particle in Figure 2.3 hits a NOR gate causing a SET in which temporarily change the output from 1 to 0. The fault is not masked by any gate and propagates to the memory element. In which the wrong value can be stored if the pulse hits a memory element during a clock event.

Soft errors in embedded processors happen when a value stored in memory elements (i.e., register or data memory) is modified. This change in the memory elements values can lead to executing incorrectly an application producing a wrong output or even never finish the execution. They can affect the data-flow and the Control-flow of a running application (CHIELLE, 2016).

The data-flow errors affect the program output. When a fault affects the data-flow, the application runs normally, but the output is incorrect at the finish of the execution. The data-flow errors are usually caused by wrong operation or incorrect data.

The control-flow errors happen when the program flow is incorrectly executed.

The possible causes of this type of error is a branch creation, a branch deletion, an incorrect branch decision, an incorrect target address, or a bit flip in the Program Counter (PC) register.

3 CODE OPTIMIZATION EFFECTS

In this chapter, it will be explained the optimizations and their effects in performance, energy consumption, and reliability in the applications.

3.1 Why optimizations?

Since the advent of the first computer, code optimizations have been a widely used tool to perform all sort of code transformations. The objectives of the optimization depend on the constraints of the application, like time or memory requirements in real time applications in embedded systems or energy consumption in a spacial application which the power is limited.

Optimizations can be applied at different levels like design level, algorithms and data structures, source code, build, compile, and assembly. In general, the higher levels in compiler optimizations provide a greater impact but requires significant changes or even to completely rewrite the code. In some cases, the performance bottleneck is in the low level of the code, and small variations in this state have a significant impact on the program performance. In this work, a study of the effect of the global compiler optimization in the code reliability is done.

In recent years, compilers have allowed users to select different levels of optimization to be applied to the code. Optimization is achieved by modifying the number, the usage, and the reference of registers. The optimized code can hugely increase the performance of the applications but also increases the registers usage.

3.2 What are the optimizations?

In order to produce an optimized code, the compiler performs one or more transformation in the source code. These transformations aim at modifying the original code to a more efficient code preserving the meaning and the output of the code. The most common optimizations that compilers can perform are: the control flow analysis, the data flow analysis, the code motion, the common subexpression elimination, the constant folding, the if-conversion, the dead code elimination, the copy propagation transformations, the inlining (HAGEN, 2011).

The Data Flow analysis identifies how and when variables are used in a program and after that applies various set equations to these usage patterns to find optimization opportunities. The Code Motion is an optimization technique which reduces the number of common subexpressions relocating them to more optimal locations in an intermediate representation of the code. Loops can be optimized using this technique decreasing, for example, the number of instructions in an inner loop and increase the part of the code outside that loop. The Common Subexpression Elimination is a standard optimization mechanism, which avoids the recalculation of previously calculated values. Thus, reducing the number of instructions executed by a program to achieve the same result. The Dead Code Elimination (DCE) is the optimization technique that removes variables and statements in the code that does not do anything permanent or useful in the program. The Constant Folding technique eliminates expressions, which can be calculated when the program is compiled. The If-conversion is a technique where branch constructs are broken into separate if statements to simplify generated code and eliminate jumps and branches wherever possible. The Copy Propagation transformation is an optimization technique to reduce or eliminate redundant computations eliminating cases in which values are copied from one location or variable to another assigning their value to another variable. The Inlining is an optimization technique where code performance can improve by replacing complex constructs, and even function calls with an inline representation of the construct or function call. The Control Flow analysis examines loops and other control constructs to identify the execution paths and based on this analysis tries to make the execution path more efficient (AHO et al., 2006).

The user manual for Linaro GNU Compiler Collection (GCC) 4.9-2014 mentions that at least 160 code optimizations can be turned on or off individually. Users also can specify more than 100 different values controlling the amount of, or determine the context for, applying a particular optimization. Individually handling all these options would not be practical for regular application development. Thus, GCC provides summary optimization options named -O0, -O1, -O2, -O3, and -Os. The default level of optimization in Release mode is O2. Table 3.1 reflects the most significant optimization flags in GCC compiler that are enabled when the -O level is applied.

Higher optimization levels perform more global transformations on the program and apply more expensive analysis algorithms to generate a faster and more compact code. The price in compilation time and the resulting improvement in execution time depending on the particular application and the hardware environment.

Table 3.1: The main GCC optimizations and the levels at which they are enabled.

	Included in level				
Optimization	**O0**	**O1**	**O2**	**Os**	**O3**
defer-pop		X	X	X	X
tree-dce		X	X	X	X
cprop-registers		X	X	X	X
loop-optimize		X	X	X	X
omit-frame-pointer		X	X	X	X
rename-register		X	X	X	X
align-loops			X	X	X
align-jumps			X	X	X
align-labels			X	X	X
align-functions			X	X	X
regmove			X	X	X
optmize-strlen			X		X
schedule-insns			X		X
cse-follow-jumps			X	X	X
cse-skip-blocks			X	X	X
peephole2			X	X	X
inline-functions					X
predictive-commoning					X
tree-partial-pre					X

Source: The Author

The GCC compiler enables a plethora of optimization to achieve better performances. Some optimizations reduce the size of the resulting machine code, while others try to create a faster code, potentially increasing its size (e.g., *loop unrolling* increases the assembly code but reduces the execution time).

O0 is the level, which GCC does not perform any optimization and compiles the source code in the most straightforward way possible. Each command in the source code is converted directly to the corresponding instructions in the executable file, without rearrangement. The compiler's goal is to reduce the cost of compilation and be able to set breakpoints, examine variables, and then continue execution.

The O1 is the first level of optimization, which the compiler tries to minimize both code size and execution time without a massive increase in the compilation time. Some optimizations are enabled at this level as shown in Table 3.1. These optimizations are described in the next paragraph.

The cprop-registers tries to reduce the number of the register copy operations.

The defer-pop accumulates function arguments on the stack. The tree-dce eliminates dead code to reduce the application size. The omit-frame-pointer avoids instructions required to set up, save, and restore frame pointers decreasing the registers usage. The loop-optimize moves constant expressions and simplifies test conditions for exit the loops in other levels of optimization this flag performs strength reduction and unrolls loops. The rename register is the GCC option that tries to use any unallocated registers to avoid false dependencies in scheduled code is frequently used on systems with a large number of registers.

The O2 level performs all O1 optimization flag and also other supported optimizations that do not involve a space-speed trade-off. Some of the main optimizations enabled in O2 level are described in the following paragraph.

The four align optimizations, which make functions, labels, jumps, and loops are aligned with the machine natural memory size boundaries avoiding no-op instructions and making the code faster. The regmove makes the GCC reassign the registers to maximize the number of registers used in the application. The optimize-strlen optimize standard C string functions using fasting alternatives. The schedule-insns reorder instructions to eliminate execution stalls when the data is unavailable.

The Os enables all O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size. The Os level disables some of the O2 level optimizations like the optimize-strlen and the schedule-insns that increment the size of the final code. This level also maintains some optimizations like the cse-follow-jumps follows jumps that, which target is not unless reached. The cse-skip-blocks follows jumps that conditionally skip blocks. The peephole2 tries to replace longer set of instructions with shorter.

The O3 is the third and highest level enabled, which emphasizes speed over size. This optimization includes all optimizations enabled at O2 and some other optimizations. The inline-function integrates all functions into the routines that call them boosting performance but also can drastically increase the size of the assembly code, which can have severe effects on the code performance. Other optimizations are enabled in the O3 level like the predictive-commoning reuse memory stores and loads performed in iterations loop, and the tree-partial-pre makes an aggressive partial redundancy elimination.

Besides the level optimizations, the compiler has variously specific optimizations that can be enabled manually. Many of these optimizations involve floating point operations making significative performance improvements but changes the ISO and IEEE

specification for math functions.

It is worth noting that optimization efficiency strongly depends on the source code. Some algorithms for which the control flow is data-dependent may be hard to be optimized. In fact, the compiler can not statically predict the branches to be taken as data is not yet available. The branch prediction can not happen, for instance, in sorting algorithms which perform branch operations based on comparisons among data.

3.3 Optimizations effects in performance and energy consumption

One of the major effects of the compiler optimizations that affect the performance is the decrease in the memory access time. This decrease is made by eliminating redundant accesses to memory and replaces with shorter latency event like register copying and value propagation through registers. As a consequence of it, the remaining accesses in the application has a higher miss rate in the cache (DEMERTZI; ANNAVARAM; HALL, 2011).

The changes in control flow is another consequence of compiler optimizations that affect the performance. A delay in branch resolution can stall the application until the branch outcome is resolved. The branch misprediction is when a wrong branch is called, the partially processed instructions in the pipeline need to be discarded, and the pipeline has to start over at the correct branch. The optimized codes have fewer committed branches and less branch misprediction than unoptimized code. The lower number of overall branch instruction allows the processor to issue and execute long basic blocks (DEMERTZI; ANNAVARAM; HALL, 2011).

The effect of the different level of compiler optimizations in the energy consumed to execute the code is directly proportional to the number of instructions. As consequence of it, optimizations that improve performance by increasing the parallelism in the program increases the energy consumption. Optimizations that decreases the number of instructions like common sub-expression elimination, induction variable elimination and unrolling also decreases the energy consumed (VALLURI; JOHN, 2001).

The influence of the different level of compiler optimizations on power dissipation is directly proportional to the number of instructions per cycle. An example of it is the optimizations such as instruction scheduling, and loop unrolling that increase the performance but also increases the IPC has a side effect which is the increases in the power dissipation (VALLURI; JOHN, 2001).

A consequence of the compiler optimizations is the significant decrease in the number of reads and writes in the main memory with the increase of the optimization level (CHIBANI et al., 2014).

3.4 State of the art

In Recent years, several studies have addressed the issue of evaluating the influence of optimizations in applications reliability.

In Sangchoolie et al. (2014) the authors evaluate the compiler optimizations injecting faults in the instruction set architecture registers and main memory locations. All the 12 benchmarks presented in the paper has an increase in the performance, but the gain varies significantly depends on the applications. The conclusion of this study is that the compiler optimization has a minor effect on the reliability of the investigated benchmark applications.

In Ferreira et al. (2013) the authors discuss how compiler optimizations influence software reliability when the applications are compiled with a technique, which detects and correct radiation control-flow errors. In order to increase the performance in the ten benchmarks selected from the Mibench embedded benchmark suite, the author uses the Compiler Optimization Level Exploration (COLE), which uses a population-based multi-objective optimization algorithm to construct a Pareto optimal set of optimizations. In the selected optimizations only 25% of the GCC optimizations appear in at most one Paretto. In this work the author implemented a software fault-injector, using GDB to perform the fault injections. The consequence of indiscriminately selecting the optimizations can decrease the software vulnerability to unacceptable levels.

In Serrano-Cases and Isaza-González (2016) the authors use the NSGA-II algorithm to choose the best optimization to reduce the code size, and execution time while improving the reliability of the final application. In the experiment, the MSP430 architecture (NAGY, 2003) has been used to inject faults in an open-sourced simulator. The method improves the fault coverage from 3% to 6% and the MWTF from 15% to 45%.

The compiler optimizations have a significant influence on the reliability of the applications. A better understanding of how to use them and how they affect the embedded processors is an essential study in the use of COTS in space applications.

4 CASE STUDY: THE EMBEDDED ARM A9

This chapter introduces the All Programmable System-on-Chip (APSoC) architecture which in our case of study is the Zynq-7000. It is also presented the setups used in the fault injection and the heavy ion experiments.

4.1 All Programmable System-on-Chip

The Application Specific Integrated Circuits (ASIC)s, which can include digital, analog and radio-frequency components, together with mixed-signal blocks implementing Analog-to-Digital Converters (ADC) and Digital-to-Analog Converters (DAC). A System On Chip (SoC) can combine all aspects of a digital system: processing, high-speed logic, interfacing, memory, and so on. The SoC solution is lower cost, enables faster and more secure data transfers between the various system elements, has higher overall system speed, lower power consumption, smaller physical size, and better reliability. The disadvantages of ASIC-based SoCs are development time and cost, and lack of flexibility.

The Programmable System-on-Chip (PSoC) is a SoC implemented in a programmable, reconfigurable device like Field Programmable Gate Array (FPGA)s, which offers a flexible platform than ASICs for implementing SoCs. The APSoC integrates the software programmability of a processor with the hardware programmability of an FPGA (CROCKETT et al., 2014).

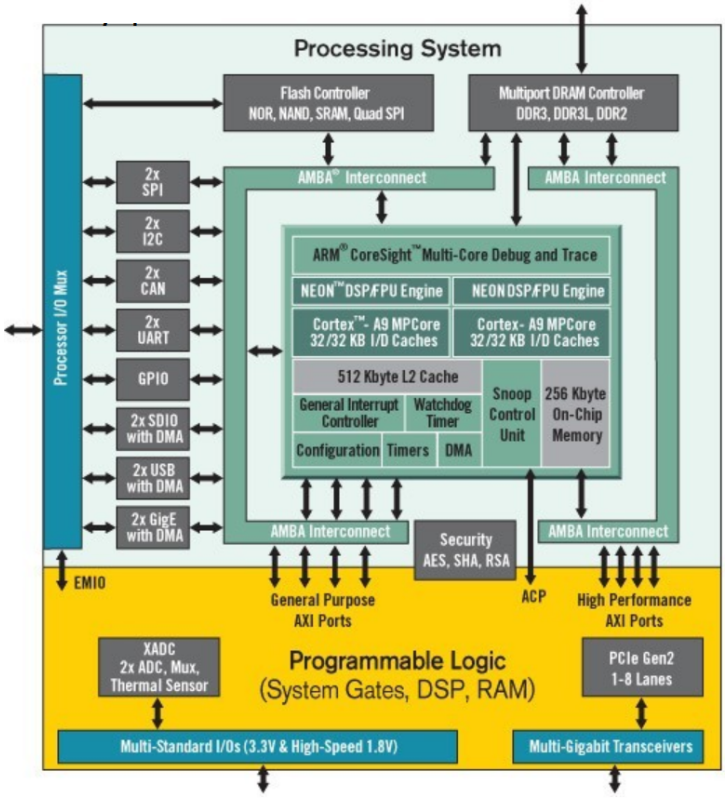
The APSoCs is formed around a Processing System (PS) and a Programmable Logic (PL). The PL section is used for high speed, arithmetic, and data flow subsystems. The PS supports operating systems and software applications. As a consequence, the system functionality can be partitioned between hardware and software.

4.1.1 Processing System

The present study is based on the COTS Zynq-7000 APSoC designed by Xilinx in Taiwan Semiconductor Manufacturing Company (TSMC) 28nm technology node. The Device Under Test (DUT), an XC7Z020-CLG484 part, is embedded in a commercially available Zed-Board Development Board. The Zynq architecture is showed in Figure 4.1.

The PS is formed around a hard-core processor the dual-core ARM Cortex-A9

Figure 4.1: An example of APSOC: the Xilinx Zynq-7000



Source: Xilinx

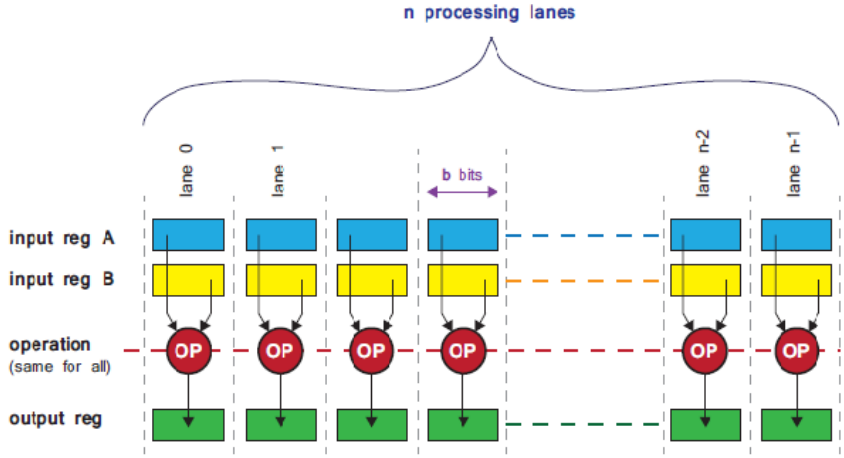
processor, other associated processing resources forming the Application Processing Unit (APU), and others peripheral interfaces, cache memory, memory interfaces, interconnect, and clock generation circuitry.

The APU embraces the two processors core each one with his NEON Media Processing Engine (MPE), Floating point Unit (FPU), a Memory Management Unit (MMU), and a Level 1 cache memory. The APU also has an On Chip Memory (OCM), a Level 2 cache memory, and a Snoop Control Unit (SCU).

The ARM Cortex-A9 processor is a general purpose choice for low power, cost-sensitive 32-bit devices, high-efficiency, dual-issue super scalar, out-of-order, and has a dynamic length pipeline (8 - 11 stages). The processor in Zynq-7000 operates with a clock frequency up to 667 MHz.

The ARM Cortex-A9 dual-core processor architecture has 56 physical 32-bit registers. Each core of the processor has 15 general purpose registers and one Program Counter (PC) register. Also, the processor has five banked registers, 15 banked general-purpose registers, and five status registers. In our Fault Injector, we inject faults in general purpose registers of one core using bare-metal applications, representing 25% of total registers.

Figure 4.2: Single Instruction Multiple Data processing in NEON



Source: (CROCKETT et al., 2014)

The two cores in the ARM processor has a separate Level 1 caches of 32KB for instructions and data being able to store the most used data and instructions to increase the processor performance and decrease the data access time. Additionally, the cores share a Level 2 cache of 512KB for instruction and data and also, have a 256KB OCM. The Memory Management Unit (MMU) translates the virtual address to physical addresses.

The SCU interfaces the processors and the Level 1 and 2 cache memories being responsible for maintaining memory coherency between the L1 data cache memories and the Level 2 cache memory. Additionally, manages the transactions between the PS and the PL via the Accelerator Coherency Port (ACP).

Some modern software, particularly media codecs and graphics accelerators, operate on large amounts of data that is less than word-sized. The 16-bit data is common in audio applications, and 8-bit data is standard in graphics and video. The implementation of the Advanced Single Instruction Multiple Data (SIMD) extension used in ARM processors is called NEON, and this is the standard terminology used outside architecture specifications.

The SIMD technology provides the use of a single instruction to perform the same operation in parallel on multiple data elements of the same type and size. In Figure 4.2 two registers in which each one contains a set of N individual input vector are operated using the same operation for all inputs producing a set of output vectors which are written in the output register. The NEON engine in ARMv7-A/R including ARM cortex A9 supports these data types: unsigned and signed integers, single and half precision floating point.

The NEON instructions are executed as part of the ARM or Thumb instruction

stream. The instructions provide memory accesses, data copying between NEON and general purpose registers, data type conversion, and data processing. The NEON register bank can be accessed as 64 registers in single precision or 32 registers in double precision or 16 registers in quad precision. This register bank is implemented on all current ARM Cortex-A series processors.

Developing for NEON can be done writing the code in assembly or writing the code using the intrinsics functions defined by ARM or using the compiler auto-vectorization. The intrinsics functions use the C or C++ high-level languages given direct access to NEON instructions. As a consequence, the compiler can optimize the operations for performance decreasing the work for the developer to consider the register allocation in the code. The compiler also can perform an automatic vectorization in C or C++ source code without writing assembly code or the developer using intrinsics functions. The advantage of the automatic vectorization is the fast development and the code now is portable between different tools and target platforms.

The FPU provides hardware acceleration of floating point operations following the IEEE 754 standard support single and double precision formats. Also, supports half precision and integer conversion.

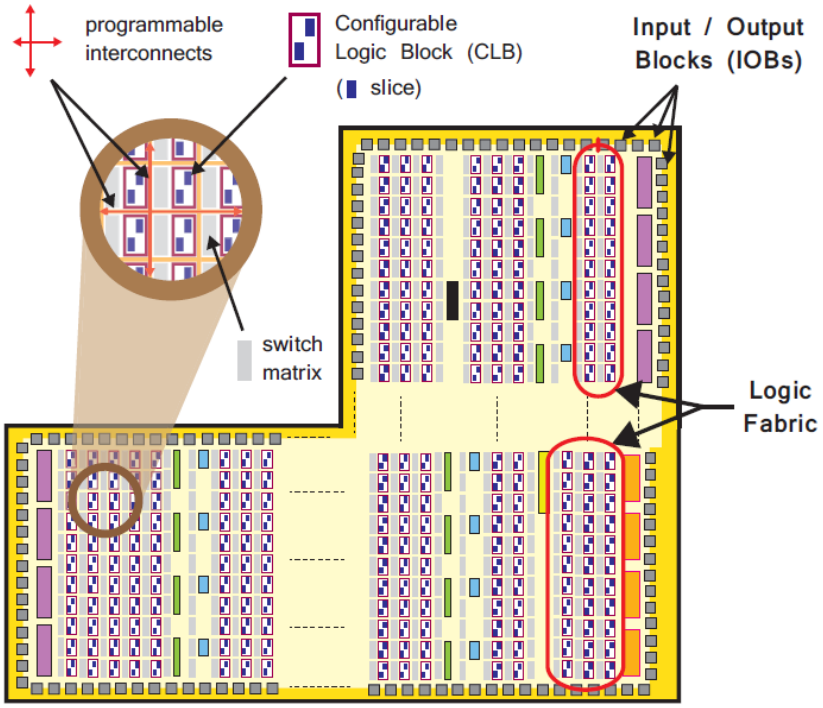
The Figure 4.1 shows that the Zynq has a variety of interfaces between the PS and external components. The Multiplexed Input/Output (MIO) is the primary communication interface and provides 54 pins of flexible connectivity. A Way to extend the communication interfaces when is required it is using the Extended MIO (EMIO), which passes through and shares the I/O resources of the PL. The Available I/O peripheral interfaces are two Serial Peripheral Interface (SPI), two I2C bus, two Controller Area Network (CAN), two Universal Asynchronous Receiver Transmitter (UART), the General Purpose Input/Output (GPIO), two SD card memory, two Universal Serial Bus (USB), and two Ethernet MAC peripheral.

4.1.2 Programmable Logic

The PL is formed around an equivalent Xilinx Artix-7 FPGA logic fabric. The available resources in the logic fabric are shown in Table 4.1. The FPGA is composed of slices, Configurable Logic Blocks (CLBs), and Input/Output Blocks (IOBs) for interfacing is showed in Figure 4.3.

The CLBs are a small, regular groupings of logic elements, which are organized

Figure 4.3: The PL structure of Zynq-7000



Source:(CROCKETT et al., 2014)

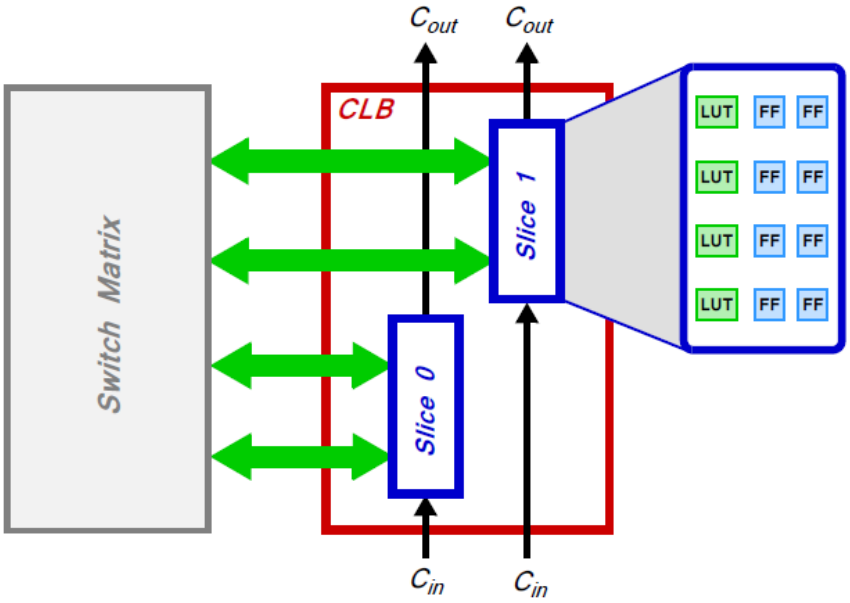
Table 4.1: The Zynq Logic Fabric Resources

Device Name	**Zynq-7020**
Xilinx 7 Series Programmable Logic Equivalent	**Artix-7 FPGA**
Programmable Logic Cells	85k
Look-up Tables (LUTs)	53,200
Flip Flops	106,400
Block RAM (# 36 KB Blocks)	4.9 Mb (140)
DSP Slices	220

Source: (XILINX, 2017)

in a two-dimensional array on the PL. They are connected with other resources via programmable interconnects and are positioned near a switch matrix and contains two logic slices.

Figure 4.4: The Configurable Logic Block (CLB)



Source:(CROCKETT et al., 2014)

The Figure 4.4 shows that the CLBs are composed of slices, Lookup Table (LUT), Flip Flops (FF), IOBs, and Carry Logic. The LUT is a flexible resource that can be used for implementing a small Read-Only Memory (ROM), a small Random Access Memory (RAM), a shift register, or a logic function of up to six inputs.

The Slice is a CLB sub-unit in which contain the resources for implement sequential and combinational logic circuits. They are composed of 4 Lookup Tables, 8 Flip-Flops, and other logic. The FF is a sequential circuit element with two stables states and can be used to store information. The Switch Matrix makes connections between elements within a CLB and from one CLB to other PL resources. The IOBs provides the interface between the PL logic resources and the physical device pads to the external circuit. The Carry Logic propagates the intermediate signals from the arithmetic circuits to other slices.

Two special purpose components are integrated into the logic array in a column arrangement the Block RAMs for dense memory requirements and the Digital Signal Processor (DSP) DSP48E1 slices for high-speed arithmetic. The Block RAMs can implement

RAM, ROM, and First In First Out (FIFO) buffers and supports Error Correction Coding (ECC).

The Block RAM can store up to 36Kb of information and also be configured as two independent 18Kb RAMs or one 36Kb RAM. An additional setup is the word size of the elements giving flexibility in a manner that a RAM can contain more, smaller elements or fewer, longer elements. Two or more Block RAMs can be combined to make larger capacity memories. In generally they can be clocked at the highest clock frequency by the device.

The DSP is a suited choice for a variety of applications in signal processing and other applications. The DSP uses the multiplex circuit to use the registers flexible and support dynamic alteration of the computation. Many computations can be done using one or more arithmetic operators which are selected via an OPMODE input. These inputs configure the internal multiplexers and determine the arithmetic functionality implemented. Also, the DSP is capable of doing some short SIMD processing. Another way to use the DSP is to perform logical functions like the fundamental boolean operations bit-wise, NOT, AND, OR, NAND, NOR, XOR, and XNOR. Multiple DSPs can be used in a connected way to solve complex arithmetic or to implement floating point arithmetic. The low power consumption and the high-frequency operation makes the DSP very attractive for implement arithmetic circuits.

4.1.3 The PS - PL Interfaces

The main link between the PS and the PL in Zynq parts are the Advanced eXtensible Interface (AXI) interconnects and interfaces. As mentioned before in subsection 4.1.2 the EMIO is also another type of connection between the PS and PL.

The AXI4 is the current version of AXI which is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) 3.0 open standard. The AXI buses are very flexible and are used to connect the processor and other IP blocks in an embedded system. There are three type of buses and choose one of them depends on the desired properties of that connection. The AXI4 is used for memory-mapped links providing the highest performance and supports burst. The AXI4-Lite is also used to memory-mapped but is a simplified protocol only supporting one data transfer per connections and no burst. The AXI4-Stream is used for high-speed streaming data, supports burst transfers of unlimited size, and is suited to direct data flow between source and destination.

A set of nine interfaces with multiple channels is the primary interface between the PS and PL. An interface is a point-to-point connection used to transfer data, addresses, and handshake signals between the master and slave of the system. There are three different types of PS-PL interfaces:(1) The General Purpose AXI is a 32-bit data bus used for low and medium rate communications which have four General purpose interfaces divided in two PS and two PL masters. (2) The Accelerator Coherency Port is a single asynchronous connection between PL and the SCU with a bus width of 64 bits used to achieve coherency between the APU caches and elements in the PL. The PL is the master of the only one interface. (3) The High-Performance Ports is used to high rate communications between the PL and the memory elements in PS and includes FIFO buffers to support burst read and write behavior. The PL is master of all four interfaces.

4.2 Algorithms

A standardized benchmark suite for the reliability evaluation has not yet been established. However, to evaluate the reliability of the microprocessor's register file, it is required that the selected benchmarks stimulate different computational resources (QUINN et al., 2015).

The proposed software benchmark includes a collection of algorithms that are realistic software codes based on standard software benchmarks. In this work, an interruption-based fault injector and a heavy ion setup were used to evaluate the compiler optimization effects on the register file reliability in the follow applications the MxM, Advanced Encryption Standard (AES), Quicksort, Fibonacci, JPEG, and FFT.

The **MxM** is an important algorithm for several applications such as signal and control algorithm, weather forecasting, and finite element analysis (KRÜGER; WESTERMANN, 2003; LIEPE et al., 2010). It is memory bounded and the main operations used are the sum and multiplication. The MxM complexity to multiply two nXn matrices is O(n^3).

The (**AES**) is a specification for the encryption of electronic data. It is a symmetric-key algorithm which means that the same key is used for encrypting and decrypting the data. The design principle of AES is based in a substitution-permutation network and has a fixed block size of 128 bits and a key size of 128, 192, or 256 bits. It is a compute-bounded application, and the main operations are bitwise. AES complexity is O(n).

The **Quicksort** also called partition exchange sort is an efficient algorithm that

uses a divide and conquers strategy to sort efficiently. The algorithm is a comparison sort which means that it can sort any items with a total order relation is defined. Another characteristic of the algorithm is that can be operated in-place on an array and when compared with the merge sort, and heapsort algorithms are up to three times faster than the others. It is a compute-bounded application the main operation is bitwise, and the algorithm is implemented recursively. Quicksort complexity best case performance is O(n log n) and O(n^2) in an exceptional worst case.

The **Fibonacci** is a sequence of numbers where each subsequent number is the sum of the previous two. The Fibonacci numbers are used in various computer algorithms like the Fibonacci search technique, the Fibonacci heap data structure, and the Fibonacci cubes which are used for interconnecting parallel and distributed systems. It is a data application, the main operation is the sum, and the algorithm is implemented recursively. Fibonacci complexity is O(n).

The **JPEG** is a commonly used method of lossy compression for digital images, particularly for those images produced by digital photography. The best usage for this algorithm is for photographs and painting of realistic scenes. The JPEG has a selectable trade off between storage size and image quality. The algorithm can achieve a 10:1 compression with little loss in the quality of the image. The method for compression is based on the Discrete Cosine Transform (DCT) which converts each image from the spatial domain into the frequency domain. It is a mix of compute-bounded and memory bounded application because it uses a large amount of data. JPEG complexity is O(n).

The **FFT** is an algorithm that computes the discrete Fourier transform (DFT) of a sequence. The DFT algorithm separates a sequence of values into components of different frequencies the usage. Directly computing the Fourier transform using the FFT instead of DFT increases the performance hugely especially for long data sets. Digital signal processing, solving partial differential equations, and quick multiplication of larger integers are some of the applications where the FFT is used. It is a memory bounded application and the main operation used is bitwise. FFT complexity is O(n log n).

4.3 Setups

In this work to evaluate the previously described benchmarks, some metrics are used. Table 4.2 shows that the metrics are separated in performance and area. The performance metrics are the number of Instructions, the number of clock cycles, and the

execution time. The area metrics are the memory footprint and the register file usage.

$$ClockCycles = 2 \times (T_{end} - T_{start}) \tag{4.1}$$

The clock cycle metric represents the amount of time between two pulses in an oscillator. The number of clock cycles in a bare metal environment in Zynq is calculated using the Xtime library which uses the global timer in the Zynq whose counts the number of ticks of the system. The Equation 4.1 shows how to calculate the number of the clock cycles which is done subtracting the final number (T_{end}) by the initial number of clock cycles (T_{start}) and the result is multiplied per two because the global timer increases every two clock cycles giving precise results.

$$ExecutionTime = \frac{(T_{end} - T_{start})}{CountsperSecond \times 10^{-6}} \quad [\mu s] \tag{4.2}$$

The execution time represents the time to run an application in microseconds. In the Zynq the execution time is calculated using the Equation 4.2 which is the subtraction of the final number (T_{end}) by the initial number (T_start) of clock cycles. The result is divided by the number of counts per second multiplied by 10^{-6} to obtain the metric in μs.

The Open Virtual Platform (OVP) is used to generate the dynamic disassembly of the application, which makes possible to obtain the total number of instructions.

$$RegUsage = \frac{N_{regused}}{N_{total}} \tag{4.3}$$

The register file usage is the ratio of the number of the registers used in the application divided by the number of registers in the processor core. The number of total registers considered for this calculus is the number of general purpose registers except in the MxM double whose is considered the NEON register plus the general purpose registers.

The memory footprint represents the amount of main memory used or referenced while the application is running. The main memory considered here is the code segment, data segment, heap memory, call stack, and any memory used to store additional data structures such as symbol tables, debugging data structures, and others.

The benchmarks used to evaluate the general purpose registers in this work are MxM, Quicksort, AES, Fibonacci, JPEG, and FFT. Each tested setup uses the O0, O2 and O3 compiler optimizations as shown in Table 4.2.

The first columns of Table 4.2 list, for each selected code, the performance and area usage of each of the three compiler optimizations considered. In most of the cases,

Table 4.2: Compiler Optimization effects on code execution, resources utilization

		Performance Info			Area Usage	
App.	Opt.	# Inst.	# Clock Cycles	Exec. Time (μs)	Mem. Footprint	Reg. File Usage
MxM	O0	271473	40606	1.22×10^{-4}	94768	0.38
	O2	73988	8010	2.40×10^{-5}	94332	0.85
	O3	73515	7550	2.26×10^{-5}	94948	0.85
AES	O0	97782	17958	5.39×10^{-5}	110080	0.38
	O2	33066	4402	1.32×10^{-5}	97972	0.31
	O3	31757	3823	1.15×10^{-5}	101056	0.38
QuickSort	O0	330946	107882	1.62×10^{-4}	48504	0.31
	O2	148302	38104	5.72×10^{-5}	48164	0.92
	O3	166582	40264	6.05×10^{-5}	50492	0.85
FFT	O0	78931	14496	4.35×10^{-4}	94040	0.38
	O2	46928	6782	2.03×10^{-5}	92604	1.00
	O3	46870	6738	2.02×10^{-5}	92668	1.00
Fibonacci	O0	10577461	1131645	3.40×10^{-3}	46516	0.31
	O2	4806704	705765	2.12×10^{-3}	46348	0.62
	O3	4217336	433657	1.30×10^{-3}	47352	0.85
JPEG	O0	5008939	724533	2.17×10^{-3}	112432	1.00
	O2	2687238	251634	7.56×10^{-2}	107764	1.00
	O3	2452638	229917	6.90×10^{-2}	121692	1.00
MxM Double	O3	362370	54202	8.13×10^{-5}	46956	0.15
	O3+	336282	50300	7.56×10^{-5}	46956	0.70

Source: The Author

the execution time is improved when compiler optimizations are applied. In the best case (i.e., MxM from 00 to 02), optimization can reduce $5\times$ the execution time. However, optimizations are not always beneficial. In fact, 03 only slightly impacts the execution time of most of the codes. Quicksort shows a very peculiar behavior, as the 03 optimization appears to be less efficient than 02. This less efficiency in the O3 optimization is not surprising, as Quicksort has a data-dependent control flow. Hence, the compiler cannot predict statically how the code will behave when executed.

In the MxM setup, the inputs used were 20x20 matrices, and the data type was an integer. A decrease of 80.72% in the number of clock cycles was observed from O0 to O2 optimization, the register file usage was increased, and the memory footprint usage was decreased from O0 to O2 optimization. Additionally, it was observed a decrease of 81.41% in the number of clock cycles from O0 to O3 optimization, the register file usage stills the same of the O2 optimization, and the memory footprint was increased above the value in the O0 optimization.

In the AES Setup uses a 128 bits block size and 16 inputs the data type is an integer. It was observed a decrease of 75.49% in the number of clock cycles and a decrease in the register file usage and the memory footprint from O0 to O2 optimizations. Additionally,

there was a decrease of 78.72% in the number of clock cycles from O0 to O3. The register file usage decreases from O0 to O2 and increases in the O3 optimization for the same of O0. The memory footprint increases from O2 to O3 optimization but is lower than the O0 optimization.

In the Quicksort setup, a vector of 400 values was used as input, and the data types were integers values. It was observed a decrease of 64.68% in the number of clock cycles from O0 to O2 optimization, the register file was increased to almost the maximum, and the memory footprint was decreased from O0 to O2 optimization. Additionally, there was a decrease of 62.68% in the number of clock cycles from O0 to O3 which is worst than O2 optimization. The register file usage was decreased from O2 to O3 optimization. The memory footprint was increased in O3 optimization above the memory footprint of O0 optimization.

In the FFT setup, a sequence of a 512 float vector was used as input. It was observed a decrease of 53.22% in the number of clock cycles from O0 to O2 optimization, the usage of the register file was increased to the maximum, and the memory footprint was decreased from O0 to O2 optimization. Additionally, there was a decrease of 53.52% in the number of clock cycles from O0 to O3, the register file usage stills the same of the O2 optimization, and the memory footprint has an insignificant increase.

In The Fibonacci setup calculates the sequence up to the twenty-fifth number and uses the integer data type. It was observed a decrease of 37.64% in the number of clock cycles from O0 to O2 optimization, the register file usage doubles, and the memory footprint was decreased. Additionally, there was a decrease of 61.68% in the number of clock cycles from O0 to O3 optimization, an increase in the register file usage above the O2 optimization, and an increase in memory footprint.

In the JPEG setup, it was used an image with 5207 pixels as input and uses the integer as the data type. It was a decrease 65.27% in the number of clock cycles from O0 to O2 optimization, the register file usage is the maximum in all optimizations, and the memory footprint decreases from O0 to O2. Additionally, there was a decrease of 68.27% in the number of clock cycles from O0 to O3 optimization, and the memory footprint was increased above the O0 optimization.

In order to evaluate the NEON register and the general-purpose registers together, it was used the MxM matrix application with a 10x10 size matrices as input and the data type used was the double. The setup was evaluated using the O3 compiler optimization, and the **O3+**, which is the O3 compiler optimizations plus the manual optimization option

unroll all loops. This option decreases 7.55% the number of clock cycles and the register usage increases as shown in Table 4.2 in MxM Double section. The register file usage increases specifically in NEON registers and the memory footprint stills the same.

The memory footprint, which refers to the amount of main memory that a program requires or references, is barely affected by optimizations meaning that the main modifications induced by compiler optimization regard registers and not main memory accesses.

The number of instructions and clock cycles decreases with the increase of optimization level, except in the Quicksort application. Table 4.2 lists the utilization of the register file to support this statement. When the optimization level is increased, in particular from O0 to O2, the percentage of the register file required by computation is increased (if not yet saturated, as in the case of JPEG). From 02 to 03 the utilization of the register is not always affected, and, as for QuickSort, at times is reduced. This increase in the register file usage is code-dependent, as O3 is not always possible or easily implemented. The compiler, then, tries to increase as much as possible the utilization of the register file to improve performances.

5 RELIABILITY EVALUATION METHODOLOGY

In this chapter, it will be described two different methodologies to analyze the reliability of microprocessors: the accelerated beam experiment and the fault injection. The two distinct methods provide different insights, both of which are essential to evaluate a system reliability fully. The fault injection is limited to a subset of available resources, while the radiation experiments evaluate all the processor core. It is worth noting that a one-to-one comparison between the two methodologies results is not feasible. Beam experiment, in fact, is a sort of black box test in which the particle corruption is only observed when it reaches the application output. As a result, while beam experiments are the only way to gather the realistic error rate of modern computing devices and all resources can be hit and corrupted, it is not possible to exactly identify what resource has been affected. On the contrary fault, injection provides a detailed analysis, as injections are controlled. However, the final result is not realistic as only some resources can be corrupted the probability of injection is equal to all resources. In other words, beam experiments are the only way to evaluate the realistic behavior and error rate of a device or applications and fault injection is required to understand how faults propagate in the circuit.

5.1 Beam experiment vs fault injection

The beam experiment and the fault injection can be characterized based on the following properties: reachability, space controllability, time controllability, repeatability, reproducibility, and non-intrusiveness (ARLAT et al., 2003) which will be discussed in the next paragraphs.

The reachability property is the ability to reach possible fault locations in the ICs that implement the target system. In the beam experiment, the radiation hits the physical device that constitutes the irradiated circuit. In our fault injection, the faults are injected in the register file and the NEON register. While the radiation experiments inject faults by all the device.

The space controllability is the ability to control which of the reachable fault locations are corrupted. The beam experiment has a low space controllability. Faults can be confined to specific blocks, if the rest of the circuit is shielded. In the fault injection, the space controllability has a better control where the faults can be injected, but some

resources are inaccessible in the system.

The time controllability is the ability to control the instant when the fault is injected. In the beam experiment, the time when the particle hits the device cannot be controlled. In the fault injection, the exact clock cycle of the injection can be defined.

Repeatability is the property to repeat experiments with a high degree of accuracy. In the beam experiment, this ability is non-existent due to the low level of control. In our fault injection, the repeatability is high however, it is necessary to know the exact clock cycle which the fault injection happens, the register where the fault was injected, and the bit that was flipped.

The reproducibility is the property to reproduce results statistically for a given setup. The beam experiment has a medium reproducibility level because it is hard to have the same beam configuration. The fault injection has a high reproducibility because it is easy to reproduce the same configuration.

The non-intrusiveness is the property to avoid the use of resources in which they are being tested. The heavy ions experiment has a low non-intrusiveness since the target IC is thinned changing some physical specifications like temperature. The fault injection has a high intrusiveness since the application is interrupted to change the value of the register.

As a consequence of these properties, a comparison between the results of the beam experiment and the fault injection is not feasible. The metrics used to evaluate the two methodologies are described in the next section.

5.2 Reliability Metrics

The cross section (σ) is the main metric to evaluate the sensitivity to radiation of a device and can be measured using four general designs the static, semi-static, semi-dynamic, and dynamic (QUINN, 2014). In this work, it was used the dynamic test where the component is active during irradiation. The dynamic cross section for a particular particle (proton, neutron, heavy-ion, and others) is calculated using the ratio between the number of errors in the output of the system and the fluence of particles hitting the system in conformance with Equation 5.1. The fluence represents the number of particles that hit the device per unit area.

$$\sigma(dynamic) = \frac{N_{errors}}{\phi_{particles}} \quad [cm^2] \tag{5.1}$$

In higher abstraction levels such as fault injection, the Architectural Vulnerability Factor (AVF) is the metric used to calculate the sensitivity to SEU. The AVF represents the probability for a low-level corruption to propagate to the output (MUKHERJEE et al., 2003). In this work, the AVF is calculated as the ratio of the number of errors detected (N_{errors}) in the output of the application and the number of the faults injected (N_{faults}) in the register file as shown in Equation 5.2.

$$AVF = \frac{N_{errors}}{N_{faults}} \tag{5.2}$$

The cross section and the AVF metrics as mentioned before do not consider the execution time and the workload of the applications but only the sensitivity of a resource. To better evaluate the reliability of a system the authors in Rech et al. (2014) introduced the MWBF metric for Graphics Processing Units (GPUs), and in Tambara et al. (2016) the authors successfully applied the MWBF for APSoCs. In this work the MWBF was calculate to have an overall evaluation of optimizations impact on performances and reliability.

$$MTBF = \frac{1}{\sigma_{dynamic}.flux} \quad [h] \tag{5.3}$$

In order to calculate the MWBF, it is necessary to consider the Mean Time Between Failures (MTBF) of a system. The MTBF in this work is considered as the average time between two radiation-induced failures on the system continuously executing a given application and is calculated using the Equation 5.3 which takes into account the flux of hitting particles and the dynamic cross section ($\sigma_{dynamic}$). The cross section and MTBF are inversely proportional which means a higher cross section implies in a shorter MTBF considering a constant flux.

$$MEBF = \frac{MTBF}{t_{execution}} \quad [executions] \tag{5.4}$$

Also, the Mean Execution Between Failures (MEBF) is considered to calculate the MWBF. The MEBF is the number of corrected executions completed between two radiation-induced output errors and is calculated using the Equation 5.4 which takes into account the MTBF and the execution time ($t_{execution}$) of the application. The cross section

and the MEBF depend on the employed resources and how efficiently they are used.

$$MWBF = MEBF.w = \frac{w}{\sigma_{dynamic}.flux.t_{execution}} \quad data \tag{5.5}$$

The MWBF, defined as the amount of data computed correctly before experiencing an output error, evaluates the trade-off between performances and error rate (RECH et al., 2014). Finally, the MWBF is calculated using the Equation 5.5 which takes into account the MEBF and the Workload (w). A higher MWBF implies that more corrected data is computed before a failure.

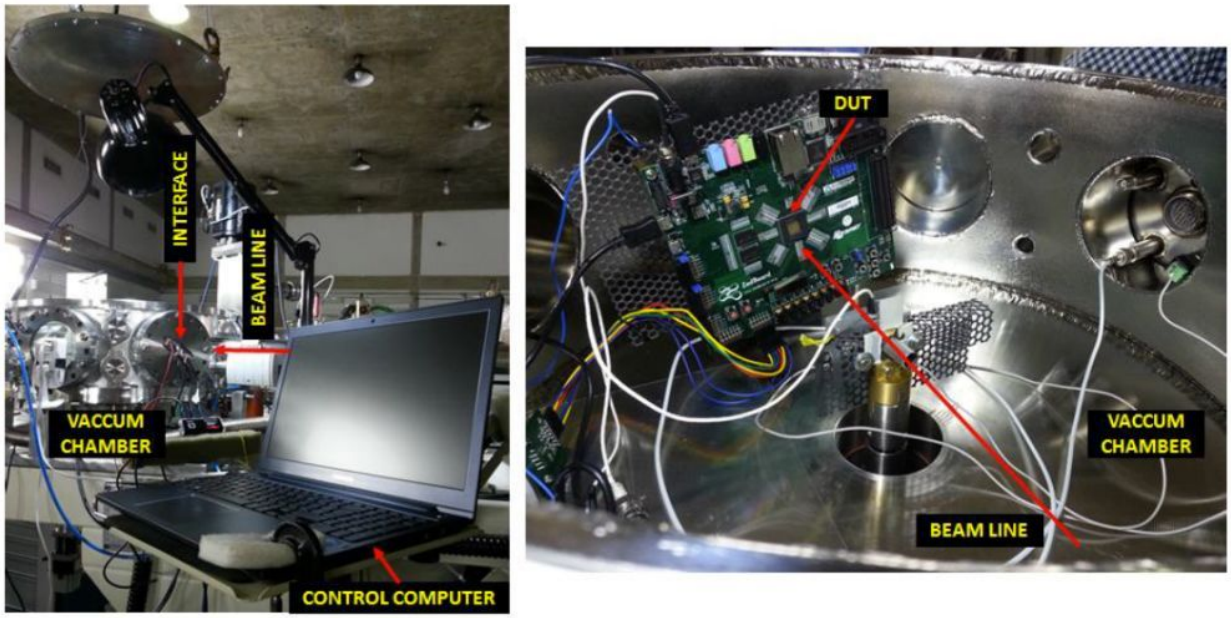
The MWBF also can be calculated using the AVF instead of the dynamic cross section. This calculus can be done because the data must be loaded in registers to be processed. As a result, the higher the AVF of a register, the higher the probability that a corrupted data loaded from the cache of that register will generate an output error.

5.3 Heavy ion experiment

While extremely precise in evaluating the probability of injections to affect the output, fault injection is still limited to a subset of available resources. In order to have a realistic evaluation of the device reliability, it is necessary to perform radiation experiments. By inducing failures in all the resources with realistic error probabilities, radiation experiments provide a reasonable prediction of the error rate of the device in a real application environment. The benchmarks evaluated under radiation are the MxM and the AES with the O0 and O3 optimizations levels which run in the ARM Cortex-A9 as a bare metal application.

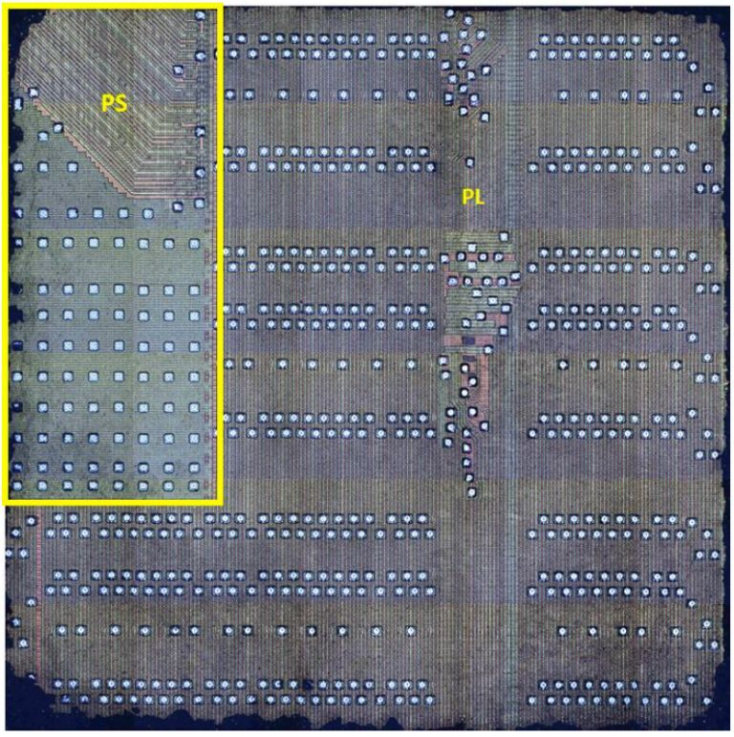
Radiation experiments were carried out with heavy ions at Laboratório Aberto de Física Nuclear at Universidade de São Paulo (LAFN-USP), Brazil (AGUIAR et al., 2014). The ion beams were produced and accelerated by the São Paulo 8UD Pelletron Accelerator. A standard Rutherford scattering setup was included using a gold foil to achieve a very low particle flux in the range from 10^2 to 10^5 $particles.cm^{-2}.s^{-1}$, as recommended by the European Space Agency (ESA) for SEU tests (TAMBARA et al., 2015). The experiment was performed in the vacuum as shown in Figure 5.1. The SEU events were observed using a 12C beam scattered by a 184 $\mu g.cm^2$ gold target, with an energy of 35 MeV, which provides an efficient LET on the active region of 4.35 $MeV.mg^{-1}.cm^{-2}$ and penetration in Si of 31 μm. In order to achieve the desired particle flow, the DUT was

Figure 5.1: Heavy ion experiment setup mounted at the beam line of the LAFN-USP



Source: (TAMBARA, 2017)

Figure 5.2: View of the surface of a Zynq-7000 device



Source: (TAMBARA, 2017)

positioned at a scattering angle of 90°, which resulted in an average flux between $1.4x10^2$ and $2.7x10^2$ $particles.cm^{-2}.s^{-1}$. This configuration was chosen based on several trials and was the most suitable concerning particle flux and number of errors. The package of an XC7Z020-CLG484 device was thinned to allow irradiated particles to penetrate the active region of the silicon as shown in Figure 5.2.

A host computer is used to monitor the experiments through a serial interface. After each execution of the application if there were no mismatch between the golden copy and the output of the application the Control DUT sent an alive signal to the computer. On the contrary, if a mismatch happens the Zynq-7020 is reset. A watchdog circuit in the Control DUT verifies if the system timeout. The host computer also has a watchdog to monitor timeout occurrences in the Control DUT.

5.4 Fault injection framework

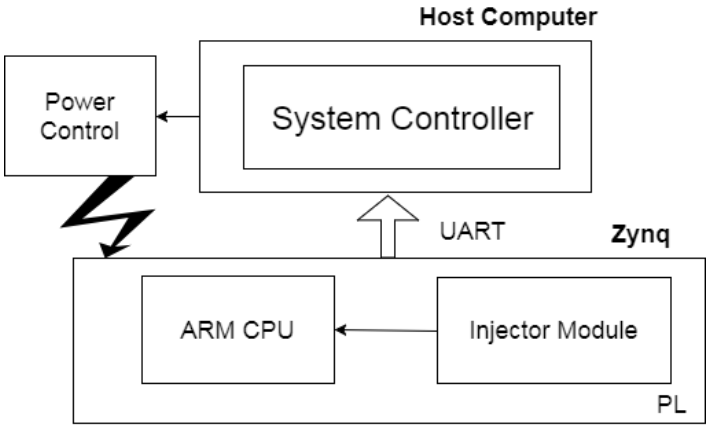
Fault injection can be performed at different levels of abstraction, from Register-Transfer Level (RTL) to software level (SAGGESE et al., 2005). As the RTL level descriptions of COTS devices are not publicly available, faults should then be injected in user-accessible resources, like register files.

Several previous works studied the sensitivity and vulnerability of register files, using fault injection or radiation experiments (ROMER; TROGER, 2011; YAN; ZHANG, 2005). Our fault injector differs from previous ones as it uses interruptions, which make it fast and easily implemented in embedded processors. Our analysis improves the knowledge on register file vulnerability by considering register criticality and by evaluating the effect of compiler optimizations on register and code vulnerability using bare-metal applications.

In order to evaluate registers criticality on the available embedded RISC processor, an interrupt-based fault injector platform able to modify the values stored in the ARM internal registers was built. Injection is triggered by an interrupt service routine which launches a dedicated procedure that modifies a randomly selected register value.

In the literature, several interesting fault injection approaches work at RTL level (like Amuse (ENTRENA et al., 2012)), instruction level (like IVM (MANIATAKOS et al., 2011)), or simulation (like gem5 (PARASYRIS et al., 2014)). Injecting faults in real hardware have several advantages compared to instruction-level or RTL simulation injections. (1) The fault injection experiment is faster because the application under test is

Figure 5.3: Fault Injector First Setup



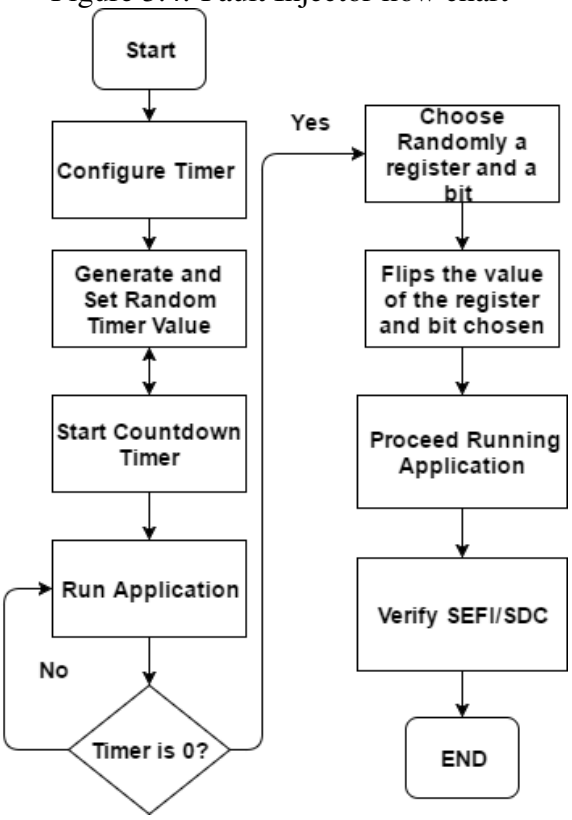
Source: The Author

executed on the hardware device itself. In our fault injection, only one interrupt service routine is triggered and can inject approximately $1,200$ per hour (2) Fault injection at instruction level can be faster than low-level injection allowing the study of large workload (KALIORAKIS et al., 2015; ROSA et al., 2015). However, results are related to software or algorithm vulnerability, without considering the architecture in which the code is executed (SRIDHARAN; KAELI, 2008). As a result, instruction level injection may not be sufficiently precise. (3) Gate-level injections are extremely precise, but they require the RTL level description of the circuit, which is hardly available for COTS systems. Finally, even if the RTL level description is available, an exhaustive fault injection at gate level is extremely time-consuming.

The authors in Carreira, Madeira and Silva (1998) investigated a system-level failure injection. They injected failures into the main internal processing units. The results of the injection are exceptions detected, output errors, and program exit codes. The authors in Arlat et al. (2003) compare three physical techniques and one software-implemented technique, analyzing the impact of the fault injection techniques on the MARS architecture. This analysis showed that the four fault injection techniques are complementary generating different types of errors. Our work injects faults in the register file aiming to analyze the results more deeply like the AVF for each register, the application execution time, and the number of reads and writes in the memory.

Our first setup of the fault injector developed in this work is implemented at software level as shown in Figure 5.3, and is based on the previous work of (VELAZCO; REZGUI; ECOFFET, 2000). Several updates were done in this fault injector. The system is composed of the following modules the Power Control, the System Controller, and the Injector Module. The Power control is an electrical device responsible for the power

Figure 5.4: Fault Injector flow chart



Source: The Author

reset of the board. The System Controller is a software application which is responsible for storing the logs and for managing the Power Control. The Injector module is part of this setup is part of the application software which is implemented in the PS part of the Zynq-7020 and is responsible for the fault injection in the registers.

In order to access all the available registers, the assembly of the Board Support Package (BSP) of Zynq-7000 was modified to store the system context when the interrupt is triggered. The assembly has been changed to store all available registers when the interrupt service routine was triggered. Additionally, to ease the access to the stored registers for injection, the BSP also stores the address of R0 in the Double Data Rate (DDR) memory. When the context is resumed (after injecting the bit flip in one of the registers), the function in the assembly code retrieves all the registers that were stored when the interrupt service routine was triggered. This technique allows injecting faults in registers that are accessible by the application and memory. In the first setup, it is not possible to inject faults in the Stack Pointer (SP), and Program counter (PC) registers because the technique of fault injection is at the software level not being possible to detect the faults.

The flowchart of our interruption-based fault injector is shown in Figure 5.4. The

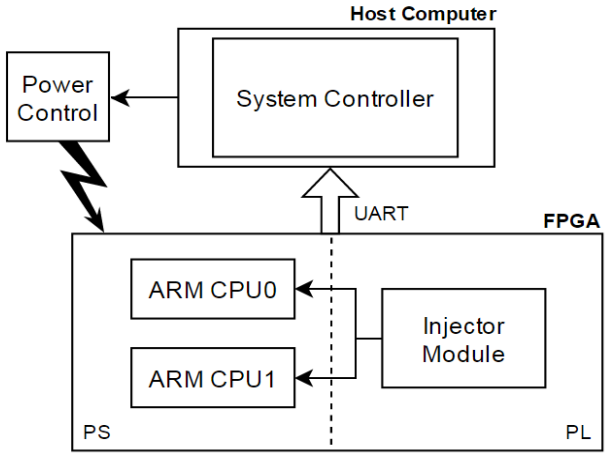
main steps performed in order to inject faults and measure the register criticality are:

(1) Configure the timer: Set a random value for injection, which is the clock cycle number at which the interrupt service routine is triggered.

(2) Execute the application and inject fault: A pre-computed input vector is sent to ARM. Application execution is triggered, and at each clock cycle, the timer is decreased. When the counter reaches 0, the interrupt service routine is triggered, which randomly selects a register and one bit. The selected bit is flipped, the system restores the context and proceeds with application execution. It is worth noting that, while not considered in this paper, the adopted procedure allows to inject also multiple bits flips or to use other error models.

(3) Error detection: When the application execution is completed, the output is compared with a pre-computed golden copy stored in the main memory if they are equal the result is an Unace. A mismatch between the two is identified as a Silent Data Corruption (SDC). Besides the corrupted and expected output value, it is generated a log which contains the register where the fault was injected. Whenever ARM becomes unresponsive or sends garbage through serial communication, an SEFI is detected by a watchdog in the host PC, and the system is rebooted.

Figure 5.5: Fault Injector Modifications of (OLIVEIRA; TAMBARA; KASTENSMIDT, 2017)



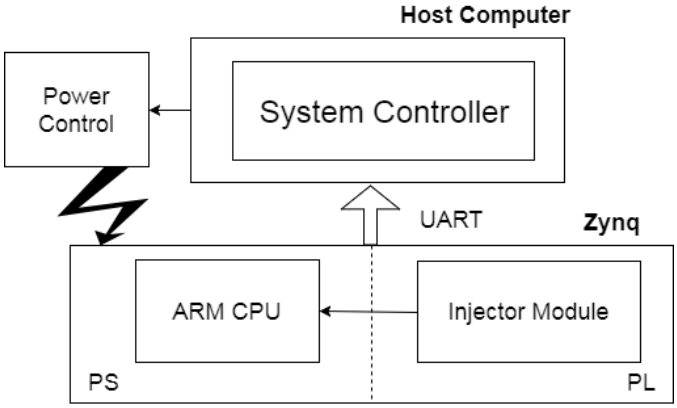
Source: (OLIVEIRA; TAMBARA; KASTENSMIDT, 2017)

The random numbers in our fault injector are generated using the Xtime_GetTime function of the Xtime library. The resulting random number is used to choose the instruction and the bit where the fault will be injected.

In Oliveira, Tambara and Kastensmidt (2017) the authors use the same approach to inject faults in both cores of ARM cortex-A9 in the zynq, but part of the Fault Injector

now is on the PL part of zynq as shown in Figure 5.5. The PS part is still responsible for the fault injection setup which chooses a random clock cycle. After that, a random register where the fault will be injected, and a random bit of the register is chosen. The interruption on the ARM core to change the value of the register chosen before in the setup, and stores the pre-computed golden copy in a Block Random Access Memory (BRAM) before the main application starts. The PL part is responsible for generating the interruption signal to the ARM-core in the correct clock cycle of the application. To compare the results between the golden copy in BRAM and the output of the application which is stored in another BRAM, to send the logs to the host computer using the UART port, and to verify using a watchdog timer when the application finishes the execution. Using this approach the fault injection can be done in all general purpose registers because now the comparison between the pre-computed data and the output of the application is made outside of the ARM.

Figure 5.6: Fault Injector Final Setup



Source: The Author

Our final Fault Injector setup uses a modified version of the injector used in Oliveira, Tambara and Kastensmidt (2017). Some changes were done in the design of the Injector module to inject faults only in a single-core of the ARM Cortex-A9 as shown in Figure 5.6. Now the metrics such as the register where the fault is injected, the bit where the bit flip happens, and the clock cycle of the fault injection are sent via UART to the host computer. In this final version, the fault injection can be done in all general-purpose registers and all NEON registers.

6 RESULTS

In this chapter is presented and discussed the fault-injection and the heavy ions experiments results. The following metrics are used to analyze the reliability: the Cross Section, the AVF, and the MWBF. The fault injection and the heavy ions experiment are the methodologies used to obtain the data to calculate the metrics. The analysis of these results is of extreme importance to evaluate the reliability for each application and configuration tested in this study.

6.1 Fault Injection Results

In this section, the data from two experiments is presented. In subsection 6.1.1 is analyzed the effects of compiler optimization in the reliability of the application injecting faults in the general purpose registers. The objective is to analyze the effect of the compiler optimization in the tested benchmarks and the different compiler optimizations. In subsection 6.1.2 is analyzed the effects of compiler optimization in the reliability of the application injecting faults in the NEON and the General purpose registers. The objective is to evaluate the effect of the NEON engine reliability of the tested benchmark with different compiler optimizations.

To correlate the observed reliability variation caused by the compiler optimizations a more detailed analysis is performed in the metrics of performance, register file usage, and memory footprint. The methodology described in the Section 5.4 is applied. The reports are showed in Table 6.1 in the Area and Performance metrics sections. These reports was showed and explained in Table 4.2. Table 6.1, also reports the following metrics the AVF, and the MWBF reliability explained in Section 5.1 for all the tested codes and configurations. The best result column in Table 6.1 shows how many times the best MWBF is better than the worst MWBF in the tested application.

The analysis of Table 6.1 was done evaluating how each compiler optimization changes the metrics for each application tested. The MWBF is calculated using the metrics obtained for each compiler optimization. The highlighted numbers in the table represent the best values for each metric in the application. These numbers will be used to analyze the correlation between these metrics and the best reliability optimization for each application.

All the AVF graphs in this section are based on the tables shown in Appendix A.

Table 6.1: Compiler Optimization effects on code execution, resources utilization, and reliability

		Performance Info			Area Usage		Fault injection		Reliability Metric	
App.	Opt.	# Inst.	# Clock Cycles	Exec. Time (μs)	Mem. Footprint	Reg. File Usage	AVF SDC	AVF SEFI	MWBF	Best Result
MxM	**O0**	271473	40606	1.22×10^{-4}	94768	**0.38**	**0.12**	**0.00**	**1083382141.45**	**83.65**
	O2	73988	8010	2.40×10^{-5}	**94332**	0.85	0.41	0.13	12951309.16	
	O3	**73515**	**7550**	$\mathbf{2.26 \times 10^{-5}}$	94948	0.85	0.36	0.07	16644578.35	
AES	O0	97782	17958	5.39×10^{-5}	110080	0.38	0.12	0.03	1745984.62	
	O2	33066	4402	1.32×10^{-5}	**97972**	**0.31**	**0.03**	**0.11**	**7098840.56**	**4.06**
	O3	**31757**	**3823**	$\mathbf{1.15 \times 10^{-5}}$	101056	0.38	0.04	0.19	6029469.18	
QuickSort	O0	330946	107882	1.62×10^{-4}	48504	**0.31**	**0.04**	**0.03**	11735117.31	
	O2	**148302**	**38104**	$\mathbf{5.72 \times 10^{-5}}$	**48164**	0.92	0.35	0.03	**18529570.01**	**1.57**
	O3	166582	40264	6.05×10^{-5}	50492	0.85	0.36	0.03	16749253.42	
FFT	**O0**	78931	14496	4.35×10^{-4}	94040	0.38	**0.06**	**0.03**	**16668641.13**	**1.65**
	O2	46928	6782	2.03×10^{-5}	**92604**	1.00	0.39	0.24	11046401.16	
	O3	**46870**	**6738**	$\mathbf{2.02 \times 10^{-5}}$	92668	1.00	0.36	0.23	12238299.75	
Fibonacci	O0	10577461	1131645	3.40×10^{-3}	46516	0.31	**0.08**	**0.03**	18862.96	
	O2	4806704	705765	2.12×10^{-3}	**46348**	0.62	0.23	0.07	13112.55	
	O3	**4217336**	**433657**	$\mathbf{1.30 \times 10^{-3}}$	47352	0.85	0.12	0.07	**22513.11**	**1.71**
JPEG	**O0**	5008939	724533	2.17×10^{-3}	112432	**1.00**	**0.11**	**0.06**	**4663876.02**	**96.11**
	O2	2687238	251634	7.56×10^{-4}	**107764**	**1.00**	0.40	0.13	54288.85	
	O3	**2452638**	**229917**	$\mathbf{6.90 \times 10^{-4}}$	121692	**1.00**	0.34	0.18	48522.93	
MxM Double	**O3**	362370	54202	8.13×10^{-5}	**46956**	**0.15**	**0.03**	**0.02**	**126155107.70**	**1.16**
	O3+	336282	**50300**	$\mathbf{7.56 \times 10^{-5}}$	**46956**	0.70	0.06	0.02	108533441.87	

Source: The Author

These tables show the data obtained from the fault injection for each optimization level applied in the tested benchmarks.

6.1.1 General purpose Register Setup Results

In order to evaluate the general purpose registers, as mentioned in the Chapter 4 Section 4.3, the benchmarks used are the MxM, the AES, the Quicksort, the FFT, the Fibonacci, and the JPEG applications and evaluate the O0, O2 and O3 optimizations levels.

As shown in Table 6.1 the MxM application has the best performance for the O3 optimization level which is 5.37 times faster than O0. The register file usage increases 2.23 times from O0 to O2 and remains the same in O3 optimization. The results obtained in the fault injection is shown using the AVF SDC metric which increases 3.41 times from O0 to O2 and decreases 0.87 times from O2 to O3 optimization level which is still higher than O0. Another metric obtained in the fault injection is the AVF SEFI metric which increases 13 times from O0 to O2 and decreases 0.53 times from O2 to O3. The highest MWBF is achieved with the O0 optimization which is 83.65 times more reliable than the

worst MWBF which is the O2 optimization.

While the highest MWBF for MxM is achieved with the O0 optimization, as shown in the highlighted numbers in Table 6.1 the best performance is achieved with the O3 optimization and the best memory footprint with the O2 optimization. As a consequence of it this means that even if O3 provides higher performances and O2 a highly optimized code, compiler optimizations are not sufficient to balance the error rate increase they bring. However, the smaller value of the sum of the AVF SDC and SEFI metrics and a lower usage of the register file is achieved in O0 optimization. The register file usage metric does not directly affect the MWBF calculation.

In Table 6.1 the AES application has the best performance in the O3 optimization level which is 4.69 times faster than the O0. The minor register usage is found in the O2 optimization which is 1.22 times better than the O2 and O3 optimization level. The best memory footprint is found in the O2 optimization which is 1.12 times better than O0 and 1.03 times better than O3 optimization. The AVF SDC metric decreases 0.75 times from O0 to O2 optimization and increases 1.33 times from O2 to O3 optimization which stills smaller than O0. The AVF SEFI metric which increases 3.66 times from O0 to O2 and increases 1.72 times from O2 to O3. The highest MWBF is achieved with the O2 optimization which is 4.02 times more reliable than the worst MWBF which is the O0 optimization.

While the highest MWBF for AES is achieved with the O2 optimization, as shown in the highlighted numbers in Table 6.1 the best performance is achieved in the O3 optimization. However, the O2 optimization has the best memory footprint, a smaller usage of the register file, and a lower value in the result of the sum of the AVF SEFI and SDC.

It is worth noting that AES is a control-flow based algorithm that performs data filtration. As such, AES is intrinsically less prone to experience SDCs than MxM is supported by the fact that, for O2 and O3 implementations, the SDC AVF for AES is 10 times lower than the AVF of MxM. The peculiarity of being control-flow-based makes AES prone to experience SEFIs. The SEFI AVF for AES compiled with O2 and O3 is about 3.50 times the AVF for SDCs. This difference between the number of SDCs and SEFIs may happen because the O0 implementation of AES is too naive to produce a representative code. Eventually, this is due to high memory latency, which reduces SEFI probability and increases SDC probability. In fact, while waiting for data the ARM is in an idle state, and no operation is executed. So, data is exposed, and corruption may lead to SDC, although unlikely to SEFI. There are rather fewer SDCs in AES, but this only

occurs because these are converted into SEFIs. So, in the end, the amount of AES errors increases also with the growth of optimization level.

In Table 6.1 the Quicksort application has the best performance in the O2 optimization level which is 2.83 times faster than the O0. The minor register usage is achieved in the O0 optimization which increases 2.96 times from O0 to O2 optimization and decreases 0.92 times from O2 to O3 optimization which stills higher than O0. The AVF SDC metric increases 8.75 times from O0 to O2 and increases 1.03 times from O2 to O3 optimization level. The AVF SEFI metric is equal for all optimizations. The highest MWBF is achieved with the O2 optimization which is 1.57 times more reliable than the worst MWBF which is the O0 optimization.

While the highest MWBF for Quicksort application is achieved in the O2 optimization, as shown in the highlighted numbers in Table 6.1 the minor register file usage, and the sum of AVF SDC and SEFI is smaller in O0 than in the other optimizations. However, the O2 optimization has the best performance and the best memory footprint. The compiler does not know $apriori$ Quicksort is a control flow application, which is why O3 is not very efficient.

In Table 6.1 the FFT application has the best performance in the O3 optimization level which is 2.15 times faster than the O0. The minor register usage is achieved in the O0 optimization which increases 2.63 times from O0 to O2 and O3 optimizations. The AVF SDC metric increases 6.5 times from O0 to O2 and decreases 0.92 times from O2 to O3 optimization which stills higher than O0. The AVF SEFI metric increases 8.0 times from O0 to O2 and decreases 0.96 times from O2 to O3 optimization which stills higher than O0. The highest MWBF is achieved with the O0 optimization which is 1.65 times more reliable than the worst MWBF which is the O2 optimization.

While the highest MWBF for FFT application is achieved in the O0 optimization, as shown in the highlighted numbers in Table 6.1. The best memory footprint is achieved in the O2 optimization, the minor register file usage is reached in O0 optimization, and the sum of AVF SDC and SEFI is smaller in O0 than in the other optimizations. However, the best performance is achieved in the O3 optimization which is almost the same of the O2 optimization.

In Table 6.1 the Fibonacci application has the best performance in the O3 optimization level which is 2.60 times faster than the O0. The minor register usage is achieved in the O0 optimization which increases 2.00 times from O0 to O2 optimization and increases 1.37 times from O2 to O3 optimization. The AVF SDC metric increases 2.87

times from O0 to O2 and decreases 0.52 times from O2 to O3 optimization level which stills higher than O0. The AVF SEFI metric increases 2.3 times from O0 to O2 and stills the same in the O3 optimization. The highest MWBF is achieved in the O3 optimization which is 1.71 times better than the worst MWBF which is the O2 optimization.

While the highest MWBF for Fibonacci application is achieved in the O3 optimization, as shown in the highlighted numbers in Table 6.1. The best memory footprint is achieved in O2 optimization, the minor register file usage is reached in the O0 optimization, and the sum of AVF SDC and SEFI is smaller in O0 than in the other optimizations. However, the best performance is achieved in the O3 optimization.

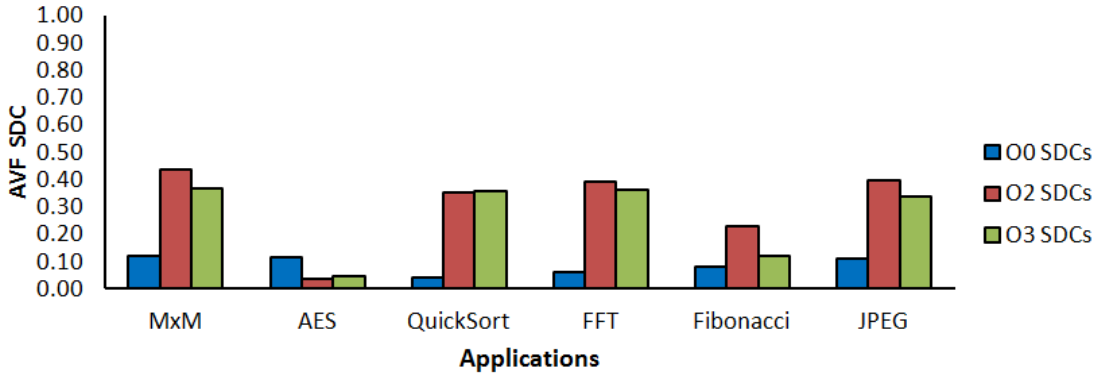
In Table 6.1 the JPEG application has the best performance in the O3 optimization level which is 3.15 times faster than the O0. The best memory footprint is achieved in the O2 optimization which is 1.12 times better than O3 and 1.04 times better than O0 optimization. The register usage is equal for all optimizations. The AVF SDC metric increases 3.63 times from O0 to O2 and decreases 0.85 times from O2 to O3 optimization level which stills higher than O0. The AVF SEFI metric increases 2.16 times from O0 to O2 and increases 1.38 times from O2 to O3 optimization. The highest MWBF is achieved in the O0 optimization which is 96.11 times more reliable than the worst MWBF which is the O3 optimization.

While the highest MWBF in the JPEG application is achieved in the O0 optimization, as shown in the highlighted numbers in Table 6.1 the best performance is achieved in the O3 optimization, and the best memory footprint is reached in O2 optimization. However, the sum of AVF SDC and SEFI is smaller in the O0 optimization.

The best memory footprint for all analyzed applications is found in O2 optimization, but the difference is less than 1.0015 times between the worst optimization making this not so relevant except in the AES and JPEG applications where this difference is up to 1.012 times.

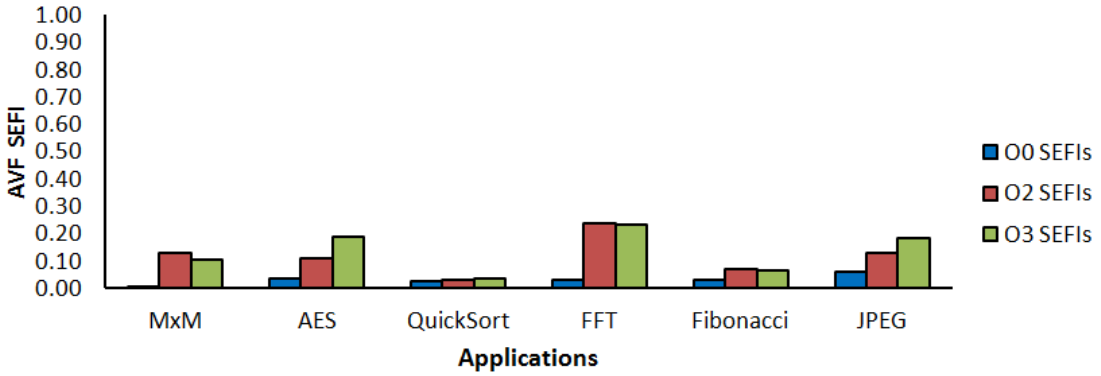
Analyzing the Table 6.1 for all tested applications shows that there is a trade-off between the AVF and the execution time. The configuration with the lowest AVF or the lowest execution time does not always bring the highest MWBF. The last column of Table 6.1 reports how much the MWBF is improved between the worst and the best case. It was observed that a higher level of optimization does not necessarily imply a higher MWBF. In fact, optimization increases the number of used registers and the exposed area. In order to be beneficial, the performance improvements must be sufficient to compensate the increased exposed area.

Figure 6.1: Total Architectural Vulnerability Factor SDCs



Source: The Author

Figure 6.2: Total Architectural Vulnerability Factor SEFIs



Source: The Author

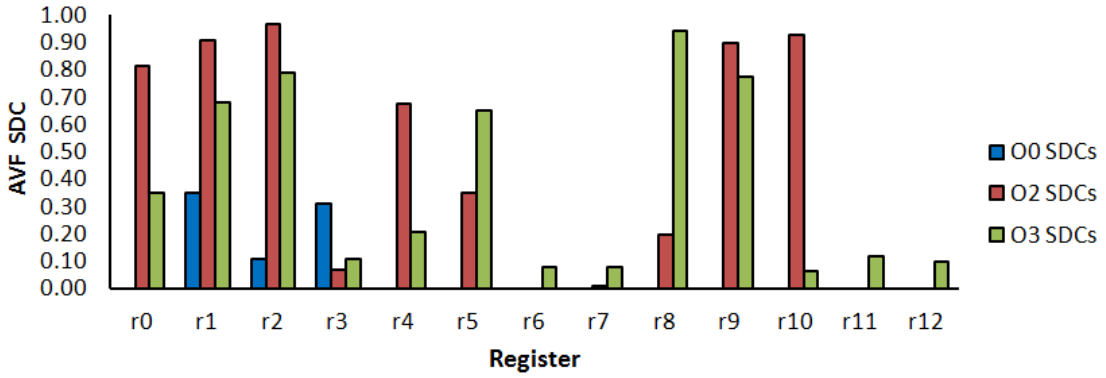
Figures 6.1 and 6.2 show how the AVF SDC and SEFI change for each application with the increase of the compiler optimization. The AVF for both SDCs and SEFIs of most applications increases between O0 and O2. On the contrary, between O2 and O3 the AVF is very similar, in most cases. It is worth noting that, as shown in Table 6.1, the execution time, number of instructions, clock cycles, and resources utilization significantly change with the application of the O2 optimization level instead of the O0. The difference between O2 and O3 is slight which is again mainly caused by the fact that O3 optimization level is not always practical and easy to be implemented. A first-order indication of how much the optimization level modifies the executed code is the execution time and resources utilization.

The Register Lifetime is the time when the data stored is still useful in the register file. Any fault occurring to the register during this period corrupts data integrity. Consequently, the higher the lifetime, the longer the register is prone to faults (RESTREPO-CALLE et al., 2015; MONTESINOS; LIU; TORRELLAS, 2007). As shown in Table 6.1 in the execution time column the O0 optimization obtained the longest execution time for

all applications. However, the Figures 6.1 and 6.2 shows that there is an increase of the AVF with the increase of the level of optimization. Thus, it can be concluded that the increase in AVF is more related to the lifetime of variables than to the execution time of the application. The lifetime and the execution time are not always directly proportional.

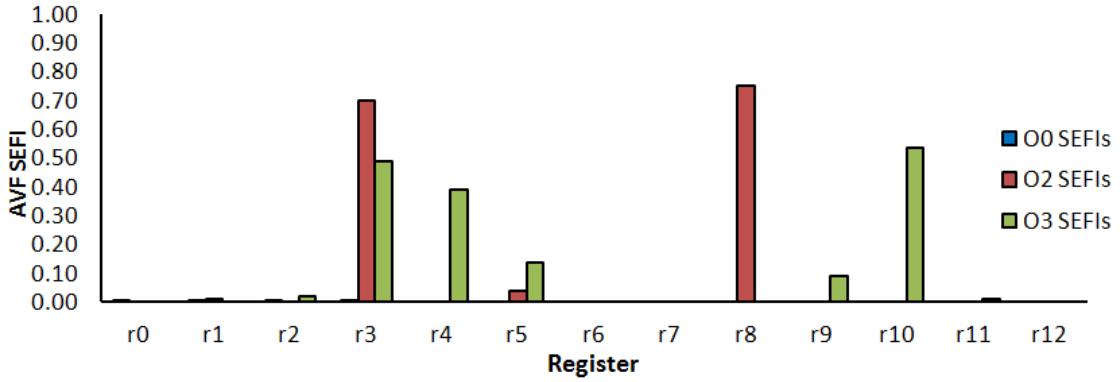
In order to better understand register criticality, it was measured the AVF for SDC and SEFI of each register, for both MxM and AES. Then, it can not be only identified the registers whose corruption is more likely to generate and SDC or an SEFI, but also if the distribution regarding the critical registers number is affected by the optimization.

Figure 6.3: MxM Architectural Vulnerability Factor SDCs



Source: The Author

Figure 6.4: MxM Architectural Vulnerability Factor SEFIs



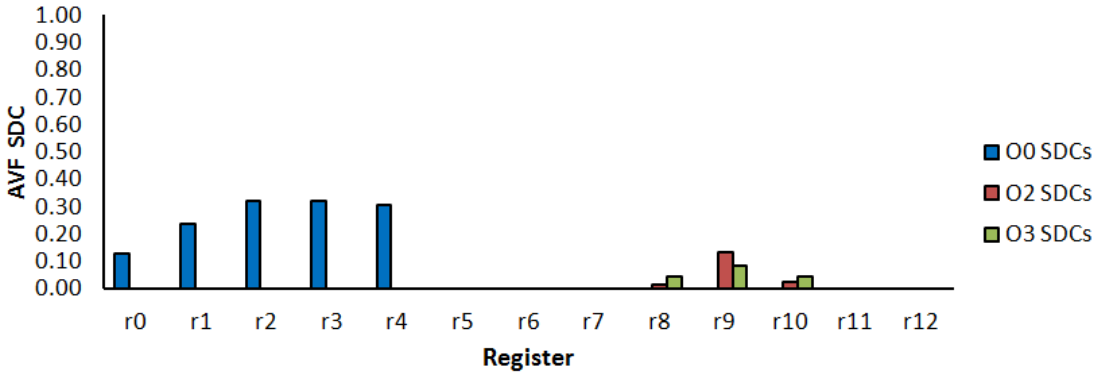
Source: The Author

The AVF SDCs for each register and how they change for each optimization in the MxM application is shown in Figure 6.3. It is clear that O2 and O3 optimizations increase the number of critical registers which is expected, as O2 and O3 try to use all available registers to improve performances. The number of critical registers is 3 in the O0 optimization, increases to 10 in 02 optimization, and 13 in O3 optimization. The AVF is lower in O0 than in others optimization. However, 6 out of 10 registers (i.e., R0, R1,

R2, R4, R9, and R10) that are critical in both O2 and O3 have a much lower AVF in O3, which is why SDC AVF for MxM is similar between O2 and O3.

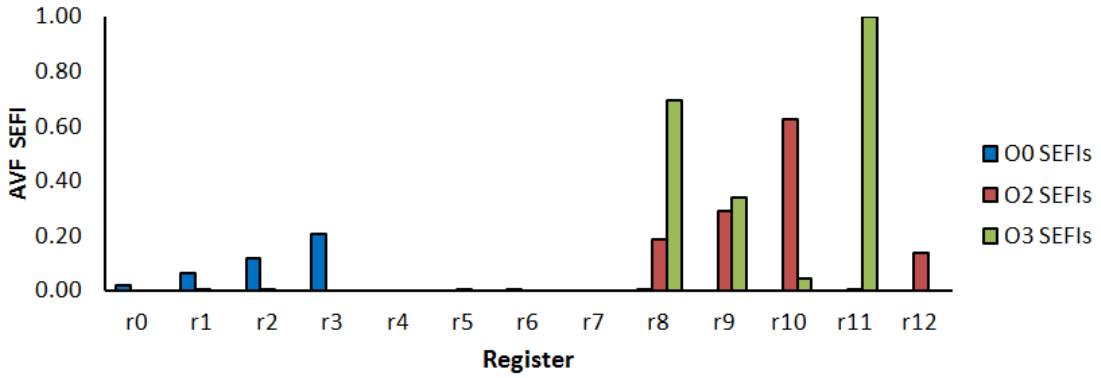
The AVF SEFI for each register in MxM application is shown in Figure 6.4. In O2 and O3 optimizations a significant increase in the AVF SEFI can be seen in accordance with data in Table 6.3. As for SDCs, O2 increases the AVF of some registers significantly, while O3 distributes the AVF SEFI among most of the available registers. The number of critical registers in O0 is 0 in O0 optimization and increases to 3 in O2 optimization and 5 in O3 optimization. The AVF is lower in O0 than in other optimizations.

Figure 6.5: AES Architectural Vulnerability Factor SDCs



Source: The Author

Figure 6.6: AES Architectural Vulnerability Factor SEFIs



Source: The Author

The AVF SDCs for each register and how they change in each optimization in AES application is shown in Figure 6.5. The number of critical registers is 5 in the O0 optimization, decreases to 3 in 02 and O3 optimization. Interestingly, in contrast with MxM, AES O2 and O3 reduce not only the number of critical registers but also their AVF. As mentioned before the O0 optimization for AES can be too naive, eventually masking the filter characteristic of the code.

The AVF SEFIs for each register in AES application is shown in Figure 6.6. The number of critical registers is 4 for all optimizations in spite the registers change. The AVF SEFI increases in O2 and O3 optimizations, and because the optimization does not use all the available registers, the AVF is concentrating in fewer registers. As showed in Table 6.1 the register usage also depends on the size of the application and their inputs. Applications such as FFT, JPEG use all the registers in the lowest optimization levels.

Table 6.2: Fault Injection Results

Application	Optimization	Execution Time	AVF SDC	AVF SEFI	MWBF SDC	MWBF SEFI
MxM	**O0**	2.1 ms	0.12	0	6.32×10^6	∞
	O2	0.8 ms	0.44	0.13	4.58×10^6	1.57×10^7
	O3	0.6 ms	0.37	0.11	7.28×10^6	2.54×10^7
AES	**O0**	5.2 ms	0.12	0.03	2.63×10^6	9.12×10^6
	O2	0.5 ms	0.03	0.11	9.23×10^7	2.96×10^7
	O3	0.4 ms	0.04	0.19	9.09×10^7	2.15×10^7

In Table 6.2 it was analyzed the fault injection results measuring the MWBF SDC and SEFI separately. The execution time used in this analysis starts with the power of the board and finishes when the application finished, which is the same considered in the radiation experiments. The AVFs in this table is the same of the Table 6.1 which was analyzed in the previous paragraphs.

In the MxM application the execution time decreases 2.62 times from O0 to O2 optimization and decreases 1.33 times from O2 to O3 optimization. The best MWBF SDC is found in the O3 optimization which is 1.59 times better than the worst MWBF which is the O2 optimization. In the MWBF SEFI in O0 optimization is found a ∞ which symbols that no error was found in the injection. In the O2 and O3 optimization, the highest MWBF is achieved in O3 which is 1.61 times better than O2. Considering the sum of both MWBF SDC and SEFI the O2 optimization stills the highest MWBF for the MxM application.

In the AES application, the execution time decreases 10.4 times from O0 to O2 optimization and decreases 1.25 times from O2 to O3 optimization. The highest MWBF SDC is achieved in the O2 optimization which is 35.09 times better than the worst MWBF SDC which is the O0 optimization. The highest MWBF SEFI is achieved in the O3 optimization which is 3.24 times better than the worst MWBF SEFI which is the O0 optimization. Considering the sum of both MWBF the O2 stills the best MWBF for AES

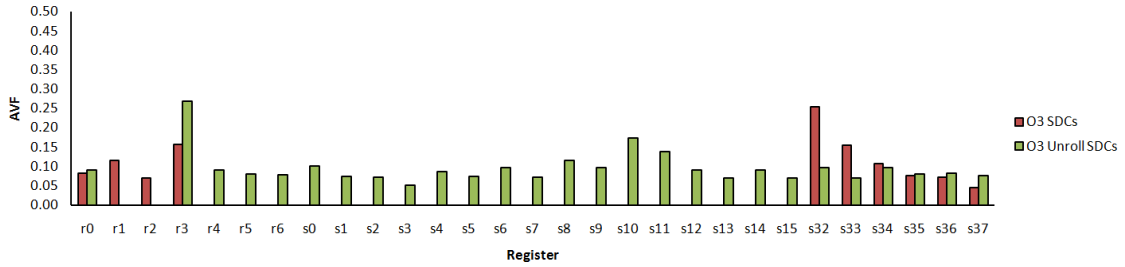
application.

6.1.2 NEON Setups Results

In order to evaluate the general-purpose registers and the NEON registers as mentioned before in the Chapter 4 Section 4.3 the benchmark tested in this experiment is the MxM Double and the optimizations levels evaluated are the O3 and the O3+.

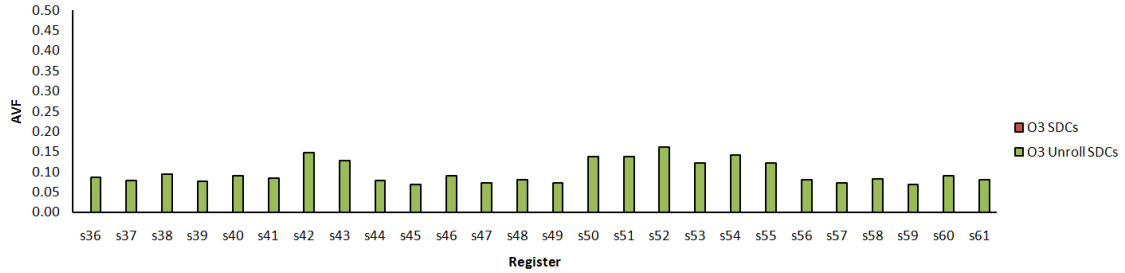
As shown in Table 6.1 the MxM Double application has the best performance in the O3+ optimization level which is 1.08 times faster than O3. The register file usage increases 4.66 times from O3 to O3+. The results obtained in the fault injection is evaluated using the AVF SDC metric which increases 2.00 times from O3 to O3+ optimizations and the AVF SEFI metric which is equal in both optimizations.

Figure 6.7: AVF SDC MxM Double O3



Source: The Author

Figure 6.8: AVF SDC MxM Double O3



Source: The Author

The AVF SDCs for each used register and how they change in each optimization in MxM Double application is shown in Figures 6.7 and 6.8. It is clear that O3 and O3+ application increases the number of critical registers which is expected, as O3 and O3+ try to use all available registers to improve performances. The number of critical registers is 10 in the O3 optimization and increases to 52 in the O3+ optimization.

While the highest MWBF in the MxM Double application is achieved in the O3, as shown in the highlighted numbers in Table 6.1 the O3+ optimization has the best per-

formance, and the O2 optimization has the best memory footprint. However, the minor register file usage, and the sum of AVF SDC and SEFI is smaller in the O3 optimization.

6.2 Heavy Ions Experiment Results

In order to evaluate the effect of the compiler optimizations in the whole system under radiation, it was used as benchmarks the MxM and the AES applications and the optimizations levels evaluated are the O0 and O3 optimizations levels. The methodology employed in this experiment was explained in the Chapter 5 Section 5.3.

Although the increase of optimization level increases the SoC heavy ions cross section, they are beneficial as they reduce the code execution time. As a result, while the probability for one impinging particle to generate a noticeable error increase, the execution time is reduced, reducing the exposure time of the device. As reported in Table 6.3 in the MxM application the execution time decreases 3.5 times from O0 to O3 optimization. The SDC cross section increases 2.25 times from O0 to O3 optimization. The SEFI cross section is considered 0 which means the no SEFI errors happens the cause of this may be the small number of events detected. The highest MWBF SDC is achieved in O3 optimization which is 1.54 times better than O0. The MWBF SEFI is considered ∞ which means the MWBF is not susceptible to SEFI errors or a lower number of events was observed. The obtained data in the heavy ions experiments for the MxM application is in accordance with fault injection.

As reported in Table 6.3 in the AES application the execution time decreases 13.00 times from O0 to O3 optimization. The SDC cross section increases 10.69 times from O0 to O3 optimization. The SEFI cross section decreases 40.59 times from O0 to O3 optimization. The highest MWBF SDC is achieved in O3 optimization which is 1.39 times better than O0. The highest MWBF SEFI is achieved in O3 optimization which is 5.28 times better than O0. On the contrary of MxM application, AES shows the opposite trend. This behavior may happen because the AES trend is biased by the small number of events detected, caused by the intrinsic AES SDC reliability.

Table 6.3: Heavy Ions Experiment Results

Application	Optimization	Execution Time	σ SDC	σ SEFI	MWBF SDC	MWBF SEFI
MxM	**O0**	2.1 ms	$(1.16 \pm 1.50) \times 10^{-5}$	$(0 \pm 1.50) \times 10^{-5}$	1.14×10^{10}	∞
	O3	0.6 ms	$(2.62 \pm 3.14) \times 10^{-5}$	$(0 \pm 3.14) \times 10^{-5}$	1.76×10^{10}	∞
AES	**O0**	5.2 ms	$(8.93 \pm 13.4) \times 10^{-7}$	$(8.93 \pm 13.4) \times 10^{-7}$	5.96×10^{10}	5.96×10^{10}
	O3	0.4 ms	$(8.35 \pm 1.09) \times 10^{-6}$	$(2.20 \pm 2.86) \times 10^{-6}$	8.29×10^{10}	3.15×10^{11}

Source: The Author

6.3 An Analysis of the Dynamic Disassembly Code

The insights provided by our fault-injector on the vulnerability of registers could be used to implement dedicated and efficient hardening strategies. By duplicating or protecting only the registers that are found to be more likely to generate output errors it may be possible to increase the code reliability without introducing useless overhead.

Table 6.4: Number of Reads and Writes in the main memory

Application	Optimization	# Read	# Write
MxM	O0	241761	252813
	O2	55900	56300
	O3	55410	55927
AES	O0	105060	91512
	O2	32894	29070
	O3	30496	29510
Quicksort	O0	312871	265035
	O2	126266	116831
	O3	144746	132375

Source: The Author

Another two important parameters that help us to understand better the obtained results are the numbers of reads and writes to the main memory according to the optimization strategy. Such instructions are very time-consuming, being some of the instructions most affected by the optimization strategies. For this analysis, it was selected the MxM, AES, and QuickSort benchmarks. The obtained results are listed in Table 6.4. The code length analysis and the number of reads and writes in the main memory were performed using the Open Virtual Platform (OVP) analyzing the dynamic disassembly of the code that was executed in ARM model. Confirming the results obtained in (CHIBANI et al., 2014) and commented before in Chapter 3 Section 3.3.

Results show that, in general, the numbers of reads and writes in the main memory are significantly reduced with the increase in the optimization level. Such behavior is explained noting that in O0 data is pre-fetched from the main memory just before the execution of the instruction that needs this data. In contrast, O2 and O3 focus on minimizing memory access time by eliminating redundant accesses to memory to optimize the overall application execution time. O2 and O3 eliminate memory accesses and replace them with shorter latency events, such as register copying and value propagation through registers. As a consequence, O2 and O3 make use of a higher number of registers, which increases the criticality of the registers as well as the overall application sensitivity. However, the reliability of an application does not depend only on its sensitive area. It also depends on the time required to complete its computation correctly. Therefore, it is also mandatory to evaluate the reliability of an application regarding MWBF.

The obtained results are in accordance with the ones obtained in (IBRAHIM; RUPP; HABIB, 2009), although authors investigated the impact of compiler optimizations on power consumption and not on reliability. They observed that O3 decreases the number of memory references by 94%, while the instructions per cycle are increased by 250% and the consumed power by 25%. The increase in the consumed power can be explicated by the fact that, with O2 and O3 the software loop pipelining is enabled which in turn avoids most of the processor stall cycles, resulting in better execution time. Given that processor stall cycles consume lower power than the normal instruction cycles, enhancing the execution time in that way leads to considerable increase in the power consumption.

This chapter aims to analyze the effects of the compiler optimizations in the reliability of the all tested applications and configurations in the ARM Cortex-A9.

From the analysis performed it can be seen that optimizations improve the reliability of the register file, but this is not sufficient to improve the overall processor reliability. However, data must be loaded in registers to be processed. As a result, the higher the AVF of a register, the higher the probability that a corrupted data loaded from the cache of that register will generate an output error.

7 CONCLUDING REMARKS

This work presented a study on the effects of the compiler optimizations in the reliability of embedded microprocessors. As a case of study, this work uses the Zynq APSOC with the embedded ARM Cortex-A9 which is a COTS device. In order to analyze these effects, two experiments methodologies explained before in the chapter 5 are selected the fault injection and the heavy ions experiment. The next sections summarize and discuss the results obtained in this work, and present future works.

7.1 Discussion

In modern devices, the radiation and the fault injection experiments are essential to evaluate the reliability. The most realistic experiment is the radiation which irradiates the overall system with similar particles to those found in harsh environments. In the heavy ions experiments, the irradiated benchmarks are the MxM and the AES. Both tested with the O0 and the O3 optimizations.

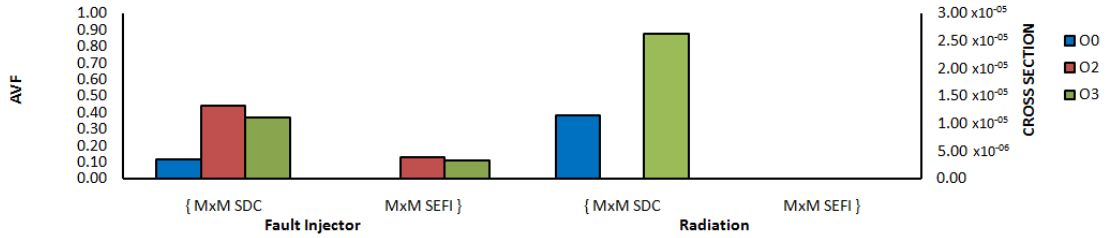
The experiment which has a more detailed evaluation of the effects of the compiler optimization in the reliability is the fault injection. In this experiment, the six benchmarks used are the MxM, AES, Quicksort, FFT, Fibonacci, JPEG with the O0, O2 and O3 optimizations. The faults are injected in the general-purpose registers.

Figures 7.1 and 7.3 shows the results in the fault injection, and in the radiation experiment for the MxM and the AES applications. The AVF was the metric used in the fault injection experiment which represents the sensitivity to SEU. The dynamic cross section was the metric used in the radiation experiment which quantifies the sensitivity of the design to any specific radiation source. The two metrics were separated in SDC and SEFI.

Figures 7.2 and 7.4 shows the result in the fault injection, and in the radiation experiment for the MxM and the AES applications. The MWBF is the metric used to evaluate the impact of the optimizations in the performance and the reliability. In the fault injection, the MWBF was calculated based on the AVF, and in the radiation experiment in the cross section. The MWBF was separated in SDC and SEFI.

In Figure 7.1, the results for MxM shows that there was an increase of the AVF SDC and SEFI in the fault injection experiment with the increase of the compiler optimization. In the radiation experiment, the Cross Section for SDC increases with the

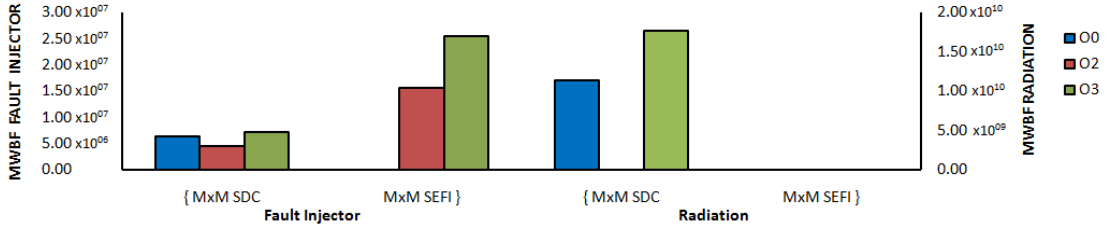
Figure 7.1: AVF and Cross Section in MxM



Source: Author

compiler optimization, and no SEFI are found in the tested compiler optimizations.

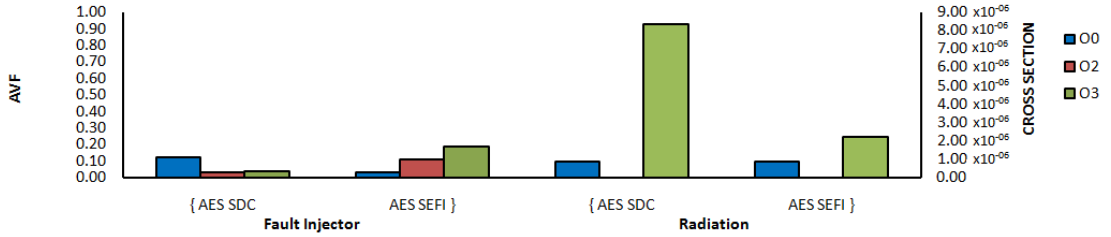
Figure 7.2: Radiation and Fault Injection MWBFs in MxM



Source: The Author

In Figure 7.2 the results for MxM in the fault injection shows that the MWBF SDC slightly change and there was an increase in the MWBF SEFI with the increase of the compiler optimization. In the radiation experiment, the MWBF SDC increases with the compiler optimization, and the MWBF SEFI does not appear in the graphic because no SEFI errors are found in the radiation experiment.

Figure 7.3: AVF x Cross Section in AES

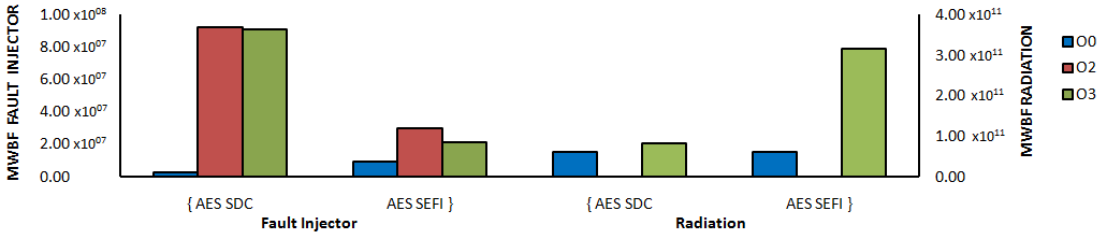


Source: The Author

In Figure 7.3, the results for AES shows that there was a decrease of the AVF SDC and an increase of the AVF SEFI in the fault injection experiment with the increase of the compiler optimization. In the radiation experiment, the Cross Section for SDC and SEFI increases with the compiler optimization.

In Figure 7.4 the results for AES in the fault injection shows that there was an increase in the MWBF SDC and SEFI with the increase of the compiler optimization. In

Figure 7.4: Radiation and Fault Injector MWBFs in AES



Source: The Author

the radiation experiment, the MWBF SDC slightly change, and there was an increase in MWBF SEFI with the increase of the compiler optimization.

The analysis of the Figures 7.1 to 7.4 shows that the impact of optimizations is strongly related to the executed algorithm. As shown in Figures 7.1 and 7.2, while O2 significantly impacts the probability of radiation-induced SDCs on MxM (O2 increases SDC AVF for MxM of about 3.70 times), O2 and O3 have almost the same AVF (the difference is lower than 10%). Furthermore, the MxM AVF for SEFI also increases when the O2 optimization is used and decreases between O2 and O3.

The fault injection experiments results show that the best performance, the minor register file usage, or the lowest AVF does not always bring the highest MWBF. The best performance changes the instructions used by the processor which can be faster but also can have a lower reliability. In fact, optimization increases the number of used registers and the exposed area. In order to be beneficial, the performance improvements must be sufficient to compensate the increased exposed area.

To evaluate the NEON register plus the general-purpose registers it was used as a benchmark is the MxM Double with the O3 and the O3+ compiler optimizations. The NEON technology as mentioned before is an SIMD accelerator processor. This technology is part of the ARM core and their highest benefits are the execution of operations with vectors and perform floating point operations in parallel.

The fault injection in the NEON registers shows that with the increase in the optimization a higher MWBF is achieved. In this case, the O3 optimization has a better performance and compensates the increase in the register file usage and the AVF. The sum of the NEON and the general purpose registers represents a total of 80 registers which means there is a higher increase in area with compared with the applications that do not use the NEON. In the newest arm architectures such as the ARMv8, the NEON technology is better supported that was in the ARMv7 used in this work providing a support of more data types, and a better increase in performance.

In this work, the effects compiler optimization was better understood in different aspects. In the future, this may become a major problem due to the increase in compiler complexity, and in the use of heterogeneous resources in processors. The results show that for most applications the best MWBF is not achieved with the highest optimization level. As mentioned before in chapter 6 this happens because optimizations reduce execution time but also an increase in the use of registers. Then, there is a trade-off between the execution time and the register file usage. The comparison between the fault injection results and radiation data follow different trends which may occur because the fault injection is limited to a subset of available resources.

7.2 Future Work

As a future work, it is interesting to analyze other applications that use the NEON technology and compare the results. Also, a Heavy Ion experiment can be done to evaluate how the NEON technology changes the overall system reliability.

Another work that can be done is use the tool developed in (PLOTNIKOV et al., 2013) which allow to identify the compiler optimizations that most contribute to the improvement in performance for a given ARM A9 application and evaluate them for reliability.

REFERENCES

AGUIAR, V. et al. Experimental setup for single event effects at the são paulo 8ud pelletron accelerator. **Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms**, Elsevier, v. 332, p. 397–400, 2014.

AHO, A. V. et al. **Compilers: Principles, Techniques, and Tools (2nd Edition)**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

ARLAT, J. et al. Comparison of physical and software-implemented fault injection techniques. **IEEE Transactions on Computers**, IEEE, v. 52, n. 9, p. 1115–1133, 2003.

ASADI, G.-H. et al. Balancing performance and reliability in the memory hierarchy. In: IEEE. **Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on**. [S.l.], 2005. p. 269–279.

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

BOUDENOT, J.-C. Radiation space environment. In: **Radiation Effects on Embedded Systems**. [S.l.]: Springer, 2007. p. 1–9.

CARREIRA, J.; MADEIRA, H.; SILVA, J. G. Xception: A technique for the experimental evaluation of dependability in modern computers. **IEEE Transactions on Software Engineering**, IEEE, v. 24, n. 2, p. 125–136, 1998.

CHIBANI, K. et al. Criticality evaluation of embedded software running on a pipelined microprocessor and impact of compilation options. In: IEEE. **Electronics, Circuits and Systems (ICECS), 2014 21st IEEE International Conference on**. [S.l.], 2014. p. 778–781.

CHIELLE, E. **Selective Software-Implemented Hardware Fault Tolerance Techniques to Detect Soft Errors in Processors with Reduced Overhead**. 2016.

CROCKETT, L. H. et al. **The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc**. [S.l.]: Strathclyde Academic Media, 2014.

DANDAMUDI, S. P. **Guide to RISC processors: for programmers and engineers**. [S.l.]: Springer Science & Business Media, 2005.

DEMERTZI, M.; ANNAVARAM, M.; HALL, M. Analyzing the effects of compiler optimizations on application reliability. In: IEEE. **Workload Characterization (IISWC), 2011 IEEE International Symposium on**. [S.l.], 2011. p. 184–193.

ENTRENA, L. et al. Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. **IEEE Transactions on Computers**, IEEE, v. 61, n. 3, p. 313–322, 2012.

FERREIRA, R. R. et al. Compiler optimizations impact the reliability of the control-flow of radiation-hardened software. **Journal of Aerospace Technology and Management**, SciELO Brasil, v. 5, n. 3, p. 323–334, 2013.

HAGEN, W. V. **The definitive guide to GCC**. [S.l.]: Apress, 2011.

IBRAHIM, M. E.; RUPP, M.; HABIB, S.-D. Compiler-based optimizations impact on embedded software power consumption. In: IEEE. **Circuits and Systems and TAISA Conference, 2009. NEWCAS-TAISA'09. Joint IEEE North-East Workshop on**. [S.l.], 2009. p. 1–4.

ISAZA-GONZÁLEZ, J. et al. Dependability evaluation of COTS microprocessors via on-chip debugging facilities. In: IEEE. **2016 17th Latin-American Test Symposium (LATS)**. [S.l.], 2016. p. 27–32.

KALIORAKIS, M. et al. Differential fault injection on microarchitectural simulators. In: IEEE. **Workload Characterization (IISWC), 2015 IEEE International Symposium on**. [S.l.], 2015. p. 172–182.

KRÜGER, J.; WESTERMANN, R. Linear algebra operators for gpu implementation of numerical algorithms. In: ACM. **ACM Transactions on Graphics (TOG)**. [S.l.], 2003. v. 22, n. 3, p. 908–916.

LIEPE, J. et al. ABC-SysBio—approximate Bayesian computation in Python with GPU support. **Bioinformatics**, Oxford Univ Press, v. 26, n. 14, p. 1797–1799, 2010.

MANIATAKOS, M. et al. Instruction-level impact analysis of low-level faults in a modern microprocessor controller. **IEEE Transactions on Computers**, IEEE, v. 60, n. 9, p. 1260–1273, 2011.

MONTESINOS, P.; LIU, W.; TORRELLAS, J. Using register lifetime predictions to protect register files against soft errors. In: IEEE. **Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on**. [S.l.], 2007. p. 286–296.

MUKHERJEE, S. S. et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: IEEE. **Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on**. [S.l.], 2003. p. 29–40.

NAGY, C. **Embedded systems design using the TI MSP430 series**. [S.l.]: Elsevier, 2003.

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>.

PARASYRIS, K. et al. GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates. In: IEEE. **2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks**. [S.l.], 2014. p. 622–629.

PLOTNIKOV, D. et al. Automatic tuning of compiler optimizations and analysis of their impact. **Procedia Computer Science**, Elsevier, v. 18, p. 1312–1321, 2013.

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

QUINN, H. et al. Using Benchmarks for Radiation Testing of Microprocessors and FPGAs. **IEEE Transactions on Nuclear Science**, v. 62, n. 6, p. 2547–2554, Dec 2015. ISSN 0018-9499.

RECH, P. et al. Impact of gpus parallelism management on safety-critical and hpc applications reliability. In: IEEE. **Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on**. [S.l.], 2014. p. 455–466.

RESTREPO-CALLE, F. et al. Application-based analysis of register file criticality for reliability assessment in embedded microprocessors. **Journal of Electronic Testing**, Springer, v. 31, n. 2, p. 139–150, 2015.

ROMER, P.; TROGER, P. Reliability implications of register utilization: An empirical study. In: IEEE. **Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on**. [S.l.], 2011. p. 105–112.

ROSA, F. et al. A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability. In: IEEE. **2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)**. [S.l.], 2015. p. 211–214.

RUDOLPH, D. et al. Csp: A multifaceted hybrid architecture for space computing. **28**th **AIAA/USU Conference on Small Satellites**, p. 1–7, 2014.

SAGGESE, G. P. et al. Microprocessor sensitivity to failures: control vs. execution and combinational vs. sequential logic. In: IEEE. **2005 International Conference on Dependable Systems and Networks (DSN'05)**. [S.l.], 2005. p. 760–769.

SANGCHOOLIE, B. et al. A study of the impact of bit-flip errors on programs compiled with different optimization levels. In: IEEE. **Dependable Computing Conference (EDCC), 2014 Tenth European**. [S.l.], 2014. p. 146–157.

SERRANO-CASES, A.; ISAZA-GONZÁLEZ, J. e. a. On the influence of compiler optimizations in the fault tolerance of embedded systems. In: IEEE. **On-Line Testing and Robust System Design (IOLTS), 2016 IEEE 22nd International Symposium on**. [S.l.], 2016. p. 207–208.

SIEGLE, F. et al. Mitigation of radiation effects in sram-based fpgas for space applications. **ACM Computing Surveys (CSUR)**, ACM, v. 47, n. 2, p. 37, 2015.

SRIDHARAN, V.; KAELI, D. R. Quantifying software vulnerability. In: ACM. **Proceedings of the 2008 workshop on Radiation effects and fault tolerance in nanometer technologies**. [S.l.], 2008. p. 323–328.

TAMBARA, L. **Analyzing the Impact of Radiation-induced Failures in All Programmable System-on-Chip Devices**. 2017.

TAMBARA, L. A. et al. Heavy Ions Induced Single Event Upsets Testing of the 28 nm Xilinx Zynq-7000 All Programmable SoC. In: IEEE. **Radiation Effects Data Workshop (REDW), 2015 IEEE**. [S.l.], 2015. p. 1–6.

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.

TAN, J. et al. Analyzing soft-error vulnerability on GPGPU microarchitecture. In: **Workload Characterization (IISWC), 2011 IEEE International Symposium on**. [S.l.: s.n.], 2011. p. 226–235.

VALLURI, M.; JOHN, L. K. Is compiling for performance—compiling for power? In: **Interaction between compilers and computer architectures**. [S.l.]: Springer, 2001. p. 101–115.

VELAZCO, R.; REZGUI, S.; ECOFFET, R. Predicting error rate for microprocessor-based digital architectures through CEU (Code Emulating Upsets) injection. **IEEE Transactions on Nuclear Science**, IEEE, v. 47, n. 6, p. 2405–2411, 2000.

XILINX. **Zynq-7000 All Programmable SoC Data Sheet: Overview**. 2017.

YAN, J.; ZHANG, W. Compiler-guided register reliability improvement against soft errors. In: ACM. **Proceedings of the 5th ACM international conference on Embedded software**. [S.l.], 2005. p. 203–209.

ZARGHAM, M. R. **Computer architecture: single and parallel systems**. [S.l.]: Prentice-Hall, Inc., 1996.

APPENDIX A — TABLES

In this appendix, it will be shown the tables with the results of the fault injection for each benchmark and their tested optimizations. Each table contains the register where the fault was injected, the number of injections in that register, the number of unaces, the number of SDCs, and the number of SEFIs. It also has the AVF Unace, SDC, and SEFI of each register. Additionally, the last line of the table represents the metrics considering all the tested registers.

A.1 MxM Tables

Table A.1: Results of Fault Injection MxM O0

Register	# of Injections	# of Unaces	# of SDCs	# of SEFIs	AVF Unace	AVF SDCs	AVF SEFIs
r0	6622	6619	0	3	1	0	0
r1	6127	3968	2149	10	0.65	0.35	0
r2	6500	5810	687	1	0.89	0.11	0
r3	6061	4162	1894	4	0.69	0.31	0
r4	7394	7394	0	0	1	0	0
r5	6038	6036	0	0	1	0	0
r6	6844	6844	0	0	1	0	0
r7	6045	6045	0	0	1	0	0
r8	6699	6699	0	0	1	0	0
r9	6016	6016	0	0	1	0	0
r10	6628	6628	0	0	1	0	0
r11	6185	6185	0	0	1	0	0
r12	6632	6632	0	0	1	0	0
Total	83791	79038	4730	18	0.94	0.06	0.00

Table A.2: Results of Fault Injection MxM O2

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	85	16	69	0	0.19	0.81	0.00
r1	243	19	221	3	0.08	0.91	0.01
r2	336	12	324	0	0.04	0.96	0.00
r3	240	56	17	168	0.23	0.07	0.70
r4	365	119	246	0	0.33	0.67	0.00
r5	250	153	87	10	0.61	0.35	0.04
r6	364	364	0	0	1.00	0.00	0.00
r7	227	225	2	0	0.99	0.01	0.00
r8	389	21	76	292	0.05	0.20	0.75
r9	260	27	233	0	0.10	0.90	0.00
r10	98	7	91	0	0.07	0.93	0.00
r11	223	223	0	0	1.00	0.00	0.00
r12	340	340	0	0	1.00	0.00	0.00
Total	3420	1582	1366	473	0.46	0.40	0.14

Table A.3: Results of Fault Injection MxM O3

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	528	343	185	0	0.65	0.35	0.00
r1	500	159	341	0	0.32	0.68	0.00
r2	485	93	383	9	0.19	0.79	0.02
r3	511	205	56	250	0.40	0.11	0.49
r4	492	198	101	193	0.40	0.21	0.39
r5	475	99	309	65	0.21	0.65	0.14
r6	458	422	36	0	0.92	0.08	0.00
r7	529	489	41	0	0.92	0.08	0.00
r8	465	26	439	0	0.06	0.94	0.00
r9	569	77	440	52	0.14	0.77	0.09
r10	488	195	31	262	0.40	0.06	0.54
r11	458	398	54	6	0.87	0.12	0.01
r12	503	453	50	0	0.90	0.10	0.00
Total	7465	3157	2466	837	0.42	0.33	0.11

A.2 AES

Table A.4: Results of Fault Injection AES O0

Register	# of Injections	# of Unaces	# of SDCs	# of SEFIs	AVF Unace	AVF SDCs	AVF SEFIs
r0	3018	2580	386	52	0.85	0.13	0.02
r1	2325	1635	550	140	0.70	0.24	0.06
r2	3048	1719	970	359	0.56	0.32	0.12
r3	2375	1122	764	489	0.47	0.32	0.21
r4	3070	2120	940	0	0.69	0.31	0.00
r5	2312	2312	0	0	1.00	0.00	0.00
r6	3244	3241	0	3	1.00	0.00	0.00
r7	2334	2334	0	0	1.00	0.00	0.00
r8	3110	3096	0	14	1.00	0.00	0.00
r9	2287	2287	0	0	1.00	0.00	0.00
r10	3382	3382	0	0	1.00	0.00	0.00
r11	2336	2336	0	0	1.00	0.00	0.00
r12	2976	2976	0	0	1.00	0.00	0.00
Total	35817	31140	3610	1057	0.87	0.10	0.03

Table A.5: Results of Fault Injection AES O2

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	741	741	0	0	1.00	0.00	0.00
r1	637	636	0	1	1.00	0.00	0.00
r2	712	711	0	1	1.00	0.00	0.00
r3	812	812	0	0	1.00	0.00	0.00
r4	688	688	0	0	1.00	0.00	0.00
r5	717	716	0	1	1.00	0.00	0.00
r6	739	739	0	0	1.00	0.00	0.00
r7	679	679	0	0	1.00	0.00	0.00
r8	652	522	8	122	0.80	0.01	0.19
r9	789	455	105	229	0.58	0.13	0.29
r10	695	244	16	435	0.35	0.02	0.63
r11	696	695	0	1	1.00	0.00	0.00
r12	747	647	0	100	0.87	0.00	0.13
Total	9304	8285	129	890	0.89	0.01	0.10

Table A.6: Results of Fault Injection AES O3

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	771	771	0	0	1.00	0.00	0.00
r1	1103	1103	0	0	1.00	0.00	0.00
r2	745	745	0	0	1.00	0.00	0.00
r3	1212	1212	0	0	1.00	0.00	0.00
r4	825	825	0	0	1.00	0.00	0.00
r5	1106	1106	0	0	1.00	0.00	0.00
r6	786	786	0	0	1.00	0.00	0.00
r7	1141	1141	0	0	1.00	0.00	0.00
r8	791	205	36	548	0.26	0.05	0.69
r9	1105	637	92	372	0.58	0.08	0.34
r10	785	719	34	32	0.92	0.04	0.04
r11	1141	0	0	1141	0.00	0.00	1.00
r12	765	765	0	0	1.00	0.00	0.00
Total	12276	10015	162	2093	0.82	0.01	0.17

A.3 Quicksort

Table A.7: Results of Fault Injection Quicksort O0

Register	# of Injections	# of Unaces	# of SDCs	# of SEFIs	AVF Unace	AVF SDCs	AVF SEFIs
r0	6816	6672	144	0	0,98	0,02	0,00
r1	6392	5952	327	113	0,93	0,05	0,02
r2	6881	6072	726	83	0,88	0,11	0,01
r3	6347	5810	440	97	0,92	0,07	0,02
r4	6865	6865	0	0	1,00	0,00	0,00
r5	6155	6155	0	0	1,00	0,00	0,00
r6	6926	6876	0	0	0,99	0,00	0,00
r7	6229	6229	0	0	1.00	0.00	0.00
r8	6933	6933	0	0	1.00	0.00	0.00
r9	6241	6241	0	0	1.00	0.00	0.00
r10	7388	7388	0	0	1.00	0.00	0.00
r11	6304	6304	0	0	1.00	0.00	0.00
r12	7525	7525	0	0	1.00	0.00	0.00
Total	87002	85022	1637	293	0.98	0.02	0.00

Table A.8: Results of Fault Injection Quicksort O2

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	2439	2305	134	0	0.95	0.05	0.05
r1	2841	2443	314	84	0.86	0.11	0.11
r2	2651	2572	58	21	0.97	0.02	0.02
r3	2816	2657	159	0	0.94	0.06	0.06
r4	2034	1510	436	88	0.74	0.21	0.21
r5	1852	415	1424	13	0.22	0.77	0.77
r6	2045	843	1201	1	0.41	0.59	0.59
r7	2873	602	2271	0	0.21	0.79	0.79
r8	2213	1058	876	279	0.48	0.40	0.40
r9	3024	394	2029	601	0.13	0.67	0.67
r10	2498	2175	323	0	0.87	0.13	0.13
r11	3124	140	2984	0	0.04	0.96	0.96
r12	2791	2791	0	0	1.00	0.00	0.00
Total	33201	19905	12209	1087	0.60	0.37	0.03

Table A.9: Results of Fault Injection Quicksort O3

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	573	553	18	2	0.97	0.03	0.00
r1	3792	3340	396	56	0.88	0.10	0.01
r2	646	623	21	2	0.96	0.03	0.00
r3	4565	4351	211	3	0.95	0.05	0.00
r4	512	399	93	20	0.78	0.18	0.04
r5	1560	202	1358	0	0.13	0.87	0.00
r6	651	527	121	3	0.81	0.19	0.00
r7	7919	4957	2962	0	0.63	0.37	0.00
r8	508	169	31	308	0.33	0.06	0.61
r9	1660	232	1318	110	0.14	0.79	0.07
r10	442	35	397	10	0.08	0.90	0.02
r11	3087	134	2900	53	0.04	0.94	0.02
r12	724	724	0	0	1.00	0.00	0.00
Total	26639	16246	9826	567	0.61	0.37	0.02

A.4 FFT

Table A.10: Results of Fault Injection FFT O0

Register	# of Injections	# of Unaces	# of SDCs	# of SEFIs	AVF Unace	AVF SDCs	AVF SEFIs
r0	2710	2594	114	2	0.96	0.04	0.00
r1	2763	2635	128	0	0.95	0.05	0.00
r2	2737	2267	345	125	0.83	0.13	0.05
r3	2841	1837	717	287	0.65	0.25	0.10
r4	2643	2643	0	0	1.00	0.00	0.00
r5	2783	2731	0	52	0.98	0.00	0.02
r6	2743	2743	0	0	1.00	0.00	0.00
r7	2766	2762	0	4	1.00	0.00	0.00
r8	2643	2643	0	0	1.00	0.00	0.00
r9	2888	2888	0	0	1.00	0.00	0.00
r10	2836	2836	0	0	1.00	0.00	0.00
r11	2637	1122	31	484	0.43	0.01	0.18
r12	2751	2751	0	0	1.00	0.00	0.00
Total	35741	32452	1335	954	0.91	0.04	0.03

Table A.11: Results of Fault Injection FFT O2

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	2669	1507	961	181	0.56	0.36	0.07
r1	2549	1296	713	537	0.51	0.28	0.21
r2	2621	1193	762	664	0.46	0.29	0.25
r3	2550	1202	726	622	0.47	0.28	0.24
r4	2522	463	1959	98	0.18	0.78	0.04
r5	2593	1144	1027	422	0.44	0.40	0.16
r6	2495	1147	1163	185	0.46	0.47	0.07
r7	2713	854	748	1096	0.31	0.28	0.40
r8	2623	324	617	1448	0.12	0.24	0.55
r9	2539	248	2284	7	0.10	0.90	0.00
r10	2509	215	1099	1192	0.09	0.44	0.48
r11	2563	355	480	1726	0.14	0.19	0.67
r12	2494	1847	273	374	0.74	0.11	0.15
Total	33440	11795	12812	8552	0.35	0.38	0.26

Table A.12: Results of Fault Injection FFT O3

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	2568	1507	921	140	0.59	0.36	0.05
r1	2540	1296	695	549	0.51	0.27	0.22
r2	2679	1193	783	703	0.45	0.29	0.26
r3	2471	1202	747	522	0.49	0.30	0.21
r4	1836	463	1300	73	0.25	0.71	0.04
r5	2260	1144	734	382	0.51	0.32	0.17
r6	2159	1147	823	189	0.53	0.38	0.09
r7	2370	854	523	993	0.36	0.22	0.42
r8	2147	324	593	1230	0.15	0.28	0.57
r9	1673	248	1402	23	0.15	0.84	0.01
r10	2198	215	1100	883	0.10	0.50	0.40
r11	2199	355	394	1450	0.16	0.18	0.66
r12	2363	1847	223	293	0.78	0.09	0.12
Total	29463	11795	10238	7430	0.40	0.35	0.25

A.5 Fibonnaci

Table A.13: Results of Fault Injection Fibonnaci O0

Register	# of Injections	# of Unaces	# of SDCs	# of SEFIs	AVF Unace	AVF SDCs	AVF SEFIs
r0	3288	3000	288	0	0.91	0.09	0.00
r1	3408	2952	343	113	0.87	0.10	0.03
r2	2857	2072	702	83	0.73	0.25	0.03
r3	3337	2810	430	97	0.84	0.13	0.03
r4	2865	2865	0	0	1.00	0.00	0.00
r5	2155	2155	0	0	1.00	0.00	0.00
r6	2876	2876	0	0	1.00	0.00	0.00
r7	2229	2229	0	0	1.00	0.00	0.00
r8	2933	2933	0	0	1.00	0.00	0.00
r9	2241	2241	0	0	1.00	0.00	0.00
r10	2387	2387	0	0	1.00	0.00	0.00
r11	2302	2302	0	0	1.00	0.00	0.00
r12	2524	2524	0	0	1.00	0.00	0.00
Total	35402	33346	1763	293	0.94	0.05	0.01

Table A.14: Results of Fault Injection Fibonnaci O2

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	1135	510	625	0	0.45	0.55	0.00
r1	1448	1241	103	104	0.86	0.07	0.07
r2	905	766	98	41	0.85	0.11	0.05
r3	918	795	123	0	0.87	0.13	0.00
r4	1111	210	822	79	0.19	0.74	0.07
r5	908	230	661	17	0.25	0.73	0.02
r6	854	42	811	1	0.05	0.95	0.00
r7	1289	1289	0	0	1.00	0.00	0.00
r8	1468	1128	0	340	0.77	0.00	0.23
r9	1563	1096	0	467	0.70	0.00	0.30
r10	1203	1203	0	0	1.00	0.00	0.00
r11	1161	1161	0	0	1.00	0.00	0.00
r12	1191	1191	0	0	1.00	0.00	0.00
Total	15154	10862	3243	1049	0.72	0.21	0.07

Table A.15: Results of Fault Injection Fibonnaci O3

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	972	833	130	9	0.86	0.13	0.01
r1	940	742	139	59	0.79	0.15	0.06
r2	823	716	99	8	0.87	0.12	0.01
r3	686	632	41	13	0.92	0.06	0.02
r4	955	863	72	20	0.90	0.08	0.02
r5	994	876	118	0	0.88	0.12	0.00
r6	884	774	103	7	0.88	0.12	0.01
r7	847	817	30	0	0.96	0.04	0.00
r8	1035	808	18	209	0.78	0.02	0.20
r9	1137	736	282	119	0.65	0.25	0.10
r10	1031	828	184	19	0.80	0.18	0.02
r11	816	751	16	49	0.92	0.02	0.06
r12	1023	1023	0	0	1.00	0.00	0.00
Total	12143	10399	1232	512	0.86	0.10	0.04

A.6 JPEG

Table A.16: Results of Fault Injection JPEG O0

Register	# of Injections	# of Unaces	# of SDCs	# of SEFIs	AVF Unace	AVF SDCs	AVF SEFIs
r0	579	513	55	11	0.89	0.09	0.02
r1	573	527	28	18	0.92	0.05	0.03
r2	592	470	111	11	0.79	0.19	0.02
r3	593	379	187	0	0.64	0.32	0.00
r4	586	480	78	28	0.82	0.13	0.05
r5	566	554	12	0	0.98	0.02	0.00
r6	562	553	9	0	0.98	0.02	0.00
r7	569	549	19	1	0.96	0.03	0.00
r8	576	565	11	0	0.98	0.02	0.00
r9	594	563	31	0	0.95	0.05	0.00
r10	586	574	12	0	0.98	0.02	0.00
r11	625	140	81	404	0.22	0.13	0.65
r12	579	556	20	3	0.96	0.03	0.01
Total	7580	6423	654	476	0.85	0.09	0.06

Table A.17: Results of Fault Injection JPEG O2

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	1759	915	637	207	0.52	0.36	0.12
r1	1724	970	594	160	0.56	0.34	0.09
r2	1734	1013	599	122	0.58	0.35	0.07
r3	1828	882	718	228	0.48	0.39	0.12
r4	1834	805	840	189	0.44	0.46	0.10
r5	1512	924	379	209	0.61	0.25	0.14
r6	1699	778	570	351	0.46	0.34	0.21
r7	1748	468	1009	271	0.27	0.58	0.16
r8	1752	540	777	213	0.31	0.44	0.12
r9	1694	624	944	115	0.37	0.56	0.07
r10	1781	774	292	667	0.43	0.16	0.37
r11	1739	374	1128	180	0.22	0.65	0.10
r12	1685	1270	386	29	0.75	0.23	0.02
Total	22489	10337	8873	2941	0.46	0.39	0.13

Table A.18: Results of Fault Injection JPEG O3

Register	**# of Injections**	**# of Unaces**	**# of SDCs**	**# of SEFIs**	**AVF Unace**	**AVF SDCs**	**AVF SEFIs**
r0	2422	1363	876	183	0.56	0.36	0.08
r1	2434	1118	748	568	0.46	0.31	0.23
r2	2402	986	710	706	0.41	0.30	0.29
r3	2426	948	984	494	0.39	0.41	0.20
r4	2463	1508	617	338	0.61	0.25	0.14
r5	2545	1278	598	569	0.50	0.23	0.22
r6	2428	1318	831	279	0.54	0.34	0.11
r7	2506	1313	728	465	0.52	0.29	0.19
r8	2385	841	816	728	0.35	0.34	0.31
r9	2486	966	1024	496	0.39	0.41	0.20
r10	2544	1297	515	732	0.51	0.20	0.29
r11	2358	1137	959	362	0.48	0.41	0.15
r12	2448	1333	956	159	0.54	0.39	0.06
Total	31847	15406	10362	6079	0.48	0.33	0.19

A.7 MxM Double

Table A.19: Results of Fault Injection MxM Double O3

Register	# of Injections	# of Unaces	# of SDCs	# of SEFIs	AVF Unace	AVF SDCs	AVF SEFIs
r0	845	776	69	0	0.92	0.08	0.00
r1	1908	1689	219	0	0.89	0.11	0.00
r2	930	865	65	0	0.93	0.07	0.00
r3	975	823	152	0	0.84	0.16	0.00
r4	810	810	0	0	1.00	0.00	0.00
r5	824	824	0	0	1.00	0.00	0.00
r6	879	879	0	0	1.00	0.00	0.00
r7	989	989	0	0	1.00	0.00	0.00
r8	855	855	0	0	1.00	0.00	0.00
r9	959	959	0	0	1.00	0.00	0.00
r10	950	950	0	0	1.00	0.00	0.00
r11	937	937	0	0	1.00	0.00	0.00
r12	927	927	0	0	1.00	0.00	0.00
sp	920	15	0	905	0.02	0.00	0.98
lr	874	4	0	870	0.00	0.00	1.00
s0	949	949	0	0	1.00	0.00	0.00
s1	872	872	0	0	1.00	0.00	0.00
s2	967	967	0	0	1.00	0.00	0.00
s3	957	957	0	0	1.00	0.00	0.00
s4	960	960	0	0	1.00	0.00	0.00
s5	895	895	0	0	1.00	0.00	0.00
s6	897	897	0	0	1.00	0.00	0.00
s7	894	894	0	0	1.00	0.00	0.00
s8	899	899	0	0	1.00	0.00	0.00
s9	905	905	0	0	1.00	0.00	0.00
s10	964	964	0	0	1.00	0.00	0.00
s11	964	964	0	0	1.00	0.00	0.00
s12	914	914	0	0	1.00	0.00	0.00
s13	838	838	0	0	1.00	0.00	0.00
s14	876	876	0	0	1.00	0.00	0.00
s15	841	841	0	0	1.00	0.00	0.00
s16	892	892	0	0	1.00	0.00	0.00
s17	877	877	0	0	1.00	0.00	0.00
s18	929	929	0	0	1.00	0.00	0.00
s19	930	930	0	0	1.00	0.00	0.00
s20	934	934	0	0	1.00	0.00	0.00
s21	867	867	0	0	1.00	0.00	0.00
s22	848	848	0	0	1.00	0.00	0.00

Table A.20: Results of Fault Injection MxM Double O3 Continuation

Register	# of Injections	# of Unaces	# of SDCs	# of SEFIs	AVF Unace	AVF SDCs	AVF SEFIs
s23	998	998	0	0	1.00	0.00	0.00
s24	835	835	0	0	1.00	0.00	0.00
s25	959	959	0	0	1.00	0.00	0.00
s26	914	914	0	0	1.00	0.00	0.00
s27	907	907	0	0	1.00	0.00	0.00
s28	933	933	0	0	1.00	0.00	0.00
s29	948	948	0	0	1.00	0.00	0.00
s30	835	835	0	0	1.00	0.00	0.00
s31	967	967	0	0	1.00	0.00	0.00
s32	893	500	393	0	0.56	0.44	0.00
s33	921	633	288	0	0.69	0.31	0.00
s34	900	611	289	0	0.68	0.32	0.00
s35	948	736	212	0	0.78	0.22	0.00
s36	945	757	188	0	0.80	0.20	0.00
s37	873	733	140	0	0.84	0.16	0.00
s38	899	899	0	0	1.00	0.00	0.00
s39	894	894	0	0	1.00	0.00	0.00
s40	845	845	0	0	1.00	0.00	0.00
s41	912	912	0	0	1.00	0.00	0.00
s42	867	867	0	0	1.00	0.00	0.00
s43	923	923	0	0	1.00	0.00	0.00
s44	993	993	0	0	1.00	0.00	0.00
s45	877	877	0	0	1.00	0.00	0.00
s46	904	904	0	0	1.00	0.00	0.00
s47	915	915	0	0	1.00	0.00	0.00
s48	949	949	0	0	1.00	0.00	0.00
s49	947	947	0	0	1.00	0.00	0.00
s50	992	992	0	0	1.00	0.00	0.00
s51	873	873	0	0	1.00	0.00	0.00
s52	991	991	0	0	1.00	0.00	0.00
s53	993	993	0	0	1.00	0.00	0.00
s54	958	958	0	0	1.00	0.00	0.00
s55	882	882	0	0	1.00	0.00	0.00
s56	973	973	0	0	1.00	0.00	0.00
s57	960	960	0	0	1.00	0.00	0.00
s58	984	984	0	0	1.00	0.00	0.00
s59	866	866	0	0	1.00	0.00	0.00
s60	874	874	0	0	1.00	0.00	0.00
s61	895	895	0	0	1.00	0.00	0.00
s62	879	879	0	0	1.00	0.00	0.00
s63	844	844	0	0	1.00	0.00	0.00
TOTAL	73116	69326	2015	1775	0.95	0.03	0.02

Table A.21: Results of Fault Injection MxM Double O3+

Register	# of Injections	# of Unaces	# of SDCs	# of SEFIs	AVF Unace	AVF SDCs	AVF SEFIs
r0	985	896	89	0	0.91	0.09	0.00
r1	887	887	0	0	1.00	0.00	0.00
r2	972	972	0	0	1.00	0.00	0.00
r3	996	729	267	0	0.73	0.27	0.00
r4	902	820	82	0	0.91	0.09	0.00
r5	910	837	73	0	0.92	0.08	0.00
r6	862	795	67	0	0.92	0.08	0.00
r7	998	998	0	0	1.00	0.00	0.00
r8	914	914	0	0	1.00	0.00	0.00
r9	884	884	0	0	1.00	0.00	0.00
r10	940	940	0	0	1.00	0.00	0.00
r11	895	895	0	0	1.00	0.00	0.00
r12	953	953	0	0	1.00	0.00	0.00
sp	935	25	0	910	0.03	0.00	0.97
lr	873	3	0	870	0.00	0.00	1.00
s0	890	801	89	0	0.90	0.10	0.00
s1	933	865	68	0	0.93	0.07	0.00
s2	961	893	68	0	0.93	0.07	0.00
s3	875	831	44	0	0.95	0.05	0.00
s4	985	901	84	0	0.91	0.09	0.00
s5	844	782	62	0	0.93	0.07	0.00
s6	843	761	82	0	0.90	0.10	0.00
s7	990	920	70	0	0.93	0.07	0.00
s8	981	869	112	0	0.89	0.11	0.00
s9	964	872	92	0	0.90	0.10	0.00
s10	860	712	148	0	0.83	0.17	0.00
s11	865	746	119	0	0.86	0.14	0.00
s12	965	879	86	0	0.91	0.09	0.00
s13	956	889	67	0	0.93	0.07	0.00
s14	900	818	82	0	0.91	0.09	0.00
s15	860	801	59	0	0.93	0.07	0.00
s16	886	886	0	0	1.00	0.00	0.00
s17	866	866	0	0	1.00	0.00	0.00
s18	963	963	0	0	1.00	0.00	0.00
s19	875	875	0	0	1.00	0.00	0.00
s20	916	916	0	0	1.00	0.00	0.00
s21	872	872	0	0	1.00	0.00	0.00
s22	830	830	0	0	1.00	0.00	0.00
s23	942	942	0	0	1.00	0.00	0.00
s24	996	996	0	0	1.00	0.00	0.00
s25	885	885	0	0	1.00	0.00	0.00

Table A.22: Results of Fault Injection MxM Double O3+ Continuation

Register	# of Injections	# of Unaces	# of SDCs	# of SEFIs	AVF Unace	AVF SDCs	AVF SEFIs
s26	999	999	0	0	1.00	0.00	0.00
s27	906	906	0	0	1.00	0.00	0.00
s28	911	911	0	0	1.00	0.00	0.00
s29	882	882	0	0	1.00	0.00	0.00
s30	940	940	0	0	1.00	0.00	0.00
s31	971	971	0	0	1.00	0.00	0.00
s32	913	826	87	0	0.90	0.10	0.00
s33	945	879	66	0	0.93	0.07	0.00
s34	878	794	84	0	0.90	0.10	0.00
s35	975	897	78	0	0.92	0.08	0.00
s36	984	903	81	0	0.92	0.08	0.00
s37	953	880	73	0	0.92	0.08	0.00
s38	912	834	78	0	0.91	0.09	0.00
s39	903	832	71	0	0.92	0.08	0.00
s40	873	790	83	0	0.90	0.10	0.00
s41	902	833	69	0	0.92	0.08	0.00
s42	958	871	87	0	0.91	0.09	0.00
s43	894	819	75	0	0.92	0.08	0.00
s44	900	768	132	0	0.85	0.15	0.00
s45	879	766	113	0	0.87	0.13	0.00
s46	912	840	72	0	0.92	0.08	0.00
s47	917	854	63	0	0.93	0.07	0.00
s48	881	802	79	0	0.91	0.09	0.00
s49	986	914	72	0	0.93	0.07	0.00
s50	947	871	76	0	0.92	0.08	0.00
s51	851	789	62	0	0.93	0.07	0.00
s52	977	842	135	0	0.86	0.14	0.00
s53	921	795	126	0	0.86	0.14	0.00
s54	859	720	139	0	0.84	0.16	0.00
s55	882	774	108	0	0.88	0.12	0.00
s56	909	780	129	0	0.86	0.14	0.00
s57	957	840	117	0	0.88	0.12	0.00
s58	956	880	76	0	0.92	0.08	0.00
s59	920	853	67	0	0.93	0.07	0.00
s60	938	860	78	0	0.92	0.08	0.00
s61	947	882	65	0	0.93	0.07	0.00
s62	899	817	82	0	0.91	0.09	0.00
s63	832	765	67	0	0.92	0.08	0.00
TOTAL	72578	66098	4700	1780	0.92	0.06	0.02