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

KARLSRUHER INSTITUTE FÜR TECHNOLOGIE
FAKULTÄT FÜR ELEKTROTECHNIK UND
INFORMATIONSTECHNIK

JOSÉ RODRIGO FURLANETTO DE AZAMBUJA

Designing and Evaluating Hybrid Techniques to Detect Transient Faults in Processors Embedded in FPGAs

A thesis submitted to evaluation
in partial fulfillment of the requirements for the Degree of
Doctor of Computer Science

Prof. Dr. FernandaGusmão de Lima Kastensmidt
Advisor

A thesis submitted to evaluation
in partial fulfillment of the requirements for the Degree of
Doctor of Engineering

Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Advisor

September 2013.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Azambuja, José Rodrigo Furlanetto de

Designing and Evaluating Hybrid Techniques to Detect Transient Faults in Processors Embedded in FPGAs / José Rodrigo Furlanetto de Azambuja – Porto Alegre: Programa de Pós-Graduação em Computação, 2013.109 p.:il.

Thesis (doctorate) – Universidade Federal do Rio Grande do Sul and Karlsruher Institute für Technologie. Programa de Pós-Graduação em Computação and Fakultät für Elektrotechnik und Informationtechnik. Porto Alegre, BR – RS, 2013. Supervisors: Fernanda Gusmão de Lima Kastensmidt and Jürgen Becker.

1. Fault tolerance. 2. Radiation effects. 3. Processors. I. Kastensmidt, Fernanda G. L. and Becker, J.. II. Designing and Evaluating Hybrid Techniques to Detect Transient Faults in Processors Embedded in FPGAs.

ACKNOWLEDGEMENTS

I would like to start by thanking my parents, Tais and Gui, and my brother, Cado, who have always been there for me, being "there" a few meters, or a few thousand kilometers away. Thank you for all the support you provided me, allowing me to devote myself entirely to the studies. I also thank my grandparents, uncles and cousins for the countless moments, which would not fit in these few lines.

I take the opportunity also to thank Georg, who despite not having any kind of kinship with me was like a second father during the time I spent in Germany.

I would also like to thank my colleagues from UFRGS, Anelise, Carol, Eduardo, Fernando, Jimmy, Jorge, Lucas, Samuel, William and Mauricio, for the countless hours discussing a variety of subjects, the chimarrões, and the terrible football championships (and let there be Radecs!), and from KIT, Christoph, Falco, Mahtab and Oliver, for the indoor sports, FPGA boards resuscitation and hat making.

To all my friends who took me out of the lab during the weekends - and also to the ones who kept me in it during the week - and my apologies for not being able to quote you all. To FDC, for all the poorly planned adventures, and to the band, for not letting rock die - and still allowing me to play in it.

To the great friends who I rediscovered and to the great friendships I made during my stay in Germany, who made my day much better and my job much easier, Ari, Bruna, Cilene, Cláudio, Elian, Fada, Irigas, Paola, Pedrão, Gabriel, Oliver, Sharon and Zatt, and to those who I barely saw in person, but have always been fighting evil with me in the fields of justice, Christoph, Dirk, Fabio and Julian.

To my entrepreneurs friends who came to believe in a small project and today are richer than Eike Batista: Inácio and Werner.

Especially to Fernanda, for putting up with me during 7 years. Thank you for the confidence and freedom of research on the most varied subjects. I also thank Professor Jürgen Becker for the many research opportunities inside - and outside - of KIT, as well as for providing me with a great work environment. To Professors Álvaro, Lisboa, Luigi, Michael, Paolo, Reis and Weber, for all their help during my PhD.

To Elisiane, Leivo and to the coordination of PPGC, Instituto de Informática, ITIV, UFRGS, and KIT and to the financial support provided by CNPq, CAPES and HIRST.

Finally, I would like to thank the supernatural forces - in which I do not believe, but I know they exist - that make us go forward and until the end.

To all of you, my sincere Thanks.

TABLE OF CONTENTS

LIST OF ABREVIATIONS AND ACRONYMS

ACCE	Automatic Correction of Control Flow Errors
ACCED	Automatic Correction of Control Flow Errors with Duplication
ALU	Arithmetic and Logic Unit
ASIC	Application Specific Integrated Circuits
AUT	Area Under Test
BB	Basic Block
BID	Block Identifier
BRA	Inverted Branches Technique
BRAM	Block Random Access Memory
CAEN	Chemically Assembled Electronic Nanotechnology
CCA	Control Flow Checking using Assertions
CEDA	Control-flow Error Detection through Assertions
CFCSS	Control Flow Checking by Software Signatures
CFID	Control Flow Identifier
CLB	Configurable Logic Block
COTS	Commercial Off The Shelf
CUT	Circuit Under Test
CWSP	Code Word State Preserving
DUT	Design Under Test
DWC	Duplication With Comparison
ECC	Error Correcting Codes
ECCA	Enhanced Control Flow Checking using Assertions
EDAC	Error Detection And Correction
ED^4I	Error Detection by Data Diversity and Duplicated Instructions
EHP	Electron-Hole-Pairs
ENSERG	Ecole Nationale Supérieure d'Electronique et de Radioélectricité de Grenoble
EoE	End of Execution

FIT	Failure In Time
FF	Flip-Flop
FP	Frame Pointer
FPGA	Field Programmable Gate Array
GUI	Graphic User Interface
GP	Global Pointer
GPP	General Purpose Processor
GPU	Graphic Processing Unit
GSR	Global Signature Register
HDL	Hardware Description Language
HETA	Hybrid Error-detection Technique using Assertions
HPCT	Hardening Post Compiling Translator
IC	Integrated Circuit
Icc	Power Supply Current
ICAP	Internal Configuration Access Port
I-IP	Infrastructure Intellectual Property
IP	Intellectual Property
ISA	Instruction Set Architecture
ITAR	International Traffic in Arms Regulation
JALR	Jump And Link to Register
JR	Jump to Register
JRE	Java Runtime Environment
KIT	Karlsruhe Institute of Technology
LANSCE	Los Alamos Nuclear Science Center
LANL	Los Alamos National Laboratory
LUT	Look-Up Table
MeV	Mega-electron Volts (10_6 electron Volts)
$MIPS_1$	Microprocessor without Interlocked Pipeline Stages
$MIPS_2$	Mega (10_6) Instructions Per Second
NES	Node Exit Signature
NGL	Next Generation Lithography
NIS	Node Ingress Signature
NoC	Network-on-Chip
NOP	No Operation
NRE	Non-recurring engineering

NT	Node Type
OCFCM	Online Control Flow Checker Module
PC	Program Counter
PLL	Phase-Locked Loop
RA	Return Address
RadHard	Radiation Hardened
RAM	Random Access Memory
RISC	Reduction Instruction Set Computer
RTL	Register Transfer Level
SEE	Single Event Effect
SEL	Single Event Latch-up
SET	Single Event Transient
SEU	Single Event Upset
SIC	Structural Integrity Checking
SIHFT	Software Implemented Hardware Fault Tolerance
SoC	System-on-Chip
SP	Stack Pointer
SRAM	Static Random Access Memory
TCL	Tool Command Language
TID	Total Ionizing Dose
TMR	Triple Modular Redundancy
UFRGS	Universidade Federal do Rio Grande do Sul
USML	United States Munitions List
VAR	Variables Technique
VHDL	Very-high-speed integrated circuits Hardware Description Language
XOR	Exclusive OR
XTMR	Xilinx Triple Modular Redundancy

LIST OF FIGURES

LIST OF TABLES

Designing and Evaluating Hybrid Techniques to Detect Transient Faults in Processors Embedded in FPGAs

ABSTRACT

Recent technology advances have provided faster and smaller devices for manufacturing circuits that while more efficient have become more sensitive to the effects of radiation. Smaller transistor dimensions, higher density integration, lower voltage supplies and higher operating frequencies are some of the characteristics that make energized particles an issue when dealing with integrated circuits in harsh environments. These types of particles have a major influence in processors working in such environments, affecting both the program's execution flow by causing incorrect jumps in the program, and the data stored in memory elements, such as data and program memories, and registers. In order to protect processor systems, fault tolerance techniques have been proposed in literature using hardware-based and software-based approaches, which decrease the system's performance, increase its area, and are not able to fully protect the system against such effects. In this context, we proposed a combination of hardware- and software-based techniques to create hybrid techniques aimed at detecting all the faults affecting the system, at low performance degradation and memory overhead. Five techniques are presented and described in detail, from which two are known software-based only techniques and three are new hybrid techniques, to detect all kinds of transient effects caused by radiation in processors. The techniques are evaluated according to execution time, program and data memories, and area overhead and operating frequency degradation. To verify the effectiveness and the feasibility of the proposed techniques, fault injection campaigns are performed by injecting faults by simulation and performing irradiation experiments in different locations with neutrons and a Cobalt-60 sources. Results have shown that the proposed techniques improve the state-of-the-art by providing high fault detection rates at low penalties on performance degradation and memory overhead.

Keywords: fault tolerance, radiation effects, processors.

Desenvolvendo e Avaliando Técnicas Híbridas para Detectar Falhas Transientes em Processadores Embarcados em FPGAs

RESUMO

Os recentes avanços tecnológicos proporcionaram dispositivos menores e mais rápidos para a fabricação de circuitos que, apesar de mais eficientes, se tornaram mais sensíveis aos efeitos de radiação. Menores dimensões de transistores, mais densidade de integração, tensões de alimentação mais baixas e frequências de operação mais altas são algumas das características que tornaram partículas energizadas um problema, quando lidando com sistemas integrados em ambientes severos. Estes tipos de partículas tem uma grande influencia em processadores funcionando em tais ambientes, afetando tanto o fluxo de execução do programa ao causar desvios incorretos, bem como os dados armazenados em elementos de memória, como memórias de dados e programas e registradores. A fim de proteger sistemas processados, técnicas de tolerância a falhas foram propostas na literatura usando propostas baseadas em *hardware*, *software*, que diminuem o desempenho do sistema, aumentam a sua área e não são capazes de proteger totalmente o sistema destes efeitos. Neste contexto, propomos a combinação de técnicas baseadas em *hardware* e *software* para criar técnicas híbridas orientadas a detectar todas as falhas que afetam o sistema, com baixa degradação de desempenho e aumento de memória. Cinco técnicas são apresentadas e descritas em detalhes, das quais duas são conhecidas técnicas baseadas puramente em *software* e três são técnicas híbridas novas, para detectar todos os tipos de efeitos transientes causados pela radiação em processadores. As técnicas são avaliadas de acordo com o aumento no tempo de execução, no uso das memórias de dados e programa e de área, e degradação da frequência de operação. Para verificar a eficiência e aplicabilidade das técnicas propostas, campanhas de injeção de falhas são realizadas ao se simular a injeção de falhas e realizar experimentos de irradiação em diferentes localidades com nêutron e fontes de Cobalto-60. Os resultados mostraram que as técnicas propostas aprimoraram o estado da arte ao fornecer altas taxas de detecção de falhas com baixas penalidades em degradação de desempenho e aumento de memória.

Palavras-Chave: tolerância a falhas, efeitos de radiação, processadores.

Entwurf und Auswertung von Hybrid-Techniken zur Erkennung von transienten Fehlern in FPGA eingebetteten Prozessoren

ZUSAMMENFASSUNG

Der aktuelle Stand der Technologie bringt schnellere und kleinere Bausteine für die Herstellung von integrierten Schaltungen mit sich, die während sie effizienter sind auch anfälliger für Strahlung werden. Kleinere Abmessungen der Transistoren, höhere Integrationsdichte, geringere Versorgungsspannungen und höhere Betriebsfrequenzen sind einige der Charakteristika, die energiegeladene Partikel zu einer Herausforderung machen, wenn man integrierte Schaltungen in rauen Umgebungen einsetzt. Diese Art der Partikel hat einen sehr großen Einfluss auf Prozessoren, die in einer solchen Umgebung eingesetzt werden. Sowohl die Ausführung des Programms, welche durch fehlerhafte Sprünge in der Programmsequenz beeinflusst wird, als auch Daten, die in speichernden Elementen wie Programmspeicher, Datenspeicher oder in Registern abgelegt sind, werden verfälscht. Um solche Prozessorsysteme abzusichern, wird in der Literatur Fehlertoleranz empfohlen, welche die Systemperformanz verringert, einen größeren Flächenverbrauch mit sich bringt und das System dennoch nicht komplett schützen kann. Diese Fehlertoleranz kann sowohl durch software- als auch durch hardwarebasierte Ansätze umgesetzt werden. In diesem Zusammenhang schlagen wir eine Kombination aus Hardware- und Software- Lösung vor, welche die Systemperformanz nur sehr wenig beeinflusst und den zusätzlichen Speicheraufwand minimiert. Diese Hybrid-Technologie zielt darauf ab, alle Fehler in einem System zu finden. Fünf solcher Techniken werden beschrieben und erklärt, zwei der vorgestellten Techniken sind bekannte Software-Lösungen, die anderen drei sind neue Hybrid-Lösungen, um alle transienten Effekte von Strahlung in Prozessoren erkennen zu können. Diese unterschiedlichen Ansätze werden anhand ihrer Ausführungszeit, Programm-, Datenspeicher, Flächenvergrößerung und Taktfrequenz analysiert und ausgewertet. Um die Effizienz und die Machbarkeit des vorgeschlagenen Ansatzes verifizieren zu können, werden Fehlerinjektionstests sowohl durch Simulation als auch durch Bestrahlungsexperimente in unterschiedlichen Positionen mit einer Cobalt-60 Quelle durchgeführt. Die Ergebnisse des vorgeschlagenen Ansatzes verbessern den Stand der Technik durch die Bereitstellung einer höheren Fehlererkennungsrate bei sehr geringer negativer Beeinflussung der Performanz und des Speicherverbrauchs.

Stichworte: Fehlertoleranz, Strahlungseffekte, Prozessoren.

1 INTRODUCTION

Since the 1950's, when computers were made with vacuum tubes and the personal computer was nothing more than a dream, fault tolerance has been an important topic of interest. In the early ages of computers, their usage was very specific, aimed at activities such as military applications, precise calculations and space missions. An error in these applications working in such harsh environments, and sometimes even in remote places like as the space, could completely jeopardize the mission, since the repair and, in some cases, the available time to repair the system was impractical. From then until nowadays many things have changed. The technology no longer relies on vacuum tubes, but on transistors, computers became ubiquitous and a personal computer can be found in any cell phone. One the other hand, a few things remained the same, such as the old topic on how to give a system the ability to cope with a fault and continue its correct operation, or in other words, fault tolerance.

The technology did not only advance in the past, but it is constant progress. Today we can observe new trends, such as the continued reduction in transistor sizes, new fabrication processes and materials, low-power systems to fit small ubiquitous microprocessors and medical applications and, as always, the need for more processing power through higher frequencies of operation and more processing cores per die. At the same time that these advances push technology forward, they increase the system's susceptibility to noises that are present in the environment. One well-known effect is the radiation effect present in space applications but also at avionic and at ground-level. Due to the reduced transistor size and low voltages, the fault effect in integrated circuits is increasing in magnitude; consequently, fault tolerant techniques will become mandatory in all devices in the close future.

This document exposes the reasons behind the concerns surrounding a system's fault tolerance. Some of the most important fault tolerant techniques presented in the literature are analyzed and three hybrid techniques to detect faults in processors are proposed as the thesis work. In order to prove the effectiveness of the proposed techniques, fault injection campaigns were performed, including configuration bitstream fault injection, and neutron and Cobalt-60 beams irradiation experiments.

1.1 Motivation and Problem Definition

The recent advances in the semiconductor industry have led in the development of more complex components and systems' architectures by allowing fabrication processes to place a higher number of transistors per area of the silicon die. The CMOS technology has developed according to Moore's law (MOORE, 1965), where the number of transistors on Integrated Circuits (IC) doubles every two years. Nowadays, with factories fabricating transistors with 32nm, we are reaching the physical limits of a couple atoms to form the transistor's gate (KIM, 2003), (HOMPSON, 2005). Still, new

technologies arise, such as Next Generation Lithography (NGL) and Chemically Assembled Electronic Nanotechnology (CAEN), promising to reduce feature sizes to 20nm and less.

However, the same technology that made possible all this progress has also reduced the transistor reliability by reducing threshold voltages, node capacitances and tightening the noise margins (BAUMANN, 2001). These have made transistors more susceptible to faults caused by radiation interference, which can be energized particles presented on space or secondary particles such as alpha particles, generated by interaction of neutron and materials at ground level. As a consequence, mission-critical applications, such as space applications or avionics, demand fault tolerant techniques capable of recovering the system from a fault with minimum implementation and performance overhead.

Processors working in harsh environments can be upset by such energized particles. They affect processors by modifying values stored in memory elements (such as registers and data memory), by leading the processor to incorrectly execute an application by jumping or re-executing some instructions, or even by entering in a loop and never finishing the application. Such faults can also modify some computed data values, generating errors in the data results. Therefore, the use of fault tolerance techniques is mandatory to detect and/or correct these types of faults. Since processors run software on top of their hardware, fault tolerance techniques used to harden them can be based on software, hardware or hybrid solutions.

Fault tolerance techniques based on software rely on adding extra instructions to the original program code to detect or correct faults (BOLCHINI, 2005). They may be able to detect faults that affect the data and the control flow. Software-based techniques provide high flexibility, low development time and cost, since there is no need to modify the hardware. In addition, new generations of microprocessors that do not have Radiation Hardened (RadHard) versions can be used. As a result, aerospace applications can use commercial off the shelf (COTS) microprocessors with RadHard software. However, some results from random fault injection have shown the impossibility of achieving complete fault coverage when using only software-based techniques, due to control flow errors (AZAMBUJA, 2011b). This limitation is due to the inability of the software in protecting against all possible control flow effects errors that can occur in the microprocessor. One example would be a fault in the Program Counter (PC), where only a single No OPeration (NOP) instruction can execute. In this case, it does not matter how many instructions or data are added to the original code, because none of them will be executed by the processor and no fault will be detected.

As a consequence of the redundant instructions inserted in the original program code, software-based techniques have as drawbacks overheads in both data and program memory, and degradation in performance, due to an increased computation time required to execute the program. The program memory increases due to additional instructions inserted in the original program code, while the data memory may increase due to variable replication, which can be, for example, the replication of all stored data in the memory. Performance degradation comes from the execution of redundant instructions.

Hardware-based techniques usually change the original processor architecture by adding redundant logic, such as module replication with majority voters, information redundancy, such as Error Correcting Codes (ECC), or time redundancy, such as the one presented by Anghel (2000). Hardware-based techniques can also be based on

hardware monitoring devices that are inserted to the system's architecture, such as a verification hardware module, and therefore change the system's architecture, but not the processor's architecture and in this case are non-intrusive. They exploit special purpose hardware modules, called watchdog processors, to monitor memory accesses.

As a consequence of the extra logic or information redundancy, hardware-based techniques increase the processor's area up to more than three times the original size, which leads to more power consumption and production costs. Time redundancy usually does not have a big impact in area, but decreases considerably the performance, since they affect the execution time and, in some cases, the operation clock frequency. Non-intrusive hardware modules, such as watchdogs, may have a smaller impact in area and operating clock frequency, but they require access to processor's connections, such as access to the data and code memory connections. Watchdogs do not detect faults that are latent inside the microprocessor, as faults in the register bank, because they do not have access to the processor's internal buses (in order to be non-intrusive).

1.2 Overview

In order to provide reliable systems that can cope with radiation effects, we believe that the solution lies in combining software-based and hardware-based techniques. The main objective of this work is finding the best trade-off between software-based and hardware-based techniques to increase existing fault detection rates up to 100%, while maintaining low overheads in performance, in the means of operating clock frequency and application execution time, and area, in the sense of both program and data memory overhead, and extra hardware modules.

The first step to achieve our goal was to analyze the radiation effects on integrated circuits. In a second step, we performed an analysis on the influence of such effects in processors, such as the effects of a fault at a given part of the processor and its influence on the results of a running program. To do so, we checked the affected area in processor in the sense of which areas are more sensitive to which types of effects, and also the effect of a particle in the processor, in the sense of control or data flow effects. Then, we searched the literature for existing fault tolerant techniques using hardware-based techniques, software-based techniques and hybrid ones. As a result, none of the available techniques in the literature could fully protect a processor against transient effects without huge drawbacks, such as performance degradation and area overhead.

We started by implementing two known software-based techniques, called Variables and Inverted Branches (AZAMBUJA, 2010b). From there, we proposed three hybrid techniques, based both on software and hardware replication characteristics. The implemented techniques are generic and could be implemented to any application, but in order to focus this work in the techniques, instead of the application, we chose a few case-study applications and implemented the techniques for each of them. Results showed low performance degradation and memory overhead, when compared to techniques in presented in the literature.

In order to check the effectiveness and feasibility of the proposed techniques, we performed three fault injection campaigns. The first one consisted of simulating faults through a commercial simulator, where we injected hundreds of thousands of faults in the chosen case-study applications. For the second injection campaign, we used a Field Programmable Gate Array (FPGA) based on Static Random Access Memory (SRAM) configuration bitstream and exhaustively injected millions of faults in the configuration memory bitstream. For the third fault injection campaign, we used cyclotrons around

the world to hit the integrated circuit hardened by our techniques with neutron and cobalt-60 beam sources. For the last campaign, we used SRAM-based and flash-based FPGAs to implement the Design Under Test (DUT) and test them for radiation effects.

Preliminary results have shown interesting results, when compared to related works in the literature. The performance degradation combined with area and memory overhead improved the state-of-the-art. These results have been backed by intense fault injection campaigns, performed both by simulation and irradiation experiments in the sense that the proposed techniques can indeed be applied to protecting processors in harsh environments.

Chapter 2 presents the terminology and general concepts used in this work. Chapter 3 describes existing fault tolerant techniques for processors presented in the literature. Chapter 4 describes the fault tolerant techniques implemented in this work to detect transient faults in processors, from which two are known software-based and three are new hybrid techniques. Chapter 5 presents experimental fault injection campaigns for the implemented fault tolerant techniques. Chapter 6 presents the configuration bitstream fault injection campaign and results. Chapter 7 presents radiation experiments on some of the proposed techniques. Chapter 8 describes future work and concludes the thesis.

2 BACKGROUND

In this chapter, we introduce the main technical terms used in the text, describe the microprocessor architecture used as a case study and discuss background information required for better enlightenment of the topics in this thesis.

2.1 Basic Concepts of Dependable and Secure Computing

This section presents the basic set of definitions that will be used throughout this work. The definitions encompass from defects and upsets that occur in individual logic gates to fault, error and failure.

2.1.1 Defect, Upset and Fault Definitions

Defect or upset is defined as unintended differences between the implemented system and its intended function. It can be commonly a manufacture defect, for example, or transient upsets that happen during some perturbation of the environment.

Fault is then defined as a logic level abstraction of a physical defect or upset. It is used to describe the change in the logic function of a device caused by the defect or upset. Fault can be described as a deviation from the expected behavior of logic. Faults can be transient, intermittent or permanent. Transient faults occur and then disappear. They are transient effects that may occur during the lifetime of the component and it exists for a short period of time. Intermittent faults are characterized by a fault occurring, then vanishing, and then reoccurring and so on. An example of intermittent faults is signal interferences such as cross-section between connection lines. Permanent faults continue to exist in the system until the faulty component is repaired or replaced. They are usually due to manufacturing problems. Some defects or upsets may be masked by the electrical properties of the device and no fault may be observed. Usually when there is a fault in the circuit, there will be an error. However, some faults may be masked by some logic, electric or application and no error will be observed. Error is considered a wrong output signal produced by a defective system.

With nanometer dimension technologies, transistors have become more susceptible to faults caused by radiation interference due to reduced threshold voltages, reduced node capacitances and tightened noise margins (BAUMANN, 2001). Such faults can be caused by energized particles present in space or secondary particles such as alpha particles, generated by the interaction of neutron and materials at ground level (INTERNATIONAL, 2005). Integrated circuits operating in a space environment are sensitive to these particles and can be affected mainly by transient ionization and long term ionizing damage.

In the following, we will discuss the main effects from radiation interference that cause upsets in integrated circuits.

2.1.1.1 Single Event Effect (SEE):

Transient ionization may occur when a single radiation ionizing particle strikes the silicon, creating a transient voltage pulse, or a Single Event Effect (SEE). This effect can be destructive or non-destructive. An example of destructive effect is Single Event Latchup (SEL) that results in a high operating current, above device specifications, that must be corrected by a power reset. Non-destructive effects, also known as soft errors, are transient effects provoked by the interaction of a single energized particle in the PN junction of an off-state transistor (DODD, 2004). When the transient pulse occurs in a memory element, such as a register, it is classified as Single Event Upset (SEU). When the particle hits a combinational element, inducing a pulse in the combinational logic, the upset is classified as Single Event Transient (SET).

Figure 2.1 shows examples of SEU and SET effects in a circuit. On the left, one can see the SEU effect. A particle, represented by the bolt, hits the sequential logic (which could be seen as a register), changing the store value from “0010” to “0110”. This effect directly affects the rest of the circuit, changing the value stored in the sequential logic on the right from “1” to “0”. In the middle, one can see a particle hitting the NOR gate and causing a voltage pulse in the combinational logic. When propagated, the pulse hits the sequential latch window on the sequential logic to the right, which registers the incorrect value “0”, instead of a “1”. Such effects may be masked by the circuit, as discussed in the following subsections.

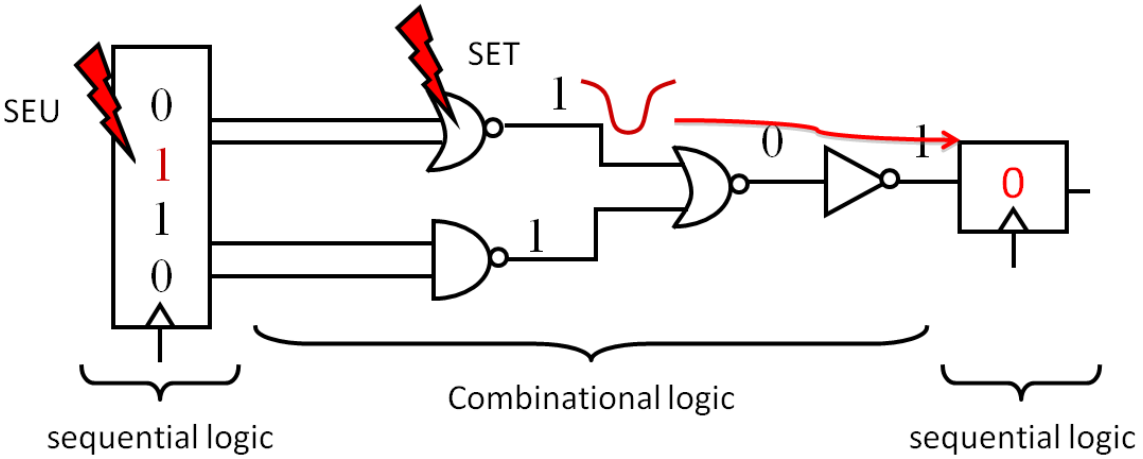

Figure 2.1: SEU and SET effects on a circuit.

Soft errors can be detected and corrected by the system’s logic, meaning that it does not require a hard reset to recover from an error. Sections 7.1.2 and 7.2.2 present neutron irradiation experiments simulating the effect of SEE in Flash-base and SRAM-based FPGAs, while Chapters 5 and 6 present fault injection simulation experiments simulating SEEs at RTL level and in the configuration memory bitstream, respectively. In this thesis, SEUs and SETs will be used to describe transient faults that the techniques presented here can cope with.

2.1.1.2 Total Ionizing Dose (TID):

The long term ionizing damage is also known as Total Ionizing Dose (TID). It is caused by the interaction of energized particles with atoms of the silicon. Photon-induced damage is initiated when Electron-Hole-Pairs (EHP) are generated along the track of secondary electrons emitted via photon-material interactions. EHPs are created from a fraction of the kinetic energy of the incident particles. Some of them are

annihilated due to recombination, but a few remain in the silicon. The remaining EHPs may fall into deep traps in the oxide bulk or near the Si/SiO_2 interface, forming trapped positive charges (BARNABY, 2006), (OLDHAM, 2003). By doing so, TID can affect the system by shifting the threshold voltage, generating leakage current and timing skews and even leading to functional failures. Sections 6.1.2 and 6.2.2 present neutron irradiation experiments using Flash- and SRAM-based FPGAs, respectively, while Section 6.1.1 discusses the effects of TID in SRAM-based FPGAs.

In this thesis, we will refer fault as the single event transient (SET) pulse that may occur in the combinational logic and as the single event upset (SEU) that is the bit-flip that may occur in the memory element.

2.1.2 Error and Failure Definitions

The design of fault tolerant systems consists in preventing a fault to cause an error and consequently a failure in the implemented system. Therefore, there is a cause-effect relationship from the particle hit (fault) to the erroneous result (system failure), as demonstrated in Figure 2.2. In this work, we will use the definition presented in Avizienis (2004).

In order to define error and failure, we first have to define a system. A system is an entity that interacts with other entities, such as other systems, hardware, software, and the physical world. A system follows a functional specification, composed of several different functions. The behavior of a system is what it does to implement its functions and is described by a sequence of states. Finally, the service delivered by a given system is its behavior, as it is perceived by its users.

Failure is the abbreviation of service failure and is defined as a system malfunction, or in other words, when the delivered service deviates from the correct one. The delivered service is considered correct when it is according to the system specification. When the service specification includes a set of several functions, the failure of one or more of the services implementing the functions may lead the system to a degraded mode that still offers a subset of needed services. We define this case as a partial failure.

Error is defined as the deviation in one of the system's sequence of states. Such deviation may compromise a system service, thus leading to a service failure. It is important to note that an error not always leads to a failure.

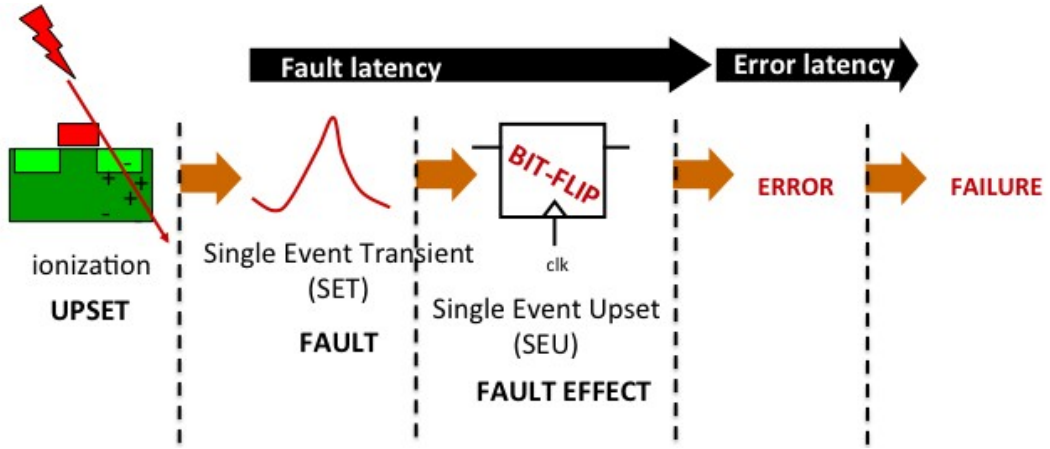

Figure 2.2: Upset, fault, error and failure chain-effect for SET and SEU.

By defining fault, error and failure, one can notice that a failure can always be seen by the user, since it leads to a system malfunction. Faults can be latent in the circuit

until manifested as an error. There are detection techniques that can detect faults and there are techniques that can detect errors.

Faults can also be masked by three different ways in the circuit: logical masking, electrical masking and latch window masking. Figures 2.3, 2.4 and 2.5 show each of them, respectively. The logical masking is when the logic of the gate being hit by a fault masks its effect. Figure 2.3 shows a NOR gate being hit by a particle and forcing its output to "1". As one can see the fault cannot propagate through the circuit as the other NOR gate has one of the inputs at value '1', which is dominant forcing the output of that gate to '0'. Electrical masking happens when the propagation of the pulse is weakened by the logic, such as the one shown in Figure 2.4. A NOT gate is hit by a particle, generation a high voltage pulse in its output. When propagating through the other three NOT gates, the pulse is weakened until electrically masked. Latch window masking happens when the pulse does not hit the latching window of a sequential logic. Figure 2.5 shows a clock cycle with its latching window. When the voltage pulse lasts for the whole latching window, the errors are stored in the logic. When the pulse does not last until the latch window (as shown in the bottom), the fault is masked.

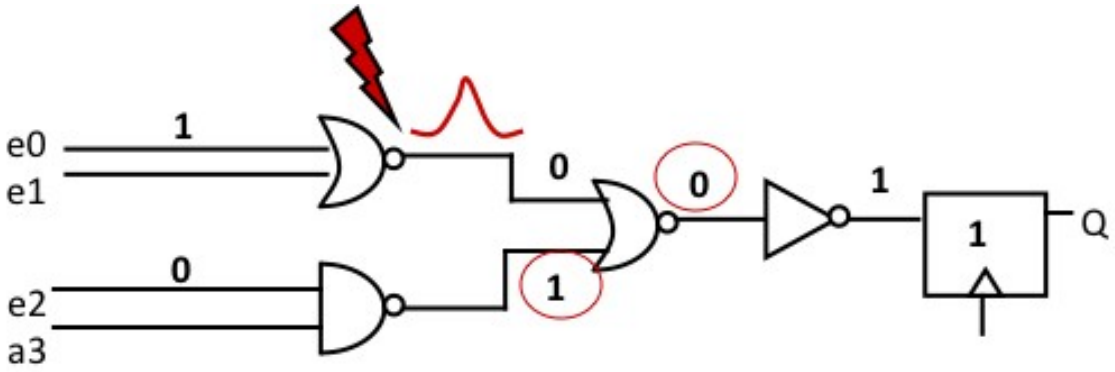

Figure 2.3: Logical masking.

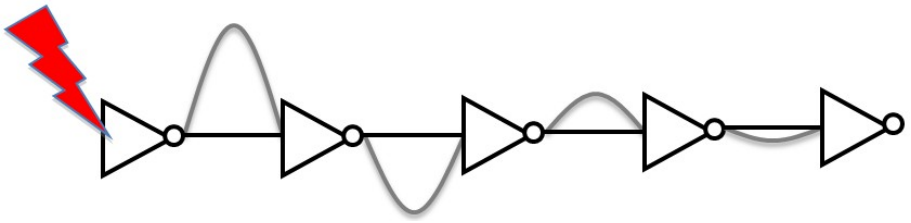

Figure 2.4: Electrical masking.

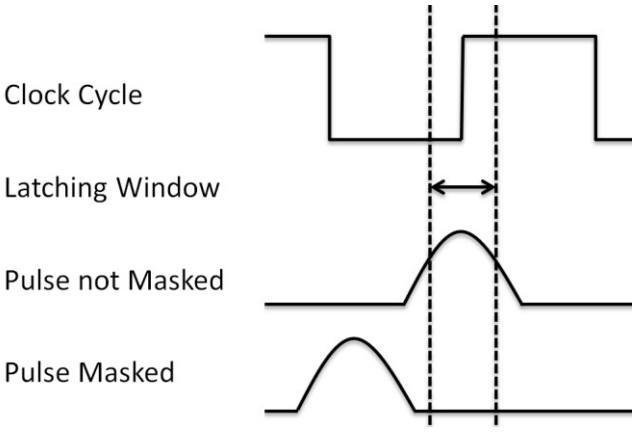

Figure 2.5: Latch window masking.

2.2 MIPS Architecture

The case study microprocessor used in this work is the Microprocessor without Interlocked Pipeline Stages (MIPS). It has a standard processor architecture based on the Reduced Instruction Set Computing (RISC) instruction set. The basic idea behind RISC is that using simple instructions, which enable easier pipelining and larger caches, the performance can be largely boosted. The MIPS architecture can be seen since 1985 in commercial applications, from workstations to Windows CE devices, routers, gateways and PlayStation gaming devices.

Among the different MIPS architecture processors, there is the miniMIPS (HANGOUT, 2013), which will be used in the work as case-study microprocessor. The miniMIPS is an open source processor that has a reduced instruction set from the original MIPS architecture, with 52 instructions. It is described in the hardware description language, Very-high-speed integrated circuits Hardware Description Language (VHDL). Consequently, it can be logically simulated and it can be synthesized into programmable circuits as FPGAs. The miniMIPS can be implemented using the Harvard memory model, where the program and data memory are separated in two different memories, or the Von Neumann, where program and data memory share the same memory. It has five stages pipeline: Instruction Address Calculation, Instruction Fetch, Instruction Decode, Execution and Memory Access. A model of the miniMIPS pipeline is shown in Figure 2.6.

The miniMIPS microprocessor has thirty-two 32-bit registers in the register bank. It also has a PC with a simplistic logic, since it has fixed size instructions and static branch prediction. Besides the PC, the microprocessor has other special purpose registers, such as the Stack Pointer (SP), Global Pointer (GP), Frame Pointer (FP), Return Register (RA) and Zero (always has the value 0), which can all be found in the register bank. The miniMIPS uses a gcc cross-compiler to translate C code into executable code.

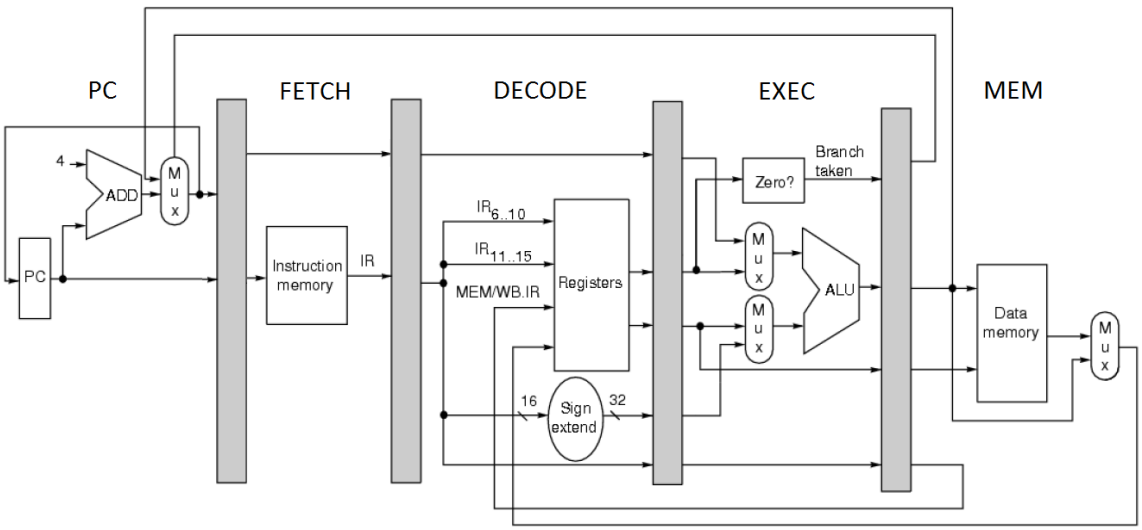

Figure 2.6: Pipeline architecture of the miniMIPS.

The instruction set used by miniMIPS has a fixed size of 32 bits, from which only two have access to the memory: the load instruction and the write instruction. Instructions have a 6-bit opcode and are divided in three classes: type-R, which specify three registers (*rs*, *rt* and *rd*), plus a shift value (*shamt*), and a function field (*funct*), type-I, which specify two registers (*rs* and *rt*) and a 16-bit immediate value, and type-J,

which do not specify any register, only a 26-bit address. Figure 2.7 shows these classes in detail.

Table 2.1: MIPS' instruction format

Type	-31- format (bits) -0-					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

The miniMIPS microprocessor has been chosen as a case-study to this work due to the following reasons. The first one is that it is largely used in the literature (also because it has been in development since 1985). It also has a simple, but efficient, architecture with RISC architecture and a 5-stage pipeline, which follows modern microprocessor models, such as Intel's and ARM's. The miniMIPS version has a very stable version, since 2009, and has been simulated and implemented in various platforms, from FPGAs (both Flash- and SRAM-based) to ASICs. The miniMIPS version used in this work was initially developed by the Ecole Nationale Supérieure d'Electronique et de Radioélectricité de Grenoble (ENSERG), made open source in Hangout (2013), and slightly improved at UFRGS (modules such as the branch prediction, as well as memory controllers).

In this thesis, we use a VHDL model of the mini-MIPS that can be logical simulated and synthesized to ASIC and to programmable platforms as FPGAs.

2.3 SEE in MIPS Processors

A processor is nothing more than a group of sequential and combinational circuits combined in one component. This combination of different circuits induces processors to be sensitive to different radiation in different areas with different effects. A processor could be roughly divided in five logical groups according to the area (program memory, data memory, register bank, control path, and data path) and in two logical groups when relating to the effect of a fault (data flow and control flow). In the following subsections, we will address how SEEs affect each part of a processor and theirs effects.

2.3.1 SEEs Divided by Sensitive Areas of a Processor

Memories are sequential circuits and therefore very sensitive to SEUs. Due to their regular physical structure, they are optimized to fit in smaller die areas than normal circuits and normally with higher operation frequencies. That means that the radiation effects, such as multiple-bit upset due to a single particle, are intensified in memory components. There are two main types of available memory: flash and SRAM memory. The first one is less sensitive to radiation effects, because it requires a high voltage to be written (change their current state), typically higher than 5V. Its main drawbacks are that it has a finite number of program-erase cycles, meaning that a memory position can only be written around 100,000 times before deteriorating its integrity, and because of the fact that normally the write circuit needs to pump up the voltage in order to reach

the high voltage for writing, and this circuit is sensitive to radiation effects. The SRAM memory is more sensitive to radiation effects, since it operates in normal voltages, but has better performance and power consumption. Besides, it doesn't have the finite number of program-erase cycles.

The memory organization of a processor can be with program and data memory combined in the same memory element (Von Neumann), or separated (Harvard). When separated, the program memory is usually stored in a flash memory and the data memory in an SRAM memory. By doing so, it is possible to reduce the number of upsets in the program memory, while the data memory can be protected by fault tolerant techniques. When sharing the same memory, program and data memories are typically implemented on SRAM memory. Because of the fact that fault tolerance techniques are too expensive to protect the program memory, low-level approaches, such as Error Detection And Correction (EDAC) must be implemented on the memory.

The register bank is mostly a sequential circuit, just like the program and data memory. Because of that, it is very sensitive to SEUs. The register bank can be implemented over an SRAM memory or by using flip-flops. In the first case, the same principles from the data program are applied. In the second case, hardware replication can be used, or even software-based technique to replicate the information stored in the registers. The miniMIPS has thirty-two 32-bit registers, resulting in a total of 1024 bits, which is a big number, when considering radiation effects.

The data path represents the computing circuit of the processor. It is defined as the circuit leading from a stored value (in the memory or in the register bank), through the Arithmetic and Logic Unit (ALU), and back to a store element. It is composed of both combinational and sequential logic, since the data path not only processes data, but also crosses the register barriers from the pipeline stages. Because of that, it is sensitive to SEUs (in the pipeline registers) and SETs (in the computing logic, such as the ALU). The effect of a fault in the data path usually leads to an erroneous result in the end of the computation, but hardly leads to an infinite loop, or a control flow error.

The control path is defined as the decision logic of a processor. It is responsible for calculating the next instruction to be fetched and setting the internal flags, such as to command the ALU to sum or subtract and a branch to be taken or not. The control path is mostly combinational, but since it has to cross the pipeline stages, also has sequential logic. The main difference between the control path and the data path is that an error in the control path leads to control flow errors, such as a branch being taken, when it should not have. Such control flow errors may lead to an erroneous result in the end of the computation.

Figure 2.8 shows a detailed view of the architecture of the miniMIPS with the sensitive areas.

2.3.2 SEEs Divided by Effect on a Processor

Faults can be classified as having data or control flow effect in a processor. Data flow effect is defined as an error in a variable during the computation. It means that the program was correctly executed, but with an erroneous result. An example would be the instruction "Registers A = Register B + Register C", where the value stored in A would be "Register B + (Register C + 1)", due to an SEU that happened in Register C. It means that the processor correctly performed the sum in the ALU, but register C had an incorrect value. Control flow effect is defined as an error in the program execution. It means that the variables were correct, but the computation was incorrect. An example

would be the same instruction, “Registers A = Register B + Register C”, where the value store in A would be “Register B - Register C”, due to an SET in the ALU that subtracted the registers, instead of summing them.

In order to differentiate a control flow from a data flow error, we check the PC evolution and compare it with a golden module. In case of a mismatch, the fault is classified as a control flow effect. If not, it is classified as a data flow effect. In some cases, a fault with a data flow effect may cause a control flow effect. An example could be an error in a register used to decide whether a branch should be taken or not. In such cases, we consider it as a control flow effect.

The relation between the location of a fault and its effect is not direct. A fault in the register bank not necessarily will have a data flow effect on a processor. Likewise, a fault in the control path will have a control flow effect.

Figure 2.8 shows the difference between a control flow and a data flow effect in a processor. On the left, one can see a control flow effect, where a jump in the program execution causes an error. On the bottom right, one can see the data memory with errors due to a data flow effect fault.

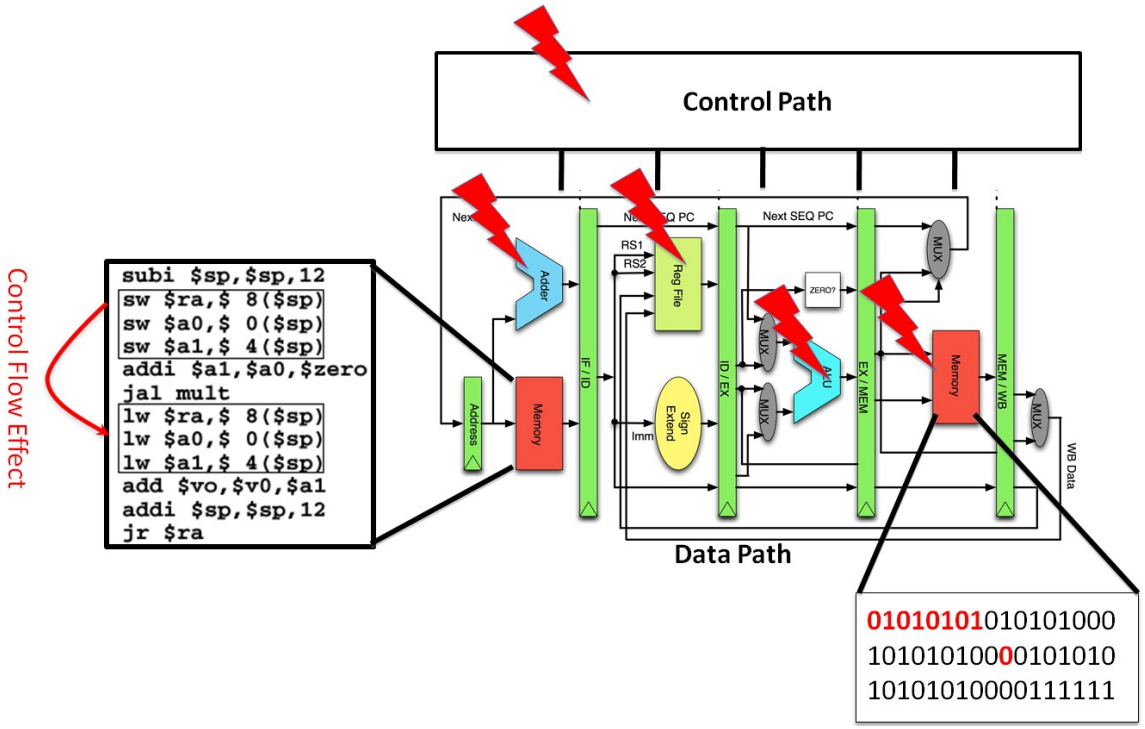

Figure 2.7: miniMIPS sensitive areas under SEE.

2.4 Fault Injection and Testing

Although the effect of faults is increasing at ground level, the rate is not yet sufficient to test fault tolerant techniques. In order to do so, fault emulation and testing is necessary. In this Section, we will go over a few options to do so, such a software fault injection by simulation, fault injection in the FPGA’s memory configuration bitstream and irradiation experiments.

2.4.1 Fault Injection by Simulation

Fault injection by simulation can be done by injecting faults at logical or electrical level in commercial simulators, such as ModelSim, from Mentor, or open source simulators, such as Spice. The main idea behind fault injection by simulation is to add interferences to the circuit. The good side about it is that it offers a huge control over the fault injection because it can be clock cycle accurate and therefore big amounts of data as a result. The drawback is that the description of the circuit is needed. If at electrical level, SPICE level description is used. If at logical level, Register Transfer Level (RTL) hardware description level can be used or logical level description. By simulating at RTL hardware description, one can get all values from all the signals implementing the circuit (and memory values) during simulation. Also, it is possible to stop the simulation, access any value inside the circuit, and resume the simulation. The main drawback is that fault injection by simulation requires huge computational power. The injection of 100,000 faults, depending on the abstraction level, may take a few days to finish.

One example was presented in Azambuja (2010b), where a fault injector by logical simulation was introduced. It was implemented in Java and could generate a script to be run in ModelSim. The software had as inputs the list of signals describing the DUT, a definition file containing the description of the faults to be injected and a definition file containing the description of the application, with information such as runtime, correct output values and the memory used by it.

2.4.2 FPGA Memory Configuration Bitstream Fault Injection

In this type of fault injection, the FPGA board is used to replace the simulator. The circuit is implemented using the FPGA, which can emulate the circuit behavior at RTL level. By doing so, instead of using the simulator, a much faster hardware circuit is used. The mechanisms to inject faults and controlling the process are more complex and harder to be implemented, but the speed improvement makes it possible to inject faults in much higher rates than by injecting faults by simulation.

There are mainly two techniques to perform fault injection in FPGA emulation. Using FPGA reconfiguration mechanisms, a fault is injected by loading a new bitstream into the FPGA, which corresponds to the original bitstream with one or more bits flipped (faulty circuit). Partial reconfiguration can be used to reduce the size of the bitstream. In this case, faults are injected in all configuration bits of the FPGA emulating upsets in the bits that control the routing, the user flip-flops and the bits that program the combinational circuits, the well-known lookup tables (LUT). Examples of FPGA fault injectors in the bitstream are FT-Unshades (NAPOLES, 2007) and the one introduced by Nazar (2012b).

Alternatively, circuit instrumentation can be used for fault injection. Circuit instrumentation consists in inserting some hardware modules, also called instruments. They can provide external controllability and observability to inject a fault and observe its effects. Circuit instrumentation is an automatic process that is basically performed by substituting cells of the DUT by their equivalent instrumented cells. Then, the instrumented circuit is prototyped in the FPGA. An example is AMUSE (ENTRENA, 2010).

2.4.3 Irradiation Experiments

The last and closest to real space radiation are irradiation experiments. It is possible, at ground level, to use energized particles to emulate the particles present in space. In order to do so, particles are accelerated and thrown at the circuit under test. Such equipment is called Cyclotron and can be found in different places in the world, using different types of energized particles. They can accelerate heavy ions, protons and neutrons.

Such experiments require a facility that can accelerate particles at high energies, such as 10MeV, with a constant flux of particles. Equipment to measure the flux, time of exposure and energy of particles is also required. Due to the complexity of the experiment, costs for renting the facilities and the danger involved, irradiation experiments can be very expensive. Because of that, irradiation campaigns take a long preparation time, in order to guarantee that the tested circuit will work, as well as the measurements. On the other hand, irradiation experiments are the closest we can get to simulate the space environment at ground level.

Examples of cyclotrons are the heady ions source in Leuven, Belgium, the proton source in Karlsruhe, Germany, and neutron sources in Didcot, United Kingdom, and Los Alamos, USA. For TID testing, a cyclotron can be found in Sao Jose dos Campos, Brazil, with a Cobalt-60 as radioactive source.

3 FAULT TOLERANCE TECHNIQUES FOR PROCESSORS

Fault tolerance techniques aiming to detect transient effects can be mainly divided in three broad categories: (1) software-based techniques, (2) hardware-based techniques and (3) hybrid techniques. Fault tolerance techniques can be applied at different levels of implementation, starting from the software level down to the architecture description level, the logical and transistor level, until the layout level. In this thesis, we will focus on techniques applied at software level and hybrid techniques.

Fault tolerance techniques based on software rely on adding extra instructions to the original program code to detect and/or correct faults (GOLOUBEVA, 2003), (OH, 2002b). They may be able to detect faults that affect the data and/or the control flow. Software-based techniques provide high flexibility, low development time and cost, since there is no need to modify the hardware. In addition, new generations of microprocessors that do not have RadHard versions can be used. As a result, aerospace applications can use Commercial Off-The-Shelf (COTS) processors with RadHard software. However, results from random fault injection campaigns have shown the impossibility of software-based techniques alone in achieving complete fault coverage for SEU (BOLCHINI, 2005), (AZAMBUJA, 2011a). This limitation is due to the inability of the software to protect all the possible control flow effects that can occur in the microprocessor.

As a consequence of the redundant instructions inserted in the original program code, software-based techniques have as drawbacks high overheads in program memory footprint and a significant increase in the execution time. The program memory increases due to the additional instructions inserted into the original code, while the data memory increases due to variable replication (in some cases, variables store in the data memory are duplicated). Performance degradation comes from the execution of redundant instructions (GOLOUBEVA, 2003), (OH, 2002).

Hardware-based techniques change the original architecture of the system or its components by adding extra hardware modules. Such techniques must be implemented during the design of the system to be hardened. Therefore, they are not suited for hardening COTS processors or closed Intellectual Property (IP) components. Their main market is Application Specific Integrated Circuits (ASICs) and FPGA based systems. Hardware-based techniques can be intrusive, when they modify the architecture of a processor, or non-intrusive, when they do not modify the processor's architecture, but the system's architecture, through communication buses or by adding extra hardware modules that do not require changes inside the components from the system. The most common non-intrusive technique is called watchdog processor (MAHMOOD, 1988), where a small hardware module uses the access between processor and memory to

check the processor's transitions and then monitor its behavior. Intrusive techniques are mainly related to replicating hardware and adding logical and arithmetic checkers.

As a consequence, hardware-based techniques can be very expensive duo to changes in the design project, or Non-Recurring Engineering (NRE) costs, development time, verification time, and testing. Also, besides the price in extra die area to fit the redundant hardware modules, it is very common that RadHard processors have lower performances than non-hardened components because they also are fabricated in older technologies compared to the state-of-the-art COTS processors due to the cost of developing new RadHard processors at any new technology release. Figure 3.1 shows a graphic comparing RadHard with COTS processors according to the processor throughput, or Mega (10_6) Instructions Per Second (MIPS). As one can see, the graphic shows an approximate 10 year gap. It is true that MIPS is not a fair comparison between processors with different architectures (which is the case of the figure), but is still valid to show that there is a gap in performance between COTS and RadHard processors. It may not be as large as 10 years, but it exists.

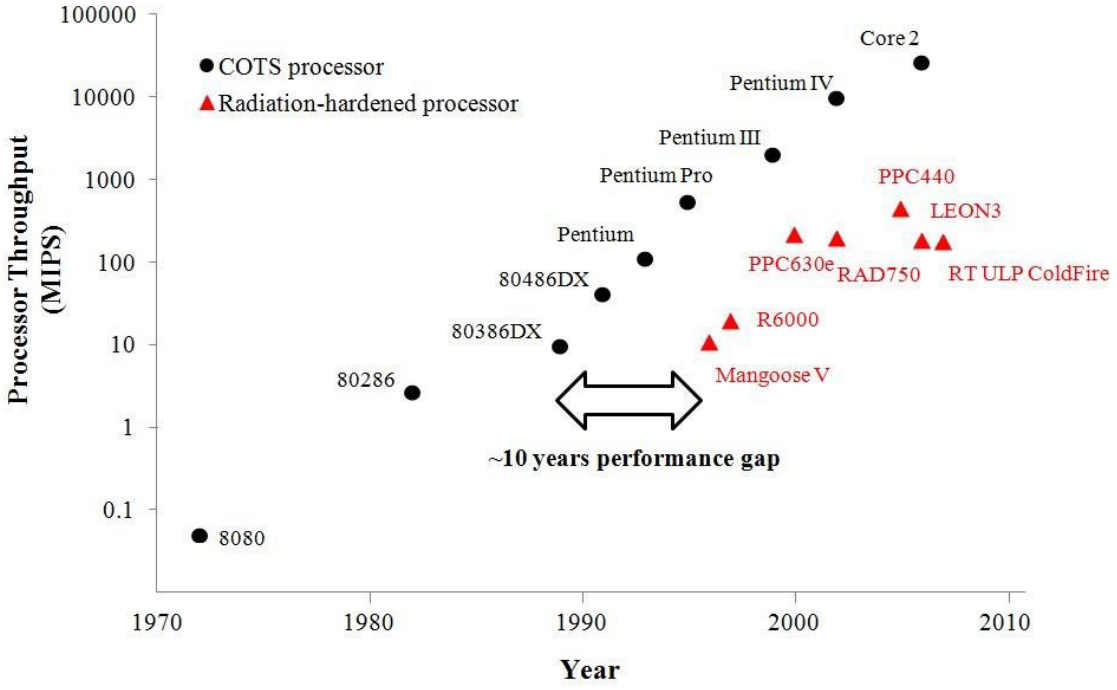

Figure 3.1: COTS x RadHard processor throughput (KEYS, 2008).

Hardware based techniques can be based on duplication with comparison, EDAC codes to protect registers and some other logical parity techniques to protect the logic. But all of them have some limitation on fault detection coverage. Without duplicating the whole processor, hardware-based techniques cannot achieve full fault detection against SEE, since part of the processor will always be unhardened. Non-intrusive modules, such as watchdogs, also cannot achieve full fault tolerance, since they do not have access to internal information from the microprocessor, like register values, for example. On the other hand, watchdogs and intrusive hardware-based techniques are the only fault tolerant techniques that can detect an infinite loop in an instruction, such as a persistent error, since software-based techniques require that the system is executing its instructions in order to detect an error.

Hybrid techniques combine software-based techniques with hardware-based techniques. The design space for hybrid techniques is quite large, since it multiplies all hardware-based possibilities per the software-based options.

The result from the use of hybrid technique is a high effectiveness, since they can provide high levels of dependability while minimizing the introduced overhead. They also offer low development time (from the software-based techniques) and small performance degradation (from the hardware-based techniques). As drawbacks, they require the application source code (in order to transform it), which is not always available, and require changes, at least, in the system's architecture.

In the following subsections, the main techniques in each category are commented and their strengths and weaknesses concerning this scenario are discussed.

3.1 Software-Based Techniques

Software-based techniques, or Software Implemented Hardware Fault Tolerance (SIHFT) techniques, use the concepts of operation, time and information redundancy to detect the occurrence of faults during program execution. In the past years techniques have been proposed so that can be automatically applied to the source code of a program, reducing significantly the development time and costs (RHOD, 2008). By doing so, the hardening is applied during software construction. The main drawbacks are the performance degradation, due to the extra instructions that will be executed by the processor, slowing the overall application runtime, and the overhead in program memory due to the extra instructions. As far as the fault tolerance is the only concern, the overhead in memory is not an issue, since EDAC techniques can be used to protect the memory. On the other hand, when power is also a concern, memory may become a constraint, since memories are responsible for most of the power dissipation and area within a chip.

Software-based techniques can be divided into two groups, according to their aim at detecting faults: (1) data flow checking techniques, which aim to detect faults affecting the data flow and (2) control flow checking techniques, which aim at detecting faults affecting the program's execution control flow. The first group comprises faults in the data structures of the processor, such as variables, registers and the data memory. Such faults may lead the processor to calculate an incorrect result, but they do not change the program flow. The second group is related to faults that affect the normal execution of a program. Such errors can be a deviation from the normal program flow, which can cause an infinite loop in a subroutine, for example, or even in an instruction. Software-based techniques usually aim to detect only one of these two groups of faults.

Among the most important solutions to detect data flow errors, there are a few techniques that exploit information and operation redundancies, such as Error Detection by Data Diversity and Duplicated Instructions (ED^4I) (OH, 2002a), the transformation technique proposed in Cheynet (2000) and Variables 1 (VAR1), Variables 2 (VAR2) and Variables 3 (VAR3) techniques proposed in Azambuja (2011b).

ED^4I consists in modifying the original version of the program by multiplying (or dividing) all variables by a constant value. The replicated version is then executed along with the unmodified program. After executing both versions, their results are compared for consistency (considering the constant value added to the replicated version). An error is detected if a mismatch is found. By running two versions of the code, instead of

one, and comparing the results, this technique introduces overheads in memory and execution time.

The technique proposed in Cheynet (2000) introduces several code transformations to modify the original program code. Rules are applied in order to duplicate all variables and operations among them. By doing so, this technique replicates the whole data path by software. With a duplicated data path, Cheynet (2000) adds instructions to the code to compare the values stored in the variables and jump to a subroutine in case of mismatch. The comparisons happen every time a variable is read.

Azambuja (2011b) proposed three techniques to cope with data effect errors. The techniques were implemented at assembly level, so that they are compiler independent, and duplicate all the values stored in registers into spare registers (register unused by the compiler). If there are not enough spare registers, the techniques choose the most important registers and replicate them, leaving the others unhardened (CHIELLE, 2012). As another option, the compiler can be set to use only half of the registers, which could introduce execution time overheads, due to the lack of registers. The main difference between the techniques is that VAR1 checks the consistency every time a register is read, while VAR2 checks when a registers is written and VAR3 checks only when the memory is accessed.

Techniques to detect data flow errors introduce overheads in memory (both program and data) and execution time, due to extra instructions in the original code and variable and registers replication. On the other hand, results presented at Azambuja (2010a) have shown 100% fault detecting for all SEU and SET injected directly in the processor's description.

Software-based techniques to detect control flow errors differ from data flow because of one main reason, which is: control flow techniques can be optimized. Data flow techniques always have to replicate data (registers, variables or memory positions) and compare it, while control flow techniques can analyze the code, comprehend it and optimize the replication and comparison. Most control flow techniques perform an analysis on the program's execution flow, divide the program into Basic Blocks (BB) and parse the program flow as a graph between different nodes (BBs). A BB is defined as a sequence of consecutive instructions that are always executed sequentially, meaning that the control flow always enters a BB in the first instructions and leaves at the end.

A set of software-based techniques has been proposed in the literature aiming at detecting control flow errors. Among the most important, there are the techniques called Control-Flow Checking using Assertions (CCA) (MCFEARIN, 1995), Enhanced Control-Flow Checking using Assertions (ECCA) (ALKHALIFA, 1999), Control-Flow Checking by Software Signatures (CFCSS) (OH, 2002b), Control-flow Error Detection through Assertions (CEDA) (VEMU, 2011) and Automatic Correction of Control Flow Errors (ACCE) (VEMU, 2007).

CCA introduced the concept of Block Identifier (BID) and Control Flow Identifier (CFID). The first identifier is a unique value for each BB, while the second is used to identify transitions between BBs. The technique uses both identifiers to monitor the behavior of the control flow, using global registers to store their values. Whenever the control flows enters a BB, the BID register is set to the BB’s unique value. The CFID is stored in a two-position array (implemented over two registers) that stores the transition between two BBs. By analyzing these values, CCA could detect most of the errors when moving from one BB to another. The main problem in this technique was that some

BBs shared the same CFID value, which led to undetected errors. In order to improve it, ECCA was proposed by Alkhalifa (1999). Alkhalifa (1999) improved CCA into ECCA by adding a new identifier and dividing the BBs into groups. He was then able to improve CCA detection rates, but without being able to detect intra-block errors (inside the BB).

CFCSS was proposed in 2002 to complement ED^{4}I in its fault detection rates. It presented a Global Signature Register (GSR) to keep track of the control flow. By assigning BB values to GSR, he was able to detect most control flow errors, but still had issues with BB identifiers sharing the same value (which also happens to CCA).

In Vemu (2011), a new technique called CEDA was proposed. CEDA uses a global register to store a control flow identifier, called Node Signature (NS). The main difference between CEDA and the other techniques is that it performs a deeper analysis on the program code, identifying networks of BBs and possible transitions between them. By doing so, it creates a transition signature to guarantee that the transition between BBs is valid. It assigns a node signature and a node exit signature to each basic block, every time the control flow enters and leaves a basic block, respectively. Results show that CEDA can detect 90% of faults that cause an incorrect transition between BBs. On the other hand, CEDA cannot detect control flow errors inside a BB.

In order to improve CEDA, Vemu (2007) proposed ACCE and ACCE with Duplication (ACCED). They use the same detection capabilities from CEDA, but improve it by allowing error correction. Despite being unable to mitigate all control flow errors, ACCE imposes low latency for error correction with performance overhead of about 20%. The only issue that remains is that neither ACCE nor any software-based technique is able to detect intra-node control flow errors, or in other words, faults causing control flow errors inside the same BB.

3.2 Hardware-Based Techniques

Hardware-based techniques must be implemented during the design phase. Because of that, such techniques cannot be applied to COTS processors or restricted IPs targeted at the general purpose market. Their use is mainly restricted to ASICs or FPGA based designs (that do not use restricted IPs). In some cases, hardware-based techniques applied to components may suffer commercial embargos, such as the International Traffic in Arms Regulation (ITAR), which is a set of United States government regulations that control the export and import of defense-related articles and services on the United States Munitions List (USML). As an example, Brazil is not able to buy RadHard Xilinx FPGAs from the United States of America. These techniques can be classified in two main groups: redundancy based, and hardware monitors. The first group relies on time or space redundancy, while the second uses watchdogs, checkers or IPs to monitor the main processor.

Techniques based on space redundancy are grounded in the single fault model, where only one of the hardware redundant copies is affected by transient upsets (ROSSI, 2005). It means that only one of the modules will be affected by a transient fault and therefore the fault detection rate should be 100%. On the other hand, studies have shown that a single fault may affect two hardware modules in case of SRAM-based FPGAs (KASTENSMIDT, 2005) due to the routing architecture, or in adjacent standard cells in ASICs as shown by Almeida (ALMEIDA, 2012).

The most well-known technique is called Duplication With Comparison (DWC) (WAKERLY, 1978). It duplicates the whole hardware and adds a comparator module to detect a mismatch between both modules. Another option would be to triplicate the hardware by using Triple Modular Redundancy (TMR), which not only detects an error, but also indicates which module generated the error, allowing correction. In case of an FPGA, the erroneous module could be partially reprogramed in the board, correcting both transient and permanent errors. The granularity of the replication may change, according to the designer's constraints (PILOTTO, 2008).

The literature also presents other approaches, such as the one proposed in Nieuwland (2006), where the critical path of combinational circuits is hardened through the duplication of gates and transient errors are masked due to the extra capacitance available in the node. It is also very common to find microprocessors partially hardened, where only the most critical registers are replicated, such as the PC and the SP.

Differently from space redundancy, space redundancy uses the same computing hardware modules to compare its value shifted in time. Usually extra hardware is added to introduce a fixed time delay δ. Anghel (2000) proposed an architecture where the outputs of a combinational circuit were duplicated and stored with different time delays (0 and δ). A comparator was then introduced to compare the stored results after δ, flagging an error in case of mismatch. It is important to mention that δ equals to the maximum transient pulse length.

Although such techniques can provide high fault detection rates, they introduce huge area overheads (a circuit hardened with TMR has about 3.5 times the size of the original circuit), which leads to higher power consumption as well. Such overheads are not acceptable in embedded systems. When using time redundancy, the value of δ tends to increase due to technology aspects. This increase impacts directly in every operation cycle, leading to unbearable performance overheads.

As an alternative to time and space redundancy, hardware-based techniques offer monitoring blocks. The second group of techniques adds special hardware modules to the system's architecture, called watchdog processors (MAHMOOD, 1988), checkers (AUSTIN, 1999) or Infrastructure Intellectual Properties (I-IP) (LISBOA, 2007). Such devices monitor the control flow of the programs inside the processors and memory accesses performed by them. In order to do that, the behavior of the processor running the application may be monitored using three types of operations: (1) Memory access checks, which look for unexpected memory accesses, such as unused memory areas and restricted function memory areas (NAMJOO, 1982), (2) Consistency checks, where the monitor checks if the value a register hold is acceptable, by exploiting information about the task performed by the program (MAHMOOD, 1983) and (3) Control flow checks, consisting in checking if the branches taken are consistent with the program graph of the application running in the processor (NAMJOO, 1983), (OHLSSON, 1995), (SCHUETTE, 1987), and (WILKEN, 1990).

Watchdog processors have one characteristic that is hardly found in other fault tolerance techniques, which is the ability to detect stuck-at errors in the execution flow, such as when a processor loops in one single instruction only. When considering watchdog processors, two types may be envisioned: active and passive watchdog processors. The active watchdog processor executes a program concurrently with the main processor, checking whether its program evolves accordingly to the one executed by the main processor. It continuously checks both programs and flags an error in case of mismatch (NAMJOO, 1983). The result is a simplified DWC approach, but still

introducing area overheads to implement the watchdog processor and small performance overhead to compare both modules.

The passive watchdog processor does not run any program. Instead, it computes a signature by observing the main processor's buses so that it can perform consistency checks. An interesting approach was proposed in Wilken (1990), where a watchdog processor observes the instructions executed by the main processor and computes a runtime signature. The code running on the main processor is modified so that when entering a basic block, an instruction is issued to inform the watchdog processor a pre-calculated signature, while the main processor executes a NOP instruction. The watchdog processor then compares the received signature with its pre-computed signature and flags an error in case of mismatch. Another similar watchdog processor was proposed in Ohlsson (1995), where it computed a signature based on the addresses of the instruction that the main processor fetched. Watchdogs are interesting approaches, since they can be implemented intrusively, by adding a new instruction to the processor's Instruction Set Architecture (ISA), for example, or non-intrusively, by making the watchdog processor to observe the buses between processor and its memory. The overheads in area can be small, depending on the watchdog processor's complexity and they usually have a small impact on the system.

As another alternative, one can use checkers as a hardware-based technique. An architecture called DIVA was proposed in Austin (1999), using a simple functional checker to verify the correctness of all computation being executed in the main processor. The technique added a functional checker to the execution stage of the pipeline, so that it allowed only correct results to reach the register barrier. The implementation of the checker was done so that it was simpler than the core processor, since it received the instruction to be executed together with the values of the input operands and the result from the main processor. By doing so, the checker did not have to care about address calculations and therefore could be implemented in a simpler way than the processor core.

Instead of only detecting an error, the authors decided for the risky assumption that the checker is always correct. The assumption came from the fact that they used oversized transistors in its construction and also that they performed an extensive verification in the design phase. By doing so, in case of a mismatch between the core processor and the checker, DIVA could send the result of the checker to the register barrier. If a new instruction would not be released for the checker after a given number of clock cycles, the core processor's pipeline would be flushed and the processor restarted from a given point. The main drawback of this technique is that the assumption that the checker is always correct is not always true. A second drawback is that this technique, unlike the monitors, requires deep changes in the processor architecture.

3.3 Hybrid Techniques

Hybrid techniques are very effective, since they offer the designer a tradeoff between hardware- and software-based techniques. They provide low cost software transformations, high detection rates and small area overheads from the software-based techniques and also high performance and high detection rates from the hardware-based techniques. The possibility to choose among the available techniques and how much of each technique to use (partially of fully) expand the design space and offer the implementation of hardening techniques targeted to specific applications.

As well as the advantages that hybrid techniques inherit from software- and hardware-based techniques, come also a few disadvantages. Among the drawbacks, one can notice the requirement of the source code of the application that processor core should run (which cannot be always fulfilled), performance degradation and memory overhead from the software-based techniques and area overhead from the hardware-based techniques. On the other hand, these drawbacks are normally smaller than when using a pure software- or hardware-based technique.

They combine software-based techniques with hardware-based techniques. A hybrid technique can be a simple combination of a software-based technique with a hardware-based technique, such as in Cuenca-Asensi (2011) and Lindoso (2012), or a combination where software- and hardware-based techniques interact with each other, such as in Schialli (2006), Rhod (2008), Bernardi (2006) and Nicolaidis (1999). The first group is more easily implemented, since it's a pure combination of techniques. It can be optimized by analyzing overheads and detection rates to better choose the techniques to be used, but it has a limited design space, since it does not consider the techniques where software- and hardware-based techniques share information. The second group, on the other hand, has a larger design space and therefore offers the designer more hardening options, when considering performance degradation and area overhead.

In Cuenca-Asensi (2011), a hardware/software co-design methodology is proposed to detect SEUs in microprocessors. The proposed hardening infrastructure receives a specification of the system requirements that takes into account constraints related to silicon area, performance, power consumption, hardware cost, reliability, availability, safety, security and recovery time. It then chooses a set of the best techniques for the given system and tests them in the real processor implementation. By doing so, it can then choose the best combination from the available design space. The proposed hardening infrastructure offers the technique called SWIFT-R (REIS, 2007) as a software-based technique and selective TMR as a hardware-based technique.

The methodology proposed by Cuenca-Asensi (2011) was then improved by Lindoso (2012) into a methodology to correct SET in microprocessors. The methodology follows the same principles and offers the same software- and hardware-based techniques. The main difference is that the system to test the chosen set of fault tolerance techniques was improved from FT-Unshades (NAPOLES, 2007) to AMUSE (ENTRENA, 2010), offering faster test results. The main drawback of both these techniques is that they are intrusive, meaning that they cannot be applied to a COTS microprocessor. Also, if the program code is protected, none of the techniques could be applied. On the other hand, they offer a methodology to choose and test different techniques from a given design space.

As an alternative to these intrusive approaches, Schialli (2006) introduced the idea of an Infrastructure IP (I-IP). He proposed a very simple I-IP to be put between the main processor and its instruction memory, so that it could substitute the fetched code with hardened one, on-the-fly. The main problem of his proposal is that the I-IP did not contain an ALU or a control unit and therefore introduced significant performance overheads without being supported by a suitable design flow environment.

Another hybrid technique was presented in Bernardi (2006) adopting software-based techniques in a minimal version along with the introduction of an I-IP. The software was modified so that it implemented instruction duplication and information redundancy. Also, instructions to communicate to the I-IP were added to exchange

information about the control flow execution. This I-IP worked concurrently with the main processor performing consistency checks among duplicated instructions and verifying the correctness of the program flow by monitoring the addresses.

An approach to minimize the overhead was then proposed in Rhod (2008). It combined the main ideas behind Schialli (2006) and DIVA by introducing a new I-IP between the main processor and its program memory with an architecture that could be customized by the main processor. It also implemented an ALU and a control unit, so that it also could compute the instruction fetched from the memory. By doing so, it could monitor the buses between the microprocessor and its program memory, get the operands of the original data processing instructions, compute them and compare to the results from the microprocessor for correctness. Also, it could check the feasibility of the address accessed by the processor (characteristic from hardware monitoring devices). Although the I-IPs could provide the ability to harden the program even without the source code, they still require intrusive changes in the main processors.

In Nicolaidis (1999), a hybrid technique based on duplication, time redundancy, and Code Word State Preserving (CWSP) was proposed. The CWSP introduced a gate topology to replace the last gates of a combinational circuit, so that it would be able to pass the correct value in the combinational logic in the presence of a SET. When concerning the duplication and the time redundancy, CWSP compared both outputs for correctness. When identical, the next state would be equal to the corresponding output function. When different, the next state would remain equal to the present state. By using time redundancy, CWSP introduced a delay δ to the circuit's outputs. As one can notice, the main disadvantage is the need to change the CMOS logic in the next stages by inserting extra transistors. Also, the time redundancy introduced significant overheads in performance.

3.4 Summary

This subsection summarizes all faults tolerant techniques mentioned in this section. Table 3.1 evaluates them according to intrusiveness, fault detection, fault correction, fault coverage and overheads in area, execution time, program memory and data memory. Classifications LOW, MEDIUM and HIGH are used when the value is not specific.

Table 3.1: Fault tolerance techniques summary

Fault Tolerance Technique	Intrusive	Fault Correction	Fault Coverage	Area Overhead	Execution Time Overhead	Memory Overhead
ED^4I	NO	NO	MEDIUM	-	HIGH	HIGH
Cheynet (2000)	NO	NO	HIGH	-	HIGH	HIGH
VAR3	NO	NO	HIGH	-	MEDIUM	MEDIUM
CCA	NO	NO	MEDIUM	-	HIGH	HIGH
ECCA	NO	NO	MEDIUM	-	HIGH	HIGH
CFCSS	NO	NO	MEDIUM	-	HIGH	HIGH
CEDA	NO	NO	HIGH	-	LOW	LOW
ACCE	NO	YES	HIGH	-	LOW	LOW
DWC	YES	NO	HIGH	2 times	-	-
TMR	YES	MASK	HIGH	3 times	-	-
Nieuwland (2006)	YES	MASK	HIGH	2 times	-	-
Anghel (2000)	YES	NO	LOW	LOW	-	-
Wilken (1990)	YES	NO	LOW	LOW	-	-
Ohlsson (1995)	YES	NO	LOW	LOW	-	-
DIVA	YES	NO	LOW	MEDIUM	-	-
Cuenca-Asensi (2011)	YES	YES	HIGH	HIGH	HIGH	HIGH
Lindoso (2012)	YES	YES	HIGH	HIGH	HIGH	HIGH
Schialli (2006)	YES	NO	MEDIUM	LOW	HIGH	-
Bernardi (2006)	YES	NO	MEDIUM	LOW	MEDIUM	MEDIUM
Rhod (2008)	NO	NO	MEDIUM	LOW	-	-
Nicolaidis (1999)	YES	MASK	HIGH	2 times	HIGH	-

4 PROPOSED TECHNIQUES TO DETECT TRANSIENT FAULTS IN PROCESSORS

This part of the thesis describes the developed techniques to detect transient errors affecting processors. As stated in the previous chapter, software-based techniques are unable to detect all faults affecting the control flow, while hardware-based techniques cannot protect processors without a huge area overhead. Hybrid techniques have presented a better tradeoff between overhead and fault detection. This chapter focuses in presenting and discussing three new hybrid fault tolerant techniques that can achieve high fault detection in processors, at small area overhead and performance degradation.

Software-based techniques have shown to be the best approach to dealing with data flow errors, since they don't require any extra hardware and offer full fault detection at a cost around 40% performance degradation (AZAMBUJA, 2011b). Because of that, this chapter presents mainly techniques to detect control flow errors, since they can be combined with software-based techniques presented in previous works (AZAMBUJA, 2011a).

This section presents the HPCT tool to transform program code, two known software-based techniques (Variables and Inverted Branches), and three innovative hybrid techniques to detect transient faults in embedded microprocessors: PODER, OCFCM and HETA. These techniques include the benefits proposed in the Variables method and in the Inverted Branch method.

4.1 Hardening Post Compiling Tool (HPCT)

Code transformation is a complex task that requires code analysis and processing, instruction replication, and instruction address correction. The code analysis and processing is required to find out code characteristics, such as branch instruction addresses, registers being used (and in some cases, the ones that are more important to be hardened), subroutines, memory area where the program and data are located, branch instruction destination address, and the program flow graph (basic block structure). The instruction replication uses the analysis and processing to insert instructions to the original program code, considering which registers are currently not being used. When instructions are added or moved, the destination addresses of branch instructions may change and they must be updated, including relative addresses, which must be recalculated. Such modifications are very hard to be done by hand, especially when dealing with large program codes with lots of branch instructions.

In order to automate the program code transformation, we used a tool called Hardening Post Compiling Translator (HPCT), introduced by Azambuja (2010b), and improved to implement the proposed hybrid techniques. It was implemented in Java, due to its portability to any operating system with Java Runtime Environment (JRE),

and easy string parsing and manipulation. It implements a class for each technique and therefore could be extended to implement the proposed hybrid techniques, including the generation of hardware modules, when necessary.

The tool receives as inputs the program's binary code, which makes it compiler and language independent, the hardening techniques to be applied, ISA definitions and some characteristics of the processor's architecture. The user is allowed to choose the hardening techniques in a Graphical User Interface (GUI), while ISA definitions and processor's architecture are described in classes. Current available processors are the miniMIPS and Leon II. The tool outputs a binary code, processor dependent, which can be directly interpreted by the target processor. This workflow can be seen in Figure 4.1.

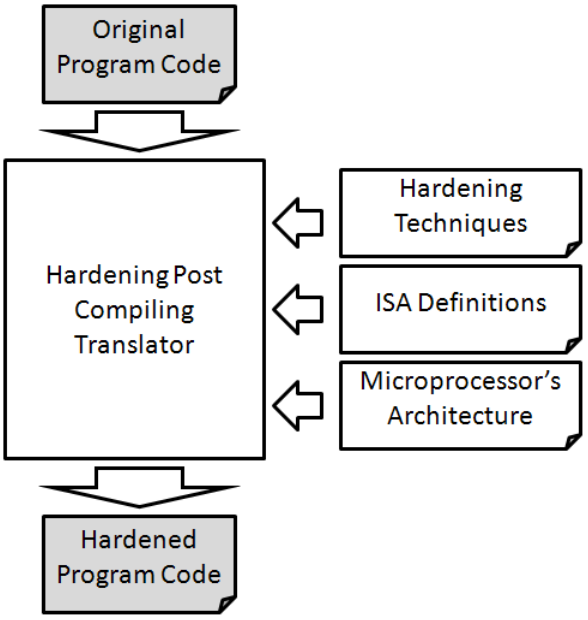

Figure 4.1: HPCT's workflow.

From the original program code, the HPCT tool extracts all the necessary information to transform the code, such as branch instruction memory locations and destination addresses, program execution flow graph, relations between the nodes of the flow graph, registers used and available, among other characteristics. Using this information, it is able to insert, remove and move instructions and blocks of instructions, such as procedures.

The extraction of branch instruction memory locations, destination addresses, used and spare registers can be easily done by reading the original program code and calculating the destination addresses for the branch instructions. The extraction of the program execution flow graph, on the other hand, can be more difficult. The control flow graph is divided in basic blocks (nodes) and control flow transitions (edges). Initially the tool adds the first non-branch line of code. The program code is than iterated until a branch instruction is found. When that happens, a node is finished and two new nodes begin in the instruction after the branch instruction and at the branch instruction's destination address. In some cases, instructions store the destination address in a register known only at runtime, such as Jump to Register (JR) or Jump and Link Register (JALR) instructions. When that happens, the code must be further analyzed in order to try to find the runtime value of the register. When the value is not found, the control flow graph can still be partially extracted and used by the techniques, but with vulnerabilities in such points.

The insertion of new instructions to the program code must take into account the control flow graph. Whenever an instruction is added or removed, all the addresses, relative or absolute, must be checked for consistency. When replicating an instruction using spare registers, such registers must also be accounted for and removed from the available registers list.

4.2 Improved Variables Technique (VAR)

The idea behind the Variables technique is to protect the data path of the processor. It does so by replicating all the registers used by the processor on spare ones (registers not used by the application running).

In order to replicate the registers, this technique assigns a spare register to each used register and replicates all write instructions performed on the original register to its copy. Also, every read operation is duplicated, so as to duplicate the data path. At given points, consistency checks are performed through branch instructions. Whenever a mismatch between the register and its copy is found, the program execution flow branches to a predefined subroutine that flags the error.

Considering that users are only interested in the values stored in the data memory (or coming out of the processor), the Variables technique assumes that if they are correct, so is the system. Because of that, consistency checks are only performed when a register is used to load or store a value in the memory. Also, in order to keep the control flow consistent, registers are also checked before branch instructions. The result is that the data being stored and read from memory is correct, as well as the data being used by the branch instructions.

4.2.1 Implementation Details

In order to evaluate the Variables techniques, four algorithms were used: 6x6 matrix multiplication, bubble sort, tiny encryption and run length algorithms were used. We used the HPCT tool to harden each application according to the Variables transformation.

The transformation follows three rules:

- Every variable *x* must be duplicated. Consider *x1* and *x2* the original variable and its copy;
- Every operation performed on a variable *x* must be performed on *x1* e *x2*;
- Before every branch instruction or instruction that accesses the memory, variables by the instruction must be checked for consistency with their copies. In case of mismatch, and error should be flagged.

Figure 4.2 illustrates a piece of code protected by Variables technique. The original code has three instructions that operate with registers and memory elements. Instructions 1 and 3 are inserted to protect the load instruction located in position 2 (ld r1, [r4]), where the first instruction verifies the register containing the base address for the load instruction (r4) and its replica (r4'). The second instruction replicates the load instruction, using the replicated memory position (r4' + offset) and loads the value into the replicated register (r1'). Instructions 8, 9 and 11 are inserted to protect the store instruction. While instructions 8 and 9 verify values stored in the base and data registers (r1 and r2, respectively) against their replicas (r1' and r2', respectively). Instruction 11

replicates the original store instruction located in position 10 (st [r1], r2) using the replicated registers r1' and r2' over a replicated memory address (r1' + offset).

Original Code	Hardened Code
1: ld r1, [r4]	**1: bne r4, r4', error** 2: ld r1, [r4] **3: ld r1', [r4' + offset]**
4: add r2, r3, 4	4: add r2, r3, 4 **5: add r2', r3', 4**
10: st [r1], r2	**6: bne r1, r1', error** **7: bne r2, r2', error** 8: st [r1], r2 **9: st [r1' + offset], r2'**

Figure 4.2: Variables technique's transformation.

The original add instruction located in position 6 (add r1, r2, r4) operates only over registers and therefore does not need any offset. In order to protect this instruction, instruction 7 is inserted, which performs the original instruction, but using the replicated registers (r2' and r4') and writing over the replicated destination register (r1').

Table 4.1: Characteristics for the variables technique program transformation

Application		Original	Variables
Matrix Multiplication	Execution Time (µs)	1,257	1,821 (1.45×)
	Code Size (bytes)	1,548	2,644 (1.71×)
	Data Size (bytes)	524	1,048 (2.00×)
Bubble Sort	Execution Time (µs)	231	375 (1.62×)
	Code Size (bytes)	1,212	1,916 (1.58×)
	Data Size (bytes)	120	240 (2.00×)
Encryption	Execution Time (µs)	157	266 (1.69×)
	Code Size (bytes)	896	1,688 (1.88×)
	Data Size (bytes)	28	56 (2.00×)
Run Length Encode	Execution Time (µs)	2,372	3,914 (1.65×)
	Code Size (bytes)	2,772	5,608 (2.02×)
	Data Size (bytes)	236	472 (2.00×)

This transformation duplicates the data being stored, i.e., the number of registers and memory addresses. Consequently, the applications are limited to a portion of the available registers and memory address. In some cases, compilers can restrict the application to a small set of registers and memory addresses, allowing the duplication. In other cases, the rules can be applied to a subset of the used registers and memory positions, although it may compromise the fault detection rate.

Table 4.1 shows the results after the Variables transformation on four different case-study applications. As one can see, the performance overhead varies from 1.45 to 1.69 the original execution time, which is a small performance degradation, when compared to other techniques to detect errors in the data flow, such as ED4I and Cheynet (2000). The memory overhead, on the other hand, is considerably big, since it varies from 1.58 to 2.02 times the original one.

4.3 Improved Inverted Branches Technique (BRA)

The Inverted Branches technique (AZAMBUJA, 2010a) was proposed to detect faults affecting the decision of branch instructions. Such errors affect the transition between different BBs and are hard to be detected, since both paths (branch taken or not) are acceptable in the program flow graph. A simple way of doing that is to replicate the branch instructions.

Branch instructions are more difficult to replicate than non-branch instructions, since they are not linear (they always have two possible next addresses). When the branch is not taken, a branch instruction can be simply replicated and inserted right after the original branch instruction, but with a destination address pointing to an error subroutine. If the branch was not taken in the original instruction, is must also not be taken in the replicated instruction, which will be executed right after the original.

In the possibility that the branch was taken, the replicated branch instruction must be inserted on the branch destination address, which is the next instruction to be executed by the microprocessor. The difference is that if the original branch was taken, the same branch must be taken again. In order to keep the original program flow, the replicated branch instruction is inverted and its destination address pointed at the error subroutine.

4.3.1 Implementation Details

In order to evaluate both the effectiveness and the feasibility of the Inverted Branches techniques, four applications were chosen: a 6x6 matrix multiplication, a bubble sort, a bit count and a Dijkstra's algorithms. The matrix multiplication and Dijkstra's algorithms require a large data processing with only a few loops and therefore uses mostly the datapath from the microprocessor. On the other hand, the bubble sort and the bit count algorithms use a large number of loops, control registers and branch instructions and therefore use mostly the controlpath, since all the data processing is related to the control registers. Each version was hardened using the HPCT.

The transformation follows three rules:

- Every branch instruction is replicated after the original instruction, with a new destination address pointing to an error subroutine;
- Every branch instruction is duplicated, inverted and inserted at the destination address of the original instruction;

- A jump instruction is added before the inverted branch instruction, pointing to the instruction after the inverted branch.

Original Code	Hardened Code
1: beq r1, r2, 6	1: beq r1, r2, 5 **2: beq r1, r2, error**
3: add r2, r3, 1	3: add r2, r3, 1
	4: jmp 6 **5: bne r1, r2, error**
6: add r2, r3, 9 7: jmp end	6: add r2, r3, 9 7: jmp end

Figure 4.3: Inverted Branches technique's transformation.

Figure 4.3 illustrates the Inverted Branches transformation applied to a program code. The conditional branch instruction Branch if Equal located in position 1 (beq r1, r2, 6) will jump to instruction 6 if registers r1 and r2 contain the same value. Initially, the branch will be replicated and inserted right after the original instruction, in position 2. The original branch instruction is then inverted and inserted in the original branch destination address (5) by using the Branch if Not Equal instruction (bne r1, r2, error). In this process, original branch instruction destination addresses must be adjusted to the new address (5, in the transformed code).

Table 4.2: Characteristics for the Inverted Branches technique program transformation

Application		Original	Inverted Branches
Matrix Multiplication	Execution Time (µs)	1,190	1,221 (1.03×)
	Code Size (bytes)	668	792 (1.19×)
	Data Size (bytes)	524	524 (–)
Bubble Sort	Execution Time (µs)	231	250 (1.08×)
	Code Size (bytes)	992	1384 (1.40×)
	Data Size (bytes)	120	120 (–)
Bit Count	Execution Time (µs)	4,073	4,593 (1.13×)
	Code Size (bytes)	460	548 (1.19×)
	Data Size (bytes)	28	28 (–)
Dijkstra	Execution Time (µs)	1,785	1,920 (1.08×)
	Code Size (bytes)	1,784	2,144 (1.20×)

	Data Size (bytes)	236	236 (–)

The insertion of the replicated inverted branch instruction may affect other execution flows. For example, instruction 5 cannot be executed after the add instruction located in position 3 (add r2, r3, 1), since it could modify the value stored in the r2 register and cause a false fault detection. In order to protect the other execution flows, the inverted branch must be protected with instruction 4, an unconditional branch that does not allow instruction 5 to be executed after instruction 3, but only after branch instruction with destination address pointing to its position.

Table 4.2 shows the original and transformed program characteristics, according to execution time, code size and data size. As one can see, the overheads in execution time vary from 1.03 (matrix multiplication) to 1.13 (bit count) times the original execution times. The overheads in program code are larger than execution time, varying from 1.13 to 1.40 times the original value, for the matrix multiplication and bubble sort algorithms, respectively.

4.4 PODER Technique

PODER is the first hybrid technique proposed in this thesis. It was based in CCA and its two-element queue to keep track of the changes in the program's control flow, BID and CFID. The technique aims at detecting a few types of control flow errors, such as: (1) incorrect jumps to the beginning of a BB, (2) incorrect jumps inside the same BB, (3) incorrect jumps to unused memory addresses and (4) control-flow loops. It is important to note that PODER cannot detect errors in branch instructions, where the execution flow should have gone to one BB, but went another. In order to do so, it must be combined with the Inverted Branches software-based technique, described in Section 4.3.

The technique is divided in software-based and hardware-based sides, which communicate through memory writes at predefined memory addresses. In order to do so, PODER exploits two main concepts:

- Software-based program code transformation: the original program code is transformed based on a set of rules and additional instructions are inserted in order to communicate with the hardware module.
- Hardware-based non-intrusive module: an additional non-intrusive hardware module is added to the architecture. This module implements watchdog and decoder characteristics in order to analyze the processor's control-flow and decode instructions sent from the inserted software instructions.

PODER starts by dividing the program's execution flow into a BB graph. In a second step, it assigns unique BID and CFID values for each BB, according to rules further described. It then starts manipulating these values, during program execution, by storing them in a two-element queue and performing operations on them to check for consistency. The main advantage of PODER is that it can be easily divided in software and hardware, so that we can improve the fault detection and reduce the overheads in memory usage and performance degradation.

The innovation of this approach relies on the use of a signature mechanism technique that works in tandem with a hardware module to be able to detect all upsets

that affect the control flow. In the following, we describe in detail how these two concepts work.

4.4.1 Software-based Side

The software-based side of PODER relies on adding extra instructions into the original program code. So it can send data to the hardware-module and, by doing so, controlling it. From the four types of control flow errors that PODER aims at, the software-base side is responsible for protecting the system against incorrect jumps to the same BB (1) and incorrect jumps to the beginning of a BB (2).

Figure 4.4 shows seven instructions divided into four BBs. Case (1) happens when a jump occurs with destination address as the first instruction of a BB (addresses 0, 3, 4, 7). Case (2) happens when a jump originates and has as destination address the same BB (addresses 0 to 2, 1 to 2, 7 to 7, for example).

Address	**BB - Instruction type**
0	BB1 - First instruction
1	BB1 - Instruction
2	BB1 - Last instruction
3	BB2 - Instruction
4	BB3 - First instruction
5	BB3 - Last instruction
6	Branch instruction
7	BB4 - Normal instruction

Figure 4.4: Examples of Incorrect jumps to the same BB (1) and to the beginning of a BB (2).

The queue management as well as the operations performed on top of BID and CFID values require a huge amount of computational time, which would lead to increased performance degradation. In order to reduce drawback, PODER migrates as much of the computation as possible to the hardware module, being responsible only for controlling the module. It is important to note that all the program transformation happens during compilation time, and not at runtime. In the following, we describe in detail how PODER detects errors (1) and (2).

4.4.1.1 Jumps to the Beginning of a Basic Block

A jump to the beginning of a basic block is a real problem because of the fact that the initialization of a BB usually contains extra instructions for control flow error detection. In some cases, the first instruction of a BB resets the control flow assertion,

which can lead to undetected errors. Because of that, the detection of control flow errors to the beginning of a BB is mandatory.

The first step to protect the system against jumps to the beginning of a BB is to analyze the program's execution flow and extract a graph containing all the BB and their transitions. By doing so, we have access to the number of BB in the program code and their sources and destinations BBs. On a second step, every BB is assigned with two identifiers: a BID and a CFID. The BID represents each BB with a unique prime number, while the CFID represents the control flow, by storing the multiplication product of its destinations' BBs.

The fact that each BID is a unique prime number combined with the fact that the CFID is the multiplication of the destination BBs' BIDS, gives PODER an interesting characteristic: the operation rest of division of the dequeued CFID by the destination BB's BID always returns zero. When the value is different than zero, some control flow error happened, causing an incorrect transition in the program's execution flow.

In order to improve the detection, CFIDs are stored in a two-element queue, initialized with the first BB's CFID. When the program flow enters a BB, its CFID is stored in the queue. When it exits a BB, the first CFID is removed from the queue and divided by the BID. Errors are detected when one of these situations occur:

- The remainder of the division is not zero;
- The queue overflows;
- The queue underflows.

Four BB are presented in Figure 4.5, that shows an example of a BB graph. BB A is the starting BB and therefore receives the BID value 3. BBs B, C and D receive the following prime numbers as BID. A has as destination BBs B and C. By multiplying B's BID value per C's BID value, we get A's CFID, which is 35. The same applies to the CFID value of BBs B and C. D does not have a CFID value, since it is the execution flow ends in in. When transitioning from B to A, PODER will divide 35 (A's CFID, stored in the two-element queue) by 5 (B's BID) and the rest of division will be 0, stating that a correct transition was performed. An incorrect transition from A to D would result in the division of 35 (stored in the queue during A's execution) by 11 (D's BID) and the rest of division would be 2, which differs from 0, therefore detecting an error.

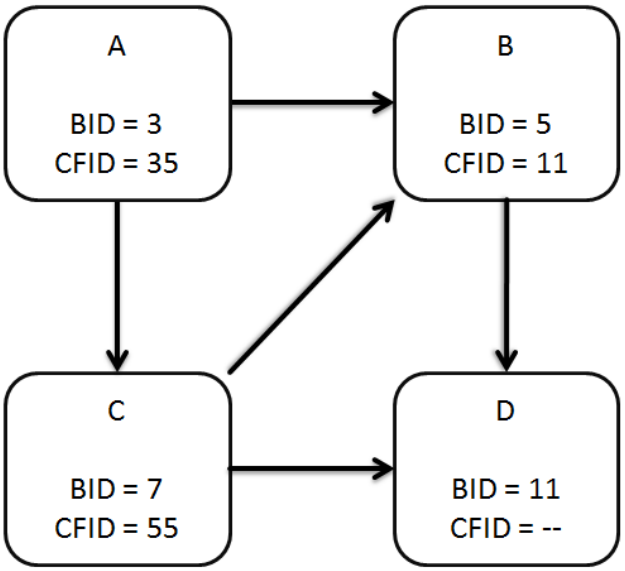

Figure 4.5: PODER technique's BB graph.

As mentioned before, the queue management is a heavy task to be performed purely in software and would result in huge memory and performance losses. Therefore, the instructions added to the original program code only control the hardware module, pushing the computation to the hardware module. By doing so, it performs the queue management when informing CFID and BID values through store instructions.

4.4.1.2 Jumps to the Same Basic Block

An incorrect jump to the same BB is an issue that cannot be solved purely by software-based techniques. The main problem is that the granularity required for that is at instruction level, while jumps to different BB require BB granularity. In order to detect errors inside the same BB, software-based technique would have to duplicate every instruction, which would lead to huge performance loss. From the related software-based techniques, none could detect such kind of error.

In order to detect such errors, PODER uses the communication between software-based techniques and hardware-based techniques. It does so by calculating a second signature for each BB, called XOR, during compilation time (by the software-based techniques), and during runtime (by the hardware module). The XOR value equals to the result of the operation eXclusive OR (XOR) between all the instructions from the BB.

XOR values are pre-computed by the compiler during the compilation phase and sent to the hardware module during runtime. The compiler adds additional instructions to send a reset value to the hardware module when the execution flow enters a BB and the calculated XOR value when the execution flow exits a BB. This is done by performing store instruction at predefined memory addresses.

By doing so, the hardware module receives a flag (when the execution flow enters a BB) to start calculating the XOR value at runtime and a check flag with the compiler-phase-computed XOR (when the execution flow exits a BB) to compare its calculated value with the one sent from the program code. When a mismatch is found, an error is notified.

4.4.2 Hardware-based Side

PODER's hardware-based side aims at complementing the software-based side in detecting incorrect jumps to the beginning of a BB (1) and incorrect jumps to the same BB (2), but also detecting incorrect jumps to unused memory addresses (3) and control flow loops (4).

Figure 4.6 shows ten instructions divided into 4 BBs. Case (3) happens when a jump occurs with destination address as the unused memory space (between addresses 8 an the end of the memory). Case (4) happens when a jump originates and has as destination address the same memory address (addresses 0 to 0, 1 to 1, 7 to 7, for example).

Address	**BB - Instruction type**
0	BB1 - First instruction
1	BB1 - Instruction
2	BB1 - Last instruction
3	BB2 - Instruction
4	BB3 - First instruction
5	BB3 - Last instruction
6	Branch instruction
7	BB4 - Normal instruction
8-end of memory	Unused memory space

Figure 4.6: Incorrect jumps to unused memory addresses (3) and control flow loops (4).

As mentioned before, the hardware module implements a two-element queue and its management circuit, and a rest of division operator to detect incorrect jumps to the beginning of a BB (1). To detect incorrect jumps to the same BB (2), it implements a XOR operator and registers. Also, it implements a small decoding unit, so that it can decode store instructions coming from the software-based side at given memory addresses. In order to have access to the instructions between the processor and the program memory, the hardware module sits between the memory buses, so that it can

read the data and address being exchanged. The overall architecture can be seen in Figure 4.7.

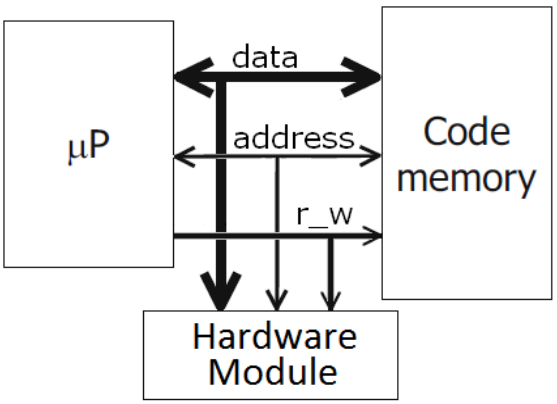

Figure 4.7: PODER technique's system architecture.

To detect incorrect jumps to unused memory addresses (3), the hardware module receives information of the memory area used by the program code (both program and data). When the processor tries to access an address that is out of range, an error is flagged.

The last kind of detection is control flow loops (4). In order to detect this kind of error, a watchdog timer is implemented. The counter is reset every time the execution flow enters a BB, with the "reset XOR" instruction, from subsection 4.4.1.2. When the counter overflows, an error is flagged. By doing so, the hardware module can detect a control flow loop that causes the execution flow to be stuck at a single instruction.

Although PODER can theoretically detect all control flow errors, it has two drawbacks. The first one is related to the BID and CFID values. Prime numbers increase at a fast pace, meaning that there is a limited quantity of BIDs that can fit in one 32-bit register. A bigger issue is that the CIFD value equals to the multiplication of BID numbers. If BIDs have a limited quantity, CFID soon reaches the 32-bit limit. When it happens, two registers must be used, increasing the complexity of the technique, as well as the performance degradation and memory usage.

The second drawback is that the hardware module has to have access to the memory buses. The buses used by some processors which use on-chip embedded cache memories may not be accessible by the hardware module. In such cases, PODER cannot be used.

4.4.3 Implementation Details

PODER is composed of two separated implementations: the software transformation implementation and the hardware module implementation. Both will be described in detail in the following.

4.4.3.1PODER's Software Transformation Implementation

The software transformation is responsible for implementing in the program code the operations required for detecting jumps to the beginning of a BB and for detecting jumps to the same BB. The first one can be seen at the two-element queue management, while the second can be seen as the XOR value calculation management.

In order to do the first, PODER has to implement three operations: (1) to send to hardware module the BID value of the BB being executed, (2) to store the CFID of the

current BB being executed in the two-element queue, and (3) to load the older CFID from the two-element queue and compare it with its BID value, previously sent by operation 1. Operation 1 and 2 must be performed in the beginning of the BB execution, while operation 3 has to be performed when exiting the BB.

The transformation follows two rules:

- A "send BID" and a "enqueue CFID" instructions are inserted at the beginning of each BB;
- A "dequeue CFID" is inserted at the end of each BB.

Figure 4.8 shows an example of the code transformation required to apply these three operations. It shows two BBs, from instructions 2 to 5 and 6 to 10. One the left column, one can see the original program code, while the right column shows the result of the transformation. Operation 1, or “send BID”, can be seen in the beginning of both BBs, represented by instructions 2 and 6. It is followed by operation (2), or “enqueue CFID”, represented by instructions 3 and 7. Operation 3, or “dequeue CFID”, can be seen in the end of both basic blocks. When comparing to the original program code, one can see that operation 1 and 2 are applied before the execution of the original instructions, while instruction 3 is inserted after.

Original Code	Hardened Code
1: beq r1, r2, 8	1: beq r1, r2, 6
4: add r2, r3, 1	**2: send BID** **3: enqueue CFID** 4: add r2, r3, 1 **5: dequeue CFID**
8: add r2, r3, 4 9: st [r1], r2	**6: send BID** **7: enqueue CFID** 8: add r2, r3, 4 9: st [r1], r2 **10: dequeue CFID**
11: jmp end	11: jmp end

Figure 4.8: PODER technique transformation for queue management.

Operations “send BID” and “enqueue CFID” are implemented in assembly, by storing the BID or CFID value at given predefined memory addresses. By doing so, the hardware module can decode the store instruction and read the values from the memory buses. The operation “dequeue CFID” does not have to send any data and therefore is performed by a store instruction with an unknown value to a given predefined memory address.

The XOR value management also requires transformations in the program code. In order to do that, PODER implements another two operations, which are: (4) to reset the XOR value in the hardware module, and (5) to check a given XOR value with the

calculated by the hardware module. Operation 4 has to be performed in the beginning of the BB, like operations 1 and 2, while operation 5 has to be performed in the end of the BB, like operation 3.

The transformation follows two rules:

- A "reset XOR" is inserted at the beginning of each BB;
- A "check XOR" is inserted at the end of each BB.

Figure 4.9 shows the same example from Figure 4.8 applied to operations 4 and 5. Operation 4, or “reset XOR”, can be seen in the beginning of both BB, represented by instructions 2 and 5, while operation 5, or “check XOR”, can be seen in the end of both BB, by instructions 4 and 8.

Original Code	Hardened Code
1: beq r1, r2, 6	1: beq r1, r2, 5
3: add r2, r3, 1	**2: reset XOR** 3: add r2, r3, 4 **4: check XOR**
6: add r2, r3, 4 7: st [r1], r2	**5: reset XOR** 6: add r2, r3, 4 7: st [r1], r2 **8: check XOR**
11: jmp end	9: jmp end

Figure 4.9: PODER technique transformation for XOR value.

Both operations are implemented by using store instruction at predefined memory addresses. The only difference between them is that “check XOR” has to send a value to the hardware module, so that it can compare to its calculated one, while “reset XOR” is a simple store instruction with an unknown value.

Original Code	Hardened Code
1: beq r1, r2, 8	1: beq r1, r2, 6
4: add r2, r3, 1	**2: reset XOR/send BID** **3: enqueue CFID** 4: add r2, r3, 1 **5: check XOR/dequeue CFID**

	6: reset XOR/send BID **7: enqueue CFID**
8: add r2, r3, 4 9: st [r1], r2	8: add r2, r3, 4 9: st [r1], r2 **10: check XOR/dequeue CFID**
11: jmp end	11: jmp end

Figure 4.10: PODER technique transformation after optimization.

When combining all operations, one can notice that instructions 1, 2 and 4 are performed in the beginning of the BB, while instruction 3 and 5 are inserted in the end of the BB. In order to optimize the technique, we combine a few instructions. The bottleneck in implementing these operations lays in the value that have to be sent to the hardware module, since only one value can be sent per instruction. The "reset XOR" does not have to send any data and therefore can be combined with "send BID" or "enqueue CFID". The same applies to "dequeue CFID", which can be combined with "check XOR". The result of both techniques applied to the same example code can be seen in Figure 4.10.

As case-study applications, we chose two algorithms: matrix multiplication and bubble sort. The first application is data flow oriented, while the second is control flow oriented. We transformed the code using the HPCT by using as inputs the original program code, the ISA definition and a file describing the microprocessor's architecture. Using these inputs, the HPC-Translator was able to generate a hardened program code.

Tables 4.3 and 4.4 show the original and modified program's execution time, code size and data size for the matrix multiplication and bubble sort algorithms, respectively. They present results for the original unhardened program, as well as the version hardened with PODER and hardened with PODER combined with Inverted Branches and Variables software-based techniques (Combined Techniques).

As one can see, PODER's execution time varies from 1.33, when applied to the matrix multiplication, to 1.61, when applied to the bubble sort, times the original unhardened program code.

Table 4.3: Characteristics for the PODER program transformation to the matrix multiplication

	Original Unhardened	PODER Technique	Combined Techniques
Execution Time (μs)	1,257	1,670 (1.33×)	2,943 (2.34×)
Code Size (bytes)	1,548	3,372 (2.18×)	5,576 (3.60×)
Data Size (bytes)	524	528 (–)	1052 (2.00×)

Table 4.4: Characteristics for the PODER program transformation to the bubble sort

	Original Unhardened	PODER Technique	Combined Techniques
Execution Time (μs)	233	374 (1.61×)	588 (2.52×)
Code Size (bytes)	1,212	2,440 (2.01×)	3,960 (3.27×)
Data Size (bytes)	120	124 (–)	244 (2.00×)

4.4.3.2 Hardware Module Implementation

The hardware module was implemented in VHDL language based on a timer that signals an error if not reset after a given number of clocks. Its enhancement was performed by adding a 16-bit register to store the real-time calculated XOR value, a 64-bit register to store the 2-element queue, a rest of division module (which is as big as a divider) and a simple decoder module.

The decoder module reads the data and address buses between the processor and the memory looking for store instruction in the program memory area. Whenever a store instruction is found, it reads the address bus to check which address the processor is accessing, in order to decode the instruction from the software-based side, and reads the data bus to read the value being sent. It then manages to perform the operation requested from the software-based side, such as a “reset XOR/send BID” or a “check XOR/dequeue CFID”.

Table 4.5 shows the size and performance of the implemented microprocessor and the hardware module. The hardware module implementation has a total of 128 registers. It was not protected against SEEs because of the fact that the worst case scenario is a incorrect fault detection, which would not compromise the system. The implemented hardware module occupies 15% of the total area of the miniMIPS microprocessor, while maintaining the same operating frequency. It is important to note that the hardware module has a fixed size, independent of the processor being used. That means that a bigger processor would lead to a smaller hardware module percentage, when compared to the processor.

Table 4.5: Area and performance of miniMIPS and the hardware module used by PODER technique synthesized in 0.18μ CMOS process technology

Source	miniMIPS	Hardware Module
Area (μm)	24,261.32	3,640.21
Frequency (MHz)	66.7	66.7

4.5 On-line Control Flow Checker Module (OCFCM)

The On-line Control Flow Checker Module (OCFCM) technique was based on checkers, watchdog processors and on the reconfigurability offered by modern FPGAs. It addresses reconfigurable systems with hardcore processors, such as FPGAs with embedded processors (for example, the Virtex an Excalibur families, from Xilinx and Altera, respectively) or closed IP processors, such as the Microblaze. It can also be applied to ASICs, but with a few restrictions.

This technique improves PODER because it can detect all errors detected by PODER (incorrect jumps to the beginning of a BB, incorrect jumps inside the same BB, incorrect jumps to unused memory addresses and control-flow loops) only by using a non-intrusive hardware module. The main drawback is that it is application-specific and therefore is not as simple to be applied to a General Purpose Processor (GPP) as PODER.

OCFCM itself is defined as a non-intrusive hardware module and therefore could be considered a pure hardware-based technique. Instead, OCFCM alone cannot achieve its main objective, which is detecting control flow errors. To do so, it has to be complemented by the Inverted Branches software-based technique (described in Section 4.3) and configured by the application running in the processor. Because of these characteristics, it is considered as a hybrid fault tolerant technique.

The technique has a clearer division between software and hardware than PODER, since the communication between them is very restricted. The division follows two main concepts:

- Software-based program code transformation: is used to configure the OCFCMs and perform small transformations in the program code, if necessary. Also, other software-based techniques are used to complement OCFCM's detection capabilities.
- Hardware-based non-intrusive module: an additional non-intrusive hardware module is added to the architecture. This module implements watchdog and decoder characteristics in order to analyze the processor's control-flow and decode instructions sent from the inserted software instructions.

OCFCM starts by analyzing the application's program code and extracting all the branch instructions and their addresses. By doing so, it creates an application-specific hardware module containing all the branch addresses and a decoder that can extract from these instructions the destination addresses. Then, during runtime, OCFCM can check the addresses that the processor is accessing and perform checks on the program's execution flow. Its main advantages are that it can be automatically generated during compilation time at small costs of area and performance degradation.

Its main drawback is that each application running on the processor requires its own OCFCM. It means that a GPP running ten different applications also requires ten OCFCMs. In order to decrease the area required to implement all the OCFCMs, we use partial reconfigurability, so that the system stores only the bitstream of each module and reprograms it on the FPGA's logic according to the running application. It is also possible to keep programmed on the FPGA a set of OCFCMs, switching between them by using software-based techniques. Since ASICS do not have reconfigurable logic, they must implement all possible OCFCMs from the start, which may lead to a huge drawback when considering such approach.

The innovation of this approach relies on the use reconfigurability and software-based techniques to use different application-specific non-intrusive hardware modules to detect all upsets that affect the control flow. In the following, we describe in detail how the technique works.

4.5.1 Hardware-based Side

OCFCM's hardware-based side aims at detecting faults that cause incorrect deviations in the execution program's flow. In order to do that, the hardware module

combines most of the non-intrusive hardware-based techniques, such as checking if the processor is accessing correct memory areas for data and program, the consistency of some variables, control flow checkpoints and also the PC evolution during runtime.

OCFCM is capable of doing that by storing some application oriented information. It sits between the processor and its memory. The hardware module, just likes PODER's, monitors the address and data buses between the microprocessor and its memory. OCFCM checks the memory accesses, branches and control flow checkpoints performed by the microprocessor. Figure 4.11 shows a set of OCFCM implemented to a processor system.

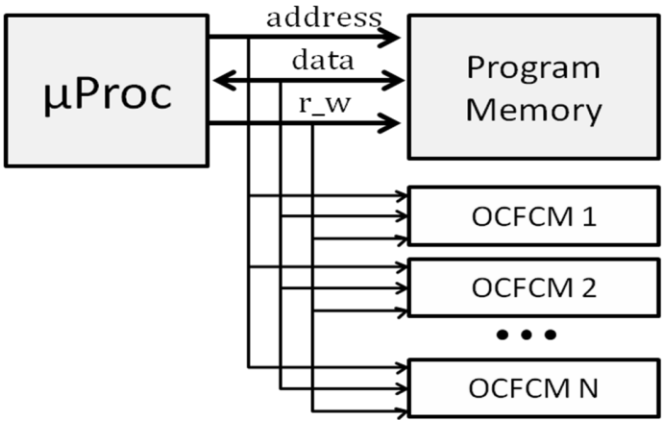

Figure 4.11: OCFCM's system architecture.

As mentioned before, OCFCM is an application specific module and has information about the portion of memory that the application is allowed to access. By doing so, it is capable of detecting incorrect memory accesses, both for data and instructions. Some variables, such as the PC and the SP are also checked through the data and address buses during runtime.

By checking the PC evolution during runtime and the number of clock cycles spent on a single instruction, OCFCM can detect if the software execution is stuck at the same memory address. Such detection is very important in processor-based systems, because software-based techniques cannot achieve such detection (in order to detect errors, redundant instructions must be executed, and a loop may hold the microprocessor in a single instruction).

In addition to these fault detection capabilities, OCFCM has the ability to check branch instructions during runtime and verify if they performed a correct branch in the program flow. To do so, OCFCM checks the PC evolution through the address bus. The program executes the instruction stored in program memory sequentially until a branch instruction is found. When performing a branch instruction, a new path becomes possible, other than the normal sequential execution. In this case, the OCFCM decodes the new possible path and checks if the microprocessor is still following the program graph. It is important to mention that the OCFCM can only check branch instructions with fixed target addresses. Branch instructions with dynamic addresses (Jump to Register, for example) must be replaced by branch instructions with fixed addresses (Jump to Address, for example).

In order to detect an inconsistency in the program flow, the instructions fetched by the microprocessor must also be fetched and decoded by the hardware module, to identify branch instructions and locate their destination address. Instead of

implementing a full generic decoder, the proposed hardware module, implements a reduced decoder composed of a list of physical memory positions of all the branch instructions in the program. The decoder can be automatically generated during compilation time and allows the OCFCM to calculate each instruction's consistent destination address based only on the address and the data buses. This leads to a significant area reduction, as well as the maintenance of the original processor's timing characteristics, such as the clock frequency.

In order to adapt the proposed technique to general-purpose microprocessors, reconfiguration must be used, so that the system can reconfigure the same area with different modules, each one specifically designed for each application. Depending on the area available on chip, designers may build more than one module on the FPGA. In this case, the system can have a set of pre-defined OCFCMs, which can be switched between different programs without affecting the overall final computation time.

OCFCM is expected to detect control flow faults that either causing the PC to freeze at the same memory address (through the watchdog) or the ones that break the sequential evolution of the PC with an inconsistent destination address. Unfortunately, these two cases do not comprehend all types of control flow errors. An incorrect decision, whether to take or not the branch, cannot be detected by the module. Therefore, a software-based side is required, with the Inverted Branches technique.

One drawback of OCFCM is that, like PODER, it has to have access to the memory buses. The buses used by some processors which use on-chip embedded cache memories may not be accessible by the hardware module. In such cases, OCFCM cannot be used.

4.5.2 Software-based Side

The software-based side on OCFCM is responsible for choosing which module will be active (when using a group of OCFCM) and performing the transformation on the software that implements the Inverted Branches software-based technique. It also has to guarantee that the program code does not use jumps with dynamic destination addresses. In such cases, a transformation must be performed to convert them into jump with static destination addresses. These characteristics are mandatory for achieving better performances and higher fault detection rates.

The first task is performed by predefining the memory address. Whenever an application starts, it writes in a given memory address the identifier of the OCFCM module to run. The active OCFCM modules decode that store instruction and put themselves on hold (if the value does not match their value) or start operating (when the value matches). The Inverted Branches transformation is described in Section 4.3.

The replacement of dynamic address branches per static address branches has to be performed by analyzing the program code and calculating the fixed address behind the register used as target. In most cases, the compiler loads a value to a given register and then performs the dynamic jump. In such cases, it is possible to replace both instructions by a fixed address branch, like Jump to Address, where the address is the one calculated by the transformation tool. When the destination address does not fit the instruction, two or more jumps can be performed. In cases where the target register value is unknown, the replacement is not possible and the technique will not be able to protect the instruction.

4.5.3 Implementation

The OCFCM implementation can be done automatically during compilation time. It inputs a C code, which is compiled into an architecture-dependent machine code file and submitted to HPC-Translator, which then generates a hardened program code (to be executed by the processor) and a Verilog file describing the customized hardware module. The Verilog description is then synthesized to generate the final FPGA configuration bitstream. The complete program transformation and hardware module generation flow is shown in Figure 4.12.

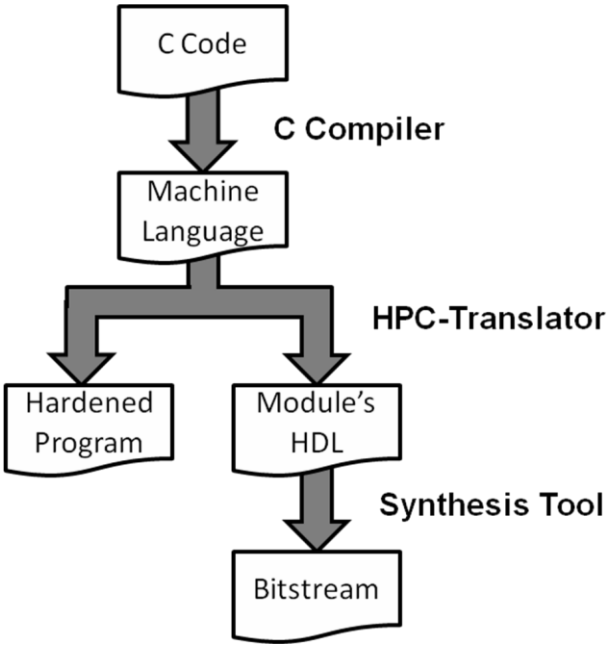

Figure 4.12: Automatic hardware generation flow.

In order to generate the OCFCMs, we input HPCT with each case-study application and receive as output a Verilog description of the hardware module. The Verilog description mainly consists of a list of every branch instruction address in the code, application definitions (such as which memory area is allowed for data and program access) and some processor definitions (such as the maximum number of clock cycles allowed to execute the same instruction).

In order to evaluate both the effectiveness and the feasibility of the presented approach, a benchmark consisting of six applications was created. As case-studies applications, we chose the following six applications: a 6x6 matrix multiplication, a bubble sort, a Dijkstra, a short encryption, a run length encoding and a bit count. The matrix multiplication requires a large amount of data processing with only a few loops and therefore uses mostly the datapath from the microprocessor. The bubble sort algorithm, on the other hand, has a large amount of loops and branch instructions and therefore uses mostly the controlpath. The Dijkstra algorithm is able to find the shortest distance between two nodes in a network and therefore is used in communication of systems on chip. The run length encoding and the short encryption are algorithms normally used in satellites in order to reduce the size of transmitted data by compressing it and to secure the communication by encrypting the transmitted data, respectively. The bit count is a small application that counts the number of bits set to '1' in a configurable fixed loop.

OCFCM's were generated for both an SRAM-based FPGA and a Flash-based FPGA, considering all six case-study applications. The area required for each of them is presented in Table 4.6, represented by the number of Look-Up Tables (LUTs) and Flip-Flops (FFs) for SRAM-based FPGAs and VersaTiles for Flash-based FPGAs. The area required for the modules is up 240 LUTs and 33 FFs, or 7.8% of the miniMIPS implemented in the SRAM-based FPGA board, and up to 528 VersaTiles, or 2.9% of minimIPS implemented in the flash-based FPGA board.

Table 4.6: OCFCM technique area results for a set of applications and the percentage of the area compared to the miniMIPS microprocessor synthesized into FPGA

	SRAM-based FPGA (Virtex 4 xc4vlx80-12ff1148)		Flash-based FPGA (ProAsic3 1500)
OCFCMs	LUTs	FFs	VersaTiles
Matrix Multiplication	200 (6.5%)	33 (2.2%)	349 (1.9%)
Bubble Sort	213 (6.9%)	33 (2.2%)	516 (2.9%)
Bit Count	192 (6.2%)	33 (2.2%)	331 (1.8%)
Dijkstra	216 (7.1%)	33 (2.2%)	424 (2.4%)
Encryption	197 (6.4%)	33 (2.2%)	340 (1.9%)
Encoding	240 (7.8%)	33 (2.2%)	528 (2.9%)

Table 4.7 compares the size of the OCFCM modules to the size of the miniMIPS microprocessor. It is important to notice that each OCFCM depends only on the application and the microprocessor's Instruction Set Architecture (ISA) and therefore is microprocessor independent. It means that more complex microprocessors would require the same resource usage for each OCFCM and therefore a smaller percentage of its total area.

Considering the possibility to dynamically reprogram the SRAM-based FPGA, we also verified the reconfiguration time required for each module, by loading the partial bitstream from the external memory and writing it into the ICAP port. The time consumed to reconfigure the modules was measured by software, using the XTime.h library, based on a Virtex-II Pro (2vp30ff896-7) platform. In the case of the ProASIC3 FPGA, there is no partial reconfiguration, so the entire system must be reconfigured in the FPGA.

Table 4.7: Partial reconfiguration time for SRAM-based FPGA (Virtex 4 xc4vlx80-12ff1148)

Source	Reconfiguration Time (ms)
Full FPGA (2vp30ff896-7)	960.0
Matrix Multiplication	8.5

Bubble Sort	9.0
Bit Count	8.1
Dijkstra	9.2
Encryption	8.3
Encoding	9.9

As shown in Table 4.7, the time required to partially reprogram an OCFCM on a SRAM-based FPGA varied from 8.1ms to 9.9ms. This value is directly proportional to the size of each OCFCM. The reconfiguration is only necessary when the module is not implemented on the board, meaning that it is not required when using the architecture shown in Figure 4.11.

4.6 Hybrid Error-detection Technique using Assertions

Hybrid Error-detection Technique using Assertions (HETA) is the third and final hybrid technique presented in this thesis. It was based in CEDA and its ability to efficiently detect control flow errors between different BBs, and PODER and its ability to detect control flow errors inside the same BB. HETA is aimed at both FPGAs and ASICs, since it implements a non-intrusive hardware module combined with transformation rules on the program code.

As mentioned before, PODER has as main drawbacks scalability issues, since the prime numbers combined with CFID grow at a fast pace, and performance and area overheads. CEDA, on the other hand, is scalable (at a given point it starts losing fault detection capabilities due to signature aliasing) and offers low performance degradation, but cannot achieve full fault detection against transient errors. HETA improves both techniques, by offering higher fault detection rates than CEDA, scalability (at a given point it also starts having aliasing issues) and small performance and area overhead.

Like PODER, HETA combines hardware- software-based techniques into a hybrid technique. Its main objective is to protect the system against control flow errors, which comprises: (1) incorrect jumps to the beginning of a BB, (2) incorrect jumps inside the same BB, (3) incorrect jumps to unused memory addresses and (4) control-flow loops. It is important to note that HETA, just like PODER and OCFCM, cannot detect errors in branch instructions, where the execution flow should have gone to one BB, but went another. In order to do so, it must be combined with the Inverted Branches software-based technique, described in Section 4.3.

The technique is divided in software-based and hardware-based sides, which communicate through memory writes at predefined memory addresses. HETA divides the computational load by exploiting two main concepts:

- Software-based program code transformation: the original program code is transformed based on a set of rules and additional instructions are inserted in order to communicate with the hardware module.
- Hardware-based non-intrusive module: an additional non-intrusive hardware module is added to the architecture. This module implements watchdog and decoder characteristics in order to analyze the processor's control-flow and decode instructions sent from the inserted software instructions.

The main idea behind HETA is to compute the same signature during compilation time, by the compiler, and during runtime, by the hardware module. By doing so, it is possible to compare the resulting signatures and detect errors in the control flow. The calculation during compilation time does not lead to performance or memory overheads and is inexpensive, since it can be done automatically. By performing the comparison in the hardware module, the overheads in performance are reduced, when comparing to software-based techniques, leaving only a few extra instruction in the program code to control the hardware module.

HETA divides the program's execution flow into a BB graph. It then assigns different signatures for each BB, according to rules discussed later in this Section. It then stores these signatures in a global register during runtime. At given points in the code, the software-based side sends these values to the hardware-module that compares with its own calculated value. The main advantage of HETA is that it offers high fault detection rates at low costs on performance and area overhead.

In the next subsections, the terminology used for HETA is presented, as well as the hardware- and software-based sides of the technique.

4.6.1 Terminology

In order to better explain this technique, we will first introduce some terminology.

Program Graph (*P*): $P = \{V, E\}$ is a control flow graph with a set of nodes, $V = \{N1, N2, N3, ..., Nm\}$ and a set of directed edges, $E = \{e1, e2, ..., en\}$.

Node (*N*): A sequence of instructions in a program for which execution always begins with the first instruction and ends with the last instruction of the sequence. There is no branching instruction inside the node except possibly the last instruction and there is no possible branching into the node except to the first instruction of the node.

Edge: A directed edge between nodes *Ni* and *Nj* (denoted $Ni \rightarrow Nj$) representing a possible execution of *Nj* after execution of *Ni* in the absence of any errors.

Successor set (*Succ*): The set of all successors of *N*. $Nj \in Succ(Ni) \Longleftrightarrow Ni \rightarrow Nj \in E$

Predecessor set (*Pred*): The set of all predecessors of *N*. $Ni \in Pred(Nj) \Longleftrightarrow Nj \in succ(Ni)$

Node type (*NT*): A node is of type A if it has multiple predecessors and at least one of its predecessors has multiple successors. A node is of type X if it is not of type A.

Signature register (*S*): A run-time register, which is continuously updated to monitor the execution of the program.

Node Ingress Signature (*NIS*): The expected value of S on ingressing the node on correct execution of the program.

Node Signature (*NS*): The expected value of S at any point within the node on correct execution of the program.

Node Exit Signature (*NES*): The expected value of S on exiting the node on correct execution of the program.

Network (*Net*): A network is a non-empty set of nodes such that $Ni \in Net \Longrightarrow (\forall Nj : pred(Ni) \cap pred(Nj) = \varphi : Nj \in Net)$, i.e., all the successors of each of the predecessors of *Ni* are also in the network, and is minimal, i.e., an empty subset of *Net* follows the above property. Each node in the program belongs to one and only one network. It can be seen on Figure 4.13.

Network predecessors (*Net_pred*): The set *network predecessors* is the union of *predecessors* of all its elements. $net\_pred(Net) = \{\cup pred(Ni) : Ni \in Net\}$. It can be seen

on Figure 4.13.

Related signature set (*A_sig*): This set is the union of *NES* of all the nodes in the network predecessors set and the *NIS* of all nodes of type A in the network.

$A\_sig(Net) = \{\cup NES(Ni) : Ni \in net\_pred(Net)\} \cup \{\cup NIS(Ni) : Ni \in Net\ NT(Ni) = A\}$

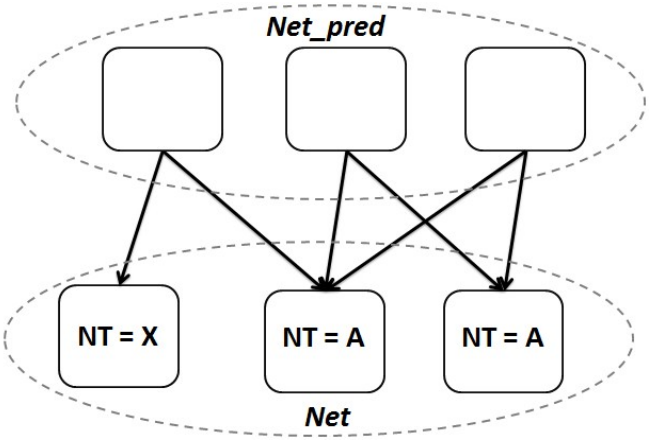

Figure 4.13: Program graph with both NT types.

4.6.2 Software-based Side

The software-based side from HETA is responsible for parsing the code and generating the assertions to be added to the program code. Also, it transforms the program code so that it keeps a global register updated with the current signature in effect and sends periodically the value stored in the register to the hardware module. By doing so, the software-based side can detect, in cooperation with the hardware module, all incorrect jumps to the beginning of a BB (1) and incorrect jumps inside the same BB (2).

HETA's software-side technique can be divided in steps that parse, analyze and add static instructions to the program code. The first step parses the program code and generates a program graph, dividing the program into nodes and edges connecting them. Each node represents a block of instructions that is always executed sequentially, while the edges are the control flow branches that interconnect them. On a second step, each node is analyzed and receives an *NT* value, according to its incoming and outgoing edges. The third step assigns *NIS*, *NS* and *NES* values to each node, according to some rules discussed later. In the final step, instructions are inserted into the original program code.

In the following subsections, the overall technique will be explained, as well as the algorithms used to choose the signatures of the proposed technique.

4.6.2.1 Description Details

As mentioned before, HETA generates signatures based on a program graph representing the application control flow. Figure 4.13 shows the generated program graph and the two types of nodes (*NT* type A and *NT* type X). As one can notice, the nodes where *NT* equals to A have multiple predecessors and at least one of the predecessors has another successor. The node where *NT* equals to X has one single predecessor.

Instructions are statically inserted into the program code to continuously update the value of S during runtime, as to monitor the program flow. When the program execution

reaches a new node, S is assigned with the node's *NIS* value. During its execution, the node's *NS* value is assigned to S. When leaving the node, S is assigned with the node's *NES* value. *NES* and *NIS* values can detect control flow errors caused by jumps between different nodes (internode errors). The *NS* value is responsible for detecting control flow errors with incorrect jumps inside the same node (intranode errors).

At run-time, S can be updated up to three times in each node transition. It varies because *NIS* and *NS*, and also *NES* and *NIS* values may be the same and, therefore, not require an update on S. The updates on S follow the sequence: (1) *NS* to *NES* – when leaving a node, (2) *NES* to *NIS* – when entering a node, and (3) *NIS* to *NS* – when executing a node. The updates 1 and 3 (*NS* to *NES* and *NIS* to *NS*, respectively) are a straight transformation based on the XOR operator, performed by the following instruction:

S = S XOR invariant (*Ni*)

The update 2 (*NES* to *NIS*), on the other hand, depends on the *NT* value of the current node. When *NT* equals to A, the instruction performed is an XOR; otherwise, the instruction performed is an AND, according to the following rule:

S = *S* AND invariant (*Ni*) if *NT*(*Ni*) = A

S = *S* XOR invariant (*Ni*) if *NT*(*Ni*) = X

At certain points of the code, which can be defined by the user, consistency checks can be added through store instructions. Such instructions store the value of S in a given preset memory address that can be identified by the hardware module.

4.6.2.2 Signature checking algorithms

This subsection explains how to assign values to the signatures *NIS*, *NS* and *NES* for each node of the program graph.

In order to allow the hardware module to perform consistency checks, *NS* must be assigned with a value that can be calculated by the module. It also must be a unique value, preventing aliasing. Therefore, each node's *NS* is set with the XOR of all its instructions plus the memory address of its first instruction. Depending on the microprocessor's architecture, only a set of the instructions can be used to generate the *NS* value, in order to avoid aliasing. As an example, only the 16 less significant bits of the instructions can be used to generate *NS*. Considering that S is never reset, but always updated, the hardware module can also detect incorrect values affecting *NIS* and *NES*.

NES and *NIS* values are divided in two parts, the upper half and the lower half. Each part is calculated differently and has a different objective. Therefore, their sizes can vary according to the program code requirements to avoid aliasing.

The upper half is used to identify the program networks (*Nets*), generating an unique value to each *Net*, called *A_sig(Net)*. That means that its minimum size, in order to avoid aliasing is $log_2(\#networks)$ bits. Once generated, *A_sig(Net)* is assigned for every node's *NIS* that belong to *Net* and every node's *NES* that belong to network predecessor *Net_pred*. By doing so, one guarantees the detection of control flow errors between different networks.

The lower half is used to identify the nodes inside the networks (nodes with the same upper half). This algorithm has to guarantee that (1) the *NES* value must be accepted by all the successor nodes and (2) an incorrect jump from a node to one of its

successor nodes must be detected. The algorithm to generate these values is described in Figure 4.14.

```
01. FOR each Network Net {
02.   FOR each node N from Net_pred {
03.     FOR each node F from Net that is not successor of N {
04.       IF F.NIS has a bit in 1
05.         SET bit 1 to 0 in P.NES and in P's successor's NIS
06.       ELSE IF P.NES has any bit in 0
07.         SET bit 0 to 1 in F.NS and in F's predecessor's NES
08.       ELSE {
09.         SET a free bit position in F.NS and in its predecessor's NES to 1
10.         SET the same bit position in its successor's P.NES and NS to 0
11.       }
12.     }
13.     SET the free bit in P.NES to 1
14.   }
15.   FOR each node N with NT=A
16.     SET the fee bits from NIS to 1
17.   FOR each node N with NT=X
18.     SET NIS to its predecessor's NES
19.}
```

Figure 4.14: Algorithm for the signature's lower half.

Figure 4.15 shows an example of the assignment of values to *NIS*, *NS* and *NES* and the operations involved in the signature updating. The example shows only the lower half of the signatures. The main idea of the technique is to allow a transition from a node's *NS* to its successor's *NS*. The transition from node A to node D, for example is quite easy, since *NS(a)*, *NES(a)* and *NIS(d)* are the same. In this case, only one XOR operation is necessary, to transform *NIS(d)* into *NS(d)* (*NS(d) = NIS(d)* xor 0111). The transition from node B to E is a bit more complicated, since node B can also branch to node F. In this case, the *NES(b)* differs from *NS(b)* and *NIS(e)*. Because of this, two operations are required to transform *NS(b)* into *NES(b)* (*NES(b) = NS(b)* xor 1101) and to transform *NES(b)* into *NIS(e)* (*NIS(e) = NES(b)* xor 1110). The most complex case is the transition from node B to node F, where all values are different. In this case, three transformations are necessary, from *NS(b)* to *NES(b)* (*NES(b) = NS(b)* xor 1101); from *NES(b)* to *NIS(f)* (*NIS(f) = NES* and 1100); and *NIS(f)* to *NS(f)* (*NS(f) = NIS(f)* XOR 1000). It is important to mention that the transformation *NES(b)* to *NIS(f)* has to be performed with an AND because node E has *NT* = A.

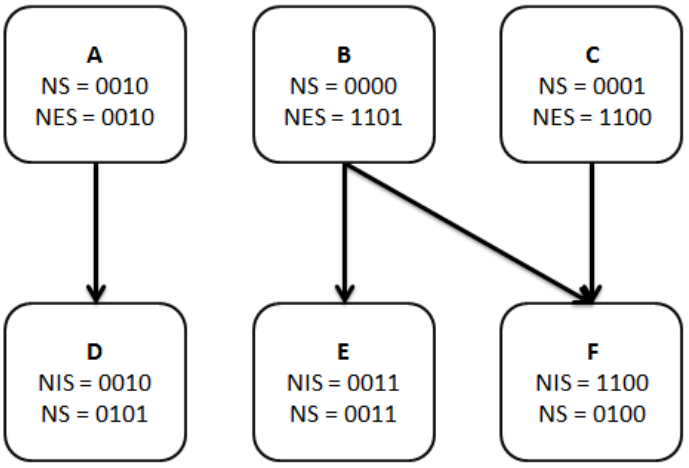

Figure 4.15: NIS, NS and NES signatures.

Another interesting fact about the above example is the possible optimizations. If *NS(f)* was equal to *NIS(f)*, a transformation would be removed, leading to performance gain and less program memory area. The same applies to *NS(b)* and *NES(b)*, *NS(c)* and *NES(c)*, *NES(b)* and *NIS(e)* and *NES(b)* and *NIS(f)*. *NS(d)* cannot have the same value as *NIS(d)*, because *NS(a)* already has that value, and it would lead to aliasing.

It is important to note that HETA, like PODER and OCFCM, cannot detect incorrect but legal jumps (according to the program graph). In order to do that, the Inverted Branches software-based technique, described in Section 4.3 is required. Also, HETA may present aliasing, when the program code has many BBs. With big applications, some signatures may start to repeat themselves and an error may not be detected by the technique.

4.6.3 Hardware-based Side

The hardware-based side of HETA is responsible for complementing the software-based side in detecting incorrect jumps to the beginning of a BB (1) and incorrect jumps to the same BB (2), but also detecting incorrect jumps to unused memory addresses (3) and control flow loops (4).

HETA only updates the value of the *S*, which means that it is never reset. By removing the reset present in PODER, for example, the beginning of a BB does not have any initialization and therefore is equal to any other instruction. Because of that, the checking performed by the hardware module can detect incorrect jumps to the beginning of a BB (1).

As mentioned in the previous subsection, a BB's value of *NS* equals to the XOR operation of all its instructions plus the memory address of its first instruction. This operation is very important, since the hardware module can calculate the same value, by XOR'ing all instruction read from the program memory by the processor. It only needs two flags that indicate the beginning and the end of a BB. By adding the *NS* value to the *S* that stores the signature values, HETA can detect incorrect jumps to the same BB (2).

In order to calculate the XOR and perform the checks, HETA relies on a small decoder that reads data and address buses and the read/write signal between the microprocessor and the memory in order to perform the instructions sent by the software-based side. The decoder reads the buses searching for two instructions: (1) Reset XOR, which resets the hardware module register that store the XOR value and (2) Check XOR, which performs a consistency check, by verifying the value in the data bus

with the internal module's registers storing the current XOR value. To perform the XOR instruction, the hardware module implements a simple accumulator which XORs itself with every new value. In order to have access to the memory buses, it sits between the processor and its memory. Figure 4.16 shows the overall architecture, with the hardware module connected to a processor.

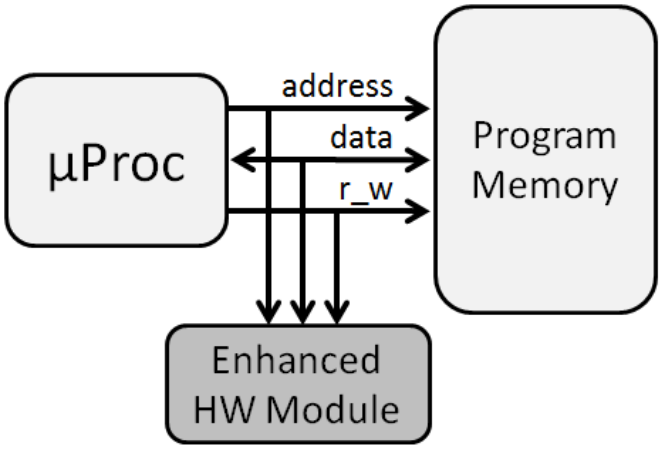

Figure 4.16: HETA's system architecture.

The watchdog characteristics allow the hardware module to detect incorrect jumps to unused memory addresses (3), by receiving information of the memory area used by the application's program code (both program and data). When the processor tries to access an address that is out of range, an error is flagged.

Like PODER and OCFCM, HETA can also detect control flow loops (4). In order to detect this kind of error, a watchdog timer is implemented. The counter is reset every time the software-based technique side enters a BB, by performing a Reset XOR instruction. When the counter overflows, an error is flagged. By doing so, the hardware module can detect a control flow loop that causes the execution flow to be stuck at a single instruction.

HETA has to main drawbacks. The first one is the signature aliasing issue that may lead to undetected errors, when the protected application has a huge number of BBs. The second one is that the technique requires access to the memory buses. Processors with on-chip embedded cache memories may not allow access of its memory buses to the hardware module. In such cases, another approach should be used.

4.6.4 Implementation Details

The implementation of HETA consists of the hardware module implementation and the software transformation. We have used the miniMIPS microprocessor as platform to implement the technique. The following subsections describe the implementations required to harden two case-study applications with HETA.

4.6.4.1 HETA Software Transformation

The software transformation is responsible for implementing two main roles: updating *S* to *NIS*, *NS* and *NES*, and controlling the hardware module. The first task is performed by adding XOR and AND instructions to the program code, according to the rules described in the previous sections. The second is performed with XOR instructions in the beginning of BBs and store instructions to predefined memory addresses placed in the end of BBs.

When the program's execution flow enters a BB, HETA adds a "xor S" instruction, which can be seen by the hardware module, informing it that a new BB has started. In order to the control the hardware module to compare its calculated signature value with the one calculated by the software-based side, HETA add a "store S" instruction in the end of a BB. By doing so, it sends S's value to the hardware module, which then compares it, flagging an error if a mismatch is found.

Figure 4.17 shows an example transformation performed when HETA is applied to an unprotected code. It presents a program code divided into two BBs, from instructions 2 to 5 and 6 to 10 in the hardened code, where the first is of has *NT* type X and the second has NT type A. The left column shows the original program code, while the right column shows the transformed hardened program code.

To perform the updates on *S*, HETA uses XOR instructions, represented by instructions 2, 5, 7 and 10, and ADD instructions, represented by instruction 7. As one can see, instruction 2 performs the update on S from NES to NS (since the BB is NT type C, the NIS equals to the NS value). The second basic block requires two instructions to update S into its NS value, represented by instruction 6 and 7. Finally, instructions 5 and 10 are used to update S from NS to NES and prepare them for the execution flow transition.

In order to control the hardware module, HETA uses the "xor S" operations, represented by instructions 2 and 7, to inform that the program's execution flow has entered a new BB. The "store S" operations, represented by instructions 4 and 9 are used to send S's value to the hardware module, so that it can compare them. The stored is performed in a predefined memory address.

Original Code	**Hardened Code**
1: beq r1, r2, 8	1: beq r1, r2, 6
4: add r2, r3, 1 **NT = X**	**2: xor S, constant** 4: add r2, r3, 1 **3: store S** **5: xor S, constant**
8: add r2, r3, 4 **NT = A**	**6: and S, constant** **7: xor S, constant** 8: add r2, r3, 4 **9: store S** **10: xor S, invariant**
11: jmp end	11: jmp end

Figure 4.17: HETA transformation.

In order to evaluate both the effectiveness and the feasibility of the presented approaches, two applications based on two algorithms: 6x6 matrix multiplication and bubble sort classification were chosen to be hardened.

One hardened program for each case study was generated using the HPCT

implementing HETA. Tables 4.8 and 4.9 show the overhead in execution time, code size and data size for the matrix multiplication and bubble sort algorithms, respectively, when comparing the original unhardened program with the version hardened with only HETA and with HETA plus variables and inverted branches (Combined Techniques).

As one can see, HETA's overhead varies from 1.08 to 1.34 times the original execution time and has 1.5 times the original unhardened code size. When combining the techniques, the overhead varied from 1.43 to 1.55 times the original code. The observed difference is because the matrix multiplication requires a large data processing with only a few loops, while the bubble sort algorithm uses a large number of loops and branch instructions. In the code size, the overhead was 2.9 and 2.8 times the original, to the matrix multiplication and bubble sort algorithms, respectively.

Table 4.8: Characteristics for the HETA program transformation to the matrix multiplication

	Original Unhardened	HETA Technique	Combined Techniques
Execution Time (µs)	1,257	1,361 (1.08×)	1,951 (1.55×)
Code Size (bytes)	1,140	1,692 (1.48×)	3,328 (2.91×)
Data Size (bytes)	288	292 (–)	580 (2.04×)

Table 4.9: Characteristics for the HETA program transformation to the bubble sort

	Original Unhardened	HETA Technique	Combined Techniques
Execution Time (µs)	201	272 (1.34x)	288 (1.43×)
Code Size (bytes)	780	1,136 (1.46x)	2,180 (2.79×)
Data Size (bytes)	40	44 (–)	84 (2.1×)

4.6.4.2 Hardware Module Implementation

The hardware module was implemented in VHDL language, based on a timer that signals an error if not reset. To calculate the XOR value, we added a 16-bit accumulator register that performs a XOR operation between its current and new values so that it is not only able to calculate the real-time XOR value, but also to store it. A decoder was also added to identify instructions from the software-based side.

The decoder module keeps reading the memory buses looking for store instruction at given predefined memory addresses. Whenever a store instruction is found, it reads the address bus to check which address the processor is accessing, in order to decode the instruction from the software-based side, and reads the data bus to read the value being sent. It then manages to perform the operation requested from the software-based side, such as a "xor S" or "store S".

The hardware implementation has a total of 64 flip-flops and is not protected against faults, since the worst case scenario is an incorrect error detection. Table 4.10 shows the

size and performance of the implemented microprocessor and the hardware module. As one can see, the hardware module has 11% of the area of the miniMIPS, while maintaining the same operation frequency.

Table 4.10: Original and modified architecture characteristics for HETA technique synthesized in 0.18µ CMOS process technology

Source	miniMIPS	Hardware Module
Area (µm)	24,261.32	2,717.26
Frequency (MHz)	66.7	66.7

It is important to note that the hardware module size is fixed. When using a bigger processor, the hardware module should remain the same and therefore with a smaller percentage of the total area of the processor. The size of the miniMIPS is used only to contextualize the actual size of the hardware module.

5 SIMULATION FAULT INJECTION EXPERIMENTAL RESULTS

We used the fault injector described in Azambuja (2010b) and simulated the circuits using different versions at ModelSim, from Mentor.

The ModelSim software, from Mentor Graphics, is a simulation tool that simulates architectures written in Hardware Description Languages (HDL), such as VHDL and Verilog. It has a Graphic User Interface (GUI) for easy access as well as a console to run scripts with Tool Command Language (TCL). It allows read and write access to any logic signal describing the system during any time of the simulation. With these commands, it is possible to inject faults with precision higher than nanoseconds and full control over the simulation time. On the other hand, ModelSim does not have a fault injection environment capable of injecting faults automatically and collecting the results. In order to do so, we used a fault injector developed by Azambuja (2010b), where it automatically generates TCL scripts that run on top of ModelSim.

The fault injector has three files as inputs: (1) fault definition file, that contains the number of faults to be injected and for how long they will be active in the system, (2) processor definition file, that contains the operating clock frequency, the signals to be upset and detection capabilities, and (3) application definition file, that contains the total runtime of the application and the memory position where the results are stored. As output, the fault injector creates a single TCL file describing the fault injection and result collection to be executed in ModelSim. Figure 5.1 shows the fault injector's role in the simulation fault injection campaign.

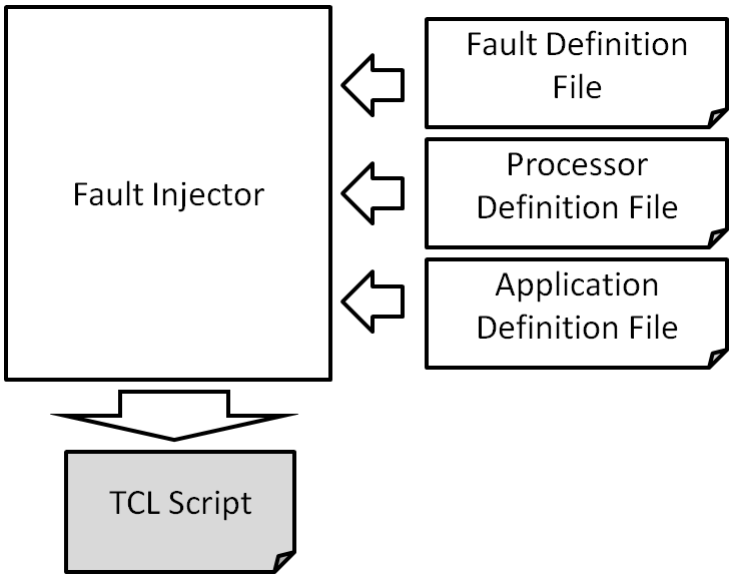

Figure 5.1: Fault injector's role.

The fault injector starts by generating a set of bits from the signals that describe the architecture (from the processor definition file) and a set of times, from the start of the application until its end (from the application definition file). It then combines these two sets and injects a fault in the architecture for the duration described in the fault definition file by using the command "force", from ModelSim. A fault is injected by running the application until the chosen time, performing the "force" command, and then running the application until the end. Figure 5.2 shows the injection of a fault in signal adr_reg1(3) with the duration of two time units.

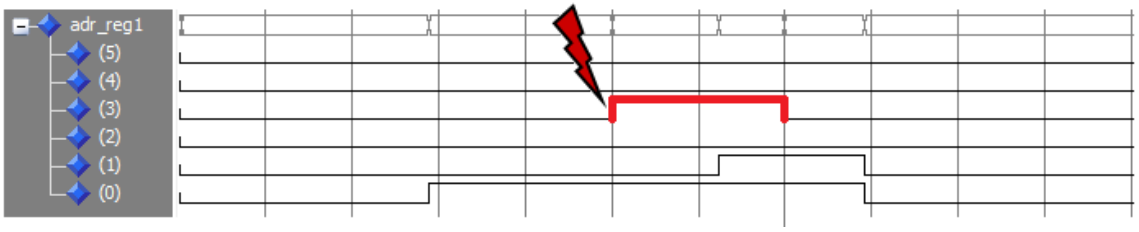

Figure 5.2: Fault injection example of a SET in signal add_reg1 bit 3.

After each fault injection and application run, results are collected and analyzed by the script. As a result, we have the number of injected faults and their effect on the system.

In the following, we describe the simulation fault injection campaign for each technique.

5.1 PODER

In order to start the fault injection campaign, 50 thousand faults were injected in all signals of the non-protected miniMIPS (including registered signals), one per program execution for the matrix multiplication and bubble sort applications. The SEU and SET types of faults were injected directly in the microprocessor VHDL code by using ModelSim XE/III 6.3c. SEUs were injected in registered signals, while SETs were injected in combinational signals, both during one and a half clock cycle. The fault injection campaign is performed automatically. At the end of each execution, the results stored in memory were compared with the expected correct values. If the results matched, the fault was discarded. The amount of faults masked by the program is application related and it should not interfere with the analysis. In the end, only faults not masked by the application were considered in the analysis. When 100% signal coverage was achieved and at least 4 faults per signal were detected we normalized the faults, varying from 4 to 5 faults per signal. Those faults were used to build the test case list.

The faults were classified by their source and effect on the system. We defined four groups of fault sources to inject SEU and SET types of faults: datapath, controlpath, register bank and ALU. Program and data memories are assumed to be protected by EDAC and therefore faults in the memories were not injected.

The fault effects were classified into two different groups: data effect and control effect, according to the fault effect. To sort the faults among these groups, we continuously compared the PC of a golden microprocessor with the PC of the faulty microprocessor. In case of a mismatch, the injected fault was classified as control effect. If the PC matched with the golden’s, the fault was classified as a data effect.

Table 5.1: Percentage of number of error from fault injection results for PODER fault tolerant technique in miniMIPS running the matrix multiplication

	Source	Data Effect	Hardened program version		Control Effect	Hardened program version	
			PODER	PODER Combined		PODER	PODER Combined
SET	Reg. Bank	9	0	100	1	0	100
	ALU	27	9	100	10	0	100
	Control	83	7	100	33	56	100
	Data	42	3	100	2	100	100
	Total	131	6	100	46	44	100
SEU	Reg. Bank	25	0	100	13	15	100
	ALU	4	-	100	0	-	-
	Control	67	13	100	36	68	100
	Data	18	0	100	7	0	100
	Total	114	7	100	56	47	100

Table 5.2: Percentage of number of error from fault injection results for PODER fault tolerant technique in miniMIPS running the bubble sort

	Source	Data Effect	Hardened program version		Control Effect	Hardened program version	
			PODER	PODER Combined		PODER	PODER Combined
SET	Reg. Bank	3	67	100	4	0	100
	ALU	7	100	100	14	14	100
	Control	22	83	100	89	42	100
	Data	14	69	100	28	0	100
	Total	46	80	100	135	29	100
SEU	Reg. Bank	2	0	100	33	18	100
	ALU	0	-	-	0	-	-
	Control	24	5	100	81	35	100
	Data	4	0	100	19	6	100
	Total	30	4	100	133	27	100

When transforming the program, new instructions were added and as a result the time in which the faults were injected changed. Since the injection time is not proportional to the total execution time, we mapped each fault locating the instruction where the fault was injected (by locating its new PC) and pipeline stage where the fault was manifested. Around 1% of the total number of faults could not be mapped and were replaced by new faults.

Tables 5.1 and 5.2 present results for the fault injection in the miniMIPS running the matrix multiplication and the bubble sort applications, respectively. Results show that PODER does not have a high detection rate, when used alone (up to 80%, when considering SETs with data effect on the bubble sort). On the other hand, when combined with Variables and Inverted Branches techniques, the result was 100% fault detection for all cases.

5.2 OCFCM

To test OCFCM, we performed a fault injection campaign where faults were injected in all signals of the non-protected microprocessor, one per program execution. The SEU and SET types of faults were injected directly in the microprocessor VHDL code by using ModelSim SE 6.6b. SEUs were injected in registered signals, while SETs were injected in combinational signals. Faults remained on the system during one and a half clock cycle, so that SETs would hit both rising and falling clock edges. The fault injection campaign was performed automatically by simulation. At the end of each execution, the results stored in memory were compared with the expected correct values.

The experiment continuously compared the PC of a golden microprocessor with the PC of the faulty microprocessor and the generated data results. Fault injection results are presented in Table 5.3. It shows the number of injected faults (Faults Injected) for each application, the number of faults that caused an error in the microprocessor (Incorrect Result) and the detection rate achieved by the proposed solution (Errors Detected). The system was simulated with a clock period of 42ns and a total of 2459 signals describing it. 40,000 faults represent 16 times the number of signals, but only 0.4% of the extensive possibilities of faults for the encryption algorithm.

This fault injection campaign simulates the effects of transient faults in the case-study system is implemented in a Flash-based FPGA, the ProASIC3 from Actel, where the user's logic (VersaTiles) can be upset by SEU and SET.

Table 5.3 shows a fault injection campaign of 40,000 faults for each application. From the total amount of faults injected, around 20% affected the system and caused an error in the final result. When protected by the OCFCM techniques, 100% of the faults were detected. In order to confirm these results, we injected more 140,000 faults in the PC (which is the most sensitive area of the microprocessor with respect to control-flow errors) of the bubble sort application, due to its low execution time and got 100% fault detection. These results mean that the studied hardening approach was able to fully protect the microprocessor system, by detecting every transient fault injected in the case-study applications. Aside from these results, an average of 1% faults with no errors per application was detected.

Table 5.3: Number of faults injected by simulation fault injection in miniMIPS protected by OCFCM and the percentage of detected errors.

Source	Faults Injected	Incorrect Results	Errors Detected
Matrix Multiplication	40,000	8,021	100%
Bubble Sort	40,000	8,746	100%
Bit Count	40,000	8,960	100%
Dijkstra	40,000	8,312	100%
Encryption	40,000	8,995	100%
Encoding	40,000	8,712	100%

5.3 HETA

In this fault injection campaign, we ran two case-study application 100,000 times each and injected one fault per execution. Faults were chosen from all signals of the non-protected microprocessor (including registered signals). The SEU and SET types of faults were injected directly in the microprocessor VHDL code by using the ModelSim simulator. SEUs were injected in registered signals, while SETs were injected in combinational signals, both during one and a half clock cycles. The fault injection campaign was performed automatically. At the end of each execution, the results stored in memory were compared with the expected correct values.

The experiment continuously compared the PC of a golden miniMIPS with the PC of the miniMIPS under fault injection and the generated data results. Fault injection results are presented in Table 5.4. It shows the number of injected faults (Faults Injected) for each application, the number of faults that caused an error in the microprocessor (Incorrect Result) and the detection rate achieved by the proposed solution (Errors Detected). The system was simulated with a clock period of 42ns and a total of 2459 signals describing it. To prove the effectiveness of the proposed technique, we also injected 100,000 faults in the PC for the bubble sort application. It represents around 21 times the total number of clock cycles that the microprocessor takes to run the application. The result was 100% fault detection.

Table 5.4: Number of faults injected by simulation fault injection in miniMIPS protected by HETA and the percentage of detected errors.

Source	Faults Injected	Incorrect Results	Errors Detected
Matrix Multiplication	100,000	12,246	12,246
Bubble Sort	100,000	10,948	10,948

6 CONFIGURATION BITSTREAM FAULT INJECTION EXPERIMENTAL RESULTS

FPGAs have their functionality defined by a large configuration memory. Faults affecting this memory, such as SEUs, can alter the device functionality, therefore changing both the function of individual components, e.g., LUTs and FFs, and the routing between them. The bitstream is defined as the sequence of bits that loads data into the configuration memory, and by changing one of its bits, we can simulate an SEU in the FPGA.

The fault injection by configuration bitstream allows fast injection time, as the Circuit Under Test (CUT) executes at the full FPGA speed, while the simulation fault injection runs at the simulator speed. When compared to radiation experiments, the amount of faults injected is much greater, as the bit flip is directly written to the memory cell. The controllability of the process is inferior to the simulation fault injection, where the designer has access to all internal signals from the implemented design, but superior to radiation experiments, since the exact location of each fault is known.

In order to inject faults in the configuration bitstream, we used the fault injector described in Nazar (2012a). Faults were injected in the configuration bitstream for PODER, OCFCM, and HETA, all combined with VAR and BRA.

The fault injection system was implemented on a Xilinx Virtex 5 device, part XC5VLX110T (XILINX, 2013a), the same device later used for radiation experiments with neutrons. When using the Internal Configuration Access Port (ICAP) to program the configuration memory, and therefore perform the fault injections, one must note not to harm the experiment control configuration. In order to avoid this problem, placement constraints were used to restrict the area occupied by the CUT. Faults were then injected only to the area defined as the Area Under Test (AUT), which is the area of the FPGA where the CUT was implemented.

The configuration memory of the used Virtex 5 is divided into frames. Each frame contains 41 words of 32 bits each, for a total of 1312 bits. Frames are accessed by their individual addresses, through the Frame Address Register (FAR), and are divided into block type, top/bottom bit, row, major address and minor address. Each frame row comprises several rows of the basic FPGA components, such 20 rows of Configurable Logic Blocks (CLBs) and 4 block RAMs.

For all techniques, we injected faults in the configuration memory in an exhaustive fashion, where all bits, from all frames, from the AUT have been changed, one per execution. By doing so, we affected CLBs, block RAMs, I/O blocks, among others. To do so, the frame was read using the ICAP in a burst access mode and stored in the frame memory. A single bit would then be bit flipped and rewritten in the FPGA's

configuration memory. It is important to note that some very specific bit positions may lead to multiple bit errors in other frames, since LUTs may be used to implement components that store user data.

Results are then transmitted to a PC for analysis. The connection is done using a serial cable. After each run, a signature is generated from the implemented fault tolerance techniques informing if any of the techniques detected an error and is the result was correct. Figure 6.1 shows the configuration bitstream fault injector system overview. As one can see, it is divided in the AUT with the CUT, the CUT I/O Ctrl, where the working, golden and init memories are located, the SEU Injector that controls the read/write on the ICAP and the System Control and Report Unit.

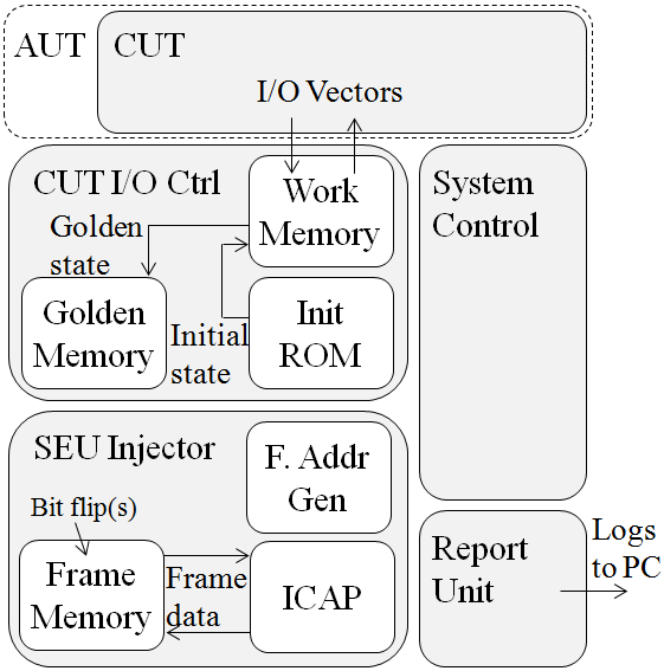

Figure 6.1: Configuration bitstream fault injector system overview (NAZAR, 2012a).

In order to compare the results from fault bitstream fault injection with radiation experiments, we can take into consideration the Virtex5 static cross section per bit from the Xilinx Reliability Report (XILINX, 2013b) measured under neutrons at the Los Alamos Nuclear Science Center (LANSCE) of 6.7×10^{-15} cm^2/bit. We can then calculate the dynamic cross section by multiplying the static cross section per bit by the number of bits that affected the design for each technique. It is important to notice, though, that faults are injected only in the configuration bits before running the application, leaving Block RAM (BRAM) memories and user flip-flops not affected.

In the following, results for PODER, OCFCM, and HETA combined with VAR and BRA are described in detail.

6.1 PODER

In order to perform the configuration bitstream fault injection campaign, we used the same hardware implementation from Chapter 5, running the 6x6 matrix multiplication algorithm. We injected 2,944,640 faults in the AUT of the FPGA board. From those faults, 48,323 caused an error in the circuit's output when considering no fault tolerance detection. Since the fault injection was exhaustive, we can assume that, except for placement and routing differences, the microprocessor core has 48,323 sensitive bits,

which represents 1.6% of the injected faults. This represents a proportion of 61 bit-flips in the configuration memory bits to cause a functional error in the design.

When considering the detection capabilities of PODER combined with VAR and BRA, we can further analyze the results from the fault injection campaign. We divided the faults that affected the resulting matrix of the application into three categories:

1. Detected faults: errors detected by PODER combined with VAR and BRA that did not affect the matrix multiplication result;
2. Detected errors: errors detected by PODER combined with VAR and BRA that corrupted the matrix multiplication results. In this case, the TMR also could detect those errors by the majority voters placed at the output.
3. Not detected errors: errors not detected by PODER combined with VAR and BRA.

Table 6.1 summarizes the bitstream fault injection campaign. As one can see, 48,323 errors affected the DUT, and only 808 faults (1.6%) were not detected by the proposed hardening approach, achieving an overall fault detection coverage of 98.4%.

Table 6.1: Classification of the total 48,323 faults in the miniMIPS protected by PODER technique with VAR and BRA

Classification	Occurences
Detected Faults	49
Detected Errors	47,466
Not Detected Errors	808

Taking into account the static cross section per bit of $6.7x10^{-15}$ cm^2/bit (XILINX, 2013b), we can calculate the dynamic cross section by multiplying it per the number of sensitive bits (48,274), resulting in a dynamic cross section of $3.2\ x10^{-10}$ cm^2. After applying PODER, the number of sensitive bits drops to 808, decreasing the dynamic cross section to $5.4x10^{-12}$ cm^2. One can notice a reduction of 59 times in the dynamic cross section when using PODER.

Such results show that VAR and BRA combined with PODER can be used in harsh environments and allow designers to reach fast fault diagnosis and correction. When comparing to hardware-based techniques, such as TMR, we can notice an area reduction higher than 66% and still acceptable fault coverage of 98.3%. On the other hand, the hardened application takes 2.34 times the original execution time and requires 15% extra area for the hardware module.

In terms of diagnosis, Table 6.2 shows the number of faults and errors in the DUT that were detected by the implemented techniques. PODER was the technique that presented the highest detection capability, with 30,336 exclusive error detections and 42,863 errors detected only with PODER-Control. The highest number of exclusive errors detected was achieved by PODER-Control, showing that it is mandatory for the hardening techniques in order to increase the fault coverage. The Variables technique also showed high error detection, with 13,527 detected errors. The Inverted Branches, on the other hand, could not exclusively detect a single error, although it was able to detect 5,614 errors.

Table 6.2: Diagnosis of detected faults and errors for PODER with VAR and BRA

Source	Incorrect Results		Errors Detected	
Flag Classification	Occurrences	Exclusive Detection	Occurrences	Exclusive Detection
Variables	0	0	13,527	2,910
Inverted Branches	0	0	5,614	0
PODER-Control	2	0	42,863	30,249
PODER-Data	0	0	0	0
PODER-Timeout	49	47	7,074	87

6.2 OCFCM

We injected 2,944,640 faults in the AUT of the FPGA board running a 6x6 matrix multiplication protected with OCFCM, VAR and BRA. From those faults, 54,024 caused an error in the circuit's output when considering no fault tolerance detection. Since the fault injection was exhaustive, we can assume that, except for placement and routing differences, the microprocessor core has 54,024 sensitive bits, which represents 1.8% of the injected faults. This represents a proportion of 54 bit-flips in the configuration memory bits to cause a functional error in the design.

We divided the errors in the same three categories as in Section 6.1. Table 6.3 summarizes the bitstream fault injection campaign. As one can see, 54,024 errors affected the DUT, and only 1,670 faults (3.1%) were not detected by the proposed hardening approach, achieving an overall fault detection coverage of 96.9%.

Table 6.3: Classification of the total 54,024 faults in the miniMIPS protected by OCFCM technique with VAR and BRA

Classification	Occurences
Detected Faults	69
Detected Errors	52,285
Not Detected Errors	1,670

Taking into account the static cross section per bit of 6.7×10^{-15} cm^2/bit (XILINX, 2013b), we can calculate the dynamic cross section by multiplying it per the number of sensitive bits (53,955), resulting in a dynamic cross section of 3.6×10^{-10} cm^2. After applying OCFCM, the number of sensitive bits drops to 1,670, decreasing the dynamic cross section to 1.1×10^{-11} cm^2. One can notice a reduction of 32 times in the dynamic cross section when using OCFCM.

Such results show that software-based techniques combined with HETA can be used in harsh environments and allow designers to reach fast fault diagnosis and correction.

When comparing to hardware-based techniques, such as Xilinx Triple Modular Redundancy (XTMR) with scrubbing, that require modifications to the microprocessor's hardware, we can notice an area reduction higher than 66% and still acceptable fault coverage of 96.9%. On the other hand, the hardened application takes 1.48 times the original time to execute and 6.5% more area to implement the OCFCM hardware module.

In terms of diagnosis, Table 6.4 shows the number of faults and errors in the DUT that were detected by the proposed technique. OCFCM was the technique that presented the highest detection capability, with 41,671 exclusive error detections and 48,576 errors detected only with OCFCM-Control. The highest number of exclusive errors detected was achieved by OCFCM-Control, showing that it is mandatory for the hardening techniques in order to increase the fault coverage. The Variables technique also showed high error detection, with 1,809 detected errors. The Inverted Branches, on the other hand, could not detect a single error.

Table 6.4: Diagnosis of detected faults and errors for OCFCM with VAR and BRA

Source	Incorrect Results		Errors Detected	
Flag Classification	Occurrences	Exclusive Detection	Occurrences	Exclusive Detection
Variables	0	0	1,809	16
Inverted Branches	0	0	0	0
OCFCM-Control	44	6	48,576	38,311
OCFCM-Data	2	0	2,216	1,185
OCFCM-Timeout	63	25	11,162	2,175

6.3 HETA

As for PODER and OCFCM, a total of 2,944,640 faults were injected in the AUT of the FPGA board. From those faults, 75,619 caused an error in the circuit's output when considering no fault tolerance detection. Since the fault injection was exhaustive, we can assume that, except for placement and routing differences, the microprocessor core has 75,507 sensitive bits, which represents 2.6% of the injected faults. This represents a proportion of 40 bit-flips in the configuration memory bits to cause a functional error in the design.

When further analyzing the results, according to the same three categories from Section 6.1, one can see in Table 6.5 that 75,619 errors affected the DUT, and only 3,247 faults (4.3%) were not detected by the proposed hardening approach, achieving an overall fault detection coverage of 95.7%.

Table 6.5: Classification of the total 75,619 faults in the miniMIPS protected by HETA technique with VAR and BRA

Classification	Occurences
Detected Faults	102
Detected Errors	72,270
Not Detected Errors	3,247

Taking into account the static cross section per bit of $6.7x10^{-15}$ cm^2/bit (XILINX, 2013b), we can calculate the dynamic cross section by multiplying it per the number of sensitive bits (75,517), resulting in a dynamic cross section of $5.1x10^{-10}$ cm^2. After applying HETA, the number of sensitive bits drops to 3,247, decreasing the dynamic cross section to $2.1x10^{-11}$ cm^2. One can notice a reduction of 24 times in the dynamic cross section when using HETA.

Such results show that software-based techniques combined with HETA can be used in harsh environments and allow designers to reach fast fault diagnosis and correction. When comparing to hardware-based techniques, such as XTMR with scrubbing, that require modifications to the microprocessor's hardware, we can notice an area reduction higher than 66% and still acceptable fault coverage of 95.7%. On the other hand, the hardened application takes 56% more time to execute and the hardware module requires extra 11.2% of area.

Table 6.6: Diagnosis of detected faults and errors for HETA with VAR and BRA

Source	Incorrect Results		Errors Detected	
Flag Classification	Occurrences	Exclusive Detection	Occurrences	Exclusive Detection
Variables	68	63	61,913	980
Inverted Branches	2	0	6,962	1
HETA-Control	4	1	23,620	4475
HETA-Data	28	26	19,657	448
HETA-Timeout	0	0	63,131	806

In terms of diagnosis, Table 6.6 shows the number of faults and errors in the DUT that were detected by the proposed technique. HETA was the technique that presented the highest detection capability, with 5,769 exclusive error detections and 63,131 errors detected only with HETA-Timeout. The highest number of exclusive errors detected was achieved by HETA-Control, showing that it is mandatory for the hardening techniques in order to increase the fault coverage. The Variables technique also showed high error detection, with 61,913 detected errors. The Inverted Branches, on the other hand, could exclusively detect one single error, although it theoretically complements HETA in control flow error detection (AZAMBUJA, 2012a) and therefore should be maintained in the system.

7 RADIATION EXPERIMENTAL RESULTS

This chapter presents radiation experimental results performed to evaluate the efficiency of the proposed techniques under an accelerated particle test. We tested FPGAs based on Flash and SRAM memory, implementing the miniMIPS microprocessor hardened with HETA, for static and dynamic test.

7.1 MIPS in Flash-based FPGAs

Flash-based FPGAs use flash memory as the configuration memory. Flash memory has low sensitiveness to radiation effects, because it requires a high voltage to be written (change its current state), typically higher than 5V. Such voltages are rarely obtained with energized particles, making flash-based FPGA a good platform to test how an ASIC would respond to the irradiation, without taking into account SEUs in the configuration memory. On the other hand, the circuit responsible for pumping up the operation voltage to the writing voltage (5V or higher) is sensitive to radiation effects and therefore reconfiguration should not be done under radiation.

The miniMIPS is implemented in flash-based FPGAs with two memories: program and data (Harvard architecture). The program memory is implemented on a flash memory and the data memory on SRAM blocks inside the FPGA, being sensitive to radiation effects.

In the following, two irradiation experiments on flash-based FPGAs are described.

7.1.1 TID Experiment

This first experiment was performed at the Instituto de Estudos Avançados (IEAv), in São José dos Campos, Brazil. We built a full-embedded system implemented in a commercial flash-based FPGA part A3P250-PQ208 fabricated in 130-nm flash-based CMOS process that retains programmed design when powered off. The aim was to analyze TID effects in a complete embedded system and to observe in detail its response to external inputs. Signal degradations were also observed during the measurements, as well as temperature increase and power supply current (Icc).

The chosen embedded system was composed of a MIPS microprocessor hardened with HETA, an unhardened SRAM memory embedded in the FPGA, two SpW links (TARRILLO, 2011) and the FPGA embedded Phase-Locked Loop (PLL) clock module. The system has some fault tolerant capabilities that are able to detect transient faults, but not necessary TID effects as radiation results will show. Figure 7.1 shows the architecture of the embedded system.

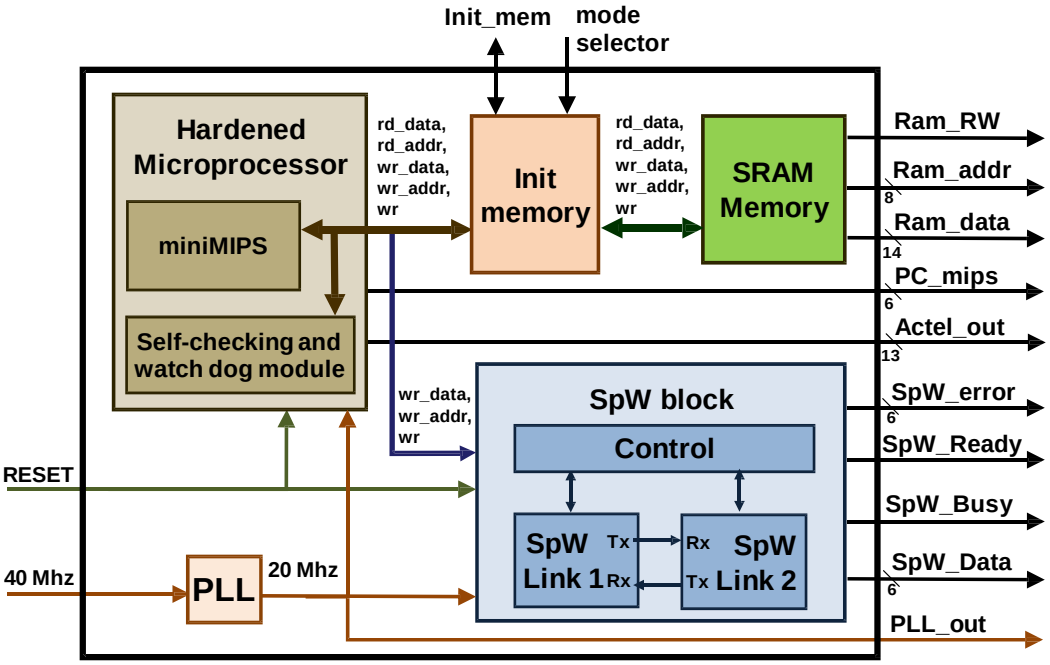

Figure 7.1: Architecture of the embedded system.

The input signals for the embedded system are supplied from an on-chip 40MHz pulse generator, divided by the PLL module into 20MHz. This signal is then connected to a global clock that reaches all modules in the system. The electrical parameters of the output signals were observed and recorded on the scope on-beam during DUT irradiation. The full system has an occupation of 88% core cells, 75% ram blocks and 70% IO cells of the FPGA. The device was irradiated with a collimated gamma-ray beam up to 68 krad(Si) with a dose rate of 2 krad per hour (0.555 rad/s) at room temperature (24.5±0,5°C) using IEAv's Co-60 source. The chip was covered with a 5mm layer of acrylic to reach the electronic equilibrium condition in order to calculate the absorbed dose in silicon from the dose in air measured with an ion chamber. Functional measurements were taken with a 1GHz oscilloscope and a 2GHz logical analyzer. The core current and noise was continuously measured with a digital multimeter. Figure 7.2 shows the experimental setup. The control of the entire experiment, including the acquisition and data storage, was performed remotely. Data acquisitions were carried through at each interval of 30 minutes, corresponding to a step of 1 krad(Si) between acquisitions, until the first functional failure at 47 krad(Si). After the first functional failure, data acquisitions were performed at each interval of 15 minutes, corresponding to a 0.5 krad(Si) acquisition step. Data was stored in text archives for posterior analyses.

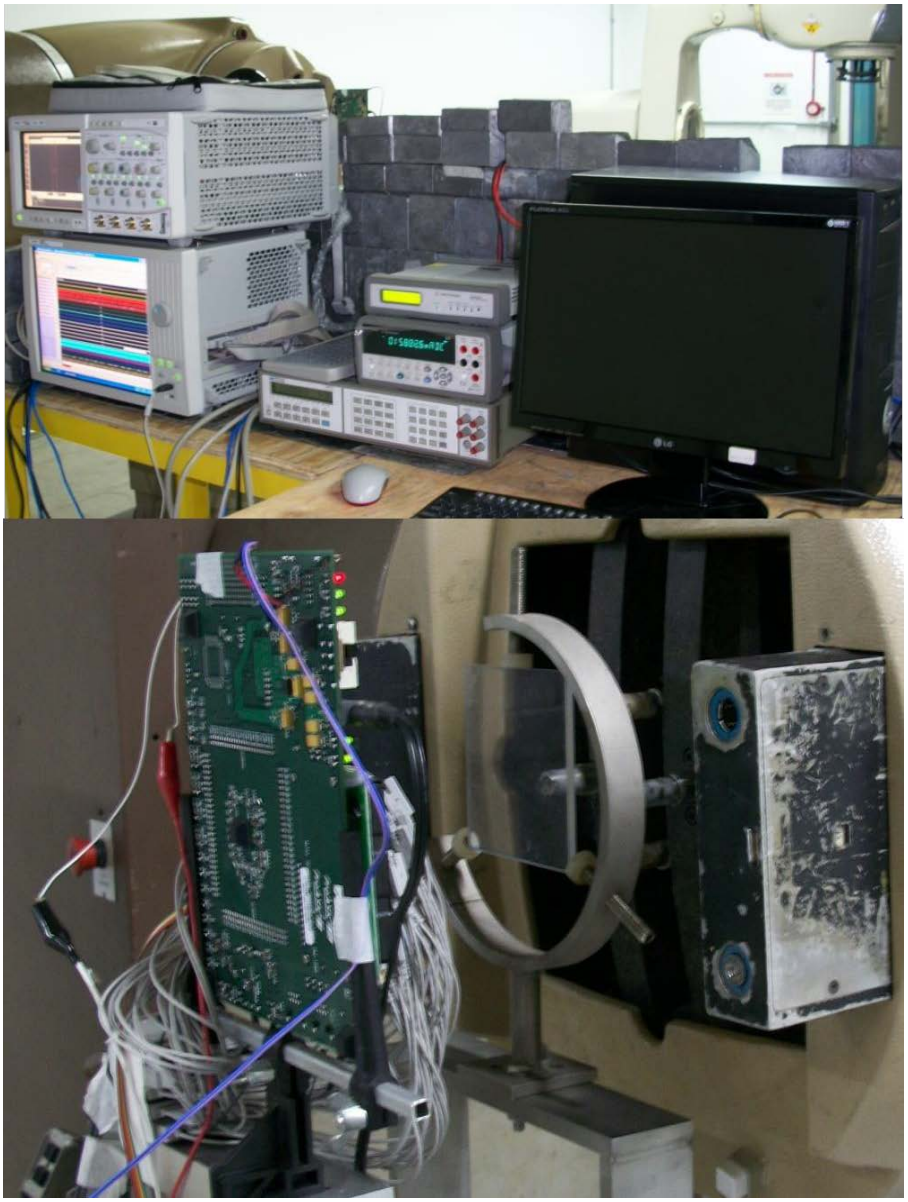

Figure 7.2: Experimental setup.

Data from all signals showed in Figure 7.1 were acquired, in order to check the systems behavior. In a fault-free environment, the microprocessor runs accessing the memory through the memory signals Ram_adr, Ram_data, and Ram_RW, while the signal PC_mips increases itself. At the end of the program's execution, the microprocessor sends the result to the SpW link through the memory data bus. The SpW link 1 then raises the signal SpW_Busy and starts transmitting data to link 2 through signal Tx1 (link 1) and Rx2 (link 2). At the end of transmission, the signal SpW_Ready informs that the signal SpW_Data_Out has a valid data output. While executing, signals Actel_Out and SpW_error_Flag inform if any error was detected in the microprocessor and in the SpW links, respectively.

Figure 7.3 shows the activity of the observed signals (black when active), according to the amount of accumulated dose. By analyzing this graphic, one can deduce the maximum accumulated dose in which each module of the circuit stop working. At 47 krad(Si) the signals SpW_Ready, SpW_Busy and SpW_Data_out stopped their activity, while the SpW_error_flag signal stopped its activity in a dose of 49 krad(Si). However, during all this period the SpW_error_flag was not able to signalize an error.

This is because the SpW error protocol was designed to detect errors such as permanent, intermittent and transient faults, delayed faults as observed under TID. In the case of the embedded SRAM memory, it is noticed that the microprocessor kept writing in the memory until 47 krad(Si) (through signal RAM_WR) and accessing it until 63 krad(Si), through signals RAM_data and RAM_adr. The fault detection HW module worked properly until 55 krad(Si), when it stopped its activity, as one can see through signal Actel_out. However, this module also was not able to detect any degradation in the propagation delay, as this module also is used to tolerate transient faults. The PLL module and the microprocessor's PC were the last parts of the embedded system to stop their activities, at 65 krad(Si) through signals PLL_Out and PC_mips, respectively.

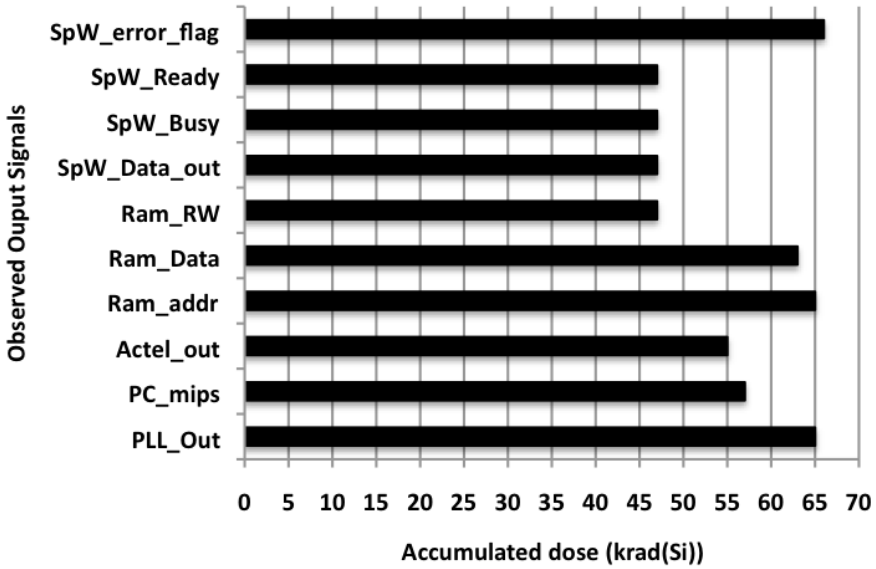

Figure 7.3: Accumulated dose for each signal output.

The Icc was measured during radiation. Figure 7.4 shows Icc and Temperature. As shown, Icc started to change after 45 krad(Si), close to the moment when some modules start stopping working. Note also that the current increases promptly and reaches 1.5 times the original current just before 65 krad(Si). Temperature and current drops abruptly when the majority of the modules fail around 65 krad(Si). The PLL output was measured in terms of frequency, duty cycle and delay compared to the board clock of 40Mhz. Figure 7.5 show the main degradations. It important to notice that the PLL maintained very well the clock output frequency up to 65 krad(Si). After 65 krad(Si), many glitches pulses were observed in the PLL clock output.

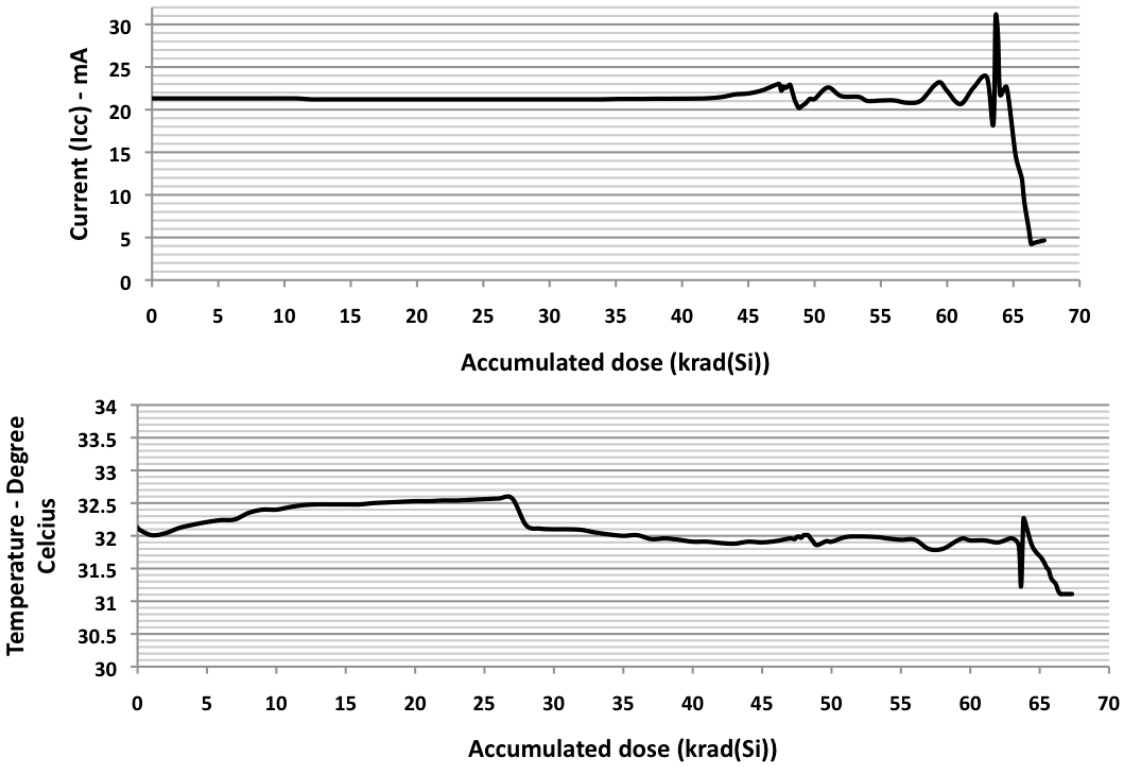

Figure 7.4: Measured current and temperature.

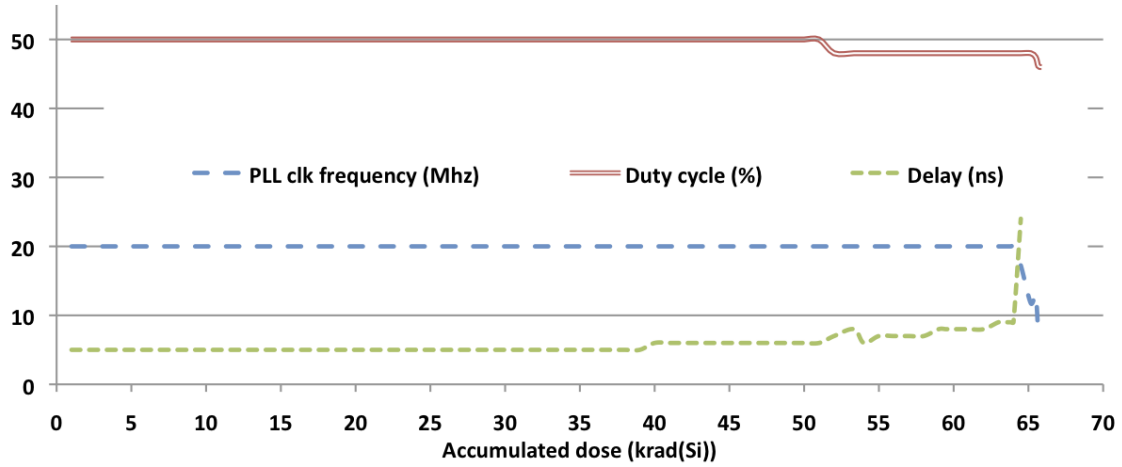

Figure 7.5: PLL clock output measurements: frequency, duty cycle and delay compared to the external 40 MHz clock.

7.1.2 Neutron Experiment

This experiment was performed at CCLRC Rutherford Appleton Laboratory, in Didcot, UK. We implemented the same embedded circuit from Section 7.1.1, shown in Figure 7.1: MIPS microprocessor hardened with HETA, an unhardened SRAM memory embedded in the FPGA, two SpW links (TARRILLO, 2011) and the FPGA embedded PLL clock module. The part used was also the same as the one used on the previous experiment: a commercial flash-based FPGA part A3P250-PQ208 fabricated in 130-nm flash-based CMOS process.

We irradiated the device with a fluence of approximately $1.5 \cdot 10^{10}$ n/(cm^2) with the available spectrum (shown in Figure 7.6), which has already been demonstrated to be suitable to emulate the atmospheric neutron flux (VIOLANTE, 2007). The available flux was of approximately $4.5 \cdot 10^4$ n/(cm^2•s) for energies above 10 MeV. The beam was focused on a spot with a diameter of 3 cm plus 1cm of penumbra, which is enough to cover the whole FPGA chip. Irradiation was performed at room temperature with normal incidence.

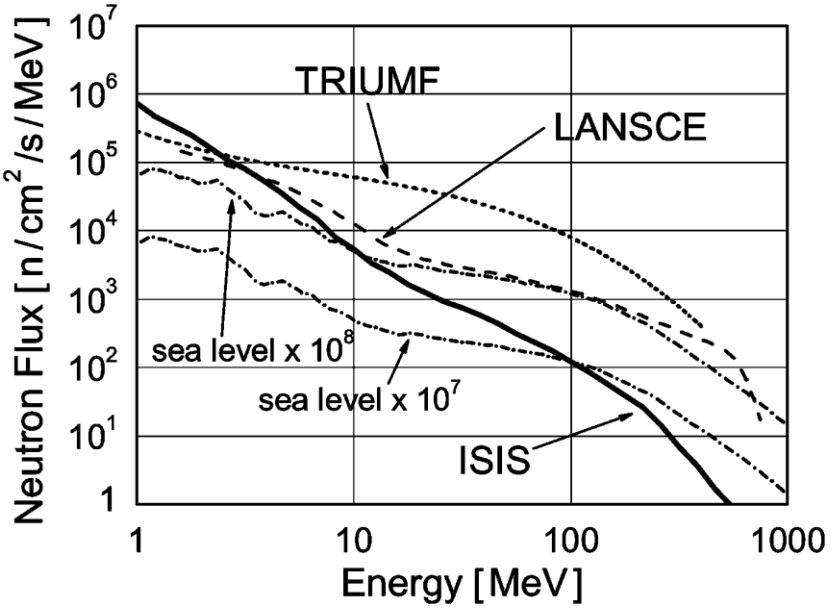

Figure 7.6: ISIS spectrum compared to those of the LANSCE and TRIUMF facilities and to the terrestrial one at sea level multiplied by 10^7 and 10^8.

No errors were observed in the outputs.

7.2 MIPS in SRAM-based FPGAs

SRAM-based FPGAs use SRAM memory as the configuration memory. Differently from flash memories, SRAM are very sensitive to radiation effects. It happens due to low write voltages, the high density of memory cells and small transistor sizes. The miniMIPS is implemented in SRAM-based FPGAs with one single memory containing program and data (Von Neumann architecture). The memory is implemented on SRAM blocks inside the FPGA, being sensitive to radiation effects.

In the following, one irradiation experiment on SRAM-based FPGAs is described.

7.2.1 TID Experiment

As would be expected from the thin gate oxides contained in these technologies, little or no parametric shift was noted during any of the radiation exposures were observed in previous tested by Xilinx (FABULA, 2000). The current was constant up to 80 krads(Si).

7.2.2 Neutron Experiment

This irradiation experiment was performed at the LANSCE facility, in Los Alamos, USA. We implemented the miniMIPS microprocessor in a Virtex5 SRAM-based FPGA, part XC5VLX110T. The main goal was to check the response of HETA when applied to SEEs in SRAM-based FPGAs.

The case-study circuit is composed of three soft-core microprocessors with hardware module and embedded memory (BRAM). Each TMR module of the DUT is a soft-core microprocessor miniMIPS using shared data and program memory. Each soft-core is connected to an embedded memory BRAM used to store the program code and data. The soft-core is protected by HETA, a non-intrusive hybrid technique that uses software redundancy in the application that is run in this processor and an extra hardware module used to monitor the communication between the processor and the BRAM. The test case application is a 32-bit 6x6 sequential matrix multiplication algorithm. After each run, the microprocessor sends the error flags from the hardware module and an End of Execution (EoE) flag to all the interface control units. The DUT circuit was implemented into the XC5VLX110T Virtex5 FPGA. Figure 7.7 shows the architecture of the embedded system and the connections between the different modules.

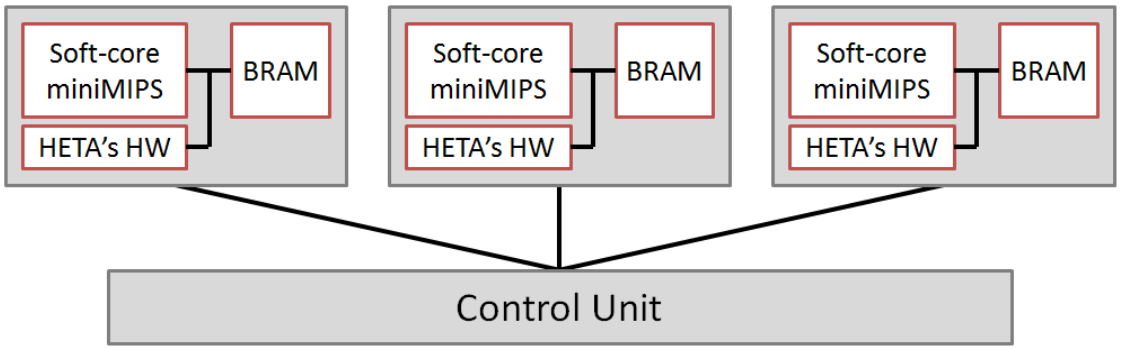

Figure 7.7: DUT's architecture with the test control unit.

The HETA's hardware module is implemented without modifying the microprocessor architecture (non-intrusive) as a logic block that works in tandem with the microprocessor. The module can monitor the data exchanged through the buses between the microprocessor and the embedded memory (BRAM) and detect unexpected deviations in the control flow and microprocessor timeouts. 5 error flags are generated, being helpful to diagnose of the error and to improve the technique itself. Flag 1, or Variables, corresponds to an error detected by the Variables technique, flag 2, or Inverted Branches, corresponds to an error in the Inverted Branches technique and flags 3 (HETA-Control), 4 (HETA-Data) and 5 (HETA-timeout) are related to errors from HETA (control flow error, data flow error and timeout, respectively). Each soft-core miniMIPS module has an occupation of 2,411 slice LUTs and 1,570 slice registers and, additionally, three 36k BRAM memories. From that area, 75 slice LUTs and 98 slice registers belong to HETA's hardware module, which corresponds to around 4% of the total area.

In order to collect results from the DUTs and send them to a computer, we implemented in the FPGA a control unit. The control unit is composed of a Finite State Machine (FSM) capable of reading the BRAMs from the DUT and comparators to detect if a fault occurred in one of the TMR modules. Once the EoE flag is received from one of the microprocessors, the FSM puts it on hold and reads its BRAM. Once the data from each of the three BRAM is read, the control unit generates the following flags: (1) difference between modules #0 and #1 – bit(0), (2) difference between modules #0 and #2 – bit(1), (3) difference between modules #1 and #2 – bit(2), (4) HETA's error flags – bit(3-7) (5) ready sign – bit(8), and (4) matrix multiplication result – 1152 bits. This module also implements a watchdog to detect if one of the TMR microprocessors is not responding and a serial interface circuit, responsible for sending the data collected during the radiation test through a serial cable to a computer. Since

our main objective is analyzing faults in the DUT, the whole control unit's circuit was duplicated and an extra error flag was added, so that we could diagnose faults affecting the control unit.

Figure 7.8 shows the FPGA placement after the routing performed by the Xilinx's tool. The top three squares represent each miniMIPS + HETA's hardware + BRAM module. The bottom three modules are the control unit, which was divided in three smaller modules: the serial interface (in the middle) and the two duplicated circuits. Boxes were added to highlight the area of each module and the real modules occupancy is represented by the dots inside the boxes (the serial interface has approximately 30% of the size of the box). Note that each redundant domain is placed far apart to minimize interference and shortcuts in presence of SEE in the configuration bits.

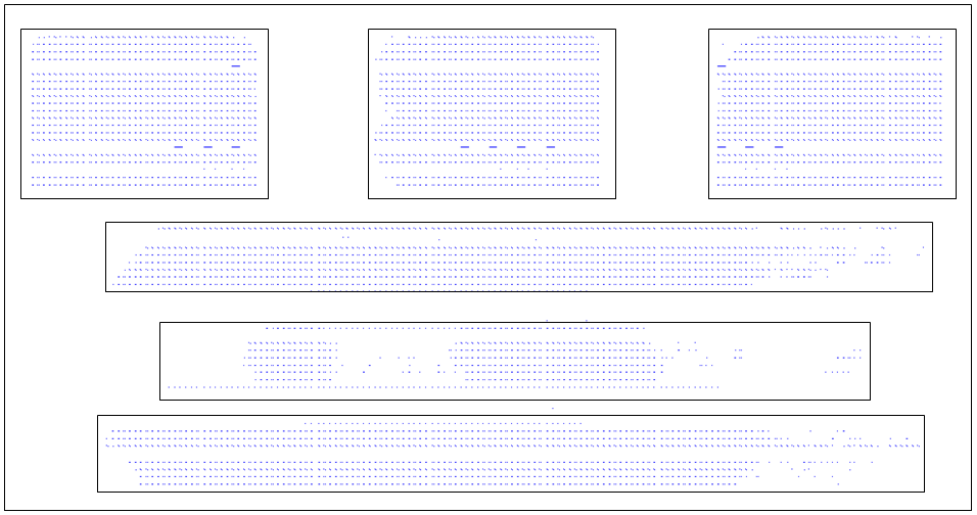

Figure 7.8: FPGA's module placement.

Radiation test was performed at Los Alamos National Laboratory's (LANL) - Los Alamos Neutron Science Center (LANSCE) Irradiation of Chips and Electronics (ICE II) House II in September 2012. We irradiated the devices with the available spectrum that emulates the atmospheric neutron flux. The available neutron flux was about $9x10^5$ $n/(cm^2 \cdot s)$ for energies above 10 MeV. The beam was focused on a spot with a diameter of 2cm plus 1cm of penumbra, thus enough for uniformly irradiate the FPGA. Irradiation was performed at room temperature with normal incidence.

Table 7.1 summarizes the neutron experiment results. As one can see, 958 errors affected the DUT after more than 97 hours of irradiation, and only 48 faults (5.3%) were not detected by the proposed hardening approach, achieving an overall fault coverage of 94.7%. We can further analyze the experimental data by dividing the errors in three subclasses (see Tab. III):

4. Detected faults: errors detected by the software-based technique and HETA's hardware module that did not affect the matrix multiplication result;
5. Detected errors: errors detected by the software-based technique and HETA's hardware module that corrupted the matrix multiplication results. In this case, the TMR also could detect those errors by the majority voters placed at the output.
6. Not detected errors: errors not detected by the software-based technique and HETA's hardware module, but detected by the TMR.

The cross-section of the application, calculated by dividing the amount of observed output errors by the fluence (about $3.5x10^{11}n/cm^2$), is $9.14x10^{-9}cm^2$. If we consider that there were three modules of the miniMIPS, the cross-section per miniMIPS is the result of the cross-section divided by three, which is $3.05\ x10^{-9}cm^2$.

Table 7.1: Classification of the total 958 faults in FPGA design tested under neutrons

Classification	Occurences
Detected Faults	157
Detected Errors	753
Not Detected Errors	48

As the available spectrum of energy resemble the atmospheric one, multiplying the experimental cross-section per miniMIPS by an average flux of 14 $n/(cm^2 \bullet h)$ at New York City (NORMAND, 1996), we can estimate the neutron-induced error rate at sea level per miniMIPS to be $4.27\ x10^{-17}$ Failure In Time (FIT). As reported in Table 7.1, the proposed technique detected most of the errors. The cross-section of the hardened application, defined as the undetected output errors divided by the fluence, is of $1.73x10^{-10}cm^2$. The undetected error rate at sea level per miniMIPS is then reduced to $8.07\ x10^{-19}$ FIT, being two order or magnitude lower with respect to the unhardened design.

Such results show that software-based techniques combined with HETA can be used in harsh environments and allow designers to reach fast fault diagnosis and correction. When comparing to hardware-based techniques, such as XTMR with scrubbing, that require modifications to the microprocessor's hardware, we can notice an area reduction higher than 66% and still acceptable fault coverage of 94.7%. On the other hand, the hardened application takes 56% more time to execute.

In terms of diagnosis, Table 7.2 shows the number of faults and errors in the DUT that were detected by the proposed techniques and the correspondent flag classification.

Table 7.2: Fault injection by partial reconfiguration in SRAM-based FPGA

Source	Incorrect Results		Errors Detected	
Flag Classification	Occurrences	Exclusive Detection	Occurrences	Exclusive Detection
Variables	0	0	502	30
Inverted Branches	0	0	38	0
HETA-Control	142	139	168	53
HETA-Data	14	14	112	104
HETA-Timeout	4	1	521	6

As shown in Table 7.2, HETA was the technique that presented the highest detection capability, with 157 fault detections, 163 exclusive error detections and 521 errors detected only with HETA-Timeout. The highest number of exclusive errors detected was achieved by HETA-Data, showing that it is mandatory for the hardening techniques in order to increase the fault coverage. The Variables technique also showed high error detection, with 502 detected errors. The Inverted Branches, on the other hand, could not exclusively detect a single error, although it theoretically complements HETA in control flow error detection and therefore should be maintained in the system.

8 CONCLUSIONS AND FUTURE WORK

In this thesis, hybrid fault tolerant techniques to detect SEE in processors were developed and verified under fault injection and radiation experiment. In chapter 3 we presented the related works through fault tolerant techniques. These works showed different approaches to deal with transient faults at hardware, software and hybrid levels. They presented interesting concepts and proved the advantage of using fault tolerant techniques at different levels to achieve high fault detection rates. In spite of that, the main disadvantages were the intrusiveness of most of hardware-based and hybrid techniques and the performance degradations and memory overhead of software-based techniques.

Chapter 4 presented our proposed fault tolerant techniques to detect transient errors in processors. Two previously known techniques were presented, called Variables and Inverted Branches. In addition, three new hybrid techniques were proposed, called PODER, OCFCM and HETA. The techniques, as well as the theory and origin behind them, were discussed in detail. Their implementations, including program transformation and hardware implementation (when required), were presented and results according to execution time (performance degradation), program memory and data memory overheads were shown. Results showed for all techniques a performance degradation varying from 1.08 to 1.69, data memory overhead up to 2 times (when using the Variables technique), and program memory overheads up to 2.18 the original one.

In Chapter 5, we presented the fault injection experimental results by simulation. A fault injector implemented in Java was used and faults were injected by running a script on top of the simulator ModelSim, from Mentor. Faults were injected, one per program execution, in all VHDL signals describing the DUT, at RTL level. Techniques PODER, OCFCM and HETA showed high detection rates for faults affecting the program's execution flow. On the other hand, as expected, they could not detect some faults affecting the data flow. In order to increase the detection, the techniques were combined with the Variables and Inverted Branches software-based techniques. The combination resulted in 100% fault detection for all techniques applied to all case-study applications. Such results show that the proposed techniques could not only be combined with data flow techniques, but also reach high detection rates when applied to real applications.

In Chapter 6, we described the fault injection campaign by modifying the configuration bitstream of a Virtex 5 SRAM-based FPGA. By doing so, we injected faults that were not guaranteed to be detected by the proposed techniques, when modifying the functions implemented in the FPGA through its configuration memory. All the proposed techniques were implemented and analyzed according to faults detected, errors detected and errors not detected. Results showed that PODER was the hybrid technique with higher detecting, reaching 98.4% error detection, which is

considered a high detection rate, especially for techniques that do no aim to protect the configuration memory of the FPGA. HETA, with the lowest detection rate, reached 95.7% error detection.

Chapter 7 presented three irradiation experimental results. FPGAs with memory configuration based on flash and SRAM were used. We analyzed the hybrid HETA technique applied to a miniMIPS microprocessor and combined with the Variables and Inverted Branches software-based techniques. We used neutron beam sources from LANSCE and CCLRC Rutherford Appleton Laboratory to test the parts XC5VLX110T (SRAM-based), from Xilinx, and A3P250-PQ208 (flash-based), from Actel, respectively. Results showed a low sensitiveness to radiation effects for the flash-based FPGA and a cross-section of 9.14x10-9cm2 for the SRAM-based FPGA. When HETA was applied to the SRAM-based FPGA, the cross-section was reduced by two orders of magnitude. We also performed TID experiments using a Co-60 source from IEAV. The part tested was a flash-based FPGA, part A3P250-PQ208, from Actel. Results showed functional failures at 45 krad(Si).

The techniques proposed in this thesis have shown interesting results, when compared to related works in the literature. The achieved detection rates combined with the performance degradation and area and memory overheads improved the state-of-the-art, by providing new ways of protecting processor system with higher fault tolerance at smaller costs of performance degradation and area overhead. These results have been backed by intense fault injection campaigns, performed by simulating upsets at RTL level and by injecting faults in the configuration memory bitstream, and TID and neutron irradiation experiments. By doing so, we tested the techniques from their early development stages until real case scenarios.

As future work, we intend to expand the techniques to cope with faults in the configuration bitstream, in order to increase the detection rates for SRAM-based FPGAs and analyze the response of the proposed techniques to multiple faults. Also, we would like to apply the techniques to more complex applications, such as operating systems and real time benchmarks. So far, we have implemented the techniques on the miniMIPS and have previous results on the LEON3 (GAISLER, 2013). We would like to extend it to and ARM architecture, such as the ARM Cortex-A9 and expand it to cope with Graphic Processing Units (GPU) and superscalar architectures.

REFERENCES

AEROFLEX Gaisler. LEON3 [Online]. Available at: http://www.gaisler.com/index.php /products/processors/leon3?task=view&id=13, 2013.

ALKHALIFA, Z.; NAIR, V.; KRISHNAMURTHY, N.; ABRAHAM, J. Design and evaluation of system-level checks for on-line control flow error detection. **IEEE Transactions on Parallel and Distributed Systems,** New York, USA: IEEE Computer Society, 1999, v. 10, n. 6, p. 627-641.

ALMEIDA, F.; KASTENSMIDT, F.; PAGLIARINI, S.; ENTRENA, L.; LINDOSO, A.; MILLAN, E.; CHIELLI, E.; NAVINER, L.; NAVINER, J. Single-event-induced charge sharing effects in TMR with different levels of granularity. In: RADIATION EFFECTS ON COMPONENTS AND SYSTEMS, RADECS 2012. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2012.

ANGHEL, L.; NICOLAIDIS, M. Cost reduction and evaluation of a temporary faults detection technique. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE, 2000, DATE 2000, Paris, FRA. **Proceedings…** New York, USA: ACM Press, 2000, p. 591-598.

AUSTIN, T. DIVA: a reliable substrate for deep submicron microarchitecture design. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 32., 1999, MICRO32, Haifa, ISR. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 1999, p. 196-207.

AVIZIENIS, A.; LAPRIE, J-C.; RANDELL, B; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, vol.1, no.1, 2004.

AZAMBUJA, J.; LAPOLLI, A.; ROSA, L.; KASTENSMIDT, F. Detecting SEEs in Microprocessors through a Non-Intrusive Hybrid Technique. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2011, v. 58, n. 3, p.993–1000.

AZAMBUJA, J.; SOUSA, F.; ROSA, L.; KASTENSMIDT, F. The limitations of software signature and basic block sizing in soft error fault coverage. In: LATIN AMERICAN TEST WORKSHOP, LATW 2010. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2010.

AZAMBUJA, J. Análise de técnicas de tolerância a falhas baseadas em software para a proteção de microprocessadores. Master Thesis. 2010.

AZAMBUJA, J.; PAGLIARINI, S.; ROSA, L.; KASTENSMIDT, F. Exploring the limitations of software-only techniques in SEE detection coverage. **Journal of Electronic Testing**, Vol. 27, p. 541-550, 2011.

AZAMBUJA, J.; PAGLIARINI, S.; ALTIERI, M.; KASTENSMIDT, F.; HUBNER, M.; BECKER, J.; FOUCARD, G.; VELAZCO, R. A Fault Tolerant Approach to Detect Transient Faults in Microprocessors Based on a Non-Intrusive Reconfigurable Hardware. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2012, v. 59, n. 4, p.1117–1124.

AZAMBUJA, J.; ALTIERI, M.; BECKER, J.; KASTENSMIDT, F. HETA: hybrid error-detection technique using assertions. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2013.

BAUMANN, R. Soft errors in advanced semiconductor devices-part I: the three radiation sources. **IEEE Transactions on Device and Materials Reliability**, Los Alamitos, USA: IEEE Computer Society, 2001, v. 1, n. 1, p.17–22.

BARNABY, H. Total-ionizing-dose effects in modern CMOS technologies. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2006, v. 53, n. 6, p.3103–3121.

BERNARDI, L.; BOLZANI, L.; REBAUDENGO, M.; REORDA, M.; RODRIGUEZ-ANDINA, J.; VIOLANTE, M. A new hybrid fault detection technique for systems-on-a-chip. **IEEE Transactions on Computers**, New York, USA : IEEE Computer Society, 2006, v. 55, n. 2, p. 185-198.

BOLCHINI, C.; MIELE, A.; SALICE, F.; SCIUTO, D. A model of soft error effects in generic IP processors. In: INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI, 2005. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2005.

BUCHNER, S.; CAMPBELL, A.; REED, R.; FODNESS, B.; KUBOYAMA, S. Angular Dependence of Multiple-Bit Upsets Induced by Protons in a 16 Mbit DRAM. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2004, v. 51, n. 6, p.3270–3277.

CHEYNET, P.; NICOLESCU, B.; VELAZCO, R.; REBAUDENGO, M.; REORDA, S.; VIOLANTE, M. Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. **IEEE Transactions On Nuclear Science**, [S.l.]: IEEE Nuclear and Plasma Sciences Society, 2000, v. 47, n. 6 (part 3), p. 2231-2236.

CHIELLE, E.; AZAMBUJA, J.; BARTH, R.; ALMEIDA, F.; KASTENSMIDT, F. Evaluating Selective Redundancy in Data-flow Software-based Techniques. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2013.

CUENCA-ASENSI, S.; MARTINEZ-ALVAREZ, A.; RESTREPO-CALLE, F.; PALOMO, F.; GUZMAN-MIRANDA, H.; AGUIRRE, M. A novel co-design approach for soft errors mitigation in embedded systems. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2011, v. 58, n. 3, p.1059-1065.

DODD, P.; SHANEYFELT, M.; FELIX, J.; SCHWANK, J. Production and propagation of single-event transients in high-speed digital logic ics. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2004, v. 51, n. 6 (part 2), p.3278–3284.

DONASSOLO, B.; AZAMBUJA, J.; KUAMOTO, L.; DIVERIO, T.; NAVAUX, P. Aplicação de Curso na Área de Cluster de Alto Desempenho. In: WORKSHOP DE PROCESSAMENTO PARALELO DISTRIBUÍDO, 2005, v. 1, p. 115-116.

ENTRENA, L.; GARCIA-VALDERAS, M.; FERNANDEZ-CARDENAL, R.; LINDOSO, A.; LOPEZ-ONGIL, C. Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. **IEEE Transactions on Computers**, 2010, v. 61, n. 3, p. 313- 322.

FABULA, J.; BOGROW, H. Total ionizing dose performance of SRAM-based FPGAs and supporting PROMs. In: MILITARY AND AEROSPACE APPLICATIONS OF PROGRAMMABLE DEVICES AND TECHNOLOGIES INTERNETIONAL CONFERENCE, 2000. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2000.

GOLOUBEVA, O.; REBAUDENGO, M.; SONZA REORDA, M.; VIOLANTE, M. Soft-error detection using control flow assertions. In INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI, 2003. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2003.

HANGOUT, L.; JAN, S: The minimips project [Online]. Available at: http://www. http://opencores.org/project,minimips, 2013.

HOMPSON, S.; CHAU, R.; GHANI, K.; MISTRY, K.; TYAGI, S.; BOHR, M. In search of forever: continued transistor scaling one new material at a time. **IEEE Transactions on Semiconductor Manufacturing**, New York, USA: IEEE Computer Society, 2005, v. 18, n.1, p. 26-36.

HUANG, K.; ABRAHAM, J. Algorithm-based fault tolerance for matrix operations. **IEEE Transactions on Computers**, New York, USA: IEEE Computer Society, 1984, v. 33, p. 518-528

INTERNATIONAL Technology Roadmap for Semiconductors: 2005 Edition, Chapter Design, 2005, pp. 6-7.

KASTENSMIDT, F.; STERPONE, L.; CARRO, L.; SONZA REORDA, M. On the optimal design of triple modular redundancy logic for SRAM-based FPGAs. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE, 2005, DATE 2005. **Proceedings…** New York, USA: ACM Press, 2005, p. 1290-1295.

KEYS, A.; ADAMS, J.; CRESSLER, J; DARTY, C.; JOHNSON, A.; PATRICK, C. High-performance, radiation-hardened electronics for space and lunar environments. AIP Space Technology and Applications International Forum. **Proceedings...** Albuquerque, NM: American Institute of Physics. 2008. p. 749-756.

KIM, N.; AUSTIN, T.; BAAUW, D.; MUDGE, T.; FLAUTNER, K.; HU, J.; IRWIN, M; KANDEMIR, M.; NARAYANAN, V. Leakage current: moore's law

meets static power. **Computer**, Los Alamitos, USA : IEEE Computer Society, 2003, v. 36, p. 68-75.

LINDOSO, A.; ENTRENA, L.; MILLAN, E.; CUENCA-ASENSI, S.; MARTINEZ-ALVAREZ, A.; RESTREPO-CALLE, F. A co-design approach for SET mitigation in embedded systems. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2012, v. 59, n. 4, p.1034–1039.

LISBOA, C.; ERIGSON, M.; CARRO, L. System level approaches for mitigation of long duration transient faults in future technologies. In: IEEE EUROPEAN TEST SYMPOSIUM, 12., ETS 2007, Freiburg, DEU. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2007, p. 165-170.

LU, D. Watchdog processors and structural integrity checking. **IEEE Transactions On Computers**, Los Alamitos, USA: IEEE Computer Society, 1982, v. 31, n. 7, p.681–685.

MCFEARIN, L; NAIR, V. Control-flow checking using assertions. In CONFERENCE ON DEPENDABLE COMPUTING FOR CRITICAL APPLICATIONS, Urbana-Champaign, USA: **Proceedings...** Washington, USA: IEEE Computer Society, 1995, p. 103-112.

MAHMOOD, A.; LU, D.; McCLUSKEY, E. Concurrent fault detection using a watchdog processor and assertions. In: IEEE INTERNATIONAL TEST CONFERENCE, ITC'83, 1983, [S.l.]. **Proceedings…** [S.l.: s.n.], 1983, p. 622-628.

MAHMOOD, A.; McCLUCKEY, E. Concurrent error detection using watchdog processors-a survey. **IEEE Transactions on Computers**. [S.l.]: 1988, v. 37, n. 2, p. 160-174.

MOORE, G. Cramming More Components onto Integrated Circuits. **Electronics Magazine**, 38, 114-117, 1965.

NAMJOO, M.; McCLUSKEY, E. Watchdog processors and capability checking. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 12., 1982, FTCS-12, Santa Monica, USA. **Proceedings…** [S.l.: s.n.], 1982, p. 245-248.

NAMJOO, M. CERBERUS-16: An architecture for a general purpose watchdog processor. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 13., 1983, FTCS-13, Milan, ITA. **Proceedings…** [S.l.: s.n.], 1983, p. 216-219.

NAPOLES, J.; GUZMAN, H.; AGUIRRE, M.; TOMBS, J.; MUNOZ, F.; BAENA, V.; TORRALBA, A.; FRANQUELO, L. Radiation environment emulation for VLSI designs A low cost platform based on xilinx FPGAs. In INTERNATIONAL SYMPOSIUM ON INDUSTRIAL ELECTRONICS, 2007. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2007.

NAZAR, G.; CARRO, L. Fast single-fpga fault injection platform. In: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, 20., DFT 2012, 2012, Monterey, USA. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2012, p. 152-157.

NAZAR, G.; RECH, P.; FROST, C.; CARRO, L. Experimental evaluation of an efficient error detection technique for FPGAs. In: RADIATION EFFECTS ON

COMPONENTS AND SYSTEMS, RADECS 2012. Proceedings… Los Alamitos, USA: IEEE Computer Society, 2012.

NICOLAIDIS, M. Time redundancy based soft-error tolerance to rescue nanometer technologies. In: IEEE VLSI TEST SUMPOSIUM, 17., VTS 1999, Dana Point, USA. **Proceedings…** Washington, DC, USA: IEEE Computer Society, 1999. p. 86-94.

NIEUWLAND, A.; JASAREVIC, S.; JERIN, G. Combinational logic soft error analysis and protection. In: IEEE INTERNATIONAL ON-LINE TEST SYMPOSIUM, 12., IOLTS 2006, Lake of Como, ITA. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2006. p. 99-104.

NORMAND, E. Single event effects in avionics. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 1996, v. 43, n. 2, p. 461-474.

OH, N.; MITRA, S.; McCLUSKEY. ED4I: error detection by diverse data and duplicated instructions. **IEEE Transactions on Computers**, 2002, v. 51, n. 2, p. 180-199.

OH, N.; SHIRVANI, E.; McCLUSKEY, E. Control-flow checking by software signatures. **IEEE Transactions on Reliability**. [S.l.]: IEEE Computer Society?], 2002, v. 51, n. 2, p. 111-122.

OLDAM, T.; MCLEAN, F. Total ionizing dose effects in MOS oxides and devices. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2003, v. 50, n. 3, p.483–499.

PATTERSON, D. A.; HENNESSY, J. L. Computer organization and design: the hardware/software interface. Morgan Kaufmann, 2009.

PILOTTO, C.; AZAMBUJA, J.; KASTENSMIDT, F. Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, 2008, SBCCI 2008, Gramado, BRA. **Proceedings…** New York, USA: IEEE Computer Society, 2007, p.199-204.

ROSSI, D.; OMANA, M.; TOMA, F.; METRA, C. Multiple transient faults in logic: an issue for next generation ICs? In: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, 20., DFT 2005, 2005, Monterey, USA. **Proceedings…** Los Alamitos, USA: IEEE Computer Society, 2005, p. 352-360.

REIS, G.; CHANG, J.; AUGUST, D. Automatic instruction-level software-only recovery. **IEEE Micro**, vol. 27, no. 1, pp. 36–47, 2007.

RHOD, E.; LISBOA, C.; CARRO, L.; SONZA REORDA, M. Hardware and software transparency in the protection of programs against SEUs and SETs. **Journal of Electronic Testing**: theory and applications. Norwell, USA: Kluwer Academic Publishers, 2008, v. 24, n. 1-3, p. 45-56.

O'GORMAN, J.; ROSS, J.; TABER, A.; ZIEGLER, J.; MUHLFELD, H.; MONTROSE, H.; CURTIS, W.; WALSH, J. Field testing for cosmic ray soft errors in semiconductor memories. **IBM Journal of Research and Development**, 1996, p. 41-49.

OHLSSON, J.; RIMEN, M. Implicit signature checking. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 25., 1995, FTCS-25, Pasadena, USA. **Digest of papers…** [S.l.: s.n.], 1995, p. 218-227.

SCHILLACI, M.; REORDA, M.; VIOLANTE, M. A new approach to cope with single event upsets in processor-based systems. In: IEEE LATIN-AMERICAN TEST WORKSHOP, 7., 2006, LATW 2006, Buenos Aires, ARG. **Proceedings…** [S.l.: s.n.], 2006, p. 145-150.

SCHUETTE, M.; SHEN, J. Processor control flow monitoring using signature instruction streams. **IEEE Transactions on Computer**, [S.l.: s.n.], 1987, v. 36, n. 3, p. 264-276.

TARRILLO, J.; AZAMBUJA, J.; KASTENSMIDT, F.; FONSECA, E.; GALHARDO, R.; GONCALEZ, O. Analyzing the effects of TID in an embedded system running in a flash-based FPGA. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2011, v. 58, n. 6, p.2855–2862.

VEMU, R.; ABRAHAM, J. CEDA: control-flow error detection through assertions. In: INTERNATIONAL ON-LINE TEST SYMPOSIUM, 12., 2006, IOLTS 06, Lake of Como, ITA. **Proceedings…** Washington, USA: IEEE Computer Society, 2006, p. 151-158.

VEMU, R.; GURUMURTHY, S.; ABRAHAM, J. ACCE: automatic correction of control-flow errors. In: INTERNATIONAL TEST CONFERENCE, 2007, ITC 2007, [Otawa, CAN]. **Proceedings…** New York, USA: IEEE Computer Society, Oct. 2007, paper 227.2, p. 1-10.

VEMU, R.; ABRAHAM, J. CEDA: control-flow error detection using assertions. **IEEE Transactions On Computers**, Los Alamitos, USA: IEEE Computer Society, 2011, v. 60, n. 9, p.1233–1245.

VIOLANTE, M.; STERPONE, L.; MANUZZATO, A.; GERARDIN, S.; RECH, P.; BAGATIN, M. PACCAGNELLA, A.; ANDREANI, C.; GORINI, G.; PIETROPAOLO, A.; CARDARILLI, G.; PONTARELLI, S.; FROST, C. A new hardware/software platform and a new 1/E neutron source for soft error studies: testing FPGAs at the ISIS facility. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2007, v. 54, n. 4, p.1184–1189.

WAKERLY, J. **Error detecting codes, self-checking circuits and applications**. New York, USA: North-Holland, 1978.

WILKEN, K.; SHEN, J. Continuous Signature Monitoring: low-cost concurrent detection of processor control errors. **IEEE Transactions on Computers Aided Design of Integrated Circuits and Systems**, New York, USA: IEEE Computer Society, 1990, v. 9, n. 6, p. 629-641.

XILINX Corp., Xilinx University Program XUPV5-LX110T Development System [Online]. Available: http://www.xilinx.com/univ/xupv5-lx110t.htm, 2013.

XILINX Corp., Device Reliability Report, UG116 (v9.4), 2013.

APENDIX - PROPOSTA DE DOUTORADO

Universidade Federal do Rio Grande do Sul
Programa de Pós-Graduação em Computação
Instituto de Informática

Plano de Trabalho: Doutorado 2010

Projetando Técnicas Eficientes e Não-Intrusivas Para a Tolerância de *Soft Errors* em Microprocessadores.

Proponente: José Rodrigo Furlanetto de Azambuja
Orientadora: Profa. Dra. Fernanda Kastensmidt

Janeiro, 2010

1. Introdução

Os avanços da última década na indústria de semicondutores aumentaram exponencialmente o desempenho dos microprocessadores. Grande parte destes ganhos em desempenho foi devido a dimensões menores e voltagens mais baixas de operação dos transistores, que levaram a arquiteturas mais complexas com maior grau de paralelismo combinado com uma alta freqüência de relógio. Entretanto, a mesma tecnologia que possibilitou todo este progresso também reduziu a confiabilidade dos transistores, reduzindo a voltagem de limiar e estreitando as margens de ruído (Baumann et al., 2001), (O'Gorman et al., 1996) e assim tornando-os mais suscetíveis a falhas causadas por partículas energizadas (O'Gorman et al., 1996). Como conseqüência, aplicações de alta confiabilidade necessitam técnicas de proteção de falhas capazes de recuperar o sistema de uma falha com um custo mínimo de implementação e desempenho.

Uma das maiores preocupações é conhecida como *soft error*, que é definido como uma falha com efeito transiente provocado pela iteração entre uma partícula energizada com a junção PN no silício. Esta perturbação carrega temporariamente os nodos do circuito, gerando pulsos de voltagem transiente que podem ser interpretados como sinais internos e assim provocando um resultado errôneo (Dodd et al., 2003). Os erros mais típicos relacionados à *soft errors* são *Single Event Upsets* (SEU), caracterizados pela mudança de estado da lógica seqüencial (registradores, flip-flops, memória, etc.) e *Single Effect Transient* (SET), que são pulsos transientes de voltagem na lógica combinacional, podendo ser registrados pela lógica seqüencial. A figura 1.1 mostra uma partícula energizada acertando a junção PN de um transistor e causando um *soft error*.

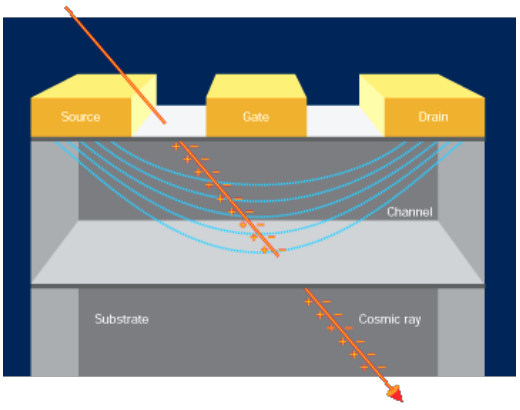

Figura 1.1: Partícula energizada ao acertar um transistor.

Técnicas de tolerância a falhas baseadas em *sofware* podem resultar em alta flexibilidade e baixo tempo de desenvolvimento e custos para sistemas computacionais. Sistemas de alto desempenho chamados *System-on-Chip* (SoC) compostos de um grande numero de microprocessadores e outros núcleos conectados através de uma *Network-on-Chip* (NoC) estão se tornando mais populares em muitas aplicações que requerem alta confiabilidade, como servidores de dados, veículos de transporte, satélites, entre outros. Ao utilizar estes sistemas, a proteção contra falhas fica a cargo do projetista. A tolerância a falhas através de *software* tem recebido muita atenção nestes sistemas, visto que não é necessário alterar o *hardware*. A figura 1.2 mostra uma NoC 3x3, conectando diferentes tipos de componentes através de nove roteadores.

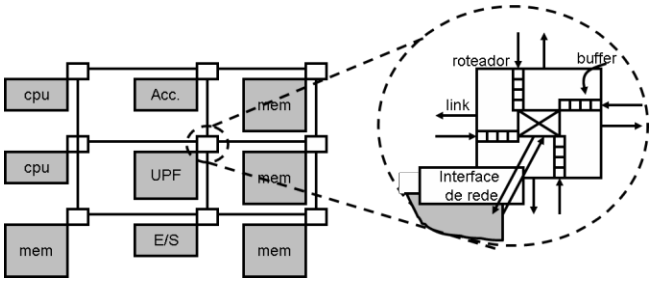

Figura 1.2: Exemplo de uma NoC 3x3.

As técnicas baseadas somente em *software* exploram a redundância de informação, instrução e tempo para detectar e até mesmo corrigir erros durante o fluxo do programa. Todas estas técnicas utilizam instruções adicionais na área de código para ou recomputar instruções ou para gravar e checar informações nas estruturas de *hardware*. Nos últimos anos, foram apresentadas ferramentas para automaticamente injetar tais instruções no código C ou assembly, reduzindo significativamente os custos de implementação.

Trabalhos relacionados apontaram problemas de técnicas baseadas somente em *software*, como a impossibilidade de alcançar uma cobertura completa de falhas do tipo SEU (Bolchini et al., 2005), alto custo de memória e degradação de desempenho. A memória aumenta devido às instruções adicionais e duplicação de memória. A degradação do desempenho acontece devido à execução repetida de instruções (Goloubeva et al., 2003), (Huang et al., 1984), (Oh et al., 2002). Entretanto, não existe estudo na literatura que analisou tanto falhas do tipo SEU quanto SET e correlacionou a localização e os efeitos das falhas injetadas com o estado de detecção. Esta informação é muito para guiar projetistas para melhorar a eficiência e as taxas de detecção de *soft errors* das técnicas de mitigação de falhas baseadas somente em *software*.

2. Estado da Arte

O estado-da-arte da área de técnicas de tolerância a falhas puramente em *software* classifica as técnicas em dois grupos: (1) que protegem erros de dados, como as técnicas propostas por (Rebaudengo et al., 1999), (Cheynet et al., 2000), (Nicolescu et al., 2003), e (2) que protegem contra erros de controle, como *Structural Integrity Checking* (SIC) (Lu, 1982), *Control-Flow Checking by Software Signatures* (CFCSS) (Oh et al., 2002), *Control Flow Checking using Assertions* (CCA) (Mcfearin et al., 1995) e *Enhanced Control Flow Checking using Assertions* (ECCA) (Alkhalifa et al., 1999).

As técnicas de proteção de dados se baseiam na replicação e comparação de instruções, registradores e memória, aumentando consideravelmente os custos com memória de programa e de dados e reduzindo o desempenho do microprocessador, visto que é necessário executar diversas instruções replicadas e de checagem. A maioria das técnicas de proteção do fluxo de programa, por outro lado, divide o código de programa em blocos básicos (partes sequenciais de programa) e atribuem valores a cada bloco básico, para então realizar checagens de fluxo de programa com variáveis globais com o mesmo fim.

As técnicas propostas obtiveram tolerância total de erros de dados do tipo SEU, conseguindo detectar todas as falhas afetando os dados, tanto em memória quanto em registradores, que fossem levar o sistema a um resultado errôneo. Entretanto, as técnicas do segundo grupo ainda não obtiveram 100% de detecção de falhas.

A técnica ECCA estende a CCA e é capaz de detectar todos os erros de desvios entre diferentes blocos básicos, mas não é capaz de detectar nem falhas de desvios dentro do mesmo bloco básico (origem e destino do desvio incorreto de fluxo) nem falhas que causam uma decisão incorreta numa instrução de desvio. A técnica CFCSS, por outro lado, consegue detectar erros dentro do mesmo bloco básico, mas não consegue detectar erros de desvio se múltiplos blocos básicos possuem o mesmo bloco básico de destino, situação muito comum em algoritmos do tipo *controlflow*.

Atualmente, as técnicas existentes na literatura conseguem proteger sistemas contra todos os tipos de erro de dados e a maioria dos erros de fluxo de programa. Entretanto, esta proteção vem com um grande custo em desempenho e em área de memória. As proteções de dados podem chegar a aumentar o tempo de computação de um algoritmo em duas vezes, enquanto a proteção completa pode ultrapassar três vezes

o tempo do algoritmo sem proteção. A memória de programa também tem um aumento considerável, chegando a três vezes o tamanho inicial.

A literatura ainda desconhece um trabalho com uma coleção de técnicas com detalhamento suficiente para o projetista combinar e proteger o seu sistema conforme a sua vontade, podendo chegar a detectar todas as falhas e com um conhecido custo em desempenho e área física.

3. Motivação

O estado da arte na área de proteção de microprocessadores está muito aquém do desejado pelos projetistas, mesmo com o esforço de grandes grupos de pesquisa, como das Universidades de Chung-Hua, de Torino, de Milão, de Atlanta, dentre outras e o grupo de pesquisa TIMA, sediado na França. Embora tenha alcançado um alto nível de detecção de falhas, as técnicas existentes ainda não obtiveram a detecção total de falhas injetadas nem traçaram uma relação entre o local das falhas injetadas com o efeito no microprocessador e o resultado de detecção de cada técnica.

Durante o seu mestrado, o candidato desenvolveu um trabalho sobre a proteção de microprocessadores através de técnicas puramente em *hardware* e *software*. Foram implementadas e simuladas diversas arquiteturas através da replicação de diferentes partes do microprocessador e adição de módulos intrusivos e extrusivos de checagem de estado e construída uma ferramenta chamada *Hardening Post-Compiling Tool* (HPCTool) para automaticamente proteger códigos em linguagem de máquina, ficando a cargo do projetista a escolha das técnicas a serem utilizadas. Apesar de ter tomado oito meses de implementação, a construção desta ferramenta foi extremamente importante, visto que a proteção de códigos-fonte é uma tarefa extremamente difícil quando realizada a mão.

Para automatizar a injeção de falhas, foi implementado um injetor de falhas capaz de injetar falhas simples ou múltiplas em qualquer sinal, em qualquer momento da execução e pelo tempo escolhido pelo projetista de forma massiva. Além de injetar e executar o programa escolhido pelo usuário, o programa gera automaticamente a coleta de resultados, compara os resultados obtidos com os esperados (corretos) e faz uma classificação das falhas.

Os resultados obtidos, a ferramenta HPCTool e o injetor de falhas, juntamente com outros aplicativos de menor expressão (classificador de falhas com relação a efeito, classificador de desvio de fluxo de programa, mapeador de falhas para

microprocessadores com *pipeline*, dentre outros), formam uma base sólida para o desenvolvimento e teste de novas técnicas em *software*, *hardware* e, principalmente, híbridas. Além disso, os resultados obtidos e a classificação de falhas oferecem uma ótima base para comparação de resultados.

4. Objetivo

O principal objetivo deste trabalho é estender o estudo realizado durante a tese de mestrado do candidato e desenvolver uma técnica híbrida de tolerância a falhas para microprocessadores capaz de detectar até 100% das falhas, ficando a cargo do projetista a melhor relação de detecção por custo. O desenvolvimento prevê a combinação de técnicas puramente em *software* e a adaptação das mesmas para um módulo não-intrusivo desenvolvido em *hardware*.

O teste da técnica híbrida, bem como a comparação com as outras técnicas, envolverá pelo menos duas aplicações de teste, sendo uma *dataflow* e outra *controlflow*. Entretanto, o ideal é que sejam implementadas todas as aplicações de um *benchmark* utilizado na indústria. Como componentes de processamento, serão utilizados os microprocessadores miniMIPS e PowerPC 405 ou superior, sendo o primeiro largamente utilizado na literatura e nos grupos de pesquisa da UFRGS e o segundo amplamente utilizado na indústria de sistemas embarcados. O processador PowerPC possui uma arquitetura diferente do miniMIPS, com *caches* L1 de memória e dados, *pipeline* de instruções modificado, interfaces para unidades de processamento auxiliar, além de uma *Instruction Set Architecture* (ISA) completamente diferente.

Atualmente, o candidato dispõe das seguintes ferramentas para a realização do trabalho:

- HPCTool (injetor de proteção em códigos em linguagem de máquina);
- Injetor de falhas completo (injeção e coleta de dados);
- Mapeador de falhas para processadores com *pipeline*;

O HPCTool deverá ser modificado para proteger o código de programa com a parte em *software* da técnica hibrida, ao mesmo tempo em que deverá ser estendido para ser compatível com o microprocessador PowerPC. O mapeador de falhas também deverá ser modificado, visto que o PowerPC este possui um *pipeline* de instruções muito diferente do utilizado pelo miniMIPS.

Ao final do processo de implementação, simulação e da campanha de injeção de

falhas, espera-se realizar um estágio no exterior com o fim de realizar uma campanha física de injeção de falhas, ou seja, irradiar ambos os processadores com partículas energizadas. Os grupos de pesquisa TIMA e da Universidade Politécnica de Torino possuem o material necessário para a injeção e histórico de parceria com a UFRGS, através da colaboração de pesquisas e intercambio de alunos.

Ao final do estudo, espera-se ter uma vasta gama de ferramentas para a proteção de códigos de programa e injeção de falhas disponível para todos os grupos de pesquisa da UFRGS e uma técnica híbrida e configurável, implementada e validada, capaz de detectar falhas em microprocessadores. Espera-se, ainda, obter dados aprofundados sobre a aplicação de tais técnicas a microprocessadores voltados para a indústria, com diferentes níveis de *cache*, com múltiplos núcleos de processamento e múltiplas memórias de dados e instruções.

5. Plano de Trabalho

Nesse capítulo são apresentadas as disciplinas cursadas durante o mestrado, para as quais será solicitado pedido de reaproveitamento dos créditos. Também é apresentado o plano de disciplinas a serem cursadas durante o doutorado. Finalmente, um cronograma com todas as atividades a serem realizadas nos quatro anos de doutorado.

5.1. Revalidação de Créditos

O pedido de reaproveitamento de disciplinas possui um total de 26 créditos. As disciplinas cursadas no mestrado são apresentadas na tabela abaixo.

Código	Disciplina	Créditos	Ano/Semestre
CMP410	Atividade Didática I	1	2009/01
CMP117	Arquitetura e Projeto de Sistemas VLSI II	4	2008/02
CMP231	Sistemas Embarcados	4	2008/02
CMP246	Teste e Confiabilidade de Sistemas de Hardware	3	2008/02
CMP401	Trabalho Individual I	2	2008/02
CMP237	Arquitetura e Organização de Processadores	3	2008/01
CMP238	Projeto e Teste de Sistemas VLSI	4	2008/01
CMP182	Redes de Computadores I B	4	2008/01
---	Proficiência em Inglês	---	2008/01

5.2. Créditos a serem realizados

Para completar os 36 créditos necessários para o doutorado, o candidato pretende cursar as seguintes disciplinas:

Código	Disciplina	Créditos	Ano/Semestre
CMP134	Introdução ao Processamento Paralelo e Distribuído	4	2010/01
CMP115	Concepção de Circuitos VLSI	4	2010/01
CMP651	Projeto Avançado de Pesquisa I	2	2010/02
CMP411	Atividade Didática II	1	2010/02

Assim, será cursado um total de 8 créditos em disciplinas e 3 créditos em projeto de pesquisa e atividade didática. Para o doutorado será realizado um total de 37 créditos (26 créditos obtidos no mestrado e mais 11 créditos cursados durante o curso de doutorado).

5.3. Cronograma

ATIVIDADES/SEMESTRE	2010/01	2010/02	2011/01	2011/02	2012/01	2012/02	2013/01	2013/02
Disciplinas								
Revisão Bibliográfica								
Exame de Qualificação								
Elaboração da Arquitetura Híbrida Tolerante a Falhas								
Defesa da Proposta de Tese								
Estágio no Exterior								
Implementação de Arquitetura Proposta								
Escrita da Tese								
Defesa da Tese								

Os grupos mais renomados na área de técnicas de tolerância a falhas para microprocessadores são os grupos do Instituto Tecnológico de Karlsruhe, coordenado pelo professor Jürgen Becker, a Universidade Politécnica de Torino, coordenada pelo professor Massimo Violante, e o laboratório de pesquisa TIMA, coordenado pelo professor Raul Velazco. Estes grupos de pesquisa são os mais indicados para realizar o estágio, visto que ambos possuem instalações para a injeção física de partículas energizadas.

Estes laboratórios de pesquisa possuem parcerias com a UFRGS, facilitando assim o estágio do candidato e trazendo melhores resultados para o curso de doutorado.

6. Experiência de Pesquisa

Durante a Graduação no curso de Engenharia de Computação pela UFRGS, o candidato participou do Programa de Formação de Treinadores Dell, onde participou ativamente do grupo de clusters de alto desempenho. Durante os vinte meses de duração do projeto, o candidato pesquisou e desenvolveu um curso abordando a construção e configuração de um cluster de alto desempenho. Este trabalho foi submetido e aceito no *workshop* WSPPD 2005 (Donassolo et al., 2005).

Durante o último ano da Graduação, o candidato foi aceito para o estágio de um ano numa parceria entre a UFRGS e a *Teschnisch Universität Kaiserslautern* (Universidade Técnica de Kaiserslautern), onde permaneceu um ano sob a tutela do professor Christophe Bobda e pesquisou e desenvolveu uma ferramenta para a configuração automática de sistemas multiprocessados. Como resultado desta experiência, foi submetido e aceito um trabalho para ReCoSoC 2007 (Azambuja et al., 2007).

Após o curso de Graduação, o candidato ingressou no Programa de Pós Graduação em Computação da UFRGS, tendo escolhido a linha de pesquisa de sistemas embarcados. No primeiro ano, o candidato desenvolveu um trabalho sobre a reconfiguração parcial de dispositivos programáveis (*Field Programable Gate Arrays* - FPGA) na presença de falhas. Como resultado, foi obtida uma redução no tempo de reconfiguração do sistema superior a 98% e a possibilidade de manter o sistema operando sem interrupção, mesmo durante a reconfiguração do sistema, fato esse ainda desconhecido pela literatura. Este trabalho foi inicialmente submetido, aceito e apresentado no congresso SBCCI 2008 (Pilotto et al., 2008).

Ainda no primeiro ano, este trabalho foi estendido com a aplicação da técnica a um sistema mais complexo para ser então submetido e aceito ao *workshop* RADECS 2008 (Azambuja et al., 2008).

Durante o segundo ano do mestrado, o trabalho apresentado no SBCCI 2008 e RADECS 2008 foi novamente estendido, com a adição de novas técnicas para melhorar seu desempenho ao se recuperar de um erro de controle. Esta extensão resultou na submissão e aceitação para o congresso IOLTS 2009 (Azambuja et al., 2009).

Além deste trabalho, o candidato adotou uma segunda linha de pesquisa. O seu interesse por microprocessadores o levou buscar um modelo ideal para a injeção de falhas e teste de técnicas de tolerância a falhas. Tendo estudado a fundo o

microprocessador miniMIPS e sua estrutura, o candidato desenvolveu um injetor de falhas, definiu grupos de falhas por local de injeção e efeito e testou diferentes técnicas de tolerância a falhas puramente em software através da criação do HPCTool. O resultado deste trabalho foi submetido e aceito para o congresso LASCAS 2010 (Azambuja et al., LASCAS 2010).

Utilizando-se das ferramentas implementadas, o candidato propôs uma nova técnica puramente em *software* para solução de erros de fluxo de programa e a testou, comparando com as demais apresentadas em (Azambuja et al., LASCAS 2010). O resultado deste estudo foi submetido e aceito para o congresso LATW 2010 (Azambuja et al., LATW 2010).

Para o curso de doutorado, espera-se aprimorar os trabalhos de ambas as linhas de pesquisa. Para a linha de pesquisa de reconfiguração parcial de FPGAs, buscar-se-á o teste físico, com a aplicação de partículas energizadas em parceria com a Universidade da Espanha (). Quanto à linha de pesquisa de proteção de microprocessadores, existe o interesse de criar uma técnica híbrida, para então aplicá-la ao microprocessador miniMIPS e ao processador PowerPC utilizado pela Universidade de Torino para testes físicos de injeção de falhas. Ao final deste trabalho, almeja-se estender os trabalhos realizados, buscando as melhores alternativas para a proteção de sistemas reprogramáveis baseados em FPGAs e para a proteção de sistemas baseados em componentes COTS, oferecendo assim uma vasta gama de possibilidades para os projetistas de sistemas.

Referências

(Baumann et al., 2001) R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. IEEE Transactions on Device and Materials Reliability, 1(1):17–22, March 2001.

(O'Gorman et al., 1996) T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. In IBM Journal of Research and Development, pages 41–49, January 1996.

(Dodd et al., 2003) P. E. Dodd, L. W. Massengill, "Basic Mechanism and Modeling of Single-Event Upset in Digital Microelectronics", IEEE Transactions on Nuclear Science, vol. 50, 2003, pp. 583-602.

(Bolchini et al., 2005) C. Bolchini, A. Miele, F. Salice, and D. Sciuto. A model of soft error effects in generic ip processors. In Proc. 20th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, pages 334–342, 2005.

(Goloubeva et al., 2003) O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante (2003) Soft-error detection using control flow assertions. In: Proceedings of the 18th IEEE international symposium on defect and fault tolerance in VLSI systems—DFT 2003, November 2003, pp 581–588.

(Huang et al., 1984) K. H. Huang, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. IEEE Trans Comput 33:518–528 (Dec).

(Oh et al., 2002) N. Oh, P. P.Shirvani, E. J. McCluskey (2002) Control flow Checking by Software Signatures. IEEE Trans Reliab 51(2):111–112 (Mar).

(Rebaudengo et al., 1999) M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, pages 210–218, 1999.

(Cheynet et al., 2000) Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Sonza Reorda M, Violante M (2000) Experimentally evaluating na automatic approach for generating safety-critical software with respect to transient errors. IEEE Trans Nucl Sci 47(6 part 3): 2231–2236 (Dec).

(Nicolescu et al., 2003) B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results", Proceedings of the Design,Automation and Test Europe Conference and Exhibition, 2003.

(Lu, 1982) D.J. Lu, "Watchdog processors and structural integrity checking," *IEEE Trans. on Computers*, Vol. C-31, Issue 7, July 1982, pp. 681-685.

(Oh) N. Oh, P.P. Shirvani, and E.J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. on Reliability*, Vol. 51, Issue 1, March 2002, pp. 111-122.

(Mcfearin et al., 1995) L.D. Mcfearin and V.S.S. Nair, "Control-flow checking using assertions," *Proc. of IFIP International Working Conference Dependable Computing for Critical Applications (DCCA-05)*, Urbana-Champaign, IL, USA, September 1995.

(Alkhalifa et al., 1999) Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy and J.A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 10, Issue 6, June 1999, pp. 627 – 641.

(Donassolo et al., 2005) J. R. Azambuja, L. S. Kuamoto, B. Donassolo. Aplicação de Curso na Área de Cluster de Alto Desempenho In: Workshop de Processamento Paralelo Distribuído, 2005, Porto Alegre. WSPPD. Porto Alegre: GPPD, 2005. v.1. p.115 - 116

(Azambuja et al., 2007) J. R. Azambuja, T. Haller, C. Bobda. Automatic Generation of Adaptive Multiprocessor Systems In: Reconfigurable Communication-centric SoCs, 2007, Montpellier. ReCoSoC, 2007.

(Pilotto et al., 2008) C. Pilotto, J. R. Azambuja, F. Kastensmidt. Synchronizing Triple Modular Redundant Designs in Dynamic Partial Reconfiguration Applications In: Symposium on Integrated Circuits and Systems Design, 2008, Gramado. Proceedings of SBCCI, 2008. p.199 – 204.

(Azambuja et al., 2008) J. R. Azambuja, C. Pilotto, F. L. Kastensmidt. Mitigating Soft Errors in SRAM-based FPGAs by Using Large Grain TMR with Selective Partial Reconfiguration In: European Workshop on Radiation Effects on Components and Systems, 2008. Proceedings of The 8th RADECS, 2008.

(Azambuja et al., 2009) J. R. Azambuja, F. Sousa, L. Rosa, F. L. Kastensmidt. Evaluating large grain TMR and selective partial reconfiguration for soft error mitigation in SRAM-based FPGAs In: 15th IEEE International On-Line Testing Symposium, 2009. 15th IEEE IOLTS, 2009. p.101 – 106.

(Azambuja et al., LASCAS 2010) J. R. Azambuja, F. Sousa, L. Rosa, F. L. Kastensmidt. Evaluating the Efficiency of Software-only Techniques to Detect SEU and SET in Microprocessors In: 15th IEEE Latin American Symposium on Circuits and Systems, 2010, Foz do Iguacu. 1st IEEE LASCAS, 2010.

(Azambuja et al., LATW 2010) J. R. Azambuja, F. Sousa, L. Rosa, F. L. Kastensmidt. The Limitations of Software Signature and Basic Block Sizing in Soft Error Fault Coverage In: 11th IEEE Latin-American Test Workshop, 2010, Punta del Este. 11th IEEE LATW, 2010.