HAL
open science

Analysing and Supporting the Reliability Decision-making Process in Computing Systems with a Reliability Evaluation Framework

Maha Kooli

To cite this version:

Maha Kooli. Analysing and Supporting the Reliability Decision-making Process in Computing Systems with a Reliability Evaluation Framework. Micro and nanotechnologies/Microelectronics. Université Montpellier II, 2016. English. NNT : . tel-01489288

HAL Id: tel-01489288

https://theses.hal.science/tel-01489288v1

Submitted on 14 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Collège Doctoral du Languedoc-Roussillon

THÈSE

Pour obtenir le grade de Docteur

Délivré par l'**Université de Montpellier**

Préparée au sein de l'école doctorale **I2S**
Et de l'unité de recherche **LIRMM-CNRS**

Spécialité: **Systèmes Automatiques et Microélectroniques**

Présentée par **Maha KOOLI**

Analysing and Supporting the Reliability Decision-making Process in Computing Systems with a Reliability Evaluation Framework

Soutenue le 01/12/2016 devant le jury composé de

Régis Leveugle	Professeur	Univ. Grenoble Alpes	Rapporteur
Paolo Prinetto	Professeur	Politecnico di Torino	Rapporteur
Ramon Canal	Professeur Associé	Politècnica de Catalunya	Examinateur
Mohamed Kaâniche	Directeur de recherche	LAAS-CNRS Toulouse	Examinateur, Président
Giorgio Di Natale	Directeur de recherche	LIRMM-CNRS	Directeur de Thèse
Alberto Bosio	MCf-HDR	LIRMM-UM	Co-encadrant de Thèse
Lionel Torres	Professeur	LIRMM-UM	Co-directeur de Thèse
Pascal Benoit	MCf-HDR	LIRMM-UM	Examinateur

UNIVERSITÉ DE MONTPELLIER

LIRMM

cnrs

À Firas
À ma mère Henda, mon père Nejib

Contents

Acknowledgment **4**

Abstract **5**

Resumé **6**

I Overview **7**

Chapter 1 Introduction **8**
1.1 Context and Objectives 8
1.2 CLERECO Project 10
1.3 Contributions 10
1.4 Author Publications 11
1.5 Thesis Structure 12

Chapter 2 Background **13**
2.1 Dependability 13
2.2 Soft Errors 15
2.3 LLVM 16

II Reliability Evaluation via Fault Injection **18**

Introduction **19**

Chapter 3 State of the Art **20**
3.1 Fault Injection 20
3.1.1 Hardware-based Fault Injection 21
3.1.2 Software-based Fault Injection 23
3.1.3 Comparison 25
3.2 Mutation Testing 26
3.2.1 Mutation Testing for Software Reliability 27
3.2.2 Mutation Testing for Hardware Verification 27
3.3 Discussion 28

Chapter 4 Proposed Fault Injection Methods **30**

4.1 Introduction . . . 31
4.2 LLVM-based Fault Injection . . . 31
4.2.1 Overview . . . 31
4.2.2 Fault Models . . . 32
4.2.3 Fault Classification . . . 34
4.2.4 Fault Injection . . . 35
4.3 C-based Fault Injection . . . 42
4.3.1 Overview . . . 42
4.3.2 Fault Classification . . . 43
4.3.3 Fault Injection . . . 43
4.4 Comparison . . . 45
4.5 Validation . . . 46
4.5.1 Simulation-based Fault Injection on Intel Processor . . . 46
4.5.2 FPGA-based Fault Injection on LEON3 Processor . . . 47
4.5.3 Experiments' Setup . . . 47
4.5.4 Results and Discussion . . . 49
4.6 Conclusion . . . 51

Chapter 5 Memory Subsystem Emulator 53
5.1 Introduction . . . 53
5.2 Subsystem Emulator . . . 54
5.2.1 RAM Emulator . . . 56
5.2.2 Cache Emulator . . . 56
5.2.3 Register Files Emulator . . . 58
5.3 Validation through Memory Emulator . . . 60
5.3.1 Emulators' Integration to Fault Injection . . . 60
5.3.2 Experimental Results . . . 62
5.4 Conclusions . . . 66

III Reliability Evaluation through Lifetime Analysis 68

Introduction 69

Chapter 6 State of the Art 70
6.1 Analytical Reliability Evaluation . . . 70
6.1.1 AVF Computation . . . 70
6.1.2 Program Analysis . . . 72
6.2 Cache Design Space Exploration . . . 73
6.2.1 DSE for Performance . . . 74
6.2.2 DSE for Reliability . . . 74
6.3 Conclusion . . . 75

Chapter 7 Analytical Reliability Evaluation 76
7.1 Introduction . . . 77
7.2 Lifetime Analysis . . . 78
7.2.1 Variable Lifetime . . . 78

7.2.2 Instruction Lifetime . . . 79
7.2.3 Fault Classification . . . 81
7.2.4 Lifetime Validation . . . 82
7.3 Fault Analysis . . . 83
7.3.1 Faults in Data . . . 83
7.3.2 Faults in Instructions . . . 85
7.4 Validation . . . 86
7.5 Industrial Case Study . . . 87
7.5.1 Flight Management System . . . 88
7.5.2 Application Analysis . . . 88
7.6 Conclusions . . . 91

Chapter 8 Memory-aware Design Space Exploration 92
8.1 Introduction . . . 92
8.2 Cache Characterizations . . . 93
8.3 Fault Evaluation . . . 94
8.3.1 Faults occurring in Data . . . 94
8.3.2 Faults occurring in Instructions . . . 98
8.4 Conclusion . . . 100

Chapter 9 Conclusion 101
9.1 Summary and Conclusion . . . 101
9.2 Application in CLERECO Project . . . 103
9.3 Future Work . . . 104

Glossary 105

List of Figures 106

List of Tables 107

Appendix 108

Bibliography 110

Acknowledgment

J'adresse mes sincères remerciements à ceux qui ont contribué à l'élaboration de cette thèse. Je tiens tout particulièrement à remercier mon directeur de thèse Giorgio Di Natale, qui m'a encadré et soutenu durant ces trois années. Avec son expérience pronfonde, ses conseils bien pertinentes et sa motivation, il m'a aprit le métier de chercheur. Je remercie également mon co-encadrant de thèse Alberto Bosio pour son aide, ses encouragements et sa confiance. Je remercie aussi Pascal Benoit et Lionel Torres qui ont collaboré dans ce projet. Je les remercies pour leur commentaires et encouragements.

Je tiens à remercier mes amis et collègues que j'ai recontré à Montpellier. J'ai partagé avec eux des moments marquants dans cette expérience. Je remercie également mes chers amis partout dans le monde pour leur soutiens et encouragements.

Un remerciement spécial à mon cher mari Firas qui m'a supporté et encouragé avec son amour, ses mots douces et sa patience. Il m'a aidé également sur le plan professionel avec ses idées originales et son expérience.

Enfin, je suis très contente de remercier ma jolie famille, qui était source de force et d'amour dans tous les moments de mon parcours. Je remercie mes très chers parents pour leur tendresse et confiance. Je remercie mes beaux parents pour leur encouragements. Je remercie ma chère soeur Nihel, mon cher frère Achraf et mon beau-frère Nabil. Je vous aime tous très fort.

Abstract

Reliability has become an important design aspect for computing systems due to the aggressive technology miniaturization and the increase of the non interrupted performance that introduce a large set of failure sources for hardware components. The hardware system can be affected by faults caused by physical manufacturing defects or environmental perturbations such as electromagnetic interference, external radiations, or high-energy neutrons from cosmic rays and alpha particles. For embedded systems and systems used in safety critical fields such as avionic, aerospace and transportation, the presence of these faults can damage their components and can lead to catastrophic failures. Investigating new methods to evaluate the system reliability helps designers to understand the effect of faults on the system, and thus to develop reliable and dependable products. Depending on the design phase of the system, the development of reliability evaluation methods can save the design costs and efforts, and will positively impact product time to-market.

The main objective of this thesis is to develop new techniques to evaluate the reliability of complex computing system running a software. The evaluation targets faults leading to soft errors. These faults can propagate through the different layers composing the full system. They can be masked during this propagation either at the technological or at the architectural level. When a fault reaches the software layer of the system, it can corrupt its data, its instructions or the control flow. These errors may impact the correct software execution by producing erroneous results or preventing the application execution and leading to abnormal termination or application hang.

In this thesis, the reliability of the different software components is analyzed at different levels of the system (depending on the design phase), emphasizing the role that the interaction between hardware and software plays in the overall system. Then, the reliability of the system is evaluated via a flexible, fast and accurate evaluation framework. Finally, the reliability decision-making process in computing systems is comprehensively supported with the developed framework (methodologies and tools).

Resumé

La fiabilité est un aspect important de conception des systèmes informatiques suite à la miniaturisation agressive de la technologie ainsi que le fonctionnement non interrompu qui introduisent un grand nombre de sources de défaillance des composantes matérielles. Le système matériel peut être affecté par des fautes causées par des défauts de fabrication ou de perturbations environnementales telles que les interférences électromagnétiques, les radiations externes ou les neutrons de haute énergie des rayons cosmiques et des particules alpha. Pour les systèmes embarqués et systèmes utilisés dans les domaines critiques pour la sécurité tels que l'avionique, l'aérospatiale et le transport, la présence de ces fautes peut endommager leurs composantes et conduire à des défaillances catastrophiques du systèmes. L'étude de nouvelles méthodes pour évaluer la fiabilité du système permet d'aider les concepteurs à comprendre les effets des fautes sur le système, et donc de développer des produits fiables et sûrs. En fonction de la phase de conception du système, le développement de méthodes d'évaluation de la fiabilité peut réduire les coûts et les efforts de conception. Ainsi, il aura un impact positif sur le temps de mise en marché du produit.

L'objectif principal de cette thèse est de développer de nouvelles techniques pour évaluer la fiabilité globale du système informatique complexe. L'évaluation vise les fautes conduisant à des erreurs dites "soft". Ces fautes peuvent se propager à travers les différentes structures qui composent le système jusqu'à provoquer une défaillance du logiciel. Elles peuvent être masquées lors de cette propagation soit au niveau technologique ou architectural. Quand la faute atteint la partie logicielle du système, elle peut endommager ses données, ses instructions ou le contrôle de flux. Ces erreurs peuvent avoir un impact sur l'exécution correcte du logiciel en produisant des résultats erronés ou empêchant l'exécution de l'application.

Dans cette thèse, la fiabilité des différentes composantes logiciels est analysée à différents niveaux du système (en fonction de la phase de conception), mettant l'accent sur le rôle que l'interaction entre le matériel et le logiciel joue dans le système global. Ensuite, la fiabilité du système est évaluée grâce à des méthodologies d'évaluation flexible, rapide et précise. Enfin, le processus de prise de décision pour la fiabilité des systèmes informatiques est pris en charge avec les méthodes et les outils développés.

Part I

Overview

Chapter 1

Introduction

Contents

1.1 Context and Objectives **8**
1.2 CLERECO Project **10**
1.3 Contributions **10**
1.4 Author Publications **11**
1.5 Thesis Structure **12**

1.1 Context and Objectives

Reliability is a main concern while designing electronic systems that are specially used in safety critical fields such as avionics, aerospace, military, and transportation. Reliability is defined as the probability that a component in the system performs continuously in a predictable way without failures. However, the increase of the non interrupted performance and the progressive miniaturization of microelectronic devices used inside these types of system introduce a large set of failure sources for hardware components. The hardware system can be affected by faults caused by physical manufacturing defects or environmental perturbations such as electromagnetic interference, external radiations, or high-energy neutrons from cosmic rays and alpha particles. These faults can damage the system components and lead to catastrophic failures.

Investigating new methods to evaluate the system reliability helps the designers to understand the effects of faults on the system, and thus to develop reliable and dependable products. Depending on the design phase of the system, the development of reliability evaluation methods can save the design cost and effort, and will positively impact the product time to-market.

For computing system, the development of methods to evaluate the reliability is a challenging task. Nowadays computing systems are based on complex architectures with multiple micro-processors, memories, external devices and several layers of software (operating system, device drives, user applications) running on the system. The hardware and software layers interact in a way that the fault affecting the hardware and leading to soft errors may propagate through the different system layers, as presented in Figure 1.1. The fault propagates through the

different hardware structures composing the full system. However, they can be masked during this propagation either at the technological or at architectural level. When a fault reaches the software layer of the system, it can corrupt its data, its instructions or the control flow. These errors may impact the correct software execution by producing erroneous results or preventing the execution of the application and leading to abnormal termination or application hang. The software stack can play an important role in masking errors, which enables the improvement of the system reliability.

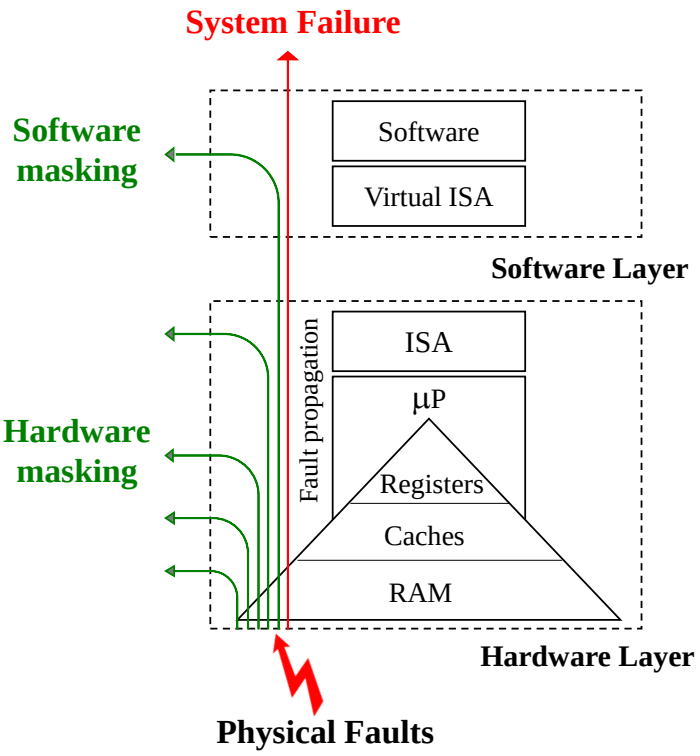

Figure 1.1: System Layers and Fault Propagation

This thesis aims to propose fast, flexible and accurate methods to evaluate complex computing system when affected by faults that manifest in the underlying hardware layer. We consider computing systems the system that are based on microprocessor and that are running software. The work proposed in this thesis is applied to microprocessor with a single core on a chip and that is running a single thread at one time. The evaluation targets faults leading to soft errors. We focus on studying the software layer role to evaluate the system reliability in different design stages of the system. At advanced design stage, the Instruction Set Architecture (ISA) is known and therefore can be used to perform simulations to analyze how faults propagate through the system components. However, at early design stage of the system when the hardware architecture is possibly not fully defined, the ISA might be unknown and cannot be used to study the effect of faults on the system behavior. The accurate reliability evaluation at this stage is a challenging task.

In this thesis, the reliability of the different software components is analyzed at different levels of the system (depending on the design phase), emphasizing the role that the interaction between hardware and software plays in the overall system. Then, the reliability of the

system is evaluated via a flexible, fast, and accurate evaluation framework. Finally, the reliability decision-making process in computing systems is comprehensively supported with the developed framework (methodologies and tools).

1.2 CLERECO Project

This thesis is proposed in the context of the joint FP7 Collaboration European CLERECO Project (Grant No. 611404). CLERECO research project recognizes early accurate reliability evaluation as one of the most important and challenging task throughout the design cycle of the computing systems across all domains. In order to continue harvesting the performance and the functionality offered by technology scaling, current methodologies to evaluate the reliability of the system should be dramatically improved. CLERECO addresses early reliability evaluation with a cross-layer approach considering different computing disciplines, computing system layers and computing market segments to address reliability for the emerging computing continuum. CLERECO methodology considers low-level information such as raw failure rates as well as the entire set of hardware and software components of the system that eventually determine the reliability delivered to the end users. The CLERECO project methodology for early reliability evaluation is comprehensively assessed and validated in advanced designs from different applications provided by the industrial partners for the full stack of hardware and software layers.

This thesis focuses on the tasks of the Work-Package number 4 of the project. The main objective of this work-package is to evaluate the impact that hardware failures may have on the overall system reliability regarding the software stack executed on the platform. In fact, this requires understanding the sensitivity of the software activities to errors generated by hardware faults, and therefore to understand its intrinsic capability of masking these errors. It is important to emphasize that CLERECO focuses only on effects of hardware faults in the software stack. Software reliability engineering including software-testing techniques that target the detection of the software design bugs is not considered in this project.

1.3 Contributions

The main contributions of this thesis are:

1. Developing new fault injection approaches that aim to evaluate the effect of hardware faults at different level of the system. These approaches allow to evaluate the reliability at different design stage of the system. Depending on the evaluation level, the approaches offer gain in the simulation time and the hardware cost.

2. Proposing a memory subsystem emulator that is able to emulate, at software-level, the behavior of the RAM, the caches, and the register files without the presence of the fully defined hardware. The structure of each unit is designed in a way to be as close as possible to the real system behavior, and as generic as possible to support different characteristics of different microprocessors.

3. Introducing a novel analytical methodology that analyzes the effect of hardware faults on the system behavior. The proposed method is accurate, fast, and flexible. It allows

to evaluate the reliability of the different system components. Furthermore, it is used to study a memory-aware design space exploration.

1.4 Author Publications

The contributions and results of this thesis have been published and presented in international conferences, journals and workshops.

- A. Vallero, S. Tselonis, N. Foutris, M. Kalioraki, M. Kooli, A. Savino, G. Politano, A. Bosio, G. Di Natale, D. Gizopoulos, S. Di Carlo. Cross-layer reliability evaluation, moving from the hardware architecture to the system level: A CLERECO EU project overview. In Microprocessors and Microsystems - Embedded Hardware Design 39(8): 1204-1214, June 2015.

- Maha Kooli, Giorgio Di Natale, Alberto Bosio. Cache Design Space Exploration in Computing Systems for Reliability Improvements. Submitted to IEEE Transactions on Dependable and Secure Computing.

- M. Kooli, G. Di Natale, A. Bosio, P. Benoit, L. Torres. Computing Reliability: On the Differences between Software Testing and Software Fault Injection Techniques. Submitted to Microprocessors and Microsystems.

- A. Vallero, A. Savino, G. M. M. Politano, S. Di Carlo, A. Chatzidimitriou, S. Tselonis, M. Kaliorakis, D. Gizopoulos, M. R. Villanueva, R. Canal, A. Gonzalez, M. Kooli, A. Bosio and G. Di Natale. Cross-Layer System Reliability Assessment Against Hardware Faults. In IEEE International Test Conference (ITC) 2016.

- M. Kooli, G. D. Natale, and A. Bosio. Cache-aware reliability evaluation through LLVM-based analysis and fault injection. In 22nd IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), Sant Feliu de Guixols, Catalunya, Spain, July 4-6, 2016.

- M. Kooli, F. Kaddachi, G. D. Natale, and A. Bosio. Cache- and register-aware system reliability evaluation based on data lifetime analysis. In 34th IEEE VLSI Test Symposium (VTS), Las Vegas, NV, USA, April 25-27, 2016, pp. 1–6.

- F. Kaddachi, M. Kooli, G. Di Natale, A. Bosio, M. Ebrahimi, M. Tahoori. System-level Reliability Evaluation through Cache-aware Software-based Fault Injection. In Proceedings of the 19th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Kosice, Slovakia, April 20-22, 2016.

- M. Kooli, A. Bosio, P. Benoit, and L. Torres. Software testing and software fault injection. In 10th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS), Napoli, Italy, April 21-23, 2015.

- M. Kooli, P. Benoit, G. Di Natale, L. Torres, and V. Sieh. Fault injection tools based on virtual machines. In 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), Montpellier, France, May 26-28, 2014.

- M. Kooli and G. Di Natale. A survey on simulation-based fault injection tools for complex systems. In Proceedings of the 9th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS), Santorini, Greece, May 6-8, 2014.

- A. Vallero, A. Savino, G. Politano, S. Di Carlo, A. Chatzidimitriou, S. Tselonis, M. Kaliorakis, D. Gizopoulos, M. Riera, R. Canal, A. Gonzalez, M. Kooli, A. Bosio, and G. Di Natale. Early component-based system reliability analysis for approximate computing systems. In 2nd Workshop On Approximate Computing (WAPCO), Prague, January, 18-20, 2016.

- M. Kooli, G. Di Natale. Evaluation of System Reliability at High Level. In GDR SoC-SiP, 2014.

1.5 Thesis Structure

The rest of this thesis is structured as follows.

Part I proceeds by Chapter 2, which presents preliminary definitions and backgrounds relevant for the thesis.

Part II is related to the reliability evaluation via fault injection. Chapter 3 discusses existing works in the literature based on fault injection and mutation testing techniques. Chapter 4 proposes new fault injection environments that are virtual-level and source-code-level. The validation of the proposed methods is discussed through comparisons to hardware-based fault injections. Chapter 5 introduces a novel memory subsystem emulator and provides a validation of the proposed fault injection environments in term of accuracy and simulation time.

Part III is related to the reliability evaluation via lifetime analysis. Chapter 6 explores the state-of-the-art of analytical reliability evaluation techniques and cache design space exploration. Chapter 7 proposes new analytical reliability evaluation technique and validates its accuracy through comparisons to hardware-based fault injections. Chapter 8 provides a memory-aware design space exploration using the proposed analytical approach.

Chapter 9 concludes the thesis, presents the application of the proposed methodologies in the CLERECO project, and discusses different perspectives.

Chapter 2

Background

Contents

2.1 Dependability **13**
2.2 Soft Errors **15**
2.3 LLVM **16**

In this chapter, we provide background and technical definitions related to this thesis, and we present the virtual instruction set architecture used in our works.

2.1 Dependability

Dependability [1] is a key decision factor in today's global business environment. It is first introduced as a global concept that subsumes the usual attributes of reliability, availability, safety, integrity, maintainability, and security [2]. It represents the ability to avoid service failures that can happen to the system more frequently and severely than acceptable. In other words, dependability is the ability to deliver products that can be trusted. Fig. 2.1 presents the schema of the complete taxonomy of dependability. It could also be defined as a measure of the system availability, reliability, and maintainability. Dependability can be determined and measured through three elements: attributes, threats, and means.

The attributes represent the way to measure the dependability. They are represented by the following elements:

- *Availability:* It represents the probability that the system is available to operate correctly at a given instant of time. A highly available system is a system that will most likely be working at a given instant in time.

- *Reliability:* It represents the probability that a component in the system performs continuously in a predictable way without failure, *i.e.*, for a prescribed time and under exact environmental conditions. A highly reliable system is a system that most likely continues to work without interruption during relatively long period of time.

  Reliability is defined over an interval of time rather than an instant in time, which is the case for availability.

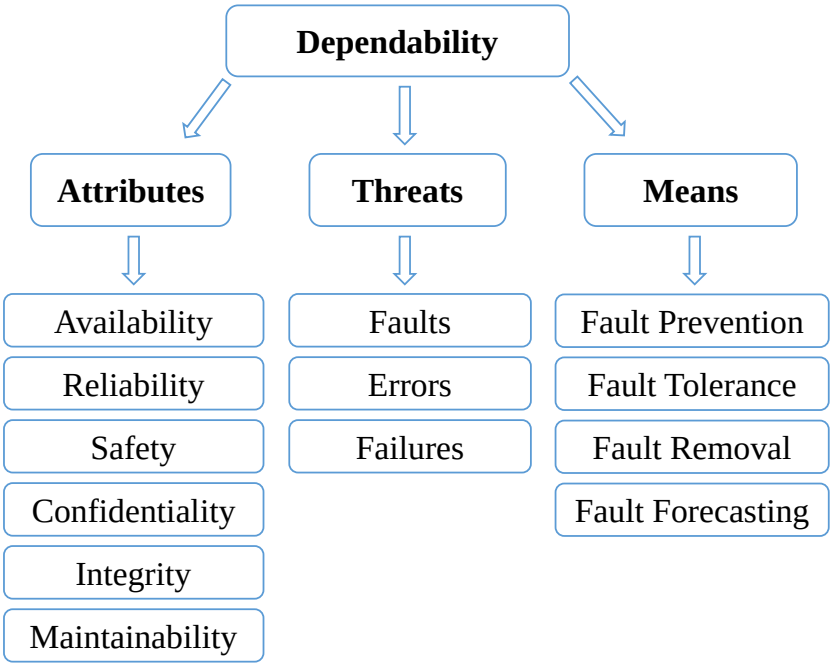

Figure 2.1: The Dependability Tree [1]

- *Safety:* It represents the probability that the system either operates correctly or interrupts its functions in a way that nothing catastrophic happens to the user or the environment.
- *Maintainability:* It represents the measure of how easily the system can be fixed in case of failure. A highly maintainable system is a system that shows high degree of availability when failures can be detected and repaired automatically.
- *Confidentiality:* It represents the absence of unauthorized disclosure of information.
- *Integrity:* It represents the absence of improper system and data modifications.
- *Security:* It is a composite of confidentiality, integrity, and authenticity attributes.

Regarding these definitions, most of the attributes are subjective and could not be measured. Only availability and reliability are quantifiable by direct measurements. For example, reliability can be measured as the failure over time, while safety is a subjective evaluation that needs judgmental information in order to have the required level of confidence. This is the reason why techniques developed to evaluate the dependability represent in reality the techniques developed to measure the availability and the reliability.

The threats represent the effects that can touch the dependability of the system. They are undesired and unexpected events possible leading to the non-dependability of the system. They represent the following elements:

- *Fault:* It is physical defect or imperfection that happens in the hardware, software or human components of the system. A fault can be the cause of specification mistakes, implementation mistakes, external disturbances, physical hardware component defects or

misuses. A fault can be permanent, intermittent or transient depending on its persistence time.

- *Error:* It is the deviation of the system internal state from the right service state. It can also be defined as contradiction between the experimental (or observed) behavior, and the theoretically (or expected) behavior of a component in the system. Errors can be observed during the test session or using special mechanisms such as System Error Detection Mechanism (*e.g.*, hardware exceptions handling, software checks).

- *Failure:* It is the result of the delivered service deviation from the correct service. A failure is also defined as an instance in time when the resulting behavior of the system does not correspond to the required specification.

The relationship between faults, errors, and failures is represented by "the chain of threats" [1]. In fact, an error is the result of the activation of a fault, and a failure occurs when an error propagates to the service interface and becomes the reason of incorrect service. The system component failure may be the cause of permanent or transient fault in the system containing this component.

The means are the techniques and methods that are able to increase the dependability of the system. They include the following techniques:

- *Fault Prevention:* It deals with avoiding the fault to occur on the system. It could be achieved by integrating development methodologies and good implementation techniques, such as design rules, modularization, use of hardened hardware, use of strongly-typed programming languages.

- *Fault Tolerance:* It means to prevent failures when faults are present in the system. The system remains working in an expected manner, according to its specifications in the presence of faults. Several studies in the literature present different methods and techniques for fault tolerance, such as control flow checking [3], Algorithm Based Fault Tolerance (ABFT) [4] and Redundancy (space redundancy, time redundancy, information redundancy).

- *Fault Removal:* It means to decrease the severity and the number of faults in the system. During the development phase, it consists on three steps: the verification, the diagnosis, and the correction. During the use of the system, fault removal is corrective or preventive maintenance. Corrective maintenance aims to remove faults that produced one or more errors, while preventive maintenance aims to uncover and remove faults before causing errors.

- *Fault Forecasting:* It is conducted by performing an evaluation of the system behavior with respect to fault occurrence or activation. The evaluation has two aspects: qualitative evaluation and quantitative evaluation.

2.2 Soft Errors

The size and complexity of the systems used in the safety-critical field lead to significant increase in the number of transient, permanent and intermittent faults in processors. Permanent

faults are defined as faults that exist indefinitely in an element if no corrective action is taken. Thus, they model permanent hardware failures such as an ALU that stops working or a cache line that has a stuck-at fault. Transient faults are defined as faults that can appear and disappear within a given period of time during computation. They are caused by events such as cosmic rays, alpha particle strikes or marginal circuit operation caused by noise for instance. Intermittent faults are malfunctions of a device or system that occur at intervals, usually irregular, in a device or system that functions normally at other times.

In electronics and computing systems, a soft error is the type of error where the signal or the data is altered to a wrong value. Soft errors can be caused by electromagnetic interference, by external radiations, such as high-energy neutrons from cosmic rays and alpha particles. For example, when a particle strikes a sensitive region of the memory cell, the charge that accumulates could exceed the minimum charge needed to flip the value stored in the cell, thus resulting in a soft error.

Soft errors are reversible and can be recovered by a reset, a power cycle or simply a rewrite of the information, which is not the case for hard errors [5] that are caused by irreversible physical changes in a chip during the manufacturing process or by aging phenomena. Soft errors can propagate through the different hardware structures composing the full system, or they can be masked during this propagation either at the technological or at architectural level [6] [7] [8]. If not masked, they can reach the software layer of the system. In computer-based systems, when a soft error reaches the software layer, it can corrupt either the data, the instructions or the control flow of the program. These errors may impact the correct software execution by producing erroneous results or preventing the execution of the application leading to abnormal termination or application hang.

Soft errors arise from Single Event Upset (SEU), Multiple Bit Upset (MBU) and Multi-Cell Upset (MCU) due to ionizing particle (ions, electrons, photons...) striking a sensitive node in a micro-electronic device. The non interrupted performance and the progressive miniaturization of microelectronic devices in the systems make them more susceptible to be affected by such errors [9].

2.3 LLVM

The main goal of this thesis is to develop methods to evaluate the reliability at early design stage of the system. Thus, we need to investigate tools that analyze the software independently from the target hardware architecture. At early design stage, the target Instruction Set Architecture (ISA), which defines the interface between the hardware and the software, might be unknown and therefore cannot be used to evaluate the reliability of the system. The solution is to resort to virtualization techniques that allow to abstract the ISA. The concept of software virtualization ensures the possibility to make analysis without previous knowladge of the actual ISA.

Different alternatives of virtualization environment implementing Virtual Instruction Set Architecture (VISA) are available in the literature [10] [11] [12] [13]. Java is widely used in web-based applications. It has the disadvantage of not being really suitable for both HPC and embedded applications. The JVM is restricted to the Java programming language, thus limiting the spectrum of software that can be analyzed. The .NET framework consists of a virtual machine able to run code in Common Languages Infrastructure (CLI), an object-oriented VISA

that is the lowest level of the framework. LLVM (Low Level Virtual Machine) [13] is a compiler framework that uses virtualization with virtual instruction sets to perform complex analysis of software applications on different architectures. LLVM provides high-level information to compiler transformations at compile-time, link-time, run-time, and idle-time between runs. LLVM uses the Intermediate Representation (IR) as a form to represent code in the compiler. It symbolizes the most important aspect of the framework, because it is designed to host mid-level analysis and transformations found in the optimizer section of the compiler. The LLVM IR is independent from the source language and the target machine.

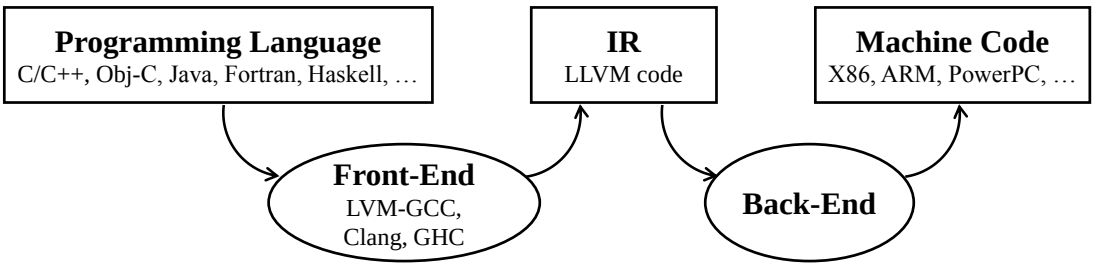

Figure 2.2: The LLVM Compiler

In this thesis, most of our contributions are based on LLVM. The advantage is the support of different programming languages, and different machine codes. In figure 2.2, we provide the LLVM compiler. As source code front ends, the LLVM compiler supports several programming languages, such as C, C++, Objective-C, Fortran, Python. As machine code back-ends, it supports many instruction set, such as ARM, MIPS, PowerPC, SPARC, x86/x86-64. This means that LLVM permits to define an abstraction layer to make the information obtained at software level and the information obtained at hardware level, compatible and easily exchangeable. In addition to the full tool chain required for software design (*e.g.*, compiler, optimizer), LLVM provides a set of additional tools explicitly devoted to perform investigation of different software properties.

Part II

Reliability Evaluation via Fault Injection

Introduction

A powerful and useful technique to evaluate the system reliability is the fault injection. It allows to assess the behavior of the system when affected by faults. The fault injection can be performed at different levels of the system. Hardware-based fault injection techniques directly inject faults in the internal processor components using manufactured processor prototype, simulation of the processor architecture or implementation of the processor on FPGA board. These techniques provide accurate evaluation of the system reliability. However, they can not be used at early design stage of the system. Software-based fault injections provide cheaper solution to evaluate the system reliability by modeling the fault injection at abstract level. Nevertheless, they are inaccurate and can not target faults in different system components.

In this part, we propose new fault injection environments that bridge the gap between software- and hardware-based fault injections. Similar to software-based fault injection, the proposed methodologies are fast, cheap and usable at early design stage. Furthermore, they are accurate and target memory system components similar to hardware-based fault injection. We introduce a virtual-level fault injection based on the LLVM virtual ISA, and a source-code-level fault injection based on the C programming language. The proposed methods target faults in both the data and instructions of the software application, and study the effect of these faults on the output of the system that is running this software. In order to target faults occurring in the system components such as the RAM and the caches, we propose a memory subsystem emulator that emulates, at software-level, the behavior of the RAM, the caches, and the register files without requiring fully defined hardware.

In order to validate the accuracy of the proposed approaches, we perform comparisons to hardware-based fault injections: simulation-based fault injection on the Intel©8086 processor and FPGA-based fault injection on the LEON3 processor. The experimental results prove the accuracy of our approaches with gain in the execution time.

The reminder of this part is structured as follows. Chapter 3 discusses the state-of-the-art on different categories of fault injection techniques, and mutation testing. Chapter 4 introduces the proposed fault injection environments. Chapter 5 presents the implementation of the memory subsystem emulator, explains the integration of this emulator within the proposed fault injection techniques, and provides experimental results to validate the proposed methods.

Chapter 3

State of the Art

Contents

3.1 Fault Injection **20**
3.1.1 Hardware-based Fault Injection 21
3.1.2 Software-based Fault Injection 23
3.1.3 Comparison 25
3.2 Mutation Testing **26**
3.2.1 Mutation Testing for Software Reliability 27
3.2.2 Mutation Testing for Hardware Verification 27
3.3 Discussion **28**

In this chapter, we discuss the state-of-the-art on the reliability evaluation methodologies based on fault simulation. The fault injection technique is a reliability evaluation technique that targets hardware faults (*e.g.*, soft errors and hard errors). Mutation testing is also a reliability evaluation technique but targets software errors occurring during the design or the implementation stage. It is also used for hardware verification.

Section 3.1 introduces the different categories of the fault injection existing in the literature. Section 3.2 presents mutation testing with its different applications on software and hardware reliability. Finally, Section 3.3 provides comparison between the two methodologies and explains their limitations regarding the goal of this thesis.

3.1 Fault Injection

Fault injection is a powerful and useful technique to evaluate the reliability of the systems under faults [14], and in particular to assess the behavior of the system when a fault affects a part of it. Fault injection is based on the realization of controlled experiments in order to evaluate the behavior of the computing systems in presence of faults. This technique can speed up the occurrence and the propagation of faults in the system to observe their effects on the system.

Fault injection is widely studied in the three last decades. Two main classifications of fault injection techniques exist in the literature: hardware-based techniques that directly inject faults

in the target hardware, and software-based techniques that model the fault injection at abstract level.

3.1.1 Hardware-based Fault Injection

Hardware-based fault injection allows to inject physical faults (*e.g.*, bit flip, stuck at fault) in the target system. It uses either manufactured processor prototype, or simulation of the processor architecture, or an implementation of the processor on FPGA board. The main advantage of these techniques is the fact that they are performed in realistic conditions which enables to provide accurate results [14].

3.1.1.1 Physical Fault Injection

The physical fault injection techniques are based on the application of external perturbations on the circuits under evaluation to assess their reliability [5]. Particle radiations, laser beams or pin forcing can be used to create realistic faults:

- **Radiation Methods:** These methods test the device in its real environment and expose it to the particle radiation [15] [16]. The device is put in realistic conditions, but it is quite hard to control where the fault is injected and to set the appropriate energy of particles up.

  FIST [17] is a fault injection tool that uses heavy ion radiation to create transient faults at random locations inside the chip and generates single or multiple bit-flips. FIST directly injects faults inside a chip, which cannot be done with pin-level injections.

- **Laser Methods:** The laser beams generate a photon-material interaction instead of a particle-material interaction. These methods can better control where the fault is injected. Radiation-based and laser-based methods provide correlation in term of fault evaluation results [18].

  Several studies in the literature use this method to inject faults. In [19], the authors use laser based equipment to inject faults at circuit level. The target faults are random spot defects that may result in discrete faults such as line breaks and short circuits. These faults could therefore contribute significantly to yield losses in stable fabrication lines of VLSI integrated circuits. In [20], the authors describe the use of a pulsed laser for studying radiation-induced single-event transients in integrated circuits. They present the basic failure mechanisms and the fundamentals of the laser testing method, and they illustrate the benefits of using a pulsed laser for studying single-event transients.

- **Pin Forcing:** The values at input/output pins of the device are directly modified to cause the same effect of radiation and laser methods [21]. This solution is cheaper, but it is only applied on simple circuits.

  RIFLE [22] is a pin-level fault injection system for dependability validation. It can be adapted to a wide range of systems, where faults are mainly injected in the processor pins. Different types of faults can be injected. The tool is able to detect whether the injected fault has produced an error or not without requiring of feedback circuits.

MESSALINE [23] is a pin-level fault forcing system. It uses both active probes and sockets to conduct pin-level fault injection. It can inject stuck-at, open, bridging, and complex logical faults. It can also control the lifetime of the fault and its frequency.

Physical fault injection techniques can access locations that are not easy to access by other techniques. They provide accurate evaluation of the system reliability. However they introduce a high risk to damage the system under test and thus a high hardware cost.

3.1.1.2 Simulation-based Fault Injection

The simulation-based fault injections do not operate on the physical device under evaluation, but they target a model of the hardware described using a simulation language, such as VHDL [24] [25] or Verilog [26]. They inject faults in the VHDL models either at run-time or at compile-time. Compared to the physical fault-injections, the simulation-based fault-injection techniques are cheaper in term of set-ups and involved hardware, and can better control the fault location. However, their application creates a computational overhead depending on the complexity of the system design[27].

VERIFY [28] is a VHDL-based fault injection technique that efficiently evaluates the effect of faults on the system reliability. The tool describes the behavior of hardware components in case of faults by extending the VHDL language with fault injection signals together with their rate of occurrence.

LIFTING [26] [29] is a simulator able to perform fault simulations for stuck-at faults and SEU on digital circuits described in Verilog. It is based on an event-driven logic simulation engine to perform the fault injection. LIFTING is different from other fault injection tools, because it provides many features for the analysis of the fault simulation results, which is meaningful for research purposes. In addition, it enables to simulate faults in complex microprocessor-based systems by describing the hardware system components (including the memory). It models the software stored in the memory, and injects faults in all elements of the hardware model.

3.1.1.3 FPGA-based Fault Injection

The FPGA-based fault injections implement the device under evaluation on an FPGA board and perform fault injection campaigns on different system components. These techniques can precisely control where the fault is injected. FPGA-based fault injection methods have recently become more popular since they provide high speed in fault injection experiments compared to simulation- and physical-based fault injection [30]. The fault injection process applies one of the following mechanisms:

- **Reconfiguration Mechanism:** The bits of the FPGA board are reconfigured to inject the fault in the specified location [31] [32] [33] [34]. The fault injection takes place either at run-time or at compile-time. However, the reconfiguration process creates a time overhead [35].

- **Instrumentation Mechanism:** Additional circuit elements in the different processor components are built to inject the fault in the target location, called *Saboteurs* [36] [37] [38]. The activation of these elements generates the required fault. The instrumentation mechanism is therefore faster than the reconfiguration mechanism.

SCFIT [27] [39] is an FPGA-based fault injection technique that is flexible and easy-to-develop. This technique utilizes debugging facilities of Altera [40] FPGAs in order to inject SEU and MBU fault models in both flip-flops and memory units. As this technique uses FPGA built-in facilities, it imposes negligible performance and area overheads on the system.

3.1.2 Software-based Fault Injection

The software-based fault injections provide cheaper solution to evaluate the system reliability. They involve modifications in the software state of the system under analysis in order to model hardware faults. The first drawback of these methods is the inability to inject faults into some locations that are not accessible by the software, such as faults occurring in the logic-level. However, these techniques are able to target applications and operating systems, which are not easy to simulate using the hardware-based fault injection. Software-based techniques model the fault injection at abstract levels, such as the operating system, the source code or the virtual machine. The fault is modeled either at compile-time or at run-time. At compile-time, the fault is injected by modifying the software executed by the system (*e.g.*, the source code). At run-time, the fault is injected during the simulation or the execution of the software.

3.1.2.1 Operating-System-level Fault Injection

The operating system presents complex and critical part of the software stack in computer-based systems. Fault injection techniques performed at the operating system level target faults in the system calls and the kernel data structure.

FERRARI [41] is an operating system fault injection technique. It uses traps and system calls in order to modify the execution state of the target application in UNIX system. The fault injection mechanism is based on the interaction between two parallel processes: the fault injection and the target program process. The first process allows to configure the fault injection environment by specifying the target program, the fault number and the fault location. Then, the *fork* system call is used to create the target program process and the *wait* system call is used to wait until the target program completes. In order to perform memory-level faults, the task memory image of the target program is modified before the execution. The target program process allows to execute the *ptrace* system call and the *execv* system call to start target program execution.

XCEPTION [42] is an operating system fault injection technique that uses advanced debugging and performance monitoring features of the actual processor in order to perform fault injections as realistically as possible. It targets the major internal processor units: the *Integer Unit*, the *Floating Point Unit*, the *Memory Management Unit*, the *Internal Data Bus*, the *Internal Address Bus*, the *General Purpose Registers* and the *Branch Processing Unit*.

3.1.2.2 Virtual-level Fault Injection

Fault injections based on virtual machine target hardware faults at the software level, and allow observing complex computer-based systems with operating system and user applications. They allow to simulate the computer system without having the real hardware thanks to the use of virtual machine.

FAUMachine [43] [44] is a virtual machine that permits to install a full operating system, such as Linux, WindowsOpenBSD or Mac OS X, and run them as an independent computers.

As microprocessors, it supports CPUs 80286, 80386, pentium, pentium II and AMD64. FAUMachine is similar in many aspect to standard virtual machines like QEMU [45] or VirtualBox [46]. The property that distinguishes FAUMachine from the other virtual machines is its ability to support fault injection functionality. The tool targets faults in memory such as transient bit flips, permanent struck-at faults, and permanent coupling faults; in disk CD/DVD drive such as transient or permanent block faults, and transient or permanent whole disk faults; in network such as transient, intermittent, and permanent send or receive faults. Compared to existing fault injection tools, FAUMachine is able to inject faults and observe the whole operating system and software applications. Thanks to the concept of virtual system, this tool provides a high simulation speed for both complex hardware and software systems [44].

Based on the Low Level Virtual Machine (LLVM) [13] [47] [48], the compiler framework that uses virtual instruction sets to perform complex analysis of software applications on different hardware architectures presented in Section 2.3 of Chapter 2, two fault injection techniques are proposed in the literature: LLFI [49] [50] and KULFI [51]. Injecting faults at the virtual instruction set architecture of the application allows to be completely independent from the source code language and the target hardware architecture.

LLFI allows to inject faults into the LLVM intermediate level of the application source code. It performs fault injections at specific program points and data types. The tool is typically used to map fault outcomes back to the source code, and understand the relationship between program characteristics and the various types of fault outcomes. The goal behind LLFI is to build source level heuristics permitting to identify optimal locations for high coverage detections of faults. Regarding the considered fault model, LLFI targets transient hardware faults affecting the processor's computation units. However, the tool does not consider faults in the memory components, the control logic of the processor and the instructions.

KULFI allows to inject random single bit flips into the instructions as well as in the data and the address registers. It permits to simulate faults occurring within the CPU state elements, providing a finer control over the fault injection process compared to LLFI. It enables the user to define some relevant options, such as the fault-occurrence probability, the byte position in which the fault will be injected, and the possibility to choose whether the fault is injected into the pointer register or the data register. KULFI considers the injection of both dynamic faults and static faults. Dynamic faults represent the transient faults and are injected randomly in time during program execution. Static faults represent the permanent faults and are injected randomly before the program execution.

3.1.2.3 Code-level Fault Injection

Fault injection techniques that perform faults at code level require the presence of the source code of the software under test. The fault injection can be either in the original source code or in the binary code of the application. The fault injection is performed by changing the code to model hardware faults.

Jaca [52] [53] is a fault injection tool that is able to inject fault in object-oriented systems. Additional Java classes are defined to generate faults in the attributes, the method return values and their parameters. The use of Javassist toolkit allows to be as independent as possible from the source code and to easily manipulate the byte-code of the loaded classes at run-time. However, this technique does not target faults affecting the instructions.

J-SWFIT [54] is a byte-code-level fault-injection technique. It directly targets the byte-

codes of the compiled Java applications. The architecture of the tool was proposed at abstract level in order to be easily understood and extended. J-SWFIT is based on a set of predefined Java operators. It consists of analyzing the byte codes of compiled Java files, finding locations where specific faults can exist and can be injected each one independently. J-SWFIT allows to evaluate the system behavior in the presence of each fault.

3.1.3 Comparison

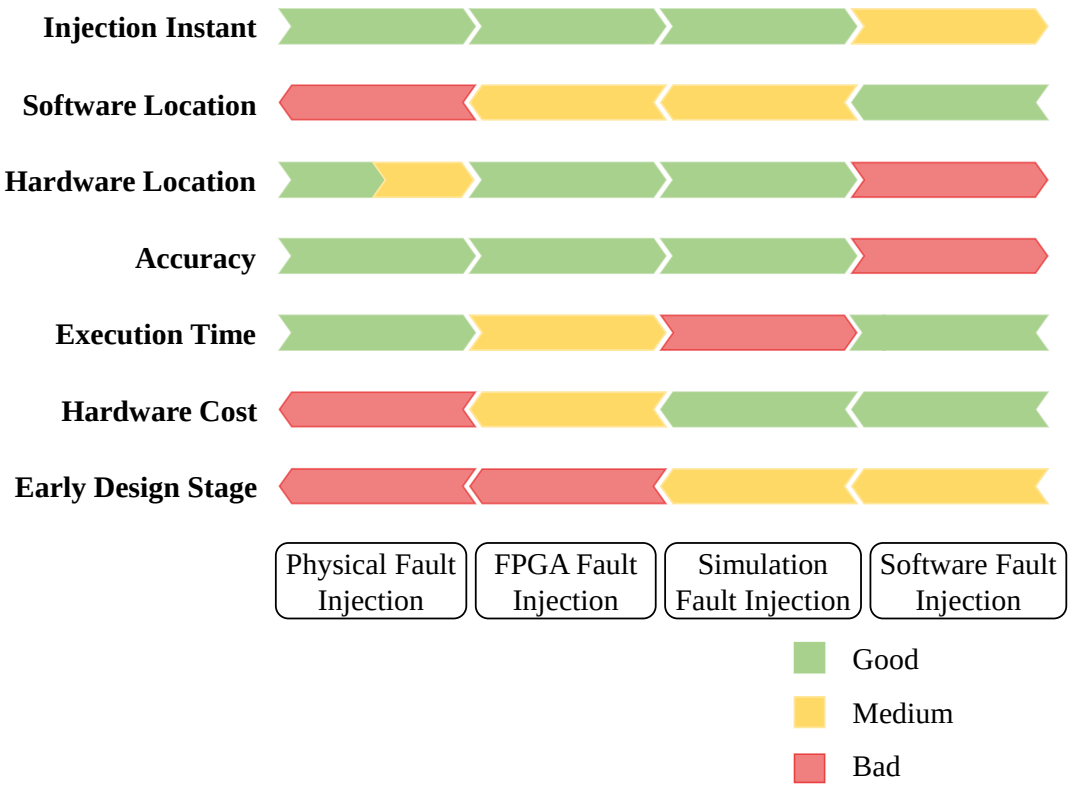

Figure 3.1: Comparison between Existing Fault Injection Methodologies

Fault injection is a powerful method to evaluate the system reliability. A comparison between the hardware- and the software-based fault injections reveals many advantages and drawbacks for both of them [55]. Based on the following criteria, we present in Figure 3.1 a comparison between the previously discussed methods:

- **Hardware Cost:** To perform fault evaluation, the hardware-based techniques require extra special hardware with different cost levels depending on the adopted method. The physical techniques are the most expensive because they present a serious risk to damage the hardware under test. However, the simulation-based and the software-based fault injection techniques do not make use of additional hardware since they consider the source code either of the simulated processor or of the software application. Thus they are cheaper in term of hardware cost.

- **Execution Time:** The software-based fault injection techniques offer an important gain in the time of the fault evaluation experiments. For the hardware-based fault injections, the physical techniques require fast fault evaluation. However, the simulation-based techniques require long simulation time. Furthermore, hardware-based techniques have less computational overhead, because they do not insert modifications at software level.

- **Fault Evaluation Accuracy:** In the literature, the main disadvantage of the software-based fault injection techniques is the lack of accuracy in terms of fault evaluation results [56]. The hardware-based techniques are considered more accurate since they perform fault injections in realistic conditions.

- **Fault Location:** The software-based methods offer better visibility of the software components, which allows better understanding of the fault effects on the software level. They enable to know exactly which variables or instructions are the most vulnerable to the injected faults. This enhances the development of reliability improvement techniques such as Software Implemented Hardware Fault Tolerance (SIHFT) [57] [4] [58]. However, the hardware-based methods offer better visibility of the internal hardware components, which allows a design space exploration of the target system by changing the characteristics of these components in order to observe the impact on the reliability. Design space exploration is a powerful methodology that helps designers to develop reliable and dependable systems.

- **Injection Instant:** For different hardware-based fault injection techniques, the fault injection instant is controlled in term of clock cycles. Thus, they provide a good precision and help to know which interval of time during the system execution is the most vulnerable to faults.

- **Early Design Stage:** The physical and the FPGA-based fault injections require fully designed, implemented and constructed hardware architecture. Thus, they can not be used at early design stage of the system. However, the simulation- and software-based fault injections, which simulate the processor under evaluation, require the definition of the ISA of the processor without needing the fully implementation and construction of the hardware.

3.2 Mutation Testing

Mutation testing is a technique for software quality improvement used during the software development phase. Its goal is to improve the ability of test cases to detect faults. While mutation testing was first introduced to target software errors (*i.g.*, bugs, design errors), lately it has been applied on hardware verification to target errors introduced during the hardware design. However, mutation testing has not been applied to target soft errors. Since in this thesis we aim to have an evaluation of the system reliability at software level, we present the technique of mutation testing in order to study the possibility to use it to target soft errors.

3.2.1 Mutation Testing for Software Reliability

Software reliability is defined as the probability of the correct software performance for a specific period and in a specified environment [59]. It is related to the field of software development, and consists on testing and modeling the ability of the software to behave correctly without failures. The software failures are caused by software faults, which mainly represent errors and bugs caused by effect such as incorrect logic, incorrect statements, incorrect input data, or misinterpretation of the specification that the software is supposed to satisfy in the design. The software reliability is evaluated in the literature through software fault injection and mutation testing [60].

Mutation testing [61] [62] [63] is a software testing technique used to assess the adequacy of test set in terms of its ability to detect software faults. The main idea consists in modifying the original program in order to obtain a faulty program behavior. Mutation testing uses supporting tools to seed artificial faults in the original code of the software in order to generate faulty programs that are supposed to produce incorrect outputs. The faults targeted by mutation analysis represent mistakes that the programmer makes during the implementation or the specification of the program.

Mutation testing is mainly applied in the field of software testing. It is used for the black box and the white box testing [61]. On the one hand, the black box testing is a validation technique for the design level. It permits to test if the program responds to customer requirements when the source code may not be available. At the software design level, mutations are generated to target faults that the programmer may introduce in the program specifications. On the other hand, the white box testing is a validation technique for the software implementation level to test the program source code. It targets the faults that the programmer may introduce in the source code such as coding errors or bugs [61]. At the software implementation level, program mutation is applied on both unit and integration testing [64]. Unit testing is a software testing method where individual units of source code are tested to determine whether they are correct for use or not[65]. Integration testing is the phase in software testing where individual software modules are combined and tested as a group [66]. This step occurs after the unit testing and before the validation testing.

Program mutation is applied on imperative programming language [63] (*e.g.*, C and Fortran), object-oriented programming language [67] [68] (*e.g.*, Java, C++ and C#), and aspect-oriented programming language [69].

As example, for imperative languages, many mutation operators have been explored by researchers, such as statement deletion, statement duplication or insertion, replacement of boolean expressions with true and false, replacement of arithmetic operations with others, *e.g.* + with *, replacement of variables with others from the same scope (variable types must be compatible).

3.2.2 Mutation Testing for Hardware Verification

Hardware reliability is defined as the ability of hardware components inside the system to perform in a predictable way, for a specific period of time under given environmental conditions, without failures [1]. A system is considered highly reliable when it continues to work without interruption during a relatively long period of time.

During the last decades, mutation testing has acquired consensus as an efficient technique to measure the software quality. In recent works, the effectiveness of mutation analysis in

hardware verification has also been proved in different levels of abstraction [70]. The purpose of the hardware verification is to detect and correct errors (*i.e.*, bugs) introduced during the hardware design. These errors can be incorrect logic, incorrect statements, or misinterpretation of the specification that the hardware is supposed to satisfy.

Mutation testing has been applied on Hardware Description Languages (HDLs), such as Verilog and VHDL. In [71], the authors use mutation analysis to study the ability of the test cases to detect errors, and improve the functional verification process of VHDL description. In [72], the authors reuse transaction level modeling mutation analysis at Register Transfer Level (RTL). They demonstrate how this can help designers to optimize the time of simulation at RTL on the one hand, and to improve the RTL test-bench quality on the other hand.

Mutation testing has also been applied to language modeling and verification for system level hardware, such as SystemC [73] [74]. In [75], the authors propose to assess the verification quality for concurrent SystemC programs. They introduce a novel mutation analysis based on coverage metrics. In [76], the authors propose a C/C++/SystemC error and mutation injection tool to facilitate the development on high-level coverage metric and diagnosis. In [77], the authors present an automatic fault localization approach for SystemC transaction level modeling designs. They target typical transaction level modeling faults, such as accidentally swapped blocking and non blocking transactions, erroneous event notification, or incorrect transaction data.

3.3 Discussion

Table 3.1: Fault Injection versus Mutation Testing

	Fault Injection	**Mutation Testing**
Faults	Physical faults leading to soft and hard errors	Software faults occurring in the design or the implementation of the software
Fault Location	The ISA of the microprocessor, the real hardware system, the virtual system, the source/bite-code code, ect	The software source code
Cost	High execution time and hardware cost	High computational cost
Automation	Automatic output analysis	Manual Analysis of equivalent mutants
Reliability Evaluation	Accurate system reliability evaluation for hardware-based fault injection	Good assessment of the test set quality

In this chapter, we presented fault injection and mutation testing as useful techniques to evaluate the reliability of different system layers (*i.e.*, software and hardware systems). In Table 3.1, we present a comparison between the two methodologies. Even if their targets and objectives are different, mutation testing and fault injection are both based on introducing artificial faults into the system under evaluation and observing their impact on its performance.

Mutation testing evaluates the quality of software test set. It quantifies the test set ability to detect faults occurring during software implementation. It has been also applied on hardware verification. However, it does not target soft errors that may occur in the hardware layer

either at run-time or at standstill, which are the main target of the reliability evaluation in this thesis. Hardware-based fault injection techniques target this type of faults, but the evaluation is hardware-dependent. They require to have the fully designed hardware architecture, which could not be available at early design stage of the system. In addition, the fault evaluation is costly in terms of execution time. Existing software-based fault injections also target soft errors, but their main disadvantage is the inaccuracy of the fault evaluation.

The goal of the next chapter is to evaluate soft errors at software level trying to be, at some points, as independent as possible from the target hardware architecture. We aim to bridge the gap between the software and hardware reliability evaluation techniques. We propose new fault injection environments that evaluate hardware faults at different software levels. We couple (i) mutation analysis to have a software-level evaluation, and (ii) fault injection to target hardware faults.

Chapter 4

Proposed Fault Injection Methods

Contents

4.1 Introduction **31**
4.2 LLVM-based Fault Injection **31**
4.2.1 Overview 31
4.2.2 Fault Models 32
4.2.3 Fault Classification 34
4.2.4 Fault Injection 35
4.3 C-based Fault Injection **42**
4.3.1 Overview 42
4.3.2 Fault Classification 43
4.3.3 Fault Injection 43
4.4 Comparison **45**
4.5 Validation **46**
4.5.1 Simulation-based Fault Injection on Intel Processor 46
4.5.2 FPGA-based Fault Injection on LEON3 Processor 47
4.5.3 Experiments' Setup 47
4.5.4 Results and Discussion 49
4.6 Conclusion **51**

In this chapter, we propose new fault injection environments: a virtual-level approach and a source-code-level approach. The validation of the proposed methods is discussed through comparisons with hardware-based fault injections.

Section 4.1 discusses the goal of the proposed methodologies and the advantages compared to related work. Section 4.2 introduces the LLVM-based fault injection. Section 4.3 presents the C-based fault injection. Section 4.4 provides a comparison between the two techniques. Section 4.5 exposes experimental results to validate the proposed approaches. Finally, Section 4.6 concludes the chapter.

4.1 Introduction

One of the goals of this thesis is to evaluate, at early design stage, the reliability of computing systems that are running software. At this stage of the system design, the hardware architecture is not yet fully defined. Thus, it is a challenging task to develop methods that accurately evaluate the system reliability. Based of fault injection, as one of the most used technique to evaluate the reliability, we present in this chapter the first contribution of this thesis. In particular, we propose two fault injection environments: a virtual-level fault injection environment based on LLVM, and a source-code-level fault injection environment based on the C programming language. Both methods allow to evaluate the behavior of the system against soft errors at software level and at different stages of the system design.

Compared to fault injection techniques existing in the literature, the proposed fault injection environments present several outcomes. In comparison with the hardware-based fault injection, the main advantage consists in the absence of the hardware under test. This makes the reliability evaluation less expensive in terms of hardware cost and execution time. In addition, while the hardware-based techniques are dependent to the target hardware architecture, the proposed approaches can be easily applied on different hardware architectures. Furthermore, the proposed fault injection environments offer a better observation of the software components, in particular the fault location in terms of space (i.e. which data or instruction is affected by the fault). This allows a better understanding of the fault effects at software level and enhances the development of reliability improvement techniques. In comparison with existing software-based fault injection, the main contribution of the proposed approaches consists in the good accuracy of the fault evaluation. Existing software-based methods are inaccurate, as discussed in Subsection 3.1.2, because they do not perform fault injection in realistic conditions. In addition, they do not target the evaluation of faults in the different system memory components such as the RAM or the caches, which is performed using the proposed methods. Furthermore, our approaches offer a better visibility of the system memory units, which allows to explore the design space of the target system by easily changing the configurations of its components in order to observe the impact on the reliability.

All the discussed advantages of the proposed fault injection environments will be proved by experimental results in Chapter 5.

4.2 LLVM-based Fault Injection

In this section, we propose new fault injection environment based on the virtual ISA of LLVM. It provides reliability evaluation of the computing system by injecting a set of fault models that represent the effect of soft errors on the software.

4.2.1 Overview

The proposed fault injection environment is based on LLVM and is able to collect data about the software reliability without resorting to predefined hardware platform. The proposed tool permits to inject a set of software fault models into the LLVM intermediate code level of the application, and to observe the outcomes on the software layer.

The main advantages of this approach are: (i) achieving speed up of the simulation time compared to hardware-based fault injection techniques, (ii) targeting faults in both the data and the instructions with only considering the software layer, (iii) targeting faults affecting hardware components such as the cache without requiring a fully designed hardware architecture, (iv) having accurate results and precisions in the fault location in space and time, which could enhance the development of reliability improvement methods.

The proposed approach is close to LLFI [49] [50] and KULFI [51], the virtual-level fault injection techniques presented in Subsection 3.1.2.2. The common point consists in the use of the same virtual ISA of LLVM. However, the proposed approach presents several advantages compared to LLFI and KULFI. The major difference is the considered fault models and the target memory components. The proposed approach considers software fault models representing the effects of the hardware SEU on the software application. The fault is injected as mutation in the readable IR of the LLVM code, while LLFI and KULFI inject faults in the binary LLVM code. This offers to the user more precision in the fault location in space and time, and thus more control on the fault propagation. In addition, thanks to the integration of the memory subsystem emulator, our approach targets faults occurring in the data and the instruction cache, which is not considered by the existing approaches. Furthermore, the proposed approach is validated through comparison to different hardware-based fault injection methods. This validation, not provided by LLFI and KULFI, proves the accuracy of our approach.

4.2.2 Fault Models

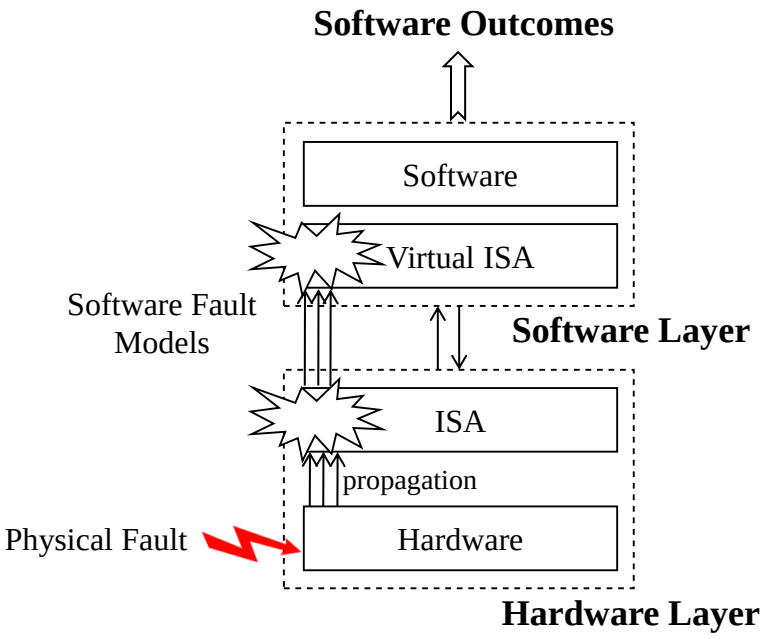

Figure 4.1: Fault Propagation through System Layers

Research approaches considering the impact of software on the full system reliability start from low level hardware faults [78] [79], trying to propagate them through the hardware architecture to the software layers in order to evaluate their impact on the final system [80] [81].

This propagation method requires complex and time consuming simulations of hardware models thus limiting the analysis of complex software stacks. In this thesis, we aim to detach the analysis of the software from the hardware system. We therefore need to model how hardware faults manifest at the software level, as modeled in Figure 4.1. A straightforward way to do this is therefore to map hardware faults into a set of fault models that affect the instructions and the data of the VISA. When the effect of hardware faults is accurately modeled at the software level, we are able to study more complex software stack architectures, and correctly analyze the effect of faults on the full system.

We define fault models that target both the data and the instructions of the software. The fault models represent the effect of single bit flip on the ISA of the microprocessor. They are defined as mutations in the source code. Each mutation represents the effect of a real fault occurring in the hardware.

- **Wrong Data in Operand (WDat):** This fault model represents the effect of a SEU occurring in the memory segment storing the data of the program. At the software level, the data stored in different memory units of the system is represented by the program variables. Thus, a SEU occurring in the data is modeled, at software level, by changing the variable value with a bitwise xor between the correct value and a randomly selected bit (*i.e.*, the bit affected by the SEU). For example, for the LLVM intermediate representation, the value of a variable (presented by operands) can be either an address or an integer, a float, etc. If the variable value is an address, then the new value is either a valid address (*i.e.*, points to an other existing address in the memory segment) or an invalid address (*i.e.*, points to non existing address in the memory segment, which results to a 'segmentation fault' error). If the variable value is an integer value, then the new value is a different integer value. And so on for the other data types.

  It is important to differentiate between this fault model and the software mutation used in software testing. In fact, in case we apply mutation technique on software testing, a variable with the value 1 can be mutated to store any different value, for example the value 2. However, in case we apply mutation technique on the reliability evaluation and we consider that the hardware faults are soft errors (*i.e.*, transient faults that appear and then disappear), we change only one bit at a time (*e.g.*, a single bit-flip in the memory storing the data). We add two new constraints for the data mutation. The first one is related to the mutated value: only a value having a hamming distance of 1 can be accepted in order to avoid that more than one bit-flip at a time occur. Considering the previous example, changing the variable value from 1 to 2 is not correct because it does not represent a single bit flip. The second constraint is related to the transient behavior of the soft errors. In software testing, a mutation permanently affects variable value and persists during the whole program execution. However, this mutation does not represent a transient bit flip. Therefore, to simulate a transient fault, we change the variable value at a randomly selected time. In particular, when the variable is in a loop, we add an index that allows to change the value in one specific iteration of the loop.

- **Instruction Replacement (InstR):** This fault model represents the effect of a SEU occurring in the memory segment storing the code of the program. The instructions are composed of only an opcode or an opcode with one or more operands. To model faults occurring in the instruction, we consider both the operand and the opcode. In the operand,

faults are modeled as WDat. In the opcode, faults are modeled by a switch to a different opcode. To be accurate, we monitor the opcode switch by computing statistics about the probabilities that a given opcode changes to an other. As presented in Figure 4.2, we consider the binary representation of the opcode. We perform a bit flip on it to obtain a new binary representation of a valid or invalid opcode. By applying this process on all the opcodes, we build the statistics that are used to switch two opcodes.

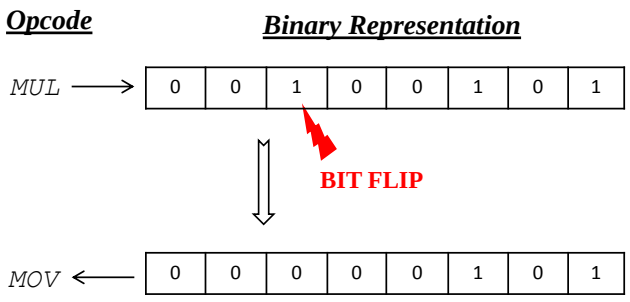

Figure 4.2: Single Bit Flip occurring on Opcode.

It is important to differentiate between the InstR fault model and the software mutation used in software testing. In case we apply mutation technique on software testing, an operator '+' can only be transformed to a new arithmetic operator like '-' or '*'. The purpose of this transformation is to detect software bugs. However, in case we apply mutation technique on the reliability evaluation, we consider that hardware faults (*e.g.*, a bit-flip in the memory segment storing the instructions) result in a random change in the instruction code. Therefore we consider four cases of opcode switch. The three last cases are not considered in software testing:

- An arithmetic opcode that switches to a different arithmetic opcode (*e.g.*, an 'add' that switches to a 'sub').
- An arithmetic opcode that switches to a memory or boolean opcode (*e.g.*, an 'add' that switches to 'load').
- An arithmetic opcode that switches to an invalid opcode (*e.g.*, an 'add' that switches to an invalid opcode that result in a crash or failure of the software).
- An arithmetic opcode may not switch despite of performing a bit-flip.

In conclusion, the application of mutation technique on reliability evaluation requires mutation operators with more constraints and precision in order to accurately model the target soft errors affecting the system.

4.2.3 Fault Classification

Once the hardware faults that can affect the software layer have been properly modeled, we need to study the impact of these faults on the final software result. Based on the literature [82] [83] [80] [84], we define the software outcomes, as a set of the software behaviors when affected by the faults defined in Subsection 4.2.2:

- *Masked*: The software produces correct results. The fault affecting the system component is masked;
- *Fail Silent Violation (FSV)*: The application outputs are different from the fault free outputs;
- *Detected*: The fault has been detected by the application. This case is possible only when a detection mechanism is implemented by the software application under test;
- *Crash / Unresponsive*: The application stops working or it never stops. If the application never stops its execution, we define a time threshold that forces the end of the execution when the execution time exceeds the threshold.

4.2.4 Fault Injection

In the next subsections, we present implementation details of the LLVM-based fault injection. We provide the steps of the fault injection starting by generating the program trace, going through the fault analysis and the fault injection processes and finishing by the outcome classifications.

4.2.4.1 Program Trace

The setup of the fault injection environment for the developed tool starts by compiling the program source code (written in any programming language) to LLVM code using the LLVM compiler. The compiled LLVM code is then executed in order to record a trace of the program. This trace will be used for analyzing and injecting faults in the data and the instructions. The trace contains information about (i) the execution time, which allows to select the random faulty instant, and (ii) the allocated memory, which allows to specify the faulty variable or instruction corresponding to the randomly selected bit.

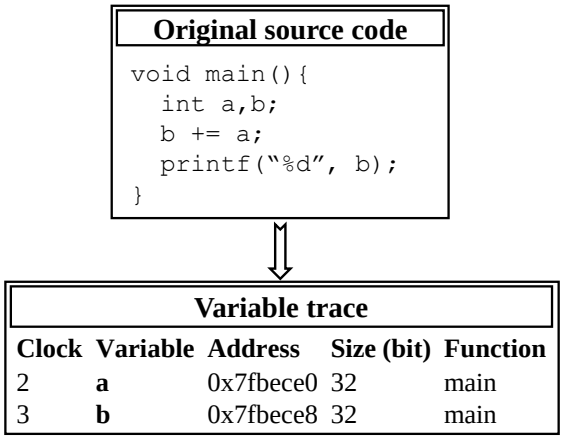

Figure 4.3: Variable Trace

In order to monitor the time during the program execution, we compute the clock cycles. Since LLVM does not provide information about the exact timing when an instruction is executed, we instrument the original LLVM code by adding the information of the current clock

cycle of the executed instruction. We consider that each LLVM instruction is executed in one clock cycle. In particular, we use a counter that is incremented after each instruction. The total number of the program clock cycles is then provided as input to the tool in order to randomly select a faulty clock cycle.

Clock Cycle	Instruction Id	Iteration Number
Clock-1	Line-1	Ex-1
Clock-2	Line-2	Ex-2
…	…	…
Clock-N	Line-M	Ex-L

Figure 4.4: Instruction Trace

Then to monitor the allocated memory during the program execution, we collect information about the flow of variables and instructions. In particular for each clock cycle, we record the allocated or/and dis-allocated variables and instructions. For variables, we record the variable name, the physical address, the size, and the function, as shown in Figure 4.3. For instructions, we record the LLVM line, which is considered as identifier of the instruction, and the number of time each instruction has been executed. The last information allows to monitor the eventual loops executed in the program. A model of the instruction trace is provided in Figure 4.4. The variable and instruction traces allow to have the flow of active variables/instructions during the program execution in order to determine the faulty variable or instruction corresponding to the randomly selected bit.

4.2.4.2 Fault Analysis

The fault can occur randomly in any memory location and at any instant of the execution time. During its execution, the program can use different spaces of the available memory, as shown in Figure 4.5. According to the program flow, it can exist some unused data or instruction memory. Thus, the faults occurring in that space do not have influence on the program execution and its outputs. In order to consider this aspect, the proposed tool proceeds by a first analysis of the program, as presented in Figure 4.6. It randomly selects an instant in the execution time and a bit in the memory. Using the program traces, the tool determines if the selected bit is located in the unused memory. If it is the case, the fault is masked without performing fault injection and executing the program, otherwise the tool proceeds by the fault injection process. This analysis process offers a reduction in the required number of fault injection, and thus a reduction in the simulation time.

The fault injection mechanism allows to inject fault directly into the readable LLVM IR code of the program. In Figure 4.7, we provide the flow of the fault injection process. For each analyzed fault (*i.e.*, non masked by the analysis process), the tool determines, using the program traces, the faulty parameters (*i.e.*, faulty variable, faulty instruction, faulty iteration, ect) that correspond to the selected faulty memory location and instant. Then, it generates from the LLVM program source code, the faulty programs. Each faulty program presentes a mutation in the original program that corresponds either to WDat or InstR. The execution of the faulty programs allows to collect the software outputs, which can be either the standard outputs or the

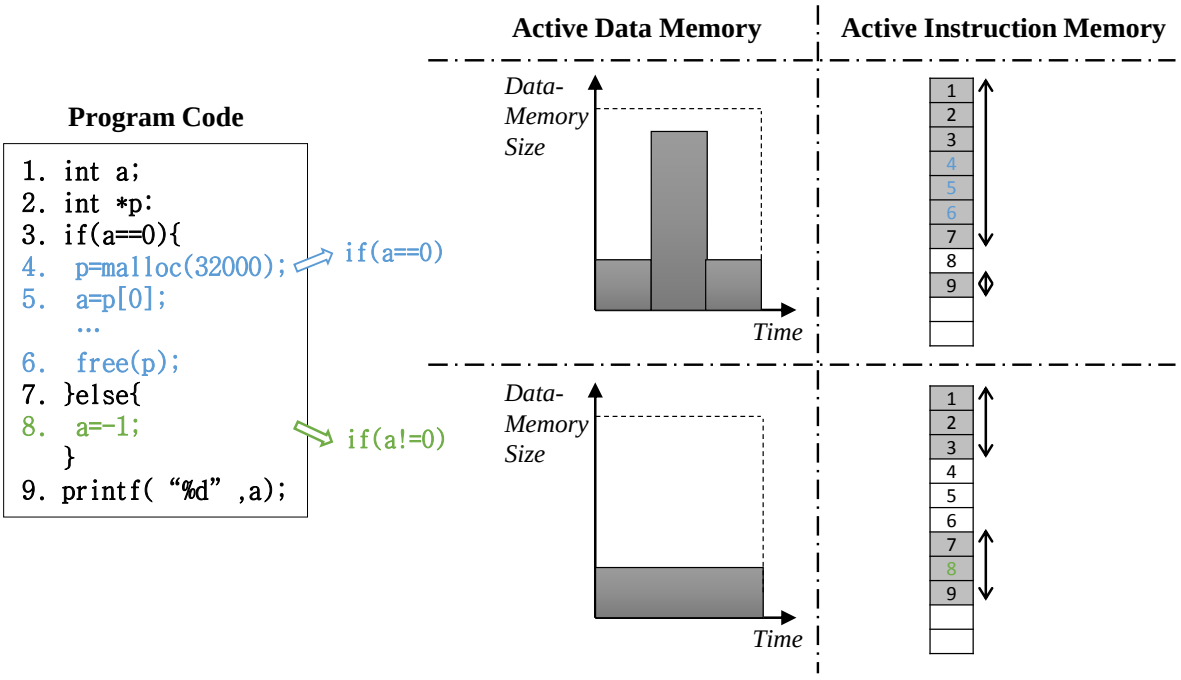

Figure 4.5: Active Data and Instruction Memory according to the Program Execution

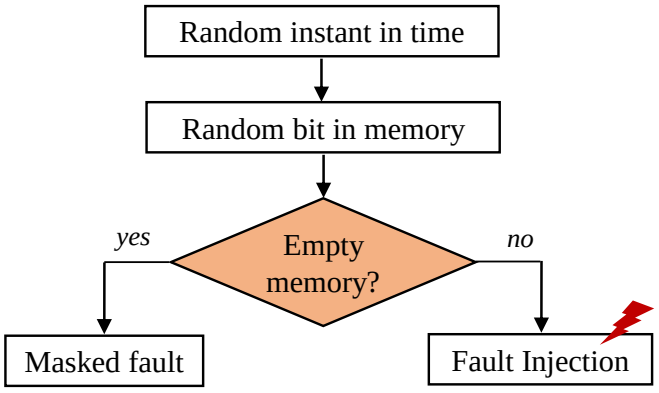

Figure 4.6: Flow of the Fault Analysis Process

erroneous outputs (*e.g.*, crash, hangs). The execution of the faulty free program allows to get the golden outputs. Finally the comparison of the golden and the faulty outputs provides the outcome classifications.

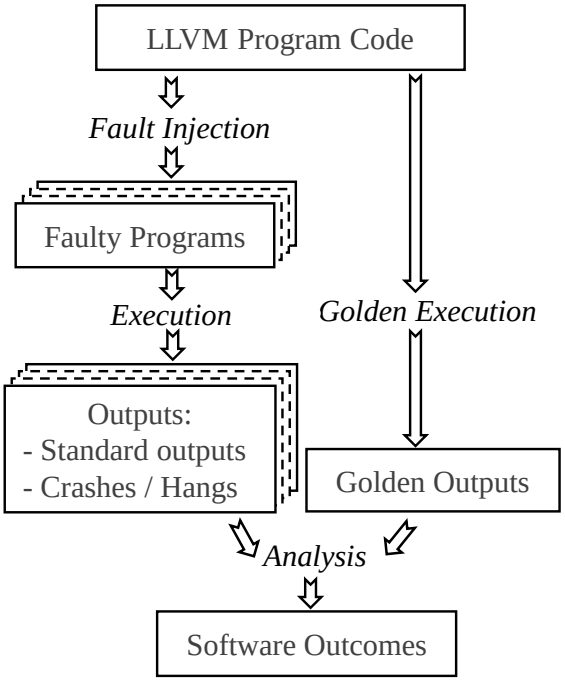

Figure 4.7: Flow of the Fault Injection Process.

Table 4.1: Wrong Data Fault Model.

Fault injection in the variable value	%var-FI1 = load i32* %var %var-FI2 = xor i32 %var-FI1, 64 store i32 %var-FI2, i32* %var
Fault injection in the variable address	%var-FI1 = load i32** %var %var-FI2 = ptrtoint i32* %var-FI1 to i64 %var-FI = xor i64 %var-FI2, 128 %var-FI3 = inttoptr i64 %var-FI to i32* store i32* %var-FI3, i32** %var

4.2.4.3 Fault Injection in Data

In LLVM, the data is represented by the program variables (operands). The fault injection in the data corresponds to the WDat fault model, as explained in Subsection 4.2.2. Thus, we model the bit flip by a bitwise xor between the correct value or address and a randomly selected bit. In order to inject the fault, we build a set of instructions that allow to model the bit flip and

to activate the fault injection in the selected variable at the selected time. This set of instructions are embedded into the LLVM program code.

In Table 4.1, we provide an example of the instructions that allow to model the bit flip in the value and the address of a variable of type 32-bit integer. For the injection in the variable value, the faulty variable contains a new different integer value. However, for the injection in the variable address, the faulty variable contains either a new valid address (i.e., the variable points to an other existing address in the memory segment) or an invalid address (i.e., the variable points to non existing address in the memory segment, which results to a 'segmentation fault' error).

In order to make the fault transient, we dynamically control the insertion of this set of instructions into the original program source code. In Figure 4.8, we provide an example of permanent and transient fault injection in the value of a variable of type 32-bit integer.

Permanent Fault Injection

```
define void @main(){
  %a = alloca i32
  store i32 1, i32* %a
  call @inject(i32* %a)
  ...
}

define void @inject(i32** %add){
  %a1 = load i32* %a
  %aFI = xor i32 %a1, 8
  store i32 %aFI, i32* %a
}
```

Permanent injection

Variable BitFlip

Transient Fault Injection

```
define void @main(){
  ...
  ; <label>:1
  call @inject(i32* %a, i32 25)
  %3 = icmp ne i32 %2, 100
  br i1 %3, label %1, label %4
  ...
}

@clk = internal global i32 0
define void @inject(i32* %a, i32 %clkFI){
  %1 = icmp eq i32 @clk, %clkFI
  br i1 %1, label %2, label %3

  ; <label>:2
  %a1 = load i32* %a
  %aFI = xor i32 %a1, 64
  store i32 %aFI, i32* %a
  br label %3

  ; <label>:3
  %4 = add i32 @clk,1
  store i32 %4, i32* @clk
}
```

Loop

Transient injection in iteration 25

Variable BitFlip

Figure 4.8: Transient and Permanent Fault Injection in Data.

4.2.4.4 Fault Injection in Instructions

In LLVM, the instruction is composed of either an opcode or an opcode and one or more operands. Thus a fault in the instruction can be either in the opcode or the operands. If the randomly selected bit corresponds to an operands, the fault is a WDat in the variable expressed by that operand. Thus, we follow the process explained in the previous subsection. However, if the selected bit corresponds to an opcode, the fault is an InstR.

The InstR fault model corresponds to the transformation of an opcode to an other. The statistics about this transformation is microprocessor dependent. For a certain ISA, we develop a tool that analyzes, for each instruction, how the opcode is transformed to another one. This information can be easily obtained using this tool. For this thesis, we implement a single bit flip on three different ISAs that we used for validation: the SPARC ISA [85], the ARM ISA [86], and the Intel x86 ISA [87]. As presented in Figure 4.2, the tool considers the binary representation of each opcode to inject a bit flip. The resulting binary representation corresponds either to a valid or an invalid opcode. By applying this process on all the opcodes, we build the statistics of each opcode to change to a valid or an invalid opcodes.

This type of analysis is very quick and can be done for many ISAs. This analysis will create several tables of statistics. When the analysis of the software is done at an early design stage of the system, *i.e.*, without the knowledge of the actual target microprocessor, we propose to calculate an average table, where each probability is the mean value coming from all the analyzed ISAs. This type of analysis would characterize the software in less accurate way, but it would allow generating very first results in terms of software reliability. When the target microprocessor is selected, simulations can be performed again (by using the correct table of instruction replacement) in order to get better results.

For the injection of the InstR fault model, given the faulty opcode, the tool randomly selects from the table of statistics, the corresponding valid or invalid opcode to which it should switch. Once we know the new opcode, we build the faulty program with the switch of the opcode. If the new opcode is invalid, we consider that the outcome of the fault injection is a crash without building the faulty program. In Figure 4.9, we provide an fault injection example of InsR for the SPARC ISA. Assuming that the faulty bit corresponds to the opcode 'mul' in the instruction number 4, and that the selected number is 0.15. Based on the statistic table of SPARC ISA, the new opcode is 'or'.

```
1. define void @main(){                    define void @main(){
     ...                                     ...
2. ; <label>:32                            ; <label>:32
3.   %33 = load i32* %i                ⇨     %33 = load i32* %i, align 4
4.   %34 = mul nsw i32 %33, 10               %34 = or i32 %33, 10
5.   %35 = load i32* %j                      %35 = load i32* %j, align 4
6.   %36 = add nsw i32 %34, %35              %36 = add nsw i32 %34, %35
     ...                                     ...
   }                                       }
```

Figure 4.9: Transient and Permanent Fault Injection in Data.

4.2.4.5 Outcome Analysis

To analyze the fault injection effect, we execute the faulty and the golden programs, and we collect the software outcomes of each execution. The golden outcome is the standard outputs of the faulty free program. However, the faulty outcome can contain either the standard outputs or the error messages. During the execution of the faulty programs, we define a time threshold. In case of program unresponsiveness (*e.g.*, infinite loop) caused by the injected fault, the execution

Table 4.2: Statistics (%) of Instruction Replacement for SPARC Architecture

	geteltptr	**store**	**and**	**or**	**xor**	**shl**	**lshr**	**ahr**	**add**	**sub**	**mul**	**udiv**	**sdiv**	**br**	**call**	**ret**	**fadd**	**fsub**	**fmul**	**fdiv**	**cmp**	**load**	**alloca**	**invalid**
geteltptr	42.8	12.4	1.4	1.4	1.4	0	0	0	2.1	0.7	2.1	0	0	0	0	0	0	0	0	0	0	0	0	35.9
store	9.6	40.6	1.6	1.6	1.1	0.5	0.5	0.5	1.6	1.6	1.6	0	0	0	0	0	0	0	0	0	0	0	0	38.5
and	6.3	9.4	25	0	6.3	3.1	0	0	6.3	9.4	0	0	0	0	0	0	0	0	0	0	0	0	0	34.4
or	6.3	9.4	0	25	6.3	0	3.1	0	9.4	6.3	6.3	6.3	0	0	0	0	0	0	0	0	0	0	0	21.9
xor	11.8	11.8	11.8	11.8	11.8	0	0	0	0	0	17.6	0	0	0	0	0	0	0	0	0	0	0	0	23.5
shl	0	12.5	12.5	0	0	0	0	12.5	0	12.5	12.5	0	0	0	0	0	0	0	0	0	0	0	0	37.5
lshr	0	12.5	0	12.5	0	0	0	12.5	12.5	0	12.5	0	0	0	0	0	0	0	0	0	0	0	0	37.5
ahr	0	12.5	0	0	0	12.5	12.5	0	0	0	12.5	0	0	0	0	0	0	0	0	0	0	0	0	50
add	6.3	6.3	4.2	6.3	0	0	2.1	0	25	10.4	8.3	0	0	0	0	0	0	0	0	0	0	6.3	0	25
sub	2.5	7.5	7.5	5	0	2.5	0	0	12.5	20	5	5	0	0	0	0	0	0	0	0	0	2.5	2.5	27.5
mul	6.1	6.1	0	4.1	6.1	2	2	2	8.2	4.1	16.3	4.1	4.1	0	0	0	0	0	0	0	0	6.1	0	28.6
udiv	0	0	0	12.5	0	0	0	0	0	12.5	12.5	12.5	12.5	0	0	0	0	0	0	0	0	0	0	37.5
sdiv	0	0	0	0	0	0	0	0	0	0	12.5	12.5	12.5	0	0	0	0	0	0	0	0	0	0	62.5
br	0	0	0	0	0	0	0	0	0	0	0	0	0	58.2	0	0	0	0	0	0	0	0	0	41.8
call	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100
ret	0	12.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12.5	0	75
fadd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7.8	5.9	5.9	0	0	0	0	80.4
fsub	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5.9	7.8	0	5.9	0	0	0	80.4
fmul	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	3.5	0	7.1	4.7	0	0	0	84.7
fdiv	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5.9	7.8	7.8	0	0	0	78.4
cmp	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100
load	0	0	0	0	0	0	0	0	9.4	3.1	9.4	0	0	0	0	3.1	0	0	0	0	0	25	0	50
alloca	0	0	0	0	0	0	0	0	0	1.8	0	0	0	0	0	0	0	0	0	0	0	0	0	98.2

Table 4.3: Statistics (%) of Instruction Replacement for ARM Architecture

	add	**and**	**br**	**icmp**	**xor**	**or**	**mul**	**load**	**store**	**sub**	**invalid**
add	0.7	1.3	4.6	5.2	1.7	1.7	9.9	12.2	37.1	3.3	22.1
and	2.7	0	3.9	5.2	1.4	1.8	11.1	10.2	35	3.6	25.2
br	4.6	2	2.1	4.7	1.9	1.9	12.5	13.2	38.7	3.8	14.6
icmp	5.2	2.6	4.7	1.4	2.6	2.6	14.9	15	42.4	3.8	4.7
br	3.4	1.4	3.8	5.2	0	2.1	11.1	11.8	37.1	3.4	20.7
mul	3.4	1.8	3.8	5.2	2.1	0	7.9	12	36.3	3.6	24.1
call	4	2.2	5	6	2.2	1.6	4	12.3	39.3	3.3	20.2
sub	4.9	2	5.3	6	2.4	2.4	12.3	4	33.8	4.1	22.9
div	6.2	2.9	6.4	7.1	3.1	3.0	16.4	14.1	19.3	4.4	17.1
ret	3.8	2.1	4.8	4.5	1.8	2.3	9.1	14.3	36.1	5.2	16.1

is suspended when the time execution exceeds the defined threshold. The final step is the comparison of the golden and the faulty outcomes, and the classification of faults according to the software outcome classes defined in Subsection 4.2.3. This allows to evaluate the software behavior against soft errors.

Table 4.4: Statistics (%) of Instruction Replacement for Intel x86 Architecture.

	add	store	load	icmp	br	mul	call	sub	div	ret	invalid
add	33	7.4	1.7	0	0.6	0	0.6	0	0	1.1	55.6
store	3.9	39.9	5.1	1.8	0.3	0	0.9	0	0.3	0.6	47.2
load	2.9	16.3	30.8	0	0	0	0	0	0	0	50
icmp	0	6.5	0	23.9	0	0	0	10.9	0	0	58.8
br	2.2	2.2	0	0	13	4.3	4.3	0	0	0	74
mul	0	0	0	0	4.5	18.2	0	0	9.1	0	68.2
call	2.6	7.9	0	0	5.3	0	5.3	0	0	0	78.9
sub	0	0	0	10.9	0	0	0	24	0	0	65.1
div	0	2.3	0	0	0	9.1	0	0	18.2	0	70.4
ret	12.5	12.5	0	0	0	0	0	0	0	12.5	62.5

4.3 C-based Fault Injection

In this section, we present a new software-based fault injection environment that evaluates the reliability of computing systems by injecting faults in the source code of the software application.

4.3.1 Overview

In order to evaluate the reliability at higher software level, we propose a new fault injector tool that aims to inject faults directly in the source code written in software programming language. It simulates realistic faults occurring in the hardware system. The considered fault models simulate the effect of soft errors (i.e. bit flip in data or instructions) on the software application of the target system. It does not consider errors in code resulting from the implementation or the design of the software.

The main advantages of the proposed approach are: (i) developing a fast, low-cost and accurate platform to evaluate the system reliability in different stages of the system design, (ii) targeting faults occurring in data and instructions of the code by evaluating their effects on the software system, (iii) targeting faults affecting hardware components such as the RAM and the cache without requiring a fully designed hardware architecture, and (iv) offering better observation of the software components and better control of the fault injection mechanism in terms of the fault location in space and time, which can enhance the development of reliability improvement methods.

Our approach is classified as a code-level fault injection technique. Compared to existing code-level fault injections presented in Subsection 3.1.2.3, the main advantage of the proposed approach is the target of faults occurring in both the data and the instructions. In addition, it targets the faults affecting the RAM and the caches of the system. Furthermore, the proposed approach offers high accuracy level of fault evaluation validated with the comparisons to hardware-based fault injections.

4.3.2 Fault Classification

In order to study the impact of faults on the software behavior, we adopt the software outcomes presented in Subsection 4.2.3.

4.3.3 Fault Injection

In the next subsections, we present the implementation details of the C-based fault injection. We provide an overview of the tool design. Then, we detail the injection mechanism and the outcome classifications.

4.3.3.1 Design

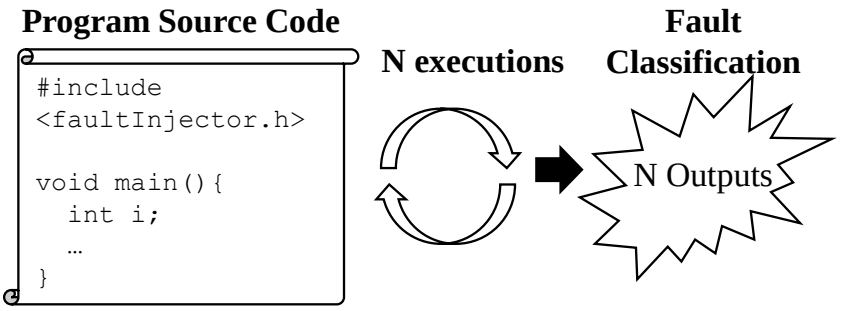

Figure 4.10: Overview of the C-based Fault Injection.

The proposed fault injection tool allows to inject bit flip in both the data and the instructions of the target application. It operates on the original source code of the software to simulate single fault injections without requiring the hardware platform. In Figure 4.10, we provide an overview of the fault injection mechanism. The program is executed N times (N is the number of the fault injections). Each program execution simulates a single fault (the fault is injected during the execution at run-time) and generates the faulty outputs that are compared with the golden outputs in order to classify the faults. This process can be easily extended to multiple faults.

The fault injection process is composed of two main threads running in parallel, as shown in Figure 4.11. The first thread consists in collecting the data and the instructions of the program. The second thread is responsible of selecting the instant and the bit, and injecting the fault.

During the program execution, the tool dynamically collects all the data and the instructions. We embed additional function calls in the original source code of the program as presented in Figure 4.12. The proposed tool collects only active software components during the program execution. When they are no longer active, they are removed from the list. For fault injection in data, we collect the program variables. These variables represent data stored in the different system memory units. For each variable, the tool records *the variable address*, which is the variable identifier; *the variable size*, which is required to identify the faulty bit; and *the variable* and *function name*, which are used to provide detailed description of the fault behavior. For fault injection in instructions, we collect the function codes of the program. The function codes

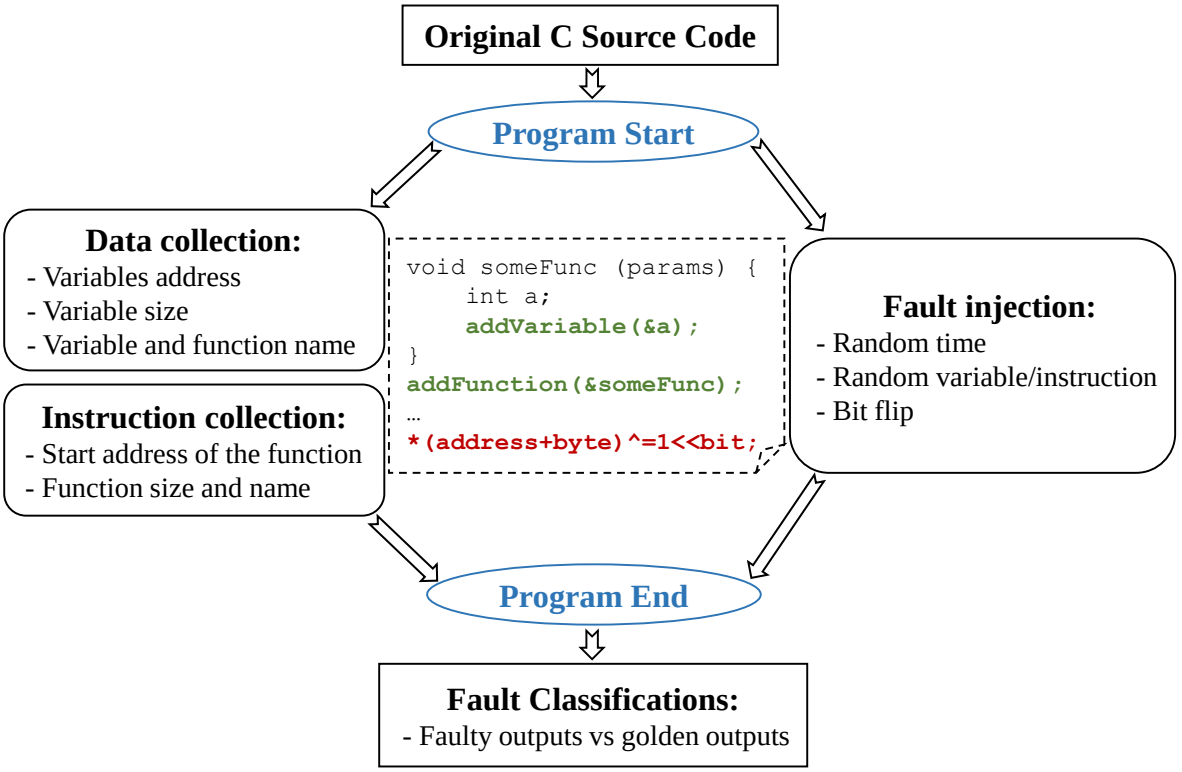

Figure 4.11: Process of the C-based Fault Injection.

represent instructions that are stored in the RAM or the instruction cache. In Listing 4.1, we show how we record, for each function, its *name* , *size* and *start address*. The start address and the size are required to select the random faulty instruction and bit. We consider for that the binary representation of the instructions.

Listing 4.1: An example of collecting a function

```
void someFunc(params) {
 int a;
 char b;
}
addFunction("someFunc", &someFunc, 257);
```

4.3.3.2 Injection Thread

An additional thread is running in parallel with the main program thread. It performs a single bit flip per program execution. First, it selects a random faulty instant from the total time of the program execution. The total program execution time is measured in nanosecond. Then, it waits until this instant is reached to select a random variable or function and a random bit to be the target of the fault injection. The selection of the faulty variable or function is done from the active variables or instructions collected at the faulty instant. Then, the thread accesses to the selected variable or function address and performs a bit flip in the selected bit, as modeled in Listing 4.2.

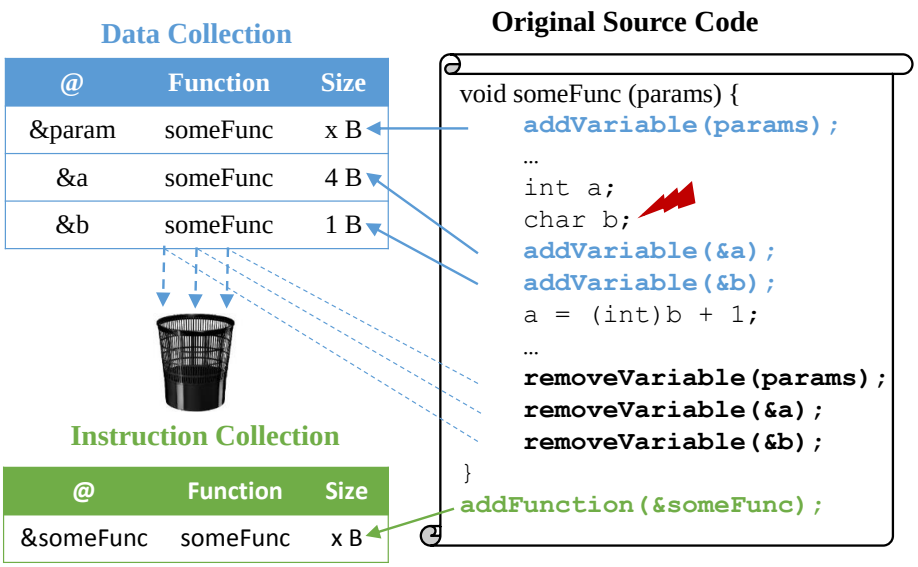

Figure 4.12: Data and Instruction Collection

Listing 4.2: The Fault Injection Function (bitflip).

```
void generateBitFlip ( ) {
 *( char* ) ( address + byte ) ^= 1 << bit;
}
```

4.3.3.3 Outcome Analysis

The fault injection tool executes the program N times, each time with a fault injection. After each injection, the tool analyzes and compares the outputs of the faulty execution to the golden output, and classifies them according to the defined software outcome classes defined in Subsection 4.3.2. The outputs of the C-fault injection tool, as presented in Figure 4.13, is precise in term of the fault controlling. This means that the user is able to know which fault (*i.e.*, location in space and time) produces which output. This enhance the development of adequate detection and protection mechanism in order to design reliable systems.

4.4 Comparison

The LLVM- and C- based fault injections developed during this thesis allow to evaluate the effect of soft errors on a given software application. They can be used at different stage of the system design. The main advantage of the fault evaluation at the software level is the gain in the execution time and the save in the hardware cost.

While the goal of both techniques is the same, we note some differences that create advantages and disadvantages of one compared to the other. In terms of hardware independence, both the LLVM- and the C-based techniques provide fault injection in data at early design stage of the system without the knowledge of the hardware architecture. For faults in instructions, the

INFO: [faults=10000]
INFO: outcome: M (masked), F (FSV), C (crash), D (detected)
****

N°	Outcome	Function	Variable	Faulty Time	Used Memory	Faulty Execution Time
1	M	main	main/matrixB	1782401422	4096	1791338418
2	F	main	main/matrixA	1408767358	4096	1459184102
3	M	unused-2845	unused-2845	1361018	1096	8726080
4	F	func_1	main/matrixA	1898564295	4096	1791341428

...

Figure 4.13: Output of C-based Fault Injection

LLVM-based tool proposes a solution that can be used at early design stage, when the hardware ISA is not yet defined. However, in order to provide more accurate characterization of the software, the statics used for the instruction replacement require the knowledge of the processor ISA. Then, in terms of source-code independence, the LLVM-based tool supports different programing languages, while the C-based tool is applied only on C programing language. The use of the virtual LLVM ISA offers more precision in terms of the fault instant because it is similar to an assembly language. The time is computed in clock cycles while for C-based tool time is computed in nanosecond. The fault injection in the C source code offers better precision in the fault location in space (*i.e.*, faulty variables and instructions) because it is a high-level programming language compared to LLVM IR, and thus more understandable for human, which enhances the development of reliability improvement methods.

While in this thesis, we simulate single fault injection, both LLVM-based and C-based fault injection methods can be easily applied on multiple fault simulation.

4.5 Validation

To validate the proposed approaches, we compare the results of the fault injections to those obtained with hardware-based fault injections. In the literature, hardware-based techniques provide accurate fault evaluation results. We make use of a simulation-based fault injection technique applied on the Intel©8086 [88] processor, and an FPGA-based fault injection technique applied on the LEON3 [89] processor. In this section, we present the first results obtained with this comparison.

4.5.1 Simulation-based Fault Injection on Intel Processor

The used simulation-based fault injection is applied on the Intel©8086 processor [88]. The fault injector embeds *Saboteurs* in the processor VHDL description in order to create transient faults in the RAM. The performed fault model is a bit flip in the selected memory address, (*i.e.*, an inversion of the target bit from logic '0' to logic '1' or *viceversa*).

Figure 4.14 illustrates the conceptual view of the simulation-based fault injection tool. The injection campaign is supervised by the fault injector manager, which takes as main input the

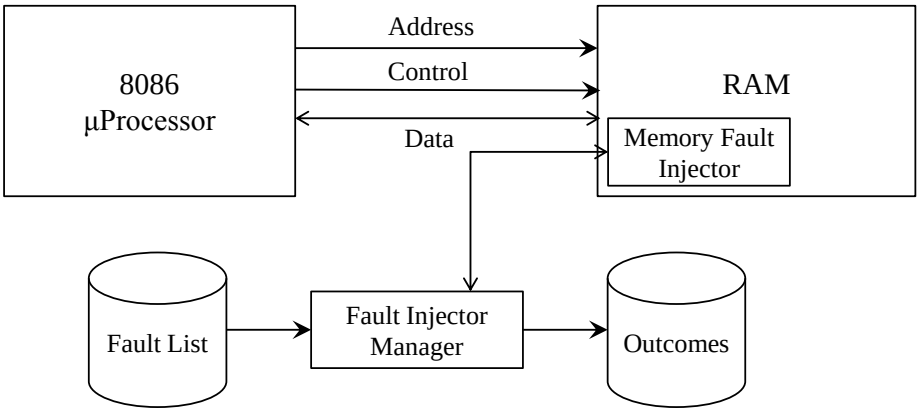

Figure 4.14: Simulation-based Fault Injector.

fault list, and delivers as outcomes the fault classification presented in Subsection 4.2.3. The fault list has the following entries:

- *Injection Time:* It describes the fault injection instant;
- *Memory Address:* It determines the memory address where the fault is injected;
- *Mask:* The mask specifies the bit of the memory address where the bit flip is generated.

4.5.2 FPGA-based Fault Injection on LEON3 Processor

LEON3 [89] is an open-source 32-bit embedded processor. We use the FPGA-based fault-injection technique SCFIT [27] on the LEON3 processor. SCFIT permits injecting faults in flip flops and memory units (*e.g.* instruction/data cache, register files and RAM) using Altera [40] debugging facilities. The SCFIT platform manages the fault-injection process and the communication between the host computer and the FPGA board, as shown in Figure 4.15. After implementing the target processor on the FPGA board, the host computer sends the program to be executed. A fault is injected in the target processor component during the execution of the program. When the faulty execution completes, snapshots of the RAM are sent back to the host computer. The faulty RAM is compared to the golden RAM in order to classify the fault.

4.5.3 Experiments' Setup

- **Benchmarks**

The target application is a simple matrix multiplication program with three versions: (1) simple version, (2) duplicated version with a detection mechanism, and (3) triplicated version with a correction mechanism. The application is implemented in C programming language and is evaluated by the proposed approaches. The provided outcome classifications are used to evaluate the reliability of the software application.

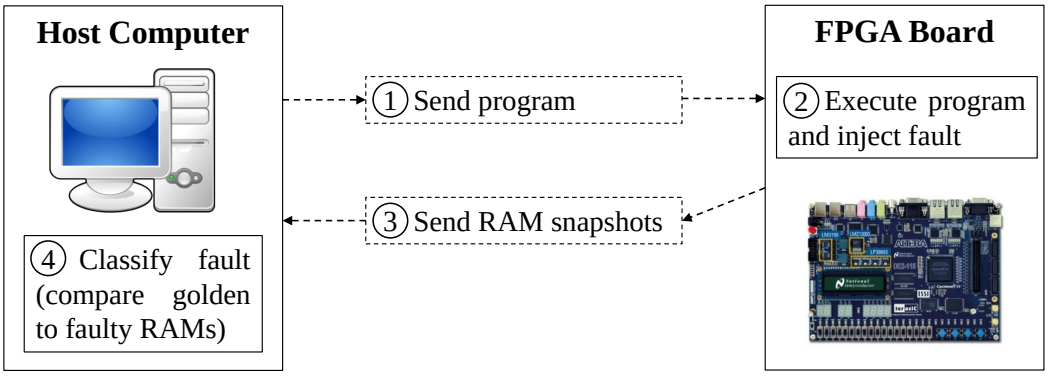

Figure 4.15: Communication between Host Computer and FPGA Board in SCFIT

- **Sample Number of the Fault Injections**

In order to obtain statistically significant results, we adopt the method proposed in [90].

$$n = \left(\frac{N}{1 + e^2 \cdot \frac{N-1}{t^2 \cdot p \cdot (1-p)}} \right) \tag{4.1}$$

The number of samples is expressed in Equation 4.1, where:

- n is the number of faults to inject;
- N is the overall number of possible injected faults;
- p is an estimation of the value being searched;
- e is the expected margin of error;
- t is the percentile (of a normal distribution) corresponding to the point having the desired confidence level 1-α. It represents the vertical boundary for the area of α/2 in the right tail of the distribution. For instance, for 95% confidence level, the t value corresponds to the 97.5 percentile of the distribution, as shown in Figure 4.16.

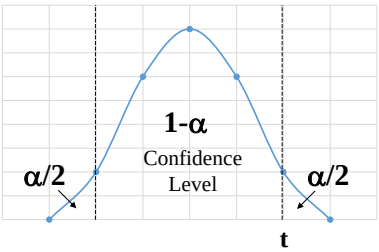

Figure 4.16: Gaussian Distribution

Since the overall number of possible faults is extremely high, we compute the limit of Equation 4.1 for N tending to infinite. We also consider the conservative result by setting p equal to 0.5.

The result is shown in Equation 4.2.

$$n = \lim_{N \to \infty} \left(\frac{N}{1 + e^2 \cdot \frac{N-1}{0.25 \cdot t^2}} \right) = \frac{t^2}{4 \cdot e^2} \tag{4.2}$$

We consider for all experiments a margin of error of 1% with a confidence level of 95%, leading to 10K fault injections per program.

4.5.4 Results and Discussion

We run experiments using the simulation-based fault injector applied on the Intel©8086 processor. We simulate transient bit flips in the memory. In parallel, we use the developed approach to simulate similar faults, then we compare the obtained results.

In Table 4.5, we present the simulation results, for the fault injection in data for the three version of the matrix multiplication workloads. The results show that there is a similarity between the LLVM-, the C- and the simulation-based fault injection for faults in data.

Table 4.5: Results of Single Transient Fault Injection in Data

Benchmark	Simulator	Masked	FSV	Detected	Crash
(1) mMul	**LLVM FI**	20.94%	78.19%	0%	0.87%
	C FI	20.88%	78.90%	0%	0.22%
	8086 FI	18.4%	81.6%	0%	0%
(2) mMul Dup	**LLVM FI**	22.92%	19.25%	57.26%	0.57%
	C FI	22.78%	21.21%	55.95%	0.06%
	8086 FI	21.4%	19.1%	59.5%	0%
(3) mMul TMR	**LLVM FI**	85.6%	13.64%	0.18%	0.58%
	C FI	86.07%	13.78%	0%	0.15%
	8086 FI	87.5%	12.5%	0%	0%

In Table 4.6, we present the simulation results, for the fault injection opcode (InstR), for the three programs using the LLVM-based approach and the simulation-based fault injection technique.

Based on the previous results, we calculate the final reliability estimation of the overall system. We set: D the data size, OC the opcode size and OP the operand size. We consider that an InstR fault affects only the opcode and a WDat fault affects the data and the operand, as shown in Figure 4.17. The equation 4.3 computes the final outcome for the LLVM-based fault injection.

$$Outcome^{LLVMFI} = \left(\frac{Outcome^{WD} * (D + OP) + Outcome^{IR} * OC}{D + OC + OP} \right) \tag{4.3}$$

Using the simulation-based fault injection, a fault in data affects only the data, and a fault in the code affects both the opcode and the operand as shown in Figure 4.17. The equation 4.4 computes the final outcome for the simulation-based fault injection.

Table 4.6: Results of Single Transient Fault Injection in Opcode

Benchmark	Simulator	Masked	SDC	Detected	Crash
(1)mMul	**LLVM FI**	27.1%	0%	0%	72.9%
	8086 FI	10.9%	18.2%	0%	70.9%
(2)mMulDup	**LLVM FI**	25.5%	0%	0%	74.5%
	8086 FI	12.3%	10.8%	13.8%	63.1%
(3)mMulTMR	**LLVM FI**	26.7%	0%	0%	73.3%
	8086 FI	23.5%	9.8%	7.8%	58.8%

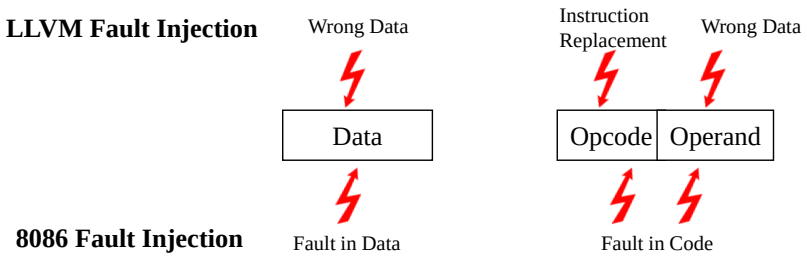

Figure 4.17: Comparison between HW FI and SW FI.

$$Outcome^{8086FI} = \left(\frac{Outcome^{Data} * D + Outcome^{Code} * (OC + OP)}{D + OC + OP} \right) \tag{4.4}$$

The final result to estimate the reliability of the whole system is presented in Table 4.7. The results of the LLVM-based fault injection and the simulation-based fault injection are very close for the three programs, which proves the accuracy of the proposed approach to evaluate the system reliability.

Table 4.7: Outcomes to Evaluate the Overall System Reliability.

Benchmark	Simulator	Masked	SDC	Detected	Crash
(1)mMul	**LLVM FI**	44.0%	55.0%	0%	1.1%
	8086 FI	43.7%	52.7%	0%	3.6%
(2)mMulDup	**LLVM FI**	22.4%	17.9%	58.8%	0.9%
	8086 FI	23.9%	18.9%	56.5%	0.7%
(3)mMulTMR	**LLVM FI**	83.7%	15.1%	0.2%	1.0%
	8086 FI	85.8%	13.3%	0.2%	0.7%

The last comparison between these two fault injectors concerning the simulation time is given in Table 4.8. The table depicts the huge difference w.r.t. to the required simulation time.

While the LLVM-based fault injection requires less then one minute for all the three programs, the simulation time of the simulation-based fault injection is 2 orders of magnitude greater (*i.e.*, up to 21 hours versus 1 minute). These results clearly show the efficiency of the LLVM-based fault injection that is based on mutation analysis.

Table 4.8: Comparison of SW FI and HW FI Simulation Times.

Benchmark	Simulator	Execution time
(1)mMul	**SW FI**	less than 1 minute
	HW FI	6 hours
(2)mMulDup	**SW FI**	less than 1 minute
	HW FI	18 hours
(3)mMulTMR	**SW FI**	less than 1 minute
	HW FI	21 hours

The results presented so far proves the accuracy of the LLVM-based fault injection to evaluate faults in the data and the instructions, and the accuracy of the C-based fault injection to evaluate faults in the data. The simulation of faults in instructions using the C-based fault injection are still not complete, thus they are not presented in this thesis.

The second validation of the proposed fault injection environments is based on the comparison to the FPGA-based fault injection applied on the LEON3 processor. For the sample version of the matrix multiplication workload, the masking probability is 74% for faults in data. This result is different from the results provided by the LLVM-, C- and simulation-based fault injection in Table 4.5. This difference comes from the activation of the caches in the LEON3 processor. That was not the case for the 8086 processor that has a sample architecture where the cache is dis-activated. The presence of the cache in the used processor has a huge impact on the evaluation of faults occurring in the RAM.

In fact, in modern microprocessors, the concept of caches is introduced to store data in order to accelerate their future requests by the CPU. To be cost-effective and to enable efficient use of data, caches are relatively small compared to the RAM. Thus, during the program execution, the frequently used data are stored in both the RAM and the caches. The CPU uses the copy of data that is stored in the cache, while the copy stored in the RAM will not be used. As a consequence, the failure probability (*i.e.*, FSV and crashes) resulting from the fault injection in the data of the RAM is reduced. This explains the increase of the masked faults resulting from the fault injection using the FPGA-based tool where the cache is activated, compared to the three other tools where the cache is not activated.

4.6 Conclusion

We presented in this chapter two new fault injection environments based on LLVM the virtual ISA, and on the C programming language. The main challenge is to evaluate the effect of hardware faults on the software. The experimental results show that the simulation of faults using

the developed approaches provides accurate results in case of targeting sample microprocessor architecture. However, when the evaluation is applied on complex architectures where one or more caches are activated, the LLVM- and C-based fault injection approaches can not provide accurate fault evaluation.

In order to adapt our approaches to modern microprocessors disposing complex architecture with one or multiple levels of cache, we propose a memory subsystem emulator that is able to emulate the behavior of the RAM, the caches, and the register files without the presence of the fully defined hardware. The memory subsystem emulator presents the second contribution of this thesis, and is introduced in Chapter 5.

Chapter 5

Memory Subsystem Emulator

Contents

5.1 Introduction **53**
5.2 Subsystem Emulator **54**
5.2.1 RAM Emulator 56
5.2.2 Cache Emulator 56
5.2.3 Register Files Emulator 58
5.3 Validation through Memory Emulator **60**
5.3.1 Emulators' Integration to Fault Injection 60
5.3.2 Experimental Results 62
5.4 Conclusions **66**

We propose in this chapter a memory subsystem emulator able to emulate, at software-level, the behavior of the RAM, the caches, and the register files without the presence of the fully defined hardware.

Section 5.1 explains the motivation behind developing the emulator. Section 5.2 provides the implemented algorithms for building the memory emulator. Section 5.3 introduces the integration of the emulator into the proposed fault injection environments, and provides experimental results to prove the accuracy of the approach as well as the fast execution time. Section 5.4 concludes the chapter.

5.1 Introduction

In modern microprocessors, the cache has been introduced to store data in order to accelerate their future requests by the processor. To be cost-effective and to enable efficient use of the data, caches are relatively small compared to the RAM. The data can be stored, at the same time, in different memory locations, as shown in Figure 5.1. Thus, the fault in the data can affect any of the memory units containing data, and the fault in the instructions can affect any of the memory units storing the instructions. The effect of these faults and their propagation to the system outputs strongly depend on the interaction between different memory components

during the program execution. Therefore, to have good precision in the hardware fault location, we consider the target memory component (*i.e.*, where the fault occurs). However, our objective is to have a fault evaluation at software level. At this level, the concept of cache and register file is not modeled. Thus, we propose a memory subsystem emulator representing a simplified model of the system components.

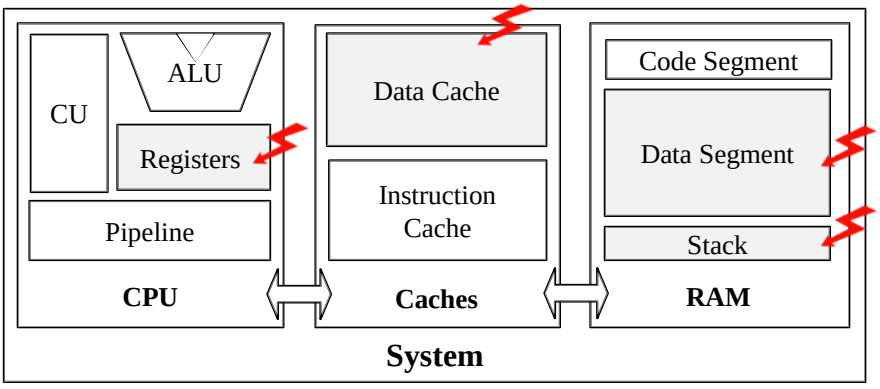

Figure 5.1: Data Location in System

The main advantage of the memory subsystem emulator proposed in this thesis compared to existing system emulator such as Gem5 [91] and SimpleScalar [92], is the independence from the specific microprocessor ISA. The implemented emulator is based on the virtual LLVM ISA, which does not require a predefined hardware architecture. In addition, the emulator supports different hardware configurations, which offers the possibility to perform system evaluation at early design stage of the system.

5.2 Subsystem Emulator

The memory subsystem emulator considers three memory units: the RAM, the caches and the register files, as presented in Figure 5.2. While in this thesis we focus on a single cache layer, the proposed emulator can be straightforwardly scaled to multiple levels of cache. The structure of each unit is designed to be: (i) as close as possible to a real system behavior, and (i) as generic as possible to support different characteristics of different microprocessors. For each component, the subsystem emulator requires a set of hardware configurations to be given as input by the user (Figure 5.2). These parameters are the only link of the proposed emulator with the hardware characteristics. In this thesis, we consider the following system components and characteristics:

- The RAM contains all the active variables and instructions used during the program execution. As input, the tool requires *the RAM size*.
- The cache is a complex queue containing the variables and/or the instructions used during the program execution. It is updated each clock cycle of the program execution when the CPU reads or writes a variable or an instruction. As input, the tool requires *the cache*

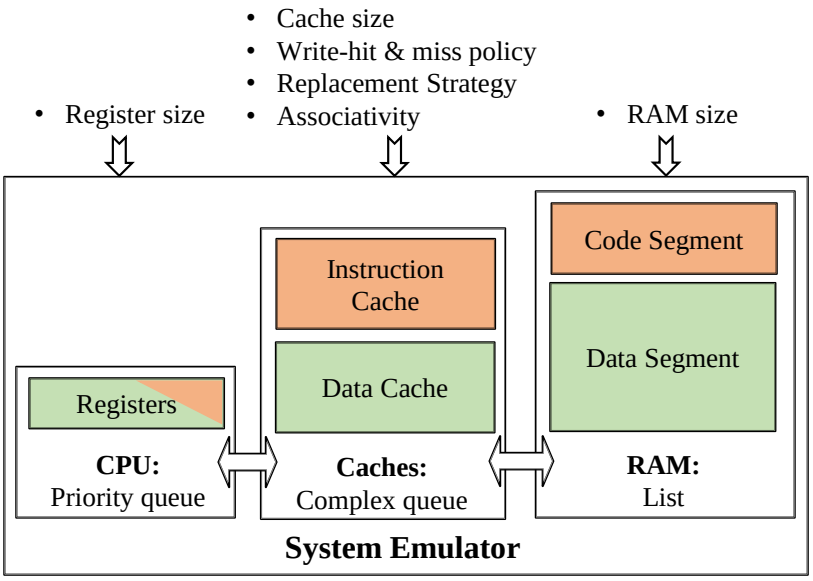

Figure 5.2: System Emulator

size; *the write-miss policy*: write allocation or no-write allocation; *the write-hit policy*: write through or write back; *the replacement strategy*: Least Recently Used (LRU), Least Recently Replaced (LRR), random; and the *associativity*: Fully Associative or Direct Mapped.

- The register files contain, for each clock cycle, a small set of the recently used variables and instructions by the program. As input, the tool requires *the size of the register files*.

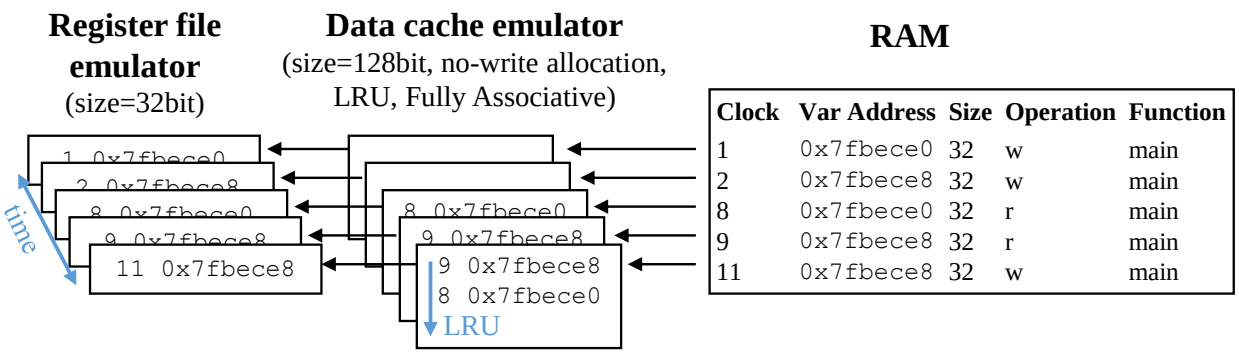

Figure 5.3: Example of Subsystem Emulator

During the program execution, we build for each clock cycle the content of each memory component. In Figure 5.3, we provide an example of a data subsystem emulator with specific configurations. The data cache emulator and the register file emulator are updated each clock cycle when a variable is either written or read. For that, we record traces of the used variables and instructions. The variables' trace contains, for each *read* or *write* operation, information about the clock cycle of the operation, the variable address, the variable size and the function

name, as shown in Figure 5.3. Similar to the variables' trace, the instructions' trace contains for each executed instruction that corresponds to one clock cycle, information about the instruction identifier (an instruction is identified by its line in the LLVM code), the instruction size, and the operation (*read* or *write*). For the first clock cycle, the operation *write* is recorded for all the instructions, since all the program code is loaded to the memory. For the rest of the clock cycles when the program is executing, a *read* operation is recorded for each executed instruction. An example of the instructions' trace is given in Figure 5.4.

Instruction Trace			
Clock cycle	**Instruction ID**	**Size(bit)**	**Operation(w/r)**
0	1. :12.	32	w
1	1.	32	r
2	2.	32	r
3	3.	32	r
4	4.	32	r
5	5.	32	r
6	6.	32	r
7	7.	32	r
8	8.	32	r
9	9.	32	r
10	10.	32	r
11	4.	32	r
12	5.	32	r
13	6.	32	r
14	11.	32	r
15	12.	32	r

Figure 5.4: Instructions' Trace

5.2.1 RAM Emulator

The RAM emulator is implemented as a list of the variables and the instructions. The memory segment storing the data in the RAM is emulated as a list of all the active variables used during the program execution. The memory segment storing the code in the RAM is emulated as a list of all the program instructions. To build, the RAM emulator for the data and the instructions, we collect all the variables and instructions recorded in the traces.

5.2.2 Cache Emulator

The cache emulator is implemented as a complex queue storing the variables or the instructions during the program execution. Figure 5.5 presents the implemented algorithm for the construction of the data and the instruction cache. When an item (*i.e.*, variable or instruction) is read or written, we first check if the item occupies the cache. If it is the case, the content of the item is updated, otherwise the item is added to its corresponding line in the cache. The process of adding the item to the cache depends on the cache configurations. In case of no-write allocation policy, the item is added to the cache only when it is read. However, in case of write allocation, the item is added to the cache both when it is read or written. Then, for

direct-mapped cache, all the items of the block containing the target item are added to the corresponding line in the cache. However, for fully associative cache, the item is loaded to the head of the cache. If the cache does not have free space for the new item, existing items are deleted successively with respect to the replacement mechanism, till enough space for the new item is available.

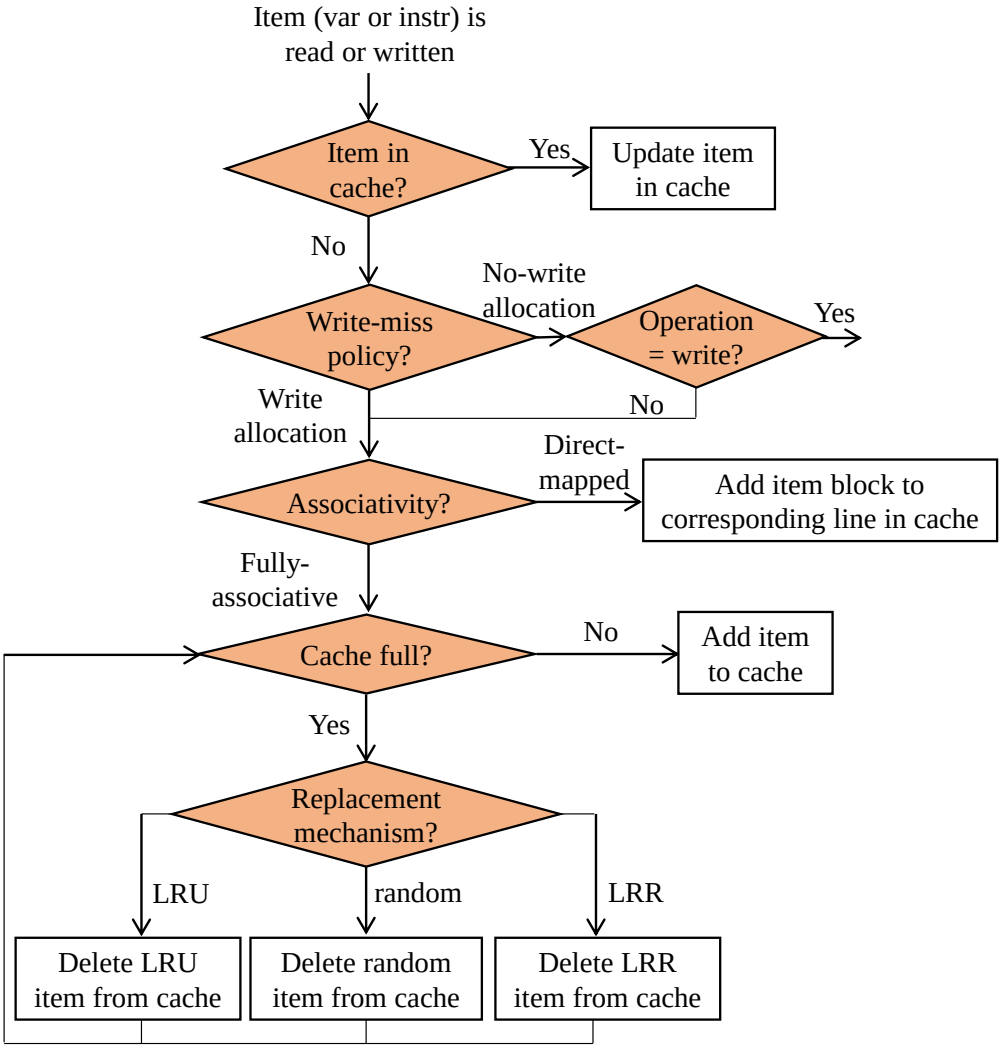

Figure 5.5: Algorithm for Cache Construction

The cache emulator is implemented to be as generic as possible to support different characteristics of different microprocessors. The cache configurations are provided as input to the emulator and are implemented as follows:

- **Size:** It permits to determine the number and the size of lines in the cache.
- **Write-hit policy:** The write-hit policy is considered only for data cache. In case of write-back policy, a *dirty bit* is defined and is set to 1 when the corresponding variable is written in the cache.

- **Write-miss policy:** The write-miss policy is considered only for the data cache since the instructions are written only in the first clock cycle. Two cases are considered: for no-write allocation, the variables are loaded to the cache only when they are read; however for write allocation, the variables are loaded to the cache both when they are read and written.

- **Replacement strategy:** Three replacement strategies are implemented: LRU, LRR and random. For LRU strategy, the cache is implemented as a priority queue. When the cache is full and there is a cache miss, the last recently used item is removed in order to have free lines in the cache. For LRR strategy, the cache is implemented as a complex queue where the older item is replaced in the cache. Finally, for random strategy, a randomly selected item is replaced in the cache. The replacement strategy is considered in case of n-associative caches and for both data and instruction cache.

- **Associativity:** Two types of associativity are implemented in this thesis: the *fully associativity* and the *direct-mapped associativity*. In Figure 5.6, we present the architecture of the fully associativity cache. The cache is implemented as a queue, where each line is 1 byte size. The items are loaded to the end of the queue. Each item can occupy any line in the cache. However, for the direct mapped cache, each item can only occupy one precise line. The index of the line is identified by the address, as presented in Figure 5.7. The size of the line is provided by the user. When the item is read (or written), the corresponding block in the RAM is identified and loaded to the exact line in the cache.

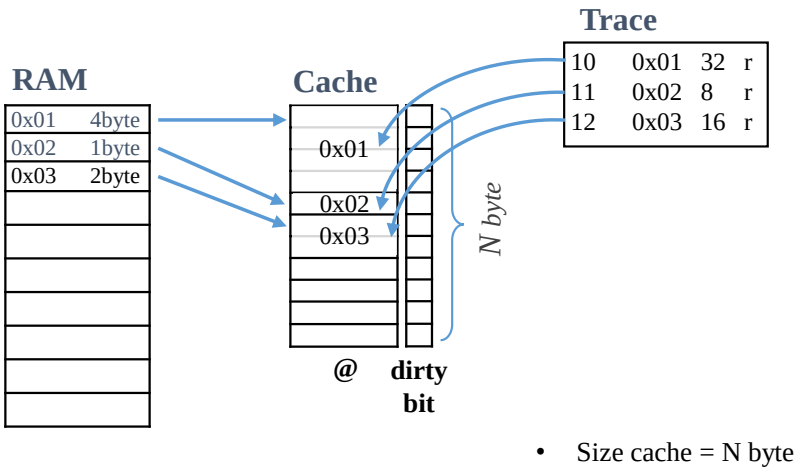

Figure 5.6: Fully-Associative Cache

5.2.3 Register Files Emulator

The register file emulator is implemented as a priority queue. The process of its construction is simpler than the cache emulator. It follows the algorithm presented in Figure 5.8. For each *read* or *write* operation, the item is either updated to the head if it is already in the register, or added to the register head if not. If all the registers are occupied, the existing items are removed successively following the LRU replacement strategy till a place for the new item is available.

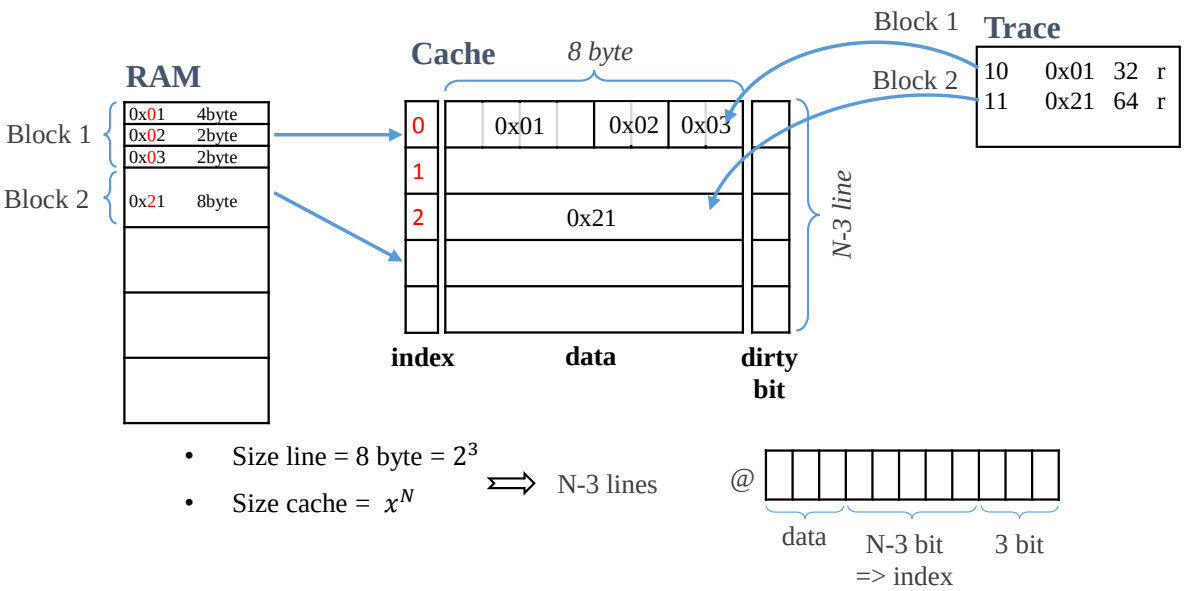

Figure 5.7: Direct-Mapped Cache

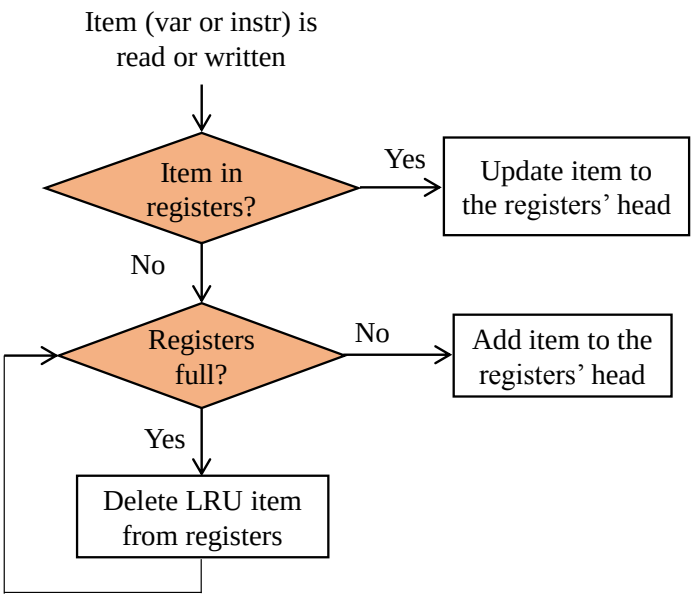

Figure 5.8: Algorithm of Register File Construction

5.3 Validation through Memory Emulator

As shown in Section 4.5, the developed fault injection approaches provide accurate results for simple microprocessors where the cache is dis-activated. The activation of the cache has a big influence on the fault simulation. Thus, in order to adapt the proposed fault injection approaches to modern processors where the cache is considered, we integrate the memory subsystem emulator introduced in Section 5.2.

5.3.1 Emulators' Integration to Fault Injection

The developed memory subsystem emulator is integrated within the LLVM- and the C-based fault injections. We adapt the tools in order to target faults occurring in the RAM and the caches.

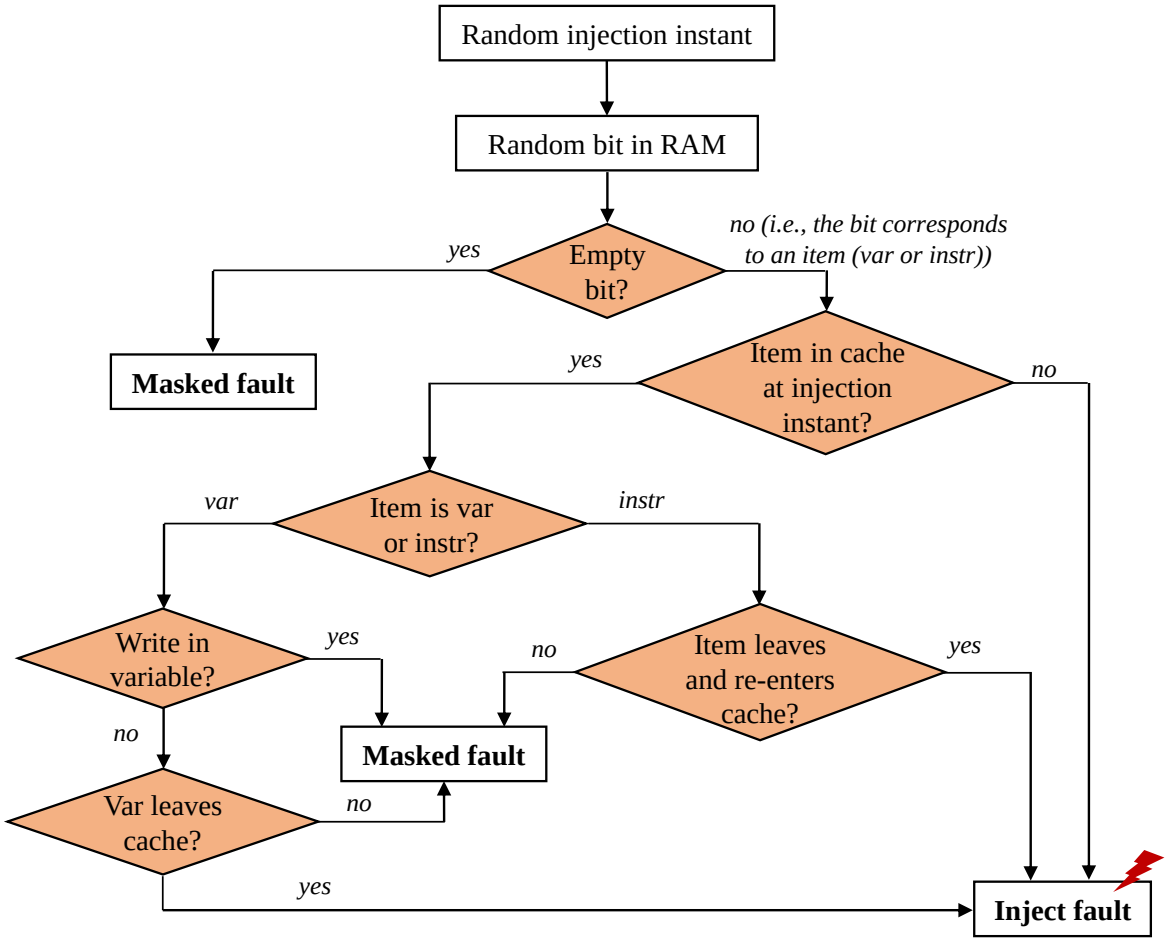

Figure 5.9: Algorithm for Fault Injection in RAM

5.3.1.1 Fault Injection in RAM

To inject faults in the RAM, we develop the algorithm provided in Figure 5.9. First, we check if the faulty bit resides in some unused memory area in the RAM. In that case, the fault is

classified as masked. Otherwise, we verify using the cache emulator whether the corresponding item resides in the cache at the injection instant. The fault injected in an item residing in the cache at this instant does not have immediate influence on the final program output. Therefore, we do not immediately inject the fault but we follow the behavior of the item in the cache.

In case the fault injection is targeting the data, we follow the *write* operations performed on the variable. If the faulty variable is written before leaving the data cache, the correct value of the variable is written back to the RAM, either immediately for write-through policy or when leaving the cache for write-back policy. Then, we do not perform the fault injection and we classify the fault as masked. Nevertheless, we inject the fault when the variable (i) does not reside in the cache at the injection instant, or (ii) resides in the cache at the injection instant then leaves without being written after the injection instant and before leaving the cache.

In case the fault injection targets instructions, we follow the residence of the instruction after the injection instant. If it leaves the cache and re-enters it, then we perform the fault injection, otherwise we classify the fault as masked.

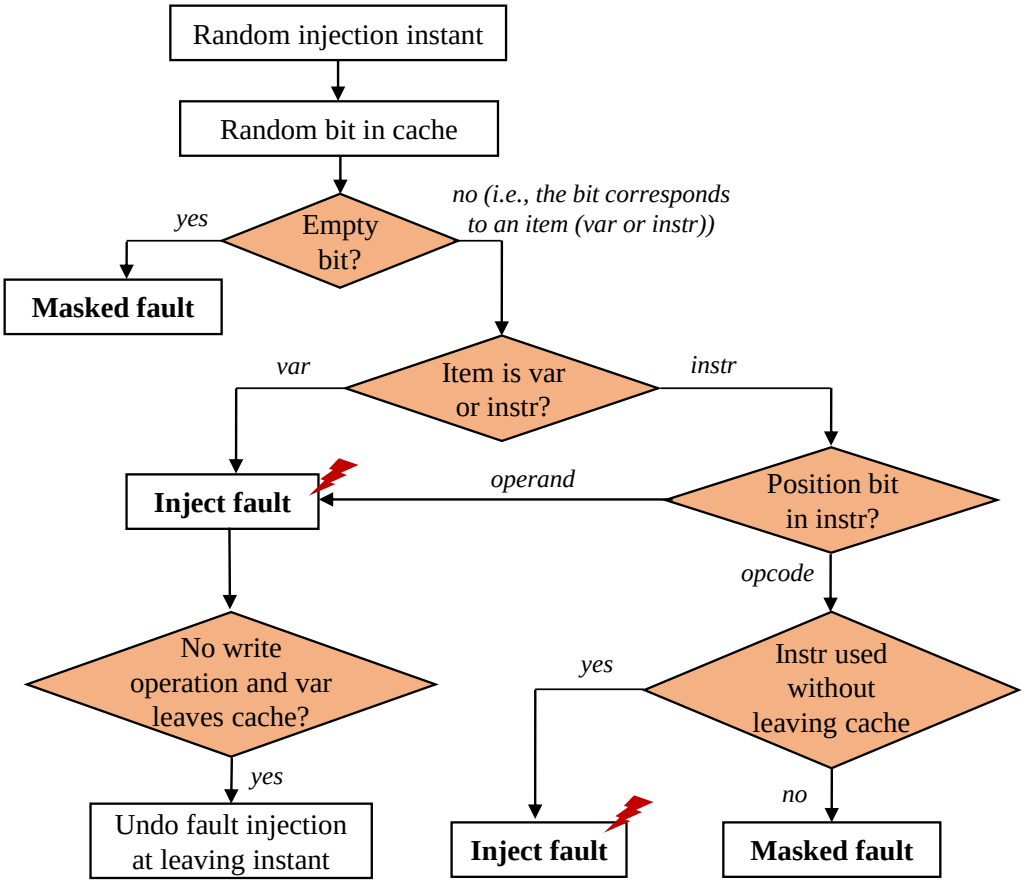

Figure 5.10: Algorithm for Fault Injection in Cache

5.3.1.2 Fault Injection in Cache

For fault injection in the cache, we develop the algorithm presented in Figure 5.10. First, we check if the faulty bit resides in some unused memory space of the cache. In that case, we

classify the fault as masked. However, if the faulty bit corresponds to a variable or an instruction residing in the cache, we consider several cases.

In case the fault injection targets data cache, we inject fault in the variable. Then, we follow its residence in the cache. If the variable leaves the data cache without being written, the correct variable is reloaded from the RAM in later CPU request and the faulty value is overwritten. To consider this case, we undo the fault injection at the leaving instant. However, if the faulty variable is written after the injection instant and before leaving the data cache, the corrupted variable is written back to the RAM, either immediately for write-through policy or when leaving the cache for write-back policy. Thus, we do not have to undo the fault injection.

In case the fault injection targets instruction cache, we check if the bit position corresponds to an opcode. Then, we follow the instruction use after the injection instant. If it is used without leaving the cache, we perform the fault injection, otherwise we classify the fault as masked.

5.3.2 Experimental Results

In this subsection, we present the experimental results of the fault injections using LLVM- and C-based methods with the integration of the memory subsystem emulator. We target faults in the RAM, the data cache and the instruction cache. The results are compared to the results of the FPGA-based fault injection presented in Subsection 4.5.2. The comparison evaluates the developed approaches in term of accuracy and simulation time.

5.3.2.1 Experiments' Setup

For the experiments, we set up a list of benchmarks that have different execution times and memory utilizations, and cover both data-intensive and control-intensive algorithms. We select a set of workloads from the open-source benchmark suite MiBench [93] (bit count, quick sort, string search, fft, crc 32).

For the three methods, we run 10K fault injections per program that provide a margin of error of 1% with a confidence level of 95%, as discussed in Subsection 4.5.3.

Finally, we set up the system configurations for the LEON3 processor, and we provide them as input to the memory subsystem emulator:

- *RAM*: 256 KB size.
- *Data Cache*: 4 KB size, write-through for the write-hit policy, no-write allocation for the write-miss policy, LRU for the replacement strategy and fully associative for cache associativity.
- *Instruction Cache*: 4 KB size, LRU for the replacement strategy and fully associative for cache associativity.
- *Register Files*: 512 B size.

5.3.2.2 LLVM-based Fault Injection

- **Accuracy Evaluation**

Figure 5.11 and 5.12 provide the percentage of masked faults injected respectively in the data cache and the instruction cache, using the LLVM-based fault injection compared to those obtained using the FPGA-based fault injection. The results of the LLVM-based approach are very close to those of the FPGA-based fault injection. The absolute difference is 1.64% for faults occurring in the data cache, and 2.12% for faults occurring in the instruction cache. This proves that the proposed LLVM-based fault injection allows to accurately evaluate the effect of faults occurring in the data and the instruction cache.

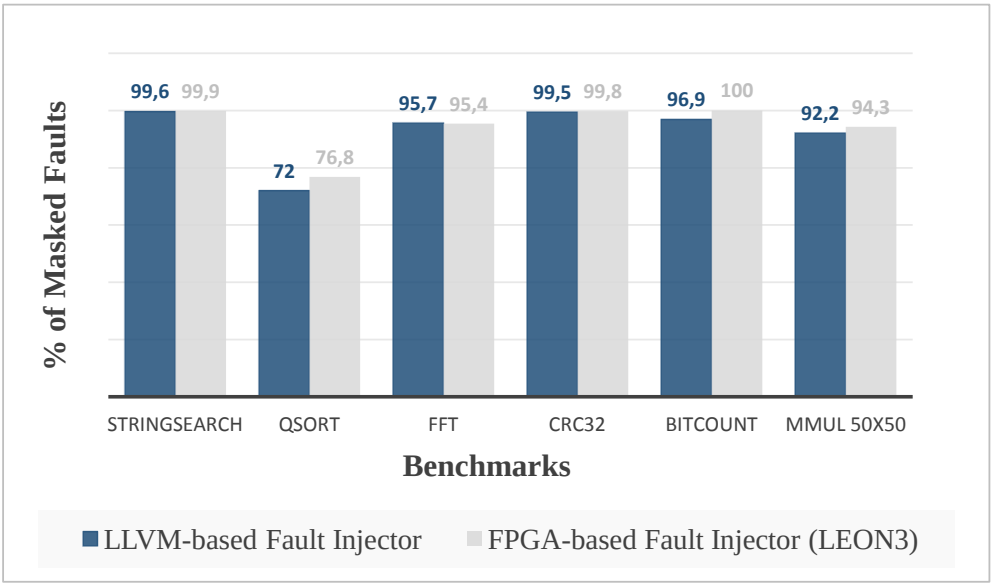

Figure 5.11: Masked Faults Injected in Data Cache using LLVM- and FPGA-based Fault Injection

- **Simulation Time Evaluation**

Table 5.1 presents the percentage of faults evaluated with the analytical process, *i.e.*, without going through the fault injection process. Depending on the occupied memory space in the cache, the percentage of faults evaluated without performing fault injection is high for some workloads (>80%). This analytical process performed before the fault injection process permits to reduce the number of performed fault injections. It allows to minimize the cost of running exhaustive number of fault injection on the same workload, which represents the main disadvantage of a full fault injection method.

Table 5.1: Faults Evaluated with the Analytical Process.

Workload	StringSearch	Qsort	FFT	CRC32	BitCount
Faults in Data	9.5%	1.1%	1.6%	2.9%	95.9%
Faults in Instruction	83.2%	32.9%	41.1%	97.5%	96.5

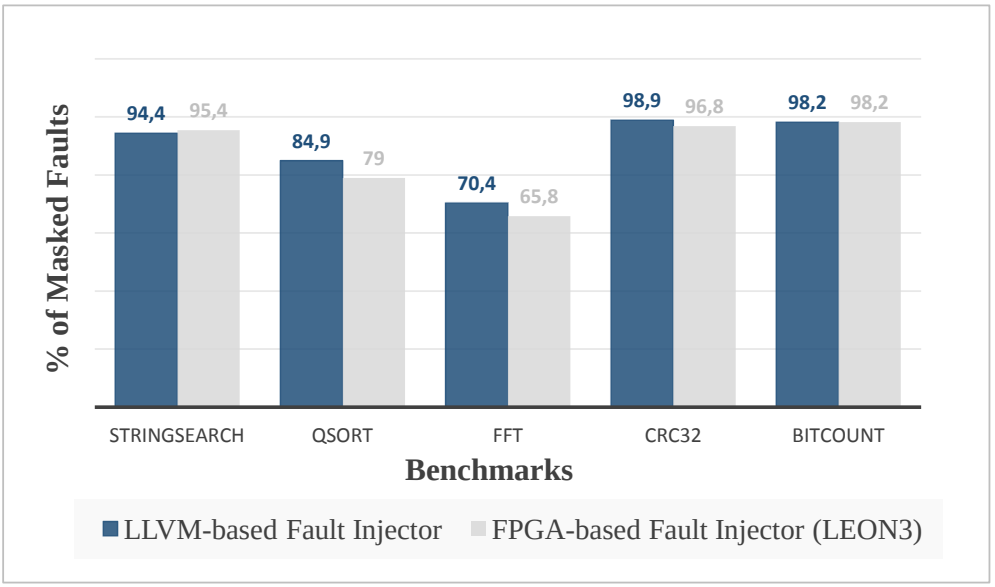

Figure 5.12: Masked Faults Injected in Instruction Cache using LLVM- and FPGA-based Fault Injection

In addition to the time saved by the analysis process, the fault injection mechanism offers as well a significant gain in the execution time. In fact, the faults are injected inside the source code of the program. Thus, the simulation time of the faulty program is the simulation time of the golden program because the mutation introduced to simulate the fault does not have influence on the program execution. For all the used benchmarks, the time of the golden execution is about 1s. Using the FPGA-based fault injection, a fault requires 10.8s to be injected. However, using the propose LLVM-based fault injection, a fault needs the same time of the golden execution. This means that the speed-up is about 10x for one simulation. It is important to mention that the used FPGA-based fault injection technique is about 3 to 4 orders of magnitude faster than a standard simulation-based fault injection.

5.3.2.3 C-based Fault Injection

- **Accuracy Evaluation**

Figure 5.13 and 5.14 provide the percentage of masked faults injected respectively in the RAM and the data cache, using the C-based fault injection compared with those obtained using the FPGA-based fault injection. The results of the C-based approach are very close to those of the FPGA-based fault injection. The absolute difference is 2.3% for faults in the RAM, and 1.4% for faults in data cache. This proves that the C-based fault injection allows to accurately evaluate the effect of faults occurring in different memory components of the system, such as the data cache and the RAM.

- **Simulation Time Evaluation**

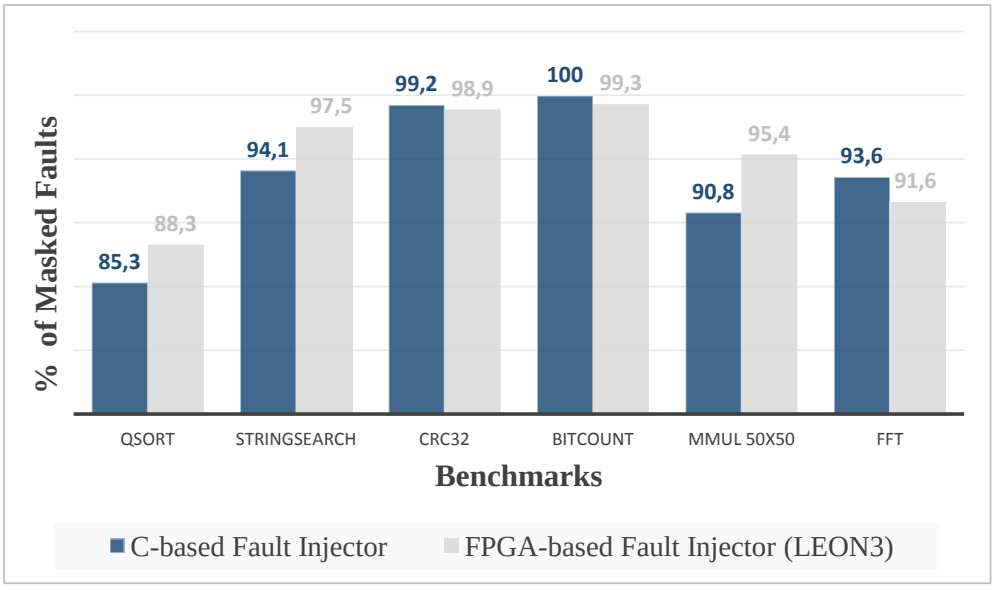

Figure 5.13: Masked Faults Injected in RAM using C- and FPGA-based Fault Injection

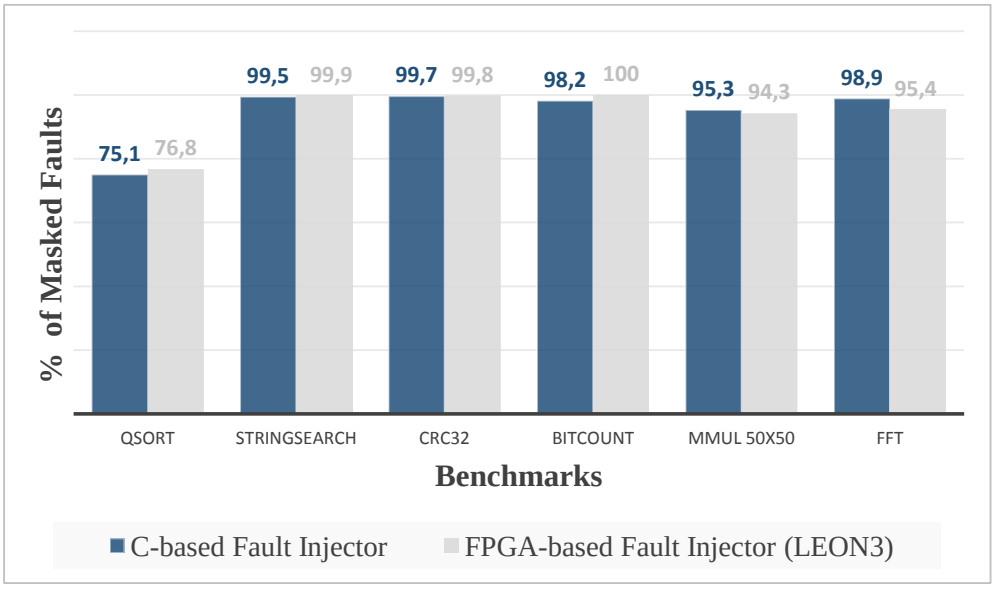

Figure 5.14: Masked Faults Injected in Data Cache using C- and FPGA-based Fault Injection

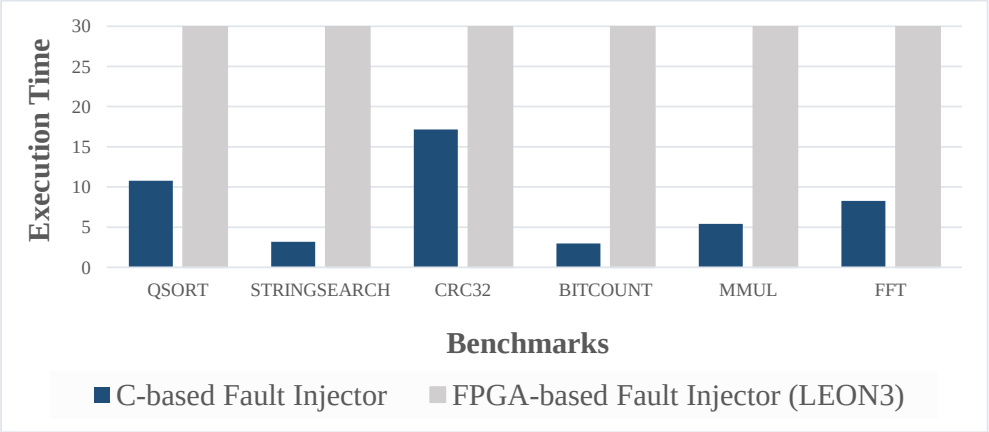

Figure 5.15: Execution Time (in hours) of C- and FPGA-based Fault Injections for Faults in Data Cache

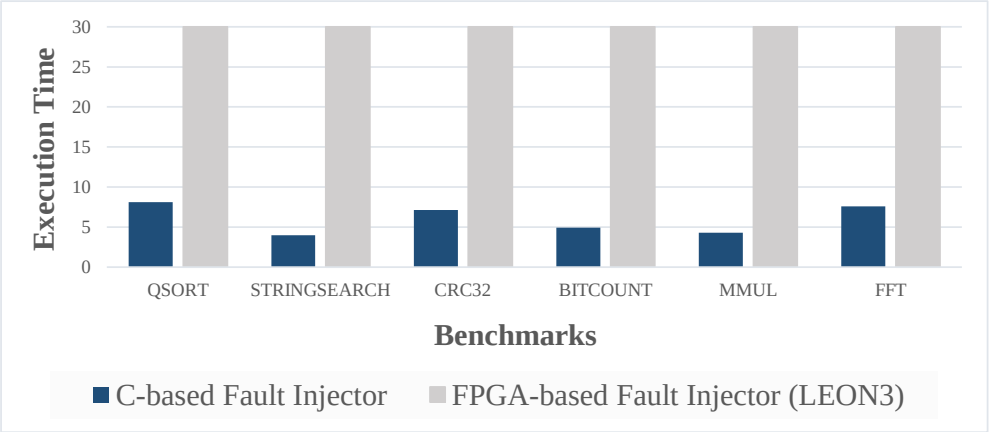

Figure 5.16: Execution Time (in hours) of C- and FPGA-based Fault Injections for Faults in RAM

Figure 5.16 and 5.15 provide a comparison of the execution time between the experiments run with the C-based and the FPGA-based fault injection for faults occurring respectively in the RAM and in the data cache. The comparison shows that the C-based approach offers a significant gain in the execution time. On average, the speed-up is 5x for the faults occurring in RAM, and 6x for faults occurring in data cache compared to the FPGA-based fault injection.

5.4 Conclusions

The emulation of the memory subsystem represents an important contribution in this thesis. It allows to emulate the behavior of the RAM, the caches and the register files at software level, without requiring a fully defined hardware architecture. As input, the emulator requires hardware configurations that are generic and easy to provide by the user.

The integration of implemented memory emulator within the proposed fault injection environments proves a significant improvement in the fault evaluation accuracy. On average, it reduces the absolute difference, for the C-based fault injection, from 58% to 2.3% for the RAM, and to 1.4% for the data cache.

To conclude, the contributions of the thesis presented so far in this part allows to accurately evaluate the reliability of computer-based systems that run complex microprocessors at different design stage.

Part III

Reliability Evaluation through Lifetime Analysis

Introduction

The results obtained so far show that the proposed fault injection environments allow to accurately evaluate the reliability of a computing system in presence of faults. The presented approaches use fault injection technique to evaluate the reliability of a given application. Even if the fault injection is executed at pure software level and does not require the presence of the hardware under evaluation, thus reducing the execution time of several order of magnitudes compared to hardware-based fault injection, it may take long times for complex applications (i.e., real industrial application) since it requires computing a big number of simulations (at least 10K to have statistically significant results) for the same workload.

To further reduce the simulation time required by the fault injection methods, we propose an analytical approach based on lifetime analysis. The proposed approach allows to evaluate the outcome of the software when affected by faults without performing long fault injection campaigns. The proposed method makes use of the concept of the lifetime and residence of variables and instructions, in order to compute the minimum percentage of masked faults. This approach provides accurate results in term of reliability evaluation of complex software application. In addition, it offers a huge save in the simulation time and energy. These advantages are proved through a comparison to accurate hardware-based fault injection. The proposed approach has been also applied on real industrial case study.

Since experimental results demonstrate that the proposed approach provides excellent results (in terms of accuracy), at very low cost (in terms of execution time), we also use it to perform fast and flexible memory-aware design space exploration in order to evaluate the effects of the faults occurring in the RAM and the caches of a computing system. The objective of this study is to evaluate the vulnerability of different system components regarding different system configurations, and by consequence comprehensively support the reliability decision-making process in computing system.

The reminder of this part is structured as follows. In Chapter 6, we present the state-of-the-art on: (i) analytical approaches for reliability evaluation, and (ii) methods for cache design space exploration to evaluate the reliability as well as the processor performance. In Chapter 7, we introduce the proposed analytical approach to evaluate the reliability of software application. We validate the accuracy of our approach through experiments on different benchmarks. We also apply the approach on real industrial case study. In Chapter 8, we present the methodology of the memory-aware design space exploration to evaluate the system reliability.

Chapter 6

State of the Art

Contents

6.1 Analytical Reliability Evaluation **70**

6.1.1 AVF Computation 70

6.1.2 Program Analysis 72

6.2 Cache Design Space Exploration **73**

6.2.1 DSE for Performance 74

6.2.2 DSE for Reliability 74

6.3 Conclusion **75**

In this chapter, we present the state-of-the-art of the analytical methods to evaluate the system reliability. We also discuss existing works that study the cache design space exploration with target to the system reliability and/or the processor performance.

6.1 Analytical Reliability Evaluation

6.1.1 AVF Computation

For computer-based system that are running software applications, soft errors become a key challenge in microprocessor design. As the largest structure in the processor, the memory elements are most susceptible to soft errors. Several techniques dealing with this type of faults exist in the literature, as discussed in Chapter 3. However, most of them are quiet costly in terms of time, performance, and energy. Designers require accurate system reliability evaluation while making appropriate cost/reliability trade-offs. Recently, some researches propose analytical methods to deal with soft errors. They are based on the assumption that not all soft errors affect the final program output and that many errors can be masked at different levels.

Mukherjee et al. [6] propose an approach allowing to quantify the error rate of a system. They assume that a transient fault affecting the system component can be masked at the architectural level. As consequence, the fault does not affect the final program output. The motivation behind this work is to deal with soft errors and provide accurate estimations of processor error

rates while making appropriate cost/reliability trade-offs. The authors introduce the Architectural Vulnerability Factor (AVF) as the probability that a fault affecting a component in the microprocessor produces a visible error in the final program output. The authors use a novel approach that tracks the subset of processor state bits required for Architecturally Correct Execution (ACE). Analytical method based on the concept of ACE divides the bits' lifetime into ACE and unnecessary for ACE (un-ACE) intervals. A bit in ACE interval is necessary for the correct program execution. A fault in that bit produces a visible error in the program output. However, a bit in the un-ACE interval is the bit which its corruption does not have influence on the final program output. The authors consider that any interval that cannot be proven as un-ACE is considered to be ACE. They identify the un-ACE bits at both the architectural and micro-architectural levels. The methodology is used to compute the AVF for the instruction queue and execution units of an Itanium ®2-like microprocessor. The AVF of a hardware structure is defined as the probability that a fault in that structure results in an error. It is computed using the Equation 6.1.

$$AVF\ of\ Structure = \frac{\sum Residency\ (in\ cycles)\ of\ ACE\ Bits\ in\ Structure}{Total\ Number\ Bits\ in\ Structure \times Total\ Execution\ Cycles} \tag{6.1}$$

The structure's error rate is the product of its raw error rate and the AVF. This AVF estimations help the designers of microprocessors to estimate the Failures In Time (FIT) rate of an entire processor at early design cycle. If the processor does not achieve the target FIT rate, then these estimations support designers to apply suitable error detection or correction mechanisms in order to design structures that are less vulnerable to single-bit upsets.

Biswas et al. [94] apply the lifetime concept to compute the AVF of address-based structures using level-one write through data cache, data translation buffer, and store buffer, each with distinctive hardware characteristics. The authors perform a detailed breaking down of the lifetime of bits' components into ACE, un-ACE, and unknown components, and compute the AVF for the data arrays. The analysis of a detailed Itanium ®2-like IA64 processor simulator shows best estimate AVFs for the data arrays of the three structures. Then, the lifetime analysis is extended to compute the AVF of the tag arrays. Furthermore, the authors use the lifetime analysis framework to show how two AVF reduction techniques (the periodic flushing and the incremental scrubbing) reduce the AVF by converting ACE lifetime components into un-ACE without affecting the system performance.

Montesinos et al. [95] use the register lifetime to propose a technique that protects register files against soft errors. They use the Error Correcting Codes (ECC) to detect and correct the errors in the register files. The full ECC support enables on-the-fly detection and correction of errors. However, it is costly in terms of power and performance. In order to design a cost-effective protection mechanism, the authors consider that the data in the registers are only useful for a small fraction of the registers' lifetime, and that not all registers are equally vulnerable. Thus, they protect the subset of the registers by generating, storing, and checking the ECC of only the most vulnerable registers. The vulnerability is identified based on the register lifetime. The authors define the lifetime of the register from its allocation to deallocation. The useful period is from the write until the last read, while the non-useful period is from allocation until the write and from last read to deallocation.

Similar to this methodology but targeting cache, Ma et al. [96] propose to improve the computation of the cache AVF. The AVF computation is based on the lifetime analysis, and

is divided into ACE, un-ACE and un-known components. The study characterizes dynamic vulnerability behavior of L1 data cache, and detects the correlations between the L1 cache AVF and various performance metrics. Based on that, the authors develop an AVF-aware ECC technique. They demonstrate that this technique provides gain in the performance and energy while maintaining a good processor reliability.

All the proposed techniques have been proposed in the context of fault tolerance. They aim to propose correction and detection techniques while making tradeoffs between reliability and performance. Based on the ACE analysis methodology, existing technique can quickly estimate the vulnerability factor of different microprocessor structures. They can be used early in the design stage. However, the lack of implementation details of the structures makes the techniques inaccurate and does not provide exact reliability evaluation of memory system components. In addition, these techniques do not consider the behavior of the software application running on the system in terms of its use to fault tolerance techniques, since they do not differentiate between hardware fault masking and software fault masking.

Recent studies demonstrate that analytical techniques provide limited accuracy as they use simplified models of error propagation to expedite the soft error rate estimation. George et al. [97] demonstrate using rigorous fault injection experiments that the ACE analysis overestimates the vulnerability of some structures up to 7x. In fact, the ACE analysis cannot consider error masking within complex logic structures. Therefore, this technique is mostly suitable for regular structures (*i.e.*, address-based structures) such as cache, register file, and reorder buffer in microprocessors. Moreover, the ACE analysis neglects circuit-level masking factors such as electrical and timing masking. Ebrahimi et al. [98] propose an approach to compute the soft error vulnerability of the entire microprocessor system consisting of regular and irregular structures. This hybrid approach uses the error propagation probability analysis to model the propagation of errors in logic structures (*i.e.*, combinational and sequential elements) and the ACE analysis to compute the vulnerability of memory components. The evaluation of the OR1200 processor reveals that the inaccuracy of this technique is less than 7% compared to simulation-based statistical fault injection for individual cells while it is five order of magnitudes faster than fault injection.

In conclusion, existing analytical studies based on AVF computation demonstrate that they overcome the cost challenge in terms of time and energy when estimating the structure vulnerability. However, the remaining challenge of analytical technique is the accuracy of the reliability evaluation compared to fault injection techniques.

6.1.2 Program Analysis

Analytical techniques based on ACE allow researchers to quantify the AVF of hardware structures, and thus to understand their vulnerability to soft errors in order to consider design tradeoffs when running specific workloads. AVF was not only applicable to hardware, but it has been also used to propose software analysis techniques. Quantifying vulnerability to hardware faults at software level provides better understanding of the program reliability independently of the micro-architecture on which it is executed.

Sridharan et Kaeli [99] adapt the technique of ACE Analysis to develop new software-level vulnerability metric called Program Vulnerability Factor (PVF). It allows to evaluate the software behavior affected by transient faults without considering the hardware architecture. The PVF computation is based on the analysis of the instruction flow during the execution. For

a software resource R with size B_R, its PVF over I instructions is computed using Equation 6.2.

$$PVF_R = \frac{\sum_I ACE\ Bits\ in\ R}{B_R \times\ I} \tag{6.2}$$

The authors of this work do not consider the influence of hardware components on the computation of the software vulnerability. This presents the main drawback of the work. In addition, it does not study the functional units and the memory.

Restrepo-Calle et al. [100] propose a metric to estimate the register file criticality in processor-based systems. The method is based on dynamic code analysis. It provides fine-grained analysis of each register criticality using the assembly source code of the program. The applicability and the accuracy of the method have been evaluated in a set of applications running in the miniMIPS and the PicoBlaze microprocessors. This work is inspired from the work proposed by Bergaoui et al. [101]. The authors in [101] propose a refined evaluation of variable and register criticalities, taking into account three criteria: the lifetime, the weight in conditional branches, and the functional dependencies. This approach aims to better classify at compile time the criticality of the registers in a given microprocessor and for a given software application.

Lee et al. [102] present static analysis approach for Register File Vulnerability (RFV) estimation. They breakdown the vulnerability of registers into intrinsic and conditional basic-block vulnerabilities. This decomposition allows to quickly compute the RFV. The authors demonstrate the practical application to compiler optimizations in order to achieve very cost-effective protection of register files against soft errors.

In opposite to the analytical methods based on AVF computation that consider only the micro-architecture of the hardware, the program analysis methods previously presented consider only the software application. The program analysis methods are not accurate enough since the vulnerability depends on the specificity of the micro-architecture.

6.2 Cache Design Space Exploration

Design Space Exploration (DSE) is defined as the activity of exploring various design alternatives preceeding the system implementation. The advantage of operating on the space of potential design candidates makes DSE useful for many engineering tasks, including rapid prototyping, optimization, and system integration. However, the size of the explored design space is very big, thus the main disadvantage is the huge number of the system parameter combinations that have to be studied. Thus, the challenge consists in exploring the design space in a cost-effective way.

In computing-based systems, DSE is mainly applied in cache. In fact, cache memories are widely used in modern microprocessors in order to improve the system performance since they store frequently accessed data and instructions to avoid the large number of costly RAM accesses. In addition to the system performance, cache memories have a big impact on the system reliability. They are most susceptible to soft errors since they are close in the same time to the main memory where the faulty data can be stored back, and to the CPU where the data is used for execution [103]. Based on this two criteria, existing works that study the cache DSE target either the system performance or the system reliability.

6.2.1 DSE for Performance

Several existing works use cache DSE to evaluate the microprocessor performance, the energy and the power consumption [104]. Some studied are based on analytical methods which provide gain in the execution time but not enough accuracy evaluation. The others are simulation-based techniques which are accurate but costly and slow.

Ghosh et Givargis [105] introduce analytical cache DSE for embedded systems that avoids exhaustive simulation. They propose an efficient algorithm to compute cache parameters satisfying the designers' performance constraint. The studied cache design space targets the cache size and the degree of associativity. Liang et Mitra [106] [107] propose a static analysis technique for rapid and accurate design space exploration of instruction caches. As cache configurations, they change the number of cache sets and the type of associativity in order to observe the impact on the cache hit rate.

In [108] and [109], the authors propose to find out, through simulation-based approach, the cache and register file size for embedded application that fits with the optimum cache performance and power consumption. They show that although having bigger cache and register file is one of the performance improvement approaches in embedded processors. However, by increasing the size of these parameters over a certain threshold level, the performance improvement is saturated and decreased.

In order to decrease the time cost of exhaustive simulation for cache DSE, tow solutions are proposed in the literature. Patel et Rajawat [110] propose an hybrid method combining both static analytical and simulation-based to speed up the design space exploration for instruction cache. This technique offers an average speed up of 10x over exhaustive simulations for the L1 cache. This technique helps to find out the cache size which saturates the performance. Li et Negi [111] discuss a time and space efficient technique for simulating a compressed trace against multiple cache configurations. They show that the use of compressed traces provides important gain in the simulation time. The target parameters of the design space considered by the two last techniques, are the number of sets, the size and number of lines, and the degree of associativity, however they do not consider the replacement strategy and cache policy.

To conclude, the previously presented works propose different approaches that aim to observe the impact of changing the cache configurations, such as the size and the associativity, on the microprocessor performance and power consumption.

6.2.2 DSE for Reliability

Only very few works use cache DSE to evaluate the system reliability. Cai et al. [112] examine the combined effect of the cache size selection on the system reliability, as well as the energy consumption and the performance. They perform extensive fault injection simulations using a cycle-accurate ARM7 microprocessor simulator, which is extremely time consuming. They find out that a careful cache size selection is needed, in order to found out the optimal energy, performance and reliability of the system.

Maghsoudloo et al. [113] present DSE of Non-Uniform Cache Access (NUCA) in many-core processors. They propose to find the best composition of different NUCA specifications that meets the desired level of reliability without violating the performance, the cache energy consumption, and the interconnection traffic. They make use of analytical method to measure the temporal vulnerability factor. They aim to find the best structure for each cache platform

which minimizes the vulnerability of L1 and L2 caches in NUCA architectures. The explored design space consists of 72 implementations, made up by combining different configurations in the current NUCA specifications. As cache configurations, they play on the cache organization, the write-hit policy, the coherence protocol, the inclusiveness, the replacement strategy, and the network topology. However, they do not target the impact of the cache size, the line size, and the cache associativity.

Wang et al. [114] propose models to facilitate the characterization of the vulnerability of lifetime phases and the optimization of the most vulnerable phases. They study the impact of some cache configurations on reducing the vulnerability of these phases. They target both the data and the instruction cache. As configurations, they play on the cache line size, the different granularities, and the write-hit policy. However, they do not observe the impact of the cache size, the cache associativity, the number of blocks, entries and cycles. The aim of this work is to use DSE to propose schemes to optimize the cache reliability against soft errors. The used platform to perform experiments is a simulator modeling a high performance microprocessor similar to Alpha 21364.

These works reveal some drawbacks for the cache DSE to evaluate the system reliability. The work proposed in [112] is based on fault injection simulations. Together with the huge number of cache configurations required by the DSE process, the number of simulations extremely increases which makes the study very complex in terms of time and energy consuming. Thus, this method is not applied when we want to target other cache configurations other different from the size. In [113] and [114], they make use of analytical methods. To compute the vulnerability factor, which is first introduced in [6], they make used of the concept of lifetime for a given instruction or variable, without considering its residence. This makes the reliability evaluation less accurate compared to fault injections. In addition, the existing work use a simulation-based environment on a fixed microprocessor ISA. This makes the study strongly dependent from the hardware architecture and does not allow a reliability evaluation at early design stage of the systems, which is the main issue of this thesis. Finally, existing works do not target a whole cache DSE. They target only some cache configurations. Other caches configurations might have impact on the system reliability and should be explored.

6.3 Conclusion

Existing analytical studies based on AVF computation allow to estimate the structure vulnerability while having time/energy trad-offs. However, the analytical technique are not able to accurately evaluate the system reliability as for fault injection techniques.

In opposite to the analytical methods that are based on AVF computation and that consider only the micro-architecture of the hardware, the program analysis methods consider only the software application. Thus, they are inaccurate since the vulnerability depends on the specificity of the micro-architecture.

Chapter 7

Analytical Reliability Evaluation

Contents

7.1 Introduction **77**
7.2 Lifetime Analysis **78**
7.2.1 Variable Lifetime 78
7.2.2 Instruction Lifetime 79
7.2.3 Fault Classification 81
7.2.4 Lifetime Validation 82
7.3 Fault Analysis **83**
7.3.1 Faults in Data 83
7.3.2 Faults in Instructions 85
7.4 Validation **86**
7.5 Industrial Case Study **87**
7.5.1 Flight Management System 88
7.5.2 Application Analysis 88
7.6 Conclusions **91**

In this chapter, we introduce a novel analytical methodology allowing to evaluate the effect of hardware faults on the system behavior. It is based on the analysis of both the lifetime and residence of the variables and the instructions. The proposed method is accurate, fast, and flexible. It is used to evaluate the reliability of different system components such as the RAM, the data cache and the instruction cache. The proposed approach has been validated through comparisons to hardware-based fault injections and has been applied on real industrial case studies.

Section 7.1 presents the motivation and the advantages of the proposed approach. Section 7.2 introduces the concept of lifetime for both variables and instructions, and provides experimental results to validate the lifetime analysis. Section 7.3 explains the implemented algorithms to analyze the fault based on the lifetime and the residence of the variables and instructions. Section 7.4 provides the experimental results to validate the accuracy of the method using a

comparison to FPGA-based fault injection. Section 7.5 presents the application of the proposed method on real industrial application which is a flight management system. Finally, Section 7.6 concludes the chapter.

7.1 Introduction

The most used methods to evaluate the system reliability are based on the fault injection technique. Even if the fault-injection is executed at pure software level, as the approaches proposed in Chapter 4, thus reducing the execution time of several order of magnitudes compared to hardware-based fault injection approaches, it may take long times for complex applications to perform big number of fault simulations. To further reduce the required simulation time, we propose in this Chapter an analytical approach that permits to evaluate the reliability of the system without performing long fault injection campaigns as usually used by existing approaches in the literature.

The proposed method allows to evaluate the outcome of the software when a single fault affects its data or instructions. We use the concept of the variable or instruction lifetime and residence to compute the percentage of masked faults. The approach is based on the LLVM Virtual ISA, which offers good accuracy in the results and more independence from the hardware architecture of the system.

Furthermore, to achieve a better characterization, we must follow a holistic approach when targeting the software layer. Thus we take into account the presence of the cache and the register files. Since we are working at high level, where the concept of caches and register files are not modeled, we use the memory subsystem emulator, presented in Chapter 5, to represent a simplified model of memory units. We emulate the behavior of different system components: the RAM, the caches, and the register files.

The main advantages of the proposed approach are: (i) it does not require a fully predefined hardware, the use of LLVM Virtual ISA permits to be independent of the hardware architecture; (ii) it accurately evaluates the faults occurring in data and instructions of the software, and considers the different system components such as the RAM, the caches and the register files, the integration of the memory subsystem emulator allows to model these memory components at software level; (iii) it significantly reduces the simulation time compared to the standard hardware reliability evaluation techniques such as fault injection techniques, the proposed method requires only one execution of the target application.

Compared to existing techniques presented in the state of the art, the proposed approach reveals many differences. First, all existing methods are strongly dependent on a predefined hardware architecture and they are applied on real processor. This can not evaluate the system reliability at early design stage. The proposed approach is based on LLVM, the virtual ISA. Thus, it does not require a fully defined hardware architecture. In addition, existing techniques propose correction and detection methods while making trade-offs between reliability and performance. They do not provide exact reliability evaluation for memory system components such as RAM and caches, which is the goal of this contribution. Finally, existing analytical approaches show that they overcome the cost challenge in terms of time and energy when evaluating the system reliability. However, the remaining challenge is the accuracy of the analytical reliability evaluation compared to fault injection techniques, as we previously discussed in Section 6.1. The proposed technique offers a good accuracy in reliability evaluation compared to

the FPGA-based fault injection on the LEON3 microprocessor that is presented in Subsection 4.5.2. The comparison of the two methods shows a similarity in the results, which proves the effectiveness and the efficiency of the proposed approach.

7.2 Lifetime Analysis

7.2.1 Variable Lifetime

At software level, the data is represented by the program variables. During the program execution, a variable can be read or written. We define the variable lifetime from the first write (followed by a read) to the last read (before the next write or the ending of the program), otherwise the variable is dead, as shown in Figure 7.1. When the variable is dead, its content is irrelevant to the correct program execution since it will be either re-written or never used again. Thus any fault affecting this variable is masked.

Figure 7.1: Variable Lifetime

In Figure 7.2, we present the steps of the lifetime computation of the data. Starting from the original source code written in any programming language (or possibly the binary code), we generate the corresponding LLVM intermediate representation (Figure 7.2.a). Since LLVM does not provide information about the exact timing when an instruction is executed, we instrument the original code by adding the information of the current clock cycle of the executed instruction (we consider that each LLVM instruction is executed in one cycle). In particular, we use a counter that is incremented after each instruction execution (Figure 7.2.b). We also instrument the code in order to log out information about the read and the written variables. For the *write* operation, we follow the *'store'* or the *'alloca'* LLVM instructions. For the *read* operation, we follow the *'load'* LLVM instruction (Figure 7.2.b). Then, we execute the program and we record a trace of the variables. The trace contains information about each read and written variable during the program execution. In particular, we record, for each *write* or *read* operation, the corresponding clock cycle, the physical address of the variable, the operation type (*read* or *write*), the variable size, and the function name where the variable is used. An example of the recoded trace is provided in Figure 7.2.c.

$$VariableLifetime = \frac{\sum Cycles\ where\ Variable\ is\ Alive}{Total\ Cycles} \tag{7.1}$$

Original source code

```
void main(){
  int a,b;
  b += a;
  printf("%d", b);
}
```

LLVM compilation ⇓ **Code instrumentation** ⇒

a. LLVM code

```
void main(){
  %a = alloca i32
  %b = alloca i32
  %4 = load i32* %a
  %5 = load i32* %b
  %6 = add i32 %4, %5
  store i32 %6, i32* %b
  %7 = load i32* %b
  %8 = call i32 @printf(i32 %7)
}
```

b. Instrumented LLVM code

```
@horloge = internal global i32 0          // set the clock to 0
@.strVarWrite = private unnamed_addr constant [18 x
i8] c"%d %p 32 w main\0A\00"              // trace for written variable
@.strVarRead = private unnamed_addr constant [18 x i8]
c"%d %p 32 r main\0A\00"                  // trace for read variable

void main(){
  %a = alloca i32
%reg1-1 = load i32* @horloge
%reg2-1 = add nsw i32 %reg1-1, 1          // increment the clock
store i32 %reg2-1, i32* @horloge
%trace1=call i32 (i8*, ...)* @printf(i8* getelementptr
inbounds ([18 x i8]* @.strVarWrite, i32 0, i32 0), i32
%reg2-1, i32* %a)                         // log written variable
  %b = alloca i32
%reg1-2 = load i32* @horloge
%reg2-2 = add nsw i32 %reg1-2, 1          // increment the clock
store i32 %reg2-2, i32* @horloge
%trace2=call i32 (i8*, ...)* @printf(i8* getelementptr
inbounds ([18 x i8]* @.strVarWrite, i32 0, i32 0), i32
%reg2-2, i32* %b)                         // log written variable
  %4 = load i32* %a
%reg1-3 = load i32* @horloge
%reg2-3 = add nsw i32 %reg1-3, 1          // increment the clock
store i32 %reg2-3, i32* @horloge
%trace3=call i32 (i8*, ...)* @printf(i8* getelementptr
inbounds ([18 x i8]* @.strVarRead, i32 0, i32 0), i32
%reg2-3, i32* %a)                         // log read variable
  %5 = load i32* %b
%reg1-4 = load i32* @horloge
%reg2-4 = add nsw i32 %reg1-4, 1          // increment the clock
store i32 %reg2-4, i32* @horloge
%trace4=call i32 (i8*, ...)* @printf(i8* getelementptr
inbounds ([18 x i8]* @.strVarRead, i32 0, i32 0), i32
%reg2-4, i32* %b)                         // log read variable
  %6 = add i32 %4, %5
%reg1-5 = load i32* @horloge
%reg2-5 = add nsw i32 %reg1-5, 1          // increment the clock
store i32 %reg2-5, i32* @horloge
  store i32 %6, i32* %b
  ...
}
```

Execution ⇓

c. Variable trace

Clock	Variable	Physical address	Size	Function	Operation
1	a	0x7fbece0	32	main	w
2	b	0x7fbece8	32	main	w
8	a	0x7fbece0	32	main	r
9	b	0x7fbece8	32	main	r
11	b	0x7fbece8	32	main	w
12	b	0x7fbece8	32	main	r

Lifetime analysis ⇓

d.

Variable	Lifetime intervals	Lifetime in the program
a	1:8	63%
b	2:9,11:12	72%

Figure 7.2: Variable Lifetime Computation

Once we have the trace, we calculate the cumulative number of cycles in which the variable was alive (Figure 7.2.d). Divided by the total number of the program clock cycles, this corresponds to the lifetime of the variable in the program, as shown in Equation 7.1. Clearly, the lifetime of the program variables is dependent on the used workload.

7.2.2 Instruction Lifetime

The computation of the instruction lifetime is quite different from the data lifetime computation since the concept of the *write* and *read* operations is not the same. For the instruction lifetime, we consider that all the instructions are written in the first clock cycle of the program execution because the program is loaded into the memory when the execution starts, and that the instruction is read when it is executed. Thus, we define the instruction lifetime from the first clock cycle to its last read, otherwise the instruction is dead. The last read represents the

Original source code

```
int a,b;
while (a){
  b ++;
}
printf("%d", b);
```

LLVM compilation ⇓

LLVM code

```
1.   %a = alloca i32, align 4
2.   %b = alloca i32, align 4
3.   br label %1

   ; <label>:1                    ; preds = %4,%0
4.   %2 = load i32* %a, align 4
5.   %3 = icmp ne i32 %2, 0
6.   br i1 %3, label %4, label %7

   ; <label>:4                    ; preds = %1
7.   %5 = load i32* %b, align 4
8.   %6 = add nsw i32 %5, 1
9.   store i32 %6, i32* %b, align 4
10.  br label %1

   ; <label>:7                    ; preds = %1
11.  %8 = load i32* %b, align 4
12.  %9 = call i32 @printf(i32 %8)
```

Execution ⇓

Instruction trace

Clock cycle	Instruction ID	Size(bit)	Operation(w/r)
0	1. :12.	32	w
1	1.	32	r
2	2.	32	r
3	3.	32	r
4	4.	32	r
5	5.	32	r
6	6.	32	r
7	7.	32	r
8	8.	32	r
9	9.	32	r
10	10.	32	r
11	4.	32	r
12	5.	32	r

Lifetime analysis ⇓

Instruction ID	Lifetime intervals	Lifetime in the program
1.	0:1	6,6 %
4.	0:11	73,3 %
7.	0:7	46,6 %
…		

Figure 7.3: Instruction Lifetime Computation

last instruction execution in the program. When an instruction is dead, it can not affect the correct program execution since it will never be executed again. Thus any fault affecting this instruction is masked.

Figure 7.3 resumes the steps of the instruction lifetime computation. After generating the LLVM intermediate representation code from the original code, we instrument it in order to compute, as for the variable lifetime computation, the clock cycle of the program and to log out information about the executed instructions. We consider again that each LLVM instruction is executed in one cycle. The next step consists in executing the program and recording the instruction trace. The trace contains, for each clock cycle, the identifier of the executed instruction (which is the line of the instruction), the instruction size, and the operation (*read* or *write*). For the first clock cycle, a *write* operation is recorded for all the instructions. Then, for each clock cycle, a *read* operation is recorded for the executed instruction. Once we have the trace, we count the number of cycles in which the instruction is alive. Divided by the total program cycle, it corresponds to the lifetime of the instruction in the program.

7.2.3 Fault Classification

The fault is classified into three categories based on the analysis of the variables and the instructions:

- **Masked Faults:** The program terminates correctly and the fault does not propagate to the final outputs.
- **Failure:** The fault affects the correct program execution. The program either terminates correctly but the output does not correspond to the golden ones, or does not terminate correctly and generates crash or hang.
- **Detected:** The fault is detected by the application. This case is only possible when a detection mechanism is implemented by the software application.

In order to be conservative and thus considering the worst case scenario, we suppose that any fault affecting a dead variable or a dead instruction is masked and that any fault affecting an alive variable or an alive instruction leads to program failure. Therefore, this analysis provides the minimum percentage of masked faults in the program. Nevertheless, the program can contain techniques of software fault tolerance on some variables or instructions, such as software masking (e.g., C=AxB, if B=0 then any fault in A will be masked) or redundancy (e.g., data/instruction duplication or triplication). In this case, the fault occurring in the variable or the instruction protected by such techniques is masked, while our analysis would consider it as failure. Thus, to consider this case, we enhance our analysis by adding a configuration file where the user can state, for each variable or instruction, if there is a fault tolerance mechanism implemented at software level. In particular, it is possible to provide the list of variables or instructions either protected or detected by the software itself. Then, during analysis we consider that the fault occurring in one of these variable or instruction is masked in case of protection, or detected in case of detection.

7.2.4 Lifetime Validation

To validate the lifetime analysis, we run experiments on simple benchmarks, and we compare the results to the those obtained with the simulation-based fault injection on Intel©8086 processor presented in Subsection 4.5.1. We use a simple matrix multiplication program of 10x10 integer array to evaluate the faults occurring in the data and the instructions. In a second step, to validate the concept of lifetime analysis on programs presenting software fault tolerance techniques, we use two other versions of the same program: the first program implements a detection mechanism, *i.e.*, duplicated variables, and the second program implements a protection mechanism *i.e.*, triplicated variables. We evaluate the faults occurring in the data using this two programs.

In Table 7.1 and 7.2, we provide the results of the fault analysis for the three programs compared to those of the simulation-based fault injection for faults occurring respectively in the data and the instructions. We do not analyze faults in the instructions for the programs presenting detection and protection mechanisms because the mechanisms are implemented on the variables.

We note that the results are very similar for the faults in data and in instructions. This proves that the lifetime analysis provides accurate results when we target a single memory, and for a sample microprocessor architecture (*i.e.*, without caches). In addition, the analysis proposed so far can perfectly evaluate programs even those presenting software fault tolerance techniques.

Table 7.1: Results of Data Lifetime Analysis compared with Simulation-based Fault Injection

Benchmark	Simulator	Masked	Failure	Detected
mMul	**LLVM-Lifetime-Analysis**	18.44%	81.56%	0%
(Sample Version)	**8086-FI**	18.4%	81.6%	0%
mMul	**LLVM-Lifetime-Analysis**	21.31%	21.12%	57.57%
(Detection Version)	**8086-FI**	21.4%	19.1%	59.5%
mMul	**LLVM-Lifetime-Analysis**	84.84%	15.16%	0%
(Protection Version)	**8086-FI**	87.5%	12.5%	0%

Table 7.2: Results of Instruction Lifetime Analysis compared with Simulation-based Fault Injection

Benchmark	Simulator	Masked	Failure	Detected
mMul	**LLVM-Lifetime-Analysis**	9.39%	90.61%	0%
(Sample Version)	**8086-FI**	10.9%	89.1%	0%

For the modern microprocessor where the cache is activated, such as LEON3, and we use the FPGA-based fault injection presented in Subsection 4.5.2 for the comparison. The masking probability for the same program with the sample version is 74%, which is very different from the masking probability provided by our analysis (18.44%). This difference is explained by the

fact that the presented lifetime analysis does not consider the presence of memory elements in the system, such as the caches and the register files. The analysis should take into consideration the variable and instruction residence, as it will be explained in the next section.

7.3 Fault Analysis

In modern microprocessors, the caches are introduced to store data or instructions in order to accelerate their future requests by the processor and increase the system performance. Therefore, we face a hardware redundancy to increase the system performance. This means that the same variable presenting the data (or the same instruction) is replicated and occupies at the same time, different memory components. For instance, a variable can occupy the memory segment storing the data in the RAM or the stack of the RAM (or the memory segment storing the code in the RAM for instructions), the data cache (or the instruction cache), or/and the registers of the CPU. While at the software level, it is the same variable (or instruction), from the hardware point of view only one replica is active in the program and influences the program execution. Therefore, a fault affecting one of the non active copies might be masked. In order to analyze the effects of faults on the software application running on complex microprocessor architecture, we have to consider the variable and the instruction residence coupled with their lifetime.

To determine the variable and the instruction residence, we require information about the program execution and the memory system units. In our analysis, we aim to have a fault evalution at software level. At this level, the concept of caches and register files are not defined. Thus, we use the memory subsystem emulator proposed in Chapter 5, in order to represent a simplified model of the actual system.

In the following, we present the implemented algorithms that permit to compute the percentage of masking, failure, and detection in the program.

7.3.1 Faults in Data

Figure 7.4 shows where and when a variable is active in the RAM and the cache, therefore leading to failure, based on the variable lifetime and residence in the RAM, the cache and the register files.

- **Fault Analysis in RAM**

A fault affecting the RAM will lead to failure in the following cases:

- The variable is alive in the program and resides only in the RAM, but not in the cache and not in the register files (Figure 7.4.a). The fault affecting the variable in the interval W.1-R.1 is propagated to the cache through the *read* R.1, and it will affect the normal behavior of the program.

- The variable is alive in the program, resides in both the RAM and the cache, but it will be reloaded to the cache before its death without any *write* operation that could overwrite the content of the variable in the RAM.

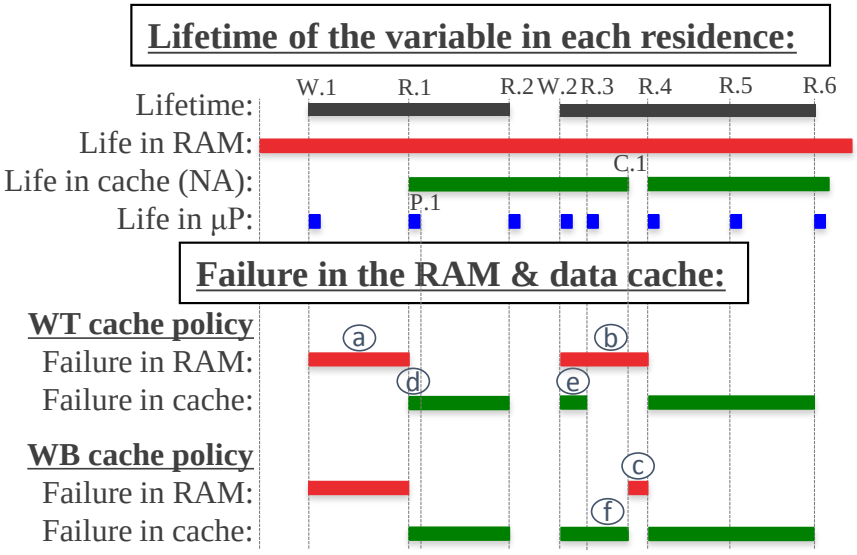

Figure 7.4: Classification of Faults Occurring in Data for the RAM and the cache.

The failure probability in that case depends on the write-hit policy of the cache. If the system uses a write-through cache policy, the contain of the variable in the RAM is updated at the instant of the *write* W.2. Thus, a fault affecting the RAM in the interval W.2-R.4 is propagated to the cache through the *read* R.4 (Figure 7.4.b). However, in case of write-back cache policy, the contain of the variable in the RAM is update at the instant C.1, because the dirty bit is set to 1 in the *write* W.2. Thus, a fault affecting the RAM in the interval C.1-R.4 is propagated to the cache through the *read* R.4 (Figure 7.4.c).

In all the rest of clock cycles the fault is considered as masked. In fact, in the interval R.1-R.2, the variable resides in the cache. Thus, the CPU uses the copy of the variable existing in the cache, any fault affecting the copy of the variable in the RAM will not be loaded to the CPU. In the interval R.2-W.2, the variable is dead, and any fault will not have any influence on the program. At the instant C.1 and in case of write-back cache policy, when the variable leaves the cache, the data of the variable is written to the RAM because the dirty bit is set to 1 when there is the *write* W.2. Thus, if we assume that there is a fault affecting the variable residing in the RAM at the interval W.2-C.1, it will be overwritten at the instant C.1.

- **Fault Analysis in Data Cache**

A fault affecting the data cache will lead to failure in the following cases:

- The variable is alive in the program, resides in the cache and not in the register files, and will be used (read/written) before leaving the cache (Figure 7.4.e). Indeed, the content of the affected variable will be stored in the RAM and used subsequently by the program.

- The variable is alive in the program, resides in both the cache and the register files, and will be reloaded to the register files before leaving the cache (Figure 7.4.d). In fact, any fault affecting the cache in that time (*i.e.*, interval R.1-P.1) will be loaded to the register files and propagated to the CPU at the instant R.2, thus influencing the program execution.

- The cache is write-back policy, the variable is alive in the program, resides in the cache, and will not be used (read/written) before leaving the cache, and the dirty bit is 1 (Figure 7.4.f). A fault affecting the cache in the interval R.3-C.1 is written back to the RAM at the instant C.1, and reloaded to the cache and the CPU at the instant R.4. In case of write-through cache policy, there is no write to the RAM at C.1, so a fault affecting the cache in the interval R.3-C.1 is not propagated to any other memory component, thus it is masked.

7.3.2 Faults in Instructions

Figure 7.5 shows where and when an instruction is active in the RAM and the cache, therefore leading to failure, based on the instruction lifetime and residence in the RAM, the cache and the register files.

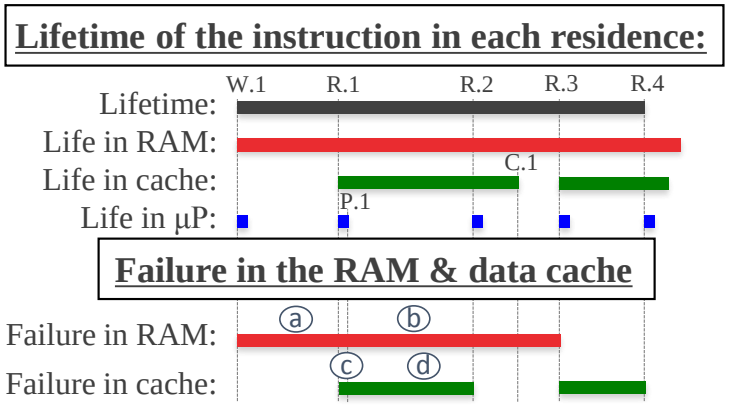

Figure 7.5: Classification of Faults Occurring in Instructions for the RAM and the cache.

- **Fault Analysis in RAM**

A fault affecting the RAM will lead to failure in the following cases:

- The instruction is alive in the program and resides only in the RAM, not in the cache and not in the register files (Figure 7.5.a). The fault affecting the instruction in the interval W.1-R.1 is propagated to the cache through the *read* R.1.
- The instruction is alive in the program, resides in both the RAM and the cache, and will be reloaded to the cache (Figure 7.5.b).

- **Fault Analysis in Instruction Cache**

A fault affecting the cache will lead to failure in the following cases:

- The instruction is alive in the program, resides in the cache and not in the register files, and will be read before leaving the cache (Figure 7.5.d).

- The instruction is alive in the program, resides in both the cache and the register files, and will be reloaded to the register files before leaving the cache (Figure 7.5.c). Any fault affecting the cache in the interval R.1-P.1 is propagated to the CPU in the instant R.2.

Any fault affecting the cache in the interval R.2-C.1 is not propagated to any of the memory component of the system since there is no write-back to the RAM at the instance C.1.

7.4 Validation

In this section, we propose to validate the reliability evaluation accuracy and the time gain provided by the proposed approach. We consider comparison with the FPGA-based fault injection technique on LEON3 processor presented in Subsection 4.5.2. As workloads, we set up a list of benchmarks that have different execution times and memory utilization, and that cover both data-intensive and control-intensive algorithms. We select a set of workloads from the open-source benchmark suite MiBench [93] (bit count, quick sort, string search, fft, crc 32), in addition to a 50x50 integer matrix multiplication.

For the fault injection, we run 10K fault injections per program in order to have an error margin of 1% with confidence level of 95%, as discussed in Subsection 4.5.3.

Finally, we set up the system configurations for the LEON3 processor, and we provide them as input to the memory subsystem emulator:

- *RAM*: 256KB size.
- *Data Cache*: 4KB size, write-through for the write hit, no-write allocate for the write miss, LRU for the replacement mechanism and fully associative for cache associativity.
- *Instruction Cache*: 4KB size, LRU for the replacement mechanism and fully associative for cache associativity.
- *Register-File*: 512B size.

For the comparison with the FPGA-based fault injection, we target the effect of faults occurring in the RAM and the data cache. In Figure 7.6 and 7.7, we provide the results of the LLVM-based analytical approach compared with the FPGA-based fault injection results for faults respectively in the RAM and the data cache.

We note that the results of the analytical method are very close to those of the FPGA-based fault injection. In average the difference is 1.3% for the RAM, and 0.2% for the data cache. This proves that the proposed approach permits to accurately evaluate the effect of faults occurring in different memory components of the system, such as the cache and the RAM.

In terms of execution time, the fault injection technique requires 10K fault simulations in order to obtain statistically significant results with an error margin of 1% and a confidence level of 95%. For 10K fault injections, the program is executed 10K times and the outcomes are analyzed 10K times. However, the proposed analytical approach requires only one program execution and one fault analysis, which provides a huge gain in the execution time. In all tested cases, our tool concludes the analysis in few seconds.

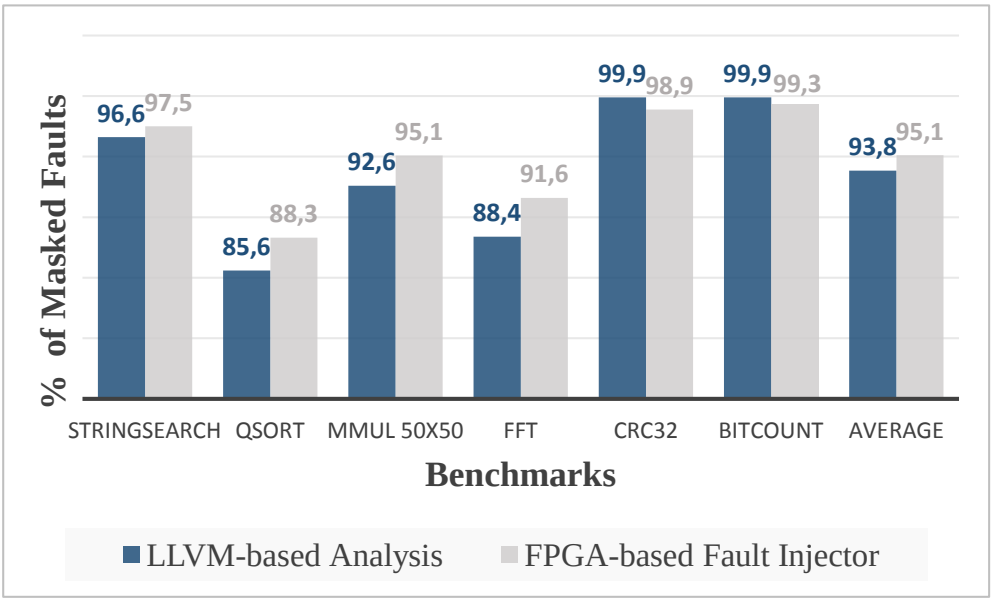

Figure 7.6: Masking Probabilities of LLVM-based Analysis and FPGA-based Fault Injection for Faults in RAM

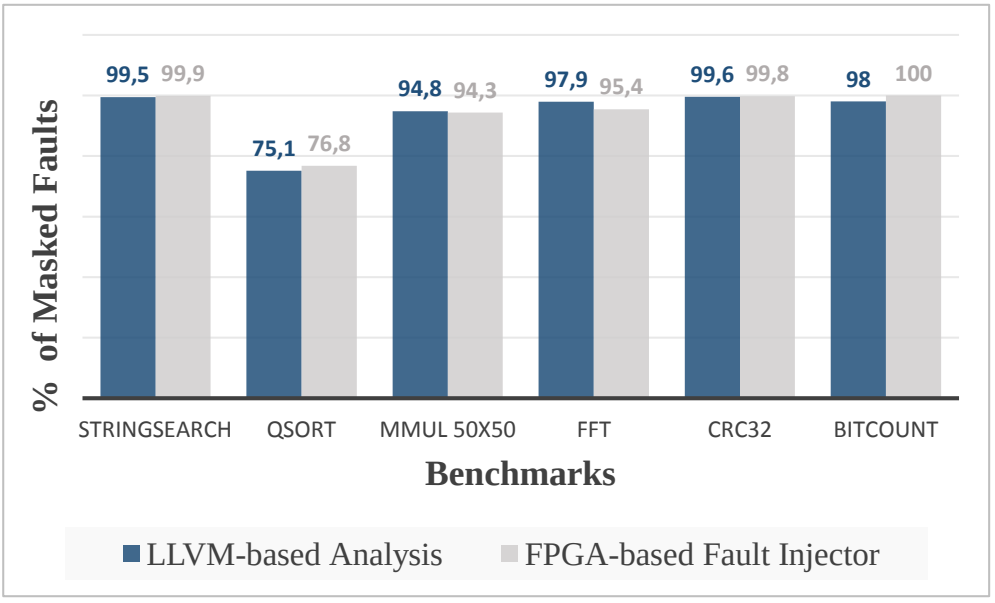

Figure 7.7: Masking Probabilities of LLVM-based Analysis and FPGA-based Fault Injection for Faults in Data Cache

7.5 Industrial Case Study

The application vulnerability strongly depends on the characteristics of the application itself. Therefore, it is important to assess the validity and usability of the proposed methodology with real applications and not only on simple benchmarks. Indeed, a small benchmark whose the code or the data entirely fit into the instruction or the data cache will not appropriately test the performance of the methodology. It is also necessary to run complex applications in order to check how the analysis effort scales with the application complexity.

Since the accuracy of the proposed approach has been validated in Section 7.4, we propose to validate in this section the applicability of the proposed methodology on complex and real programs. Therefore, we use a real industrial case study and we evaluate the fault effect on that application using the analytical method.

Table 7.3: Flight Plan Configuration Options

Flight Route	Distance	Estimated Flight Time	Execution Time (Speed Mul=10)
Grenoble-Valence	31.2 km	4 min	0 min 24
Lyon-Geneve	61.9km	7 min	0 min 42
Grenoble-Torino	121.3 km	14 min	1 min 24
Torino-Pisa	161.6 km	18 min	1 min 48
Tarbes-Limoges	174.9 km	20 min	1 min 00
Torino-Montpellier	256.7 km	30 min	3 min 00

7.5.1 Flight Management System

The used application is an industrial avionic application provided by Thales Group (one of the industrial partners of the CLERECO project). The case study selected to be representative of safety-critical application is a Flight Management System (FMS). The FMS function represents a fundamental part of the modern aircraft's avionics that automates a wide variety of in-flight tasks, such as localization, flight planning, guidance, and predictions. The purpose of the application in modern avionics is to provide the crew with centralized control for the aircraft navigation sensors, computer based flight planning, fuel management, radio navigation management, and geographical situation information. Taking charge of a wide variety of in-flight tasks, the FMS allows to reduce the workload of the flight crew in order to reduce the crew size. The FMS is mainly responsible of the flight management services allowing in-flight guidance of the plane. From the preset flight plans (take-off airport to landing airport), the FMS is responsible for the plane localization, the trajectory computation allowing the plane to follow the flight plan, and the reaction to pilot directives.

As a safety-critical application, the FMS implemented by the used test case is industrial-level application characterizing a real-time performance of avionic system, with different kinds of hardware real-time specifications and with high reliability and availability requirements. The application can be used to study the propagation of faults in microprocessor architecture. It is representative of avionic applications in terms of performance and memory requirements.

7.5.2 Application Analysis

The industrial application presents different flight plan options as provided in Table 7.3, each with specific configurations. The configurations include the flight route, the estimated distance, the estimated flight time and the execution time of the application reduced based on a certain speed multiplication.

The flight plan used in our experiments is Grenoble-Valence. The FMS application is about 15317 lines of C++ source code. Compiled into LLVM IR code, it measures 37607 lines of LLVM code. The compiled LLVM code is instrumented in order to generate the variables' trace and clock cycles. The instrumented application takes 53.633 seconds to be executed, while the sample version of the application takes 23.250 seconds to be executed. Thus, the instrumentation of the code increases the execution time of about 2x times the normal program execution, which is not too much compared to the fault injection techniques that require at least 10K injections in order to have statistically accurate results. The execution of the instrumented LLVM code generates the trace of the variables that will be used to analyze the lifetime. The high complexity of the application provides a total clock cycles of 267M, and a size of the variables' trace of 6.21GB.

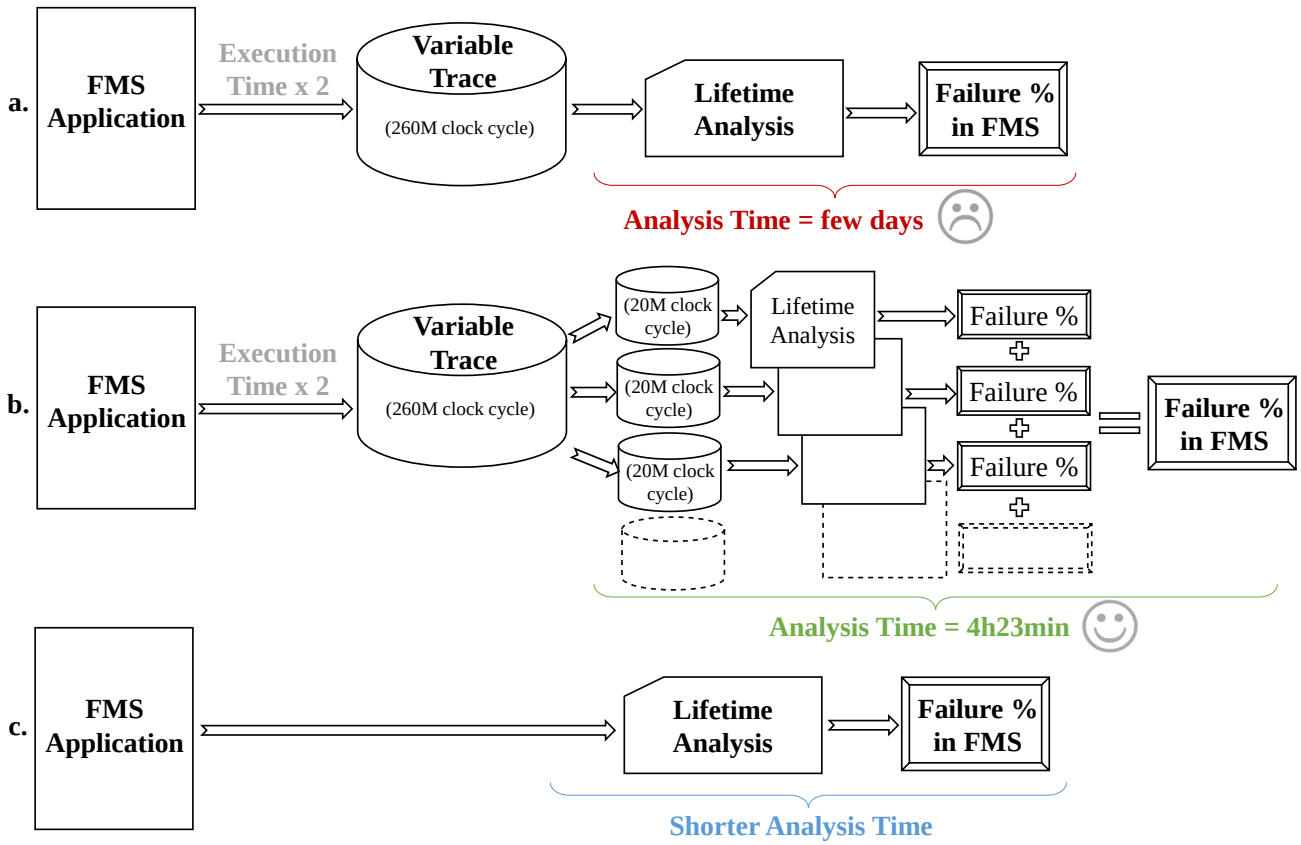

Figure 7.8: Lifetime Analysis for FMS Application

In order to quickly analyze the fault effect in each clock cycle, we divide the trace variables into sub-traces, each one with 20M clock cycles. Then, we compute the failure probability for

all the sub-traces in parallel, as shown in Figure 7.8.b, and we perform the sum of the resulting failure probabilities in order to have the failure probability for the whole application.

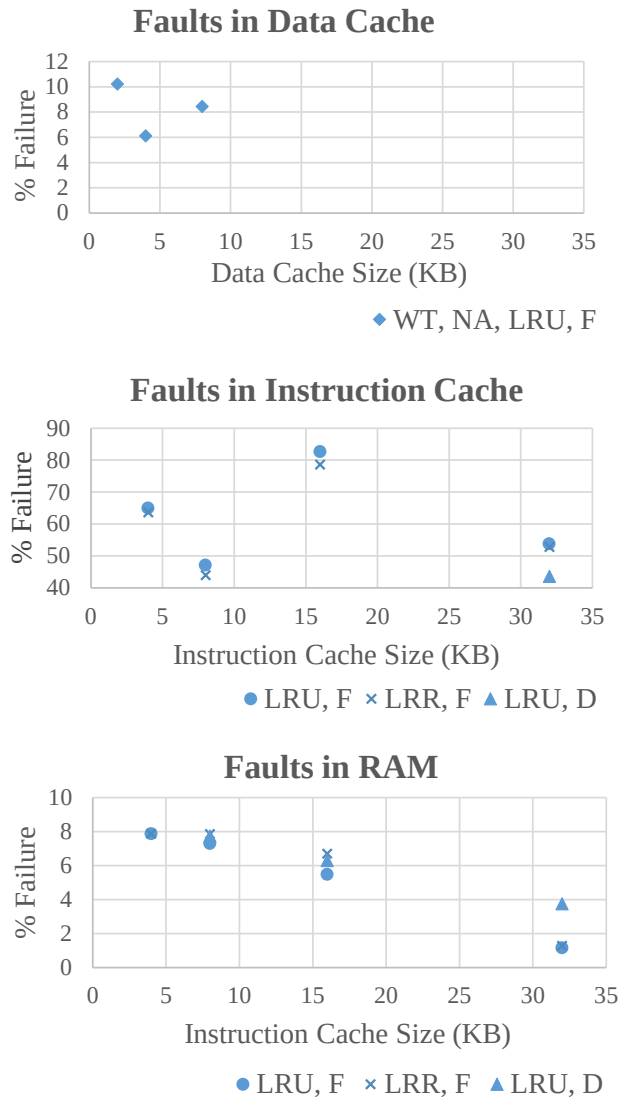

Figure 7.9: Failure Probabilities of FMS Application using LLVM-based Analysis

We compute the failure probability for fault occurring in the data cache. We studied two cache size: 2KB and 4KB. We specify as cache configurations: write-through for the write hit, no-write allocate for the write miss, LRU for the replacement mechanism and fully associative for cache associativity. In Figure 7.9, we provide the failure probabilities for faults occurring in the data cache, the instruction cache and the RAM.

The results show that the proposed analytical approach can be applied, not only on simple benchmarks, but also on complex application. For such complex application, the recorded trace of the variables or the instructions is quite long. The regular lifetime analysis based on successively analyzing the trace is very time consuming. It takes few days to be concluded as shown in Figure 7.8.a. Therefore, we propose to split the trace into small sub-traces and to analyze them in parallel. This method allows to maintain a fast fault analysis as for simple applications. The experiments performed to compute the results presented in 7.9 takes about

4 hours and 23 minutes to be concluded as shown in Figure 7.8.b. The perfect solution will be to analyze the lifetime of the variables and instructions without logging out the long trace. This solution will provide faster analysis time (*i.e.*, few minutes), as shown in Figure 7.8.c. This solution can be used in case the proposed approach is commercialized to analyze several complex applications. It is important to mention that regarding the FPGA-based fault injection results provided in Chapter 4, the fault simulation of the FMS application would take 733 hours for 10K fault injection using the FPGA-based technique which is extremely high.

7.6 Conclusions

In this chapter, we presented a new approach to evaluate the system reliability without performing long fault-injection campaign. It consists in analyzing the variable and the instruction lifetime as well as their residence in different components of the system. Based on this analysis, the proposed approach evaluates the effect of faults occurring in the data and the instructions. To achieve a better characterization of the system reliability at software level, we consider the presence of the RAM, the data cache and the register files thanks to the integration of the proposed memory subsystem emulator. The proposed analytical approach supports different cache configurations, which makes the evaluation generic enough to be applied on different hardware architectures. Furthermore, to be independent from a specific hardware architecture, we used the LLVM virtual instruction set.

To validate the approach, we consider a comparison to an FPGA-based fault injection on LEON3 processor. The experimental results prove that the proposed approach provides accurate fault evaluation. In addition, in terms of execution time, the analytical methodology does not require the presence of a fully defined hardware and offers a huge gain in the time compared to fault injection techniques without loosing the accuracy. Finally, the proposed approach is applied on real industrial test case to prove its flexibility to target complex applications.

To conclude, the main advantage of this contribution is the possibility to quickly and accurately evaluate the reliability of system running a software application. In addition, the proposed method is independence of the hardware architecture, thus can be used at early design stage of the system when the hardware is not fully defined.

Chapter 8

Memory-aware Design Space Exploration

Contents

8.1 Introduction **92**

8.2 Cache Characterizations **93**

8.3 Fault Evaluation **94**

8.3.1 Faults occurring in Data 94

8.3.2 Faults occurring in Instructions 98

8.4 Conclusion **100**

In this chapter, we present a fast and flexible memory-aware design space exploration in order to accurately evaluate the impact of the cache configurations on the system reliability. We target the effect of faults occurring in the data cache and the RAM.

Section 8.1 introduces the motivation and the objectives of this study. Section 8.2 provides the explored cache design space. Section 8.3 presents experimental results for memory-aware design space exploration to evaluate faults occurring in data cache and RAM. Section 8.4 concludes the chapter.

8.1 Introduction

A constant trend in the computer architecture is the continuous reduction of the distance between the microprocessor and the memory. From a single external RAM, designers use now up to four cache levels. The advantage of using the caches is the reduction of the data access time. Cache memories are high-speed buffers used to temporarily hold data as well as instructions that are likely to be used during the programs execution. The main consequence is that up to 60 percent of the microprocessors die area of are nowadays devoted to cache memory blocks [115].

The large portion of occupied die area, make cache arrays critical components for the whole microprocessor reliability especially for safety critical fields such as avionic, aerospace, military, and transportation systems. Thus, any fault in the cache can be the cause of failures at the system level, as shown by the experimental results in Chapter 5 and 7. Evaluating the reliability

at early design stage of the system can save the cost, the effort and consequently positively impact the product time-to-market. During this stage, the designer should define different system characterizations such as the cache configurations. The good choice of these parameters impacts the system behavior and thus can improve its reliability. Investigating methods to explore the memory design space can help the designers to evaluate the vulnerability of different system components. Consequently, it enhances the development of techniques that improve the overall system reliability.

In order to find the optimal system configurations in terms of reliability, we have to study each single possibility and observe its impact on the system reliability. However, this task is very complex and time consuming. In fact, in addition to the very big number of the system parameter combinations that should be studied, the existing reliability evaluation techniques such as the fault injection, are costly in terms of time and hardware. Depending on the category of the fault injection (*i.e.*, hardware-based, software-based, simulation-based, as discussed in Chapter 3), the execution time and hardware costs can differ, but still very high. This is due to the big number of fault injections (*e.g.*, tens of thousands) required to obtain statistically accurate results [90]. Other techniques of reliability evaluation existing in the literature, such as analytical methods, are less time consuming but not enough accurate, as discussed in Chapter 6.

The goal of this chapter is to perform a fast and flexible memory-aware DSE methodology to evaluate the system reliability. We propose to use the analytical approach proposed in Chapter 7. This methodology offers fast and accurate evaluation of faults that occur in the data and the instructions of the RAM and the caches. The main advantages of the proposed methodology are: (i) the integration of the memory subsystem emulator that permits to perform DSE by easily changing its parameters, (ii) the software-level fault analysis that provides gain in the execution time, which facilitates the big number of experiments required by the DSE, and (iii) the use of the LLVM virtual ISA that allows to be independent of the hardware ISA and thus usable at early design stage of the system.

Compared to the DSE methodologies existing in the literature, the proposed study targets additional cache configurations that might have impact on the system reliability. In addition, we explore these cache configurations in order to evaluate the effect of faults occurring in both the RAM and the cache, which allows to have a memory-aware design space exploration. The use of the analytical method offers fast, flexible and accurate evaluation.

8.2 Cache Characterizations

Using the analytical method proposed in Chapter 7, we perform a memory-aware DSE taking into accounts faults occurring in both the RAM and the caches. The methodology consists in changing the parameters of cache configurations and observing their effect on the system outputs when a fault affects the RAM or the caches.

In the proposed study, we explore the following cache design space:

- **The cache size:** 2KB, 4KB, 8KB, 16KB and 32KB.
- **The write-hit policy:** Write-Back (WB) and Write-Through (WT).
- **The write-miss policy:** Write Allocation (A) and No-Write Allocation (NA).

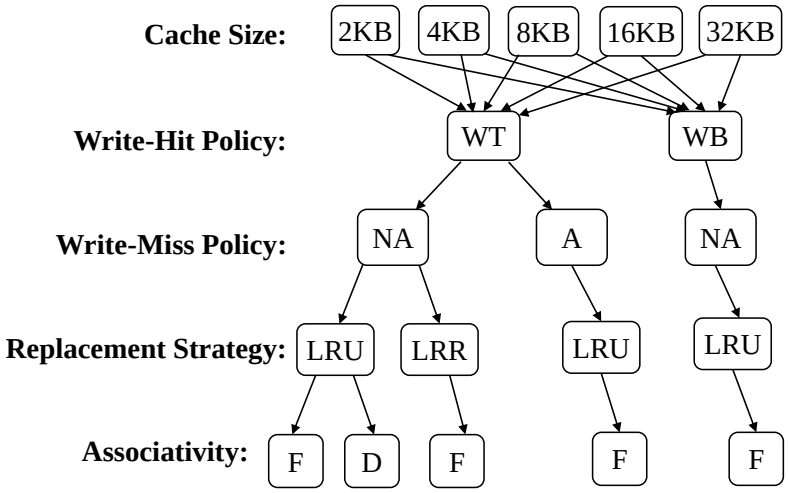

Figure 8.1: The Combinations of the Cache Configurations

- **The replacement strategy:** Last Recently Used (LRU) and Last Recently Replaced (LRR).
- **The cache associativity:** Full associativity (F) and Direct mapped (D).

Regarding these parameters, we build 25 different combinations of cache configurations, as presented in Figure 8.1. These combinations are constructed by changing each time one parameter. The use of the memory subsystem emulator at software level enables to easily change the parameters of the cache configurations. The parameters are provided by the user as input of the emulator.

8.3 Fault Evaluation

We use the analytical approach to observe the impact of changing the cache configurations on the system reliability. We evaluate the faults affecting the data cache and the RAM, and we compute for each combination of the cache configurations the failure probabilities of the target application.

As workloads, we set up a list of benchmarks with different execution time and memory utilization, and which cover both data-intensive and control-intensive algorithms. We use the matrix multiplication program with 50x50 integer array. We also select a set of workloads from the open-source benchmark suite MiBench [93] (bit count, quick sort, string search, fft, crc 32).

8.3.1 Faults occurring in Data

The first set of results concerns the faults occurring in the data of the RAM and the data cache. Figure 8.2 presents the failure probabilities of the fault analysis in the data cache and the RAM, using the different cache configurations described in Figure 8.1.

In addition to the reliability evaluation and in order to study the cache performance, we compute the hit rate of the cache for each cache size. The hit rate is calculated following the Equation 8.1. The results for all the benchmarks are presented in Figure 8.3.

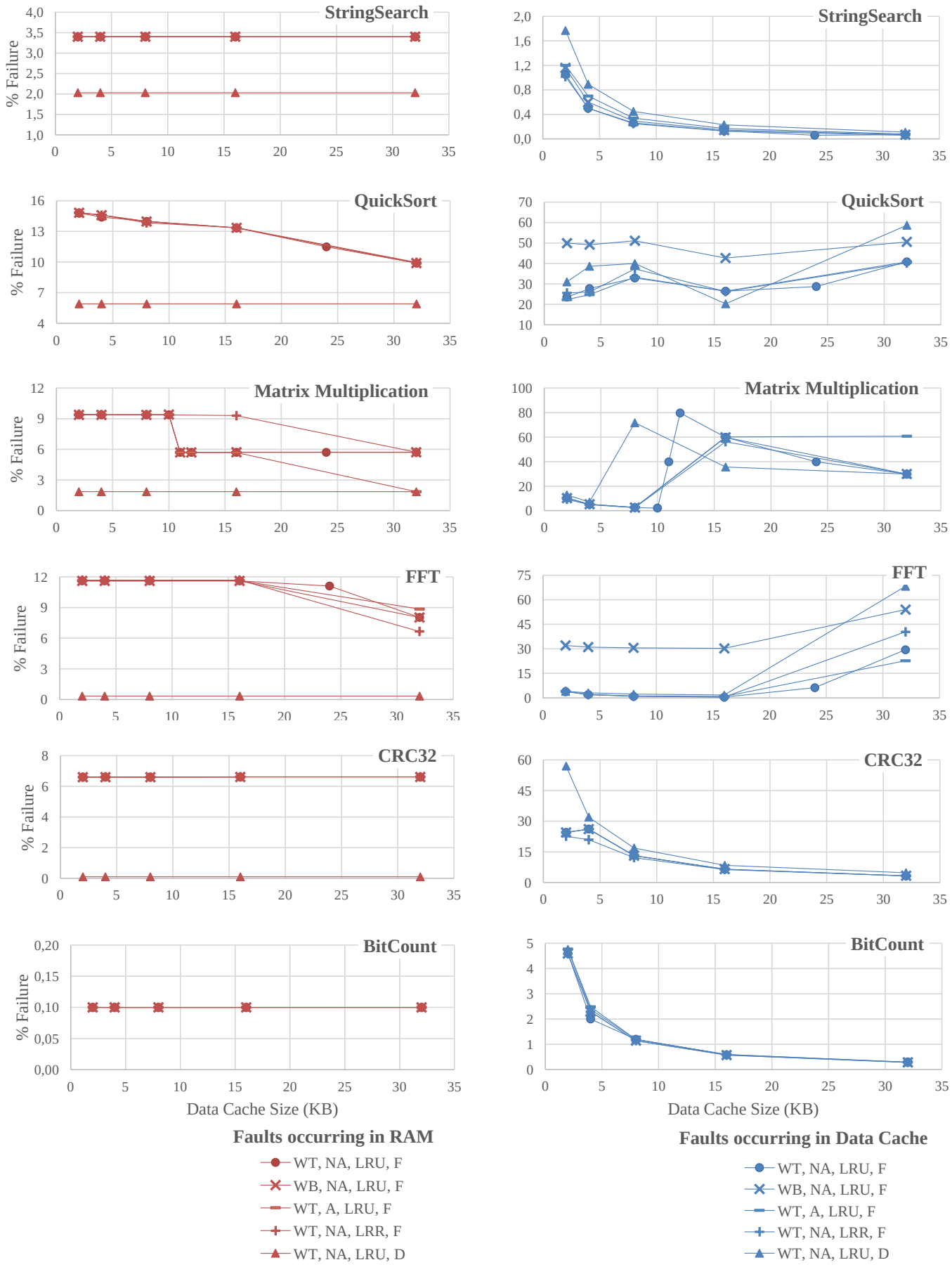

Figure 8.2: Failures Probabilities of Faults occurring in Data Cache and RAM, for 25 different Cache Configurations

$$Cache\ Hit\ Rate = \frac{\sum Cache\ Hit}{\sum Cache\ Hit + \sum Cache\ Miss} \tag{8.1}$$

The first conclusion deducted from the given results is that the failure probability strongly depends on the workloads under evaluation. The change of the cache configurations does not impact the reliability of the computing system in the same way, as shown in Figure 8.2. As consequence, it is very important to have a methodology allowing to accurately, quickly and flexibly perform memory-aware DSE on a given computing system that is running a software. The methodology proposed in this paper is accurate since it has been validated through a comparison to hardware-based fault injection (FPGA-based fault injection using LEON3 processor). In addition, it also needs low execution time and energy consumption to analyze the faults, since it requires only one program execution and it analyzes the lifetime in few seconds for all the used workloads. Furthermore, the methodology allows a flexible DSE because the user is able to easily change the system configurations that are provided as input.

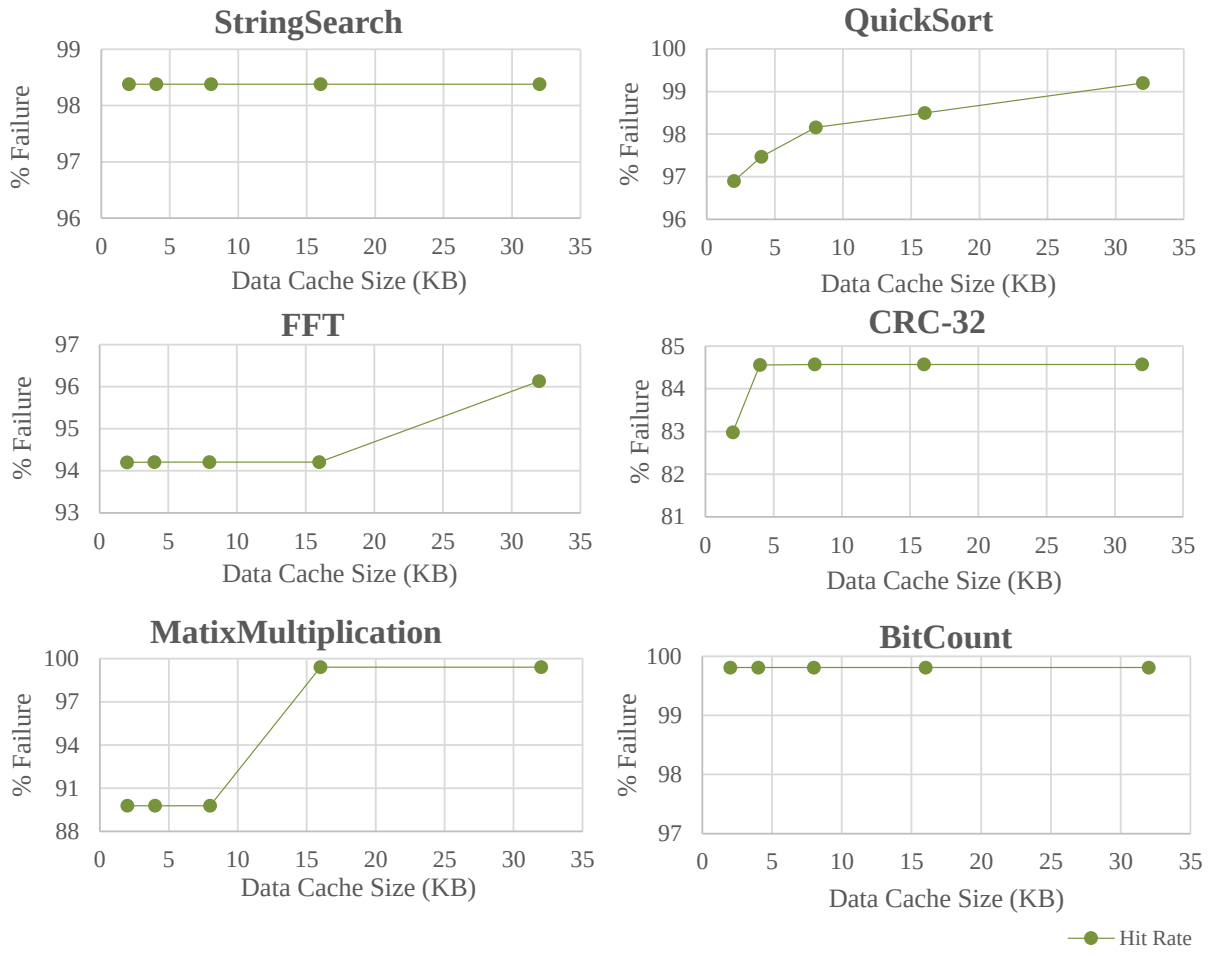

Figure 8.3: Cache Hit Rate depending on the Cache Size

Even if the reliability evaluation strongly depends on the workload, we note that some cache parameters have more impact on the system reliability. Concerning the cache size, we observe

that in most cases, when the cache size increases, the failure probability decreases for faults occurring in the data cache. In fact, the increase of the cache size makes the unused cache space bigger. Thus, the probability that a fault occurs in the unused space increases. As consequence, the masking probability increases and the failure probability decreases. In the results presented in Figure 8.2, this behavior is not noticed for two workloads: the quick sort and the fft. In fact, the data size of these programs is higher then the biggest explored cache size (*i.e.*, 32KB). Thus, during the program execution, there will not be unused space in the cache, and as consequence the failure probability does not decrease.

The second cache parameter that has important influence on the failure probability is the write-hit policy. In fact, for write-back policy, the failure probability in data cache increases compared to write-through policy. In fact, the variables are written back to the RAM when they are leaving the cache, thus their lifetime in the write-back data cache is longer than their lifetime in the write-through data cache. As consequence their failure probability increases. However, for faults in the RAM, it is the opposite, because the faults in the RAM will be erased on the write back instant. This concept is clearly visible for four benchmarks: the quick sort, the fft, the string search and the matrix multiplication. For the bit count workload, the size of data is less than 8KB. Thus, this concept is visible only in the first part (*i.e.*, when the cache size is less than 8KB). However, for the second part (*i.e.*, when the cache size is greater than 8KB), the influence of the write-hit policy is not noticed. In fact, for this cache size, the variable will not leave the cache (since there is free space), thus the variable will not be written-back to the RAM. As consequence, the behavior of the variables in the cache is the same for both type of cache policies, which results in the same failure probability.

The third cache parameter that has important influence on the failure probability is the cache associativity. For faults in data cache, we notice that the failure probability for direct-mapped cache is higher than the failure probability for fully-associative cache. However, for faults in RAM, the failure probability for direct-mapped cache is less than the failure probability for fully-associative cache. In fact, for direct-mapped cache, the whole block corresponding to the requested variable is loaded to the cache. Thus, the lifetime in cache of certain variables will increase and as consequence their failure probability in cache increases. When the variable lifetime in cache increases, the use of the copy residing in the RAM will decrease. As consequence, the failure probability in RAM decreases.

For all the explored cache configurations, the failure probability of faults occurring in the cache and the RAM changes oppositely, *i.e.*, when the failure probability in cache decreases, the failure probability in RAM increases. In fact, the failure probability of one memory component depends on its lifetime and its residence. When the lifetime of a variable in cache increases, the copy residing in the RAM will not be used during this period, thus it will not have influence on the program execution. As consequence, the vulnerability of the copy in cache increases while the vulnerability of the copy in RAM decreases. This concept is clearly visible for all workloads.

In order to well understand the effects of the cache configurations on the reliability, we study in detail the results of two workloads presented in Figure 8.2: the matrix multiplication and the bit count.

- *Case study №1: Matrix Multiplication*

We notice that the curve of the failure probability for faults in data cache dependently of the cache size, reaches a maximum at around 10KB. We explain this behavior by the fact that, at

this cache size, the exact vulnerable variables occupy all the cache lines, *i.e.*, the lifetime of the variables in the program is equal to their lifetime in the cache. Given the source code of the matrix multiplication in Listing 8.1, the exact vulnerable variables at a given clock cycle are: all the cells of the matrix *B*, one line of the matrix *A*, and one line of the matrix *C*. The size of these data corresponds to 10.4KB, which corresponds to the maximum value of failure probability. In case the cache size is smaller than 10.4KB, not all the cells of the matrix *B* will occupy the cache at the same time. Then they will enter and leave the cache several times since they are requested by the CPU for each iteration of the *loop 2*. Thus, their lifetime in cache is shorter than their lifetime in the program. As consequence their failure probability decreases. The same thing for the line of the matrix *A* and *C*. However, in case the cache size is bigger than 10.4KB, more than one line of the matrix *A* and *C* will occupy the cache at the same time. The variables of the extra lines residing in the cache are no longer alive in the program since they are used in only one iteration of the *loop 2*. Thus, their lifetime in cache is longer than their lifetime in the program. In that case, the failure probability decreases. In conclusion, the maximum of failure probability of faults in the cache is achieved for a 10.4KB cache size. This cache size provides the minimum failure probability of faults in the RAM shown in Figure 8.2, and also provides the maximum of the cache hit rate as shown in Figure 8.3.

Listing 8.1: Matrix Multiplication Source Code

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) { // loop 2
    for (k=0; k<N; k++) {
      MatrixC[i][j] += MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

- *Case study №2: Bit Count*

The bit count workload is not a data-intensive algorithm (less than 8KB). Thus, the failure probability in RAM is very low since the size of RAM is much more bigger than the size of the program data. In addition, the failure probability in cache significantly decreases (tending to 0) when the cache size increases.

8.3.2 Faults occurring in Instructions

The second set of results concerns the faults occurring in the instructions of the RAM and the instruction cache. In Figure 8.4, we present the failure probabilities of the analysis of faults occurring in the instruction cache and the RAM, using different cache configurations.

We note from these results that the failure probability of the evaluated applications for faults occurring in the instruction cache decreases when the cache size increases. Furthermore, we observe that the failure probability for faults occurring in the RAM is slightly influenced by the change of the cache configurations. This behavior is due to the fact that the used benchmarks have low instructions' size compared to the RAM size.

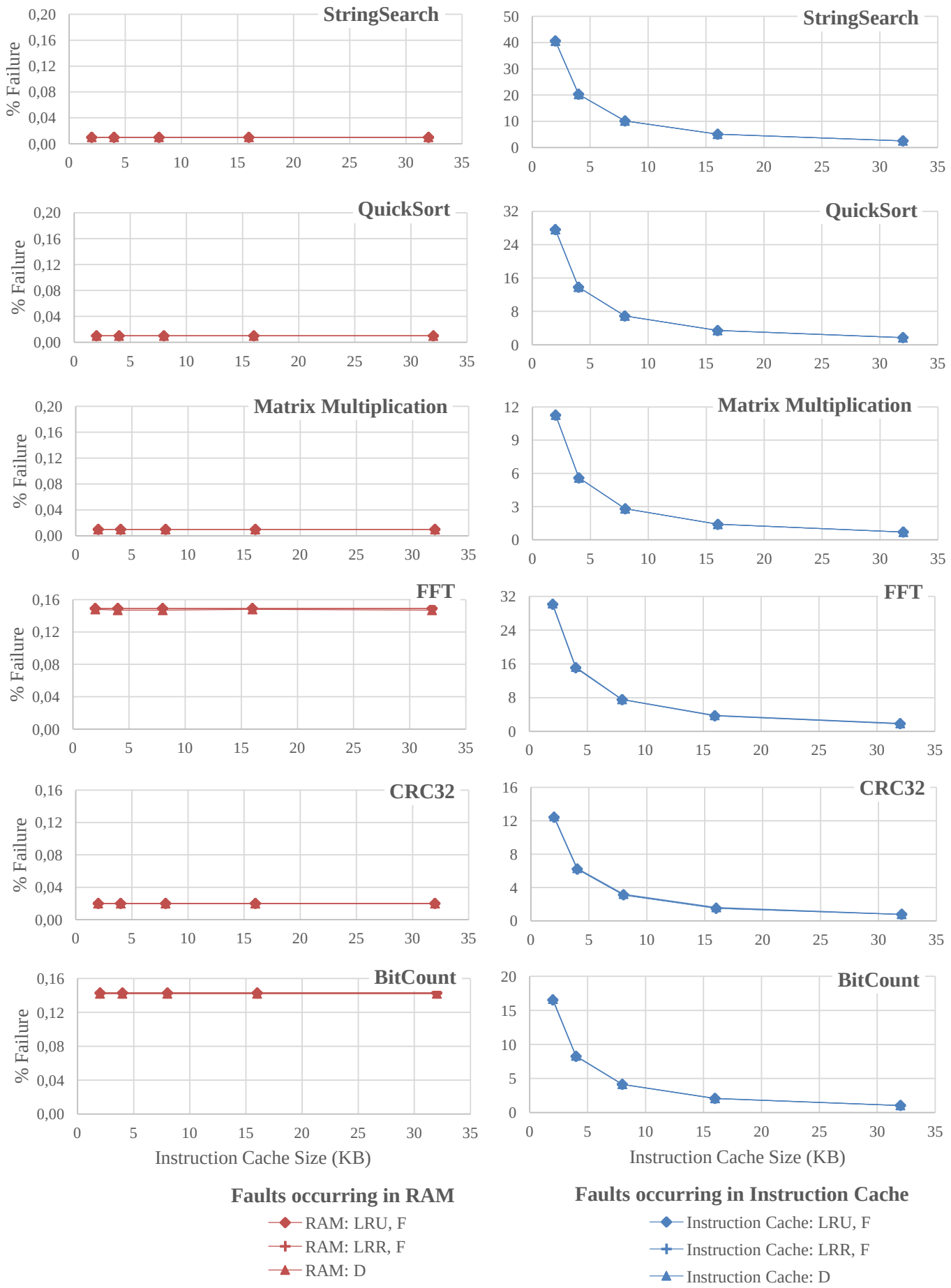

Figure 8.4: Failures Probabilities of Faults occurring in RAM and Instruction Cache, for 15 different Cache Configurations.

8.4 Conclusion

We presented in this chapter a memory-aware design space exploration methodology to evaluate the system reliability. We used the analytical approach presented in Chapter 7, and we studied the impact of different hardware configurations on the reliability of memory components, in particular, the effect of faults occurring in the cache and the RAM. The experimental results show that the reliability evaluation strongly depends on the target software. In addition, we observed that some hardware configurations have more influence on the fault propagation than others. These configurations are the cache size, the cache write-hit policy, and the cache associativity.

The proposed memory-aware design space exploration methodology highlights two main contributions of this thesis: the memory system emulator and the analytical reliability evaluation approach. We demonstrated that the proposed subsystem emulator is flexible since it can easily be applied on different hardware architectures and system configurations. In addition, we proved that the proposed analytical approach is fast since it can quickly and accurately provide a memory design space exploration.

An important application of the proposed memory-aware design space exploration proposed in this chapter is the development of techniques that are able to increase the dependability of the system. For instant, it can be used to apply suitable detection and protection mechanisms on the software application in order to improve the system reliability.

Chapter 9

Conclusion

Contents

9.1 Summary and Conclusion **101**

9.2 Application in CLERECO Project **103**

9.3 Future Work **104**

9.1 Summary and Conclusion

In this thesis, we developed new fault injection environments able to evaluate the effect of soft errors on the system reliability. We presented a first fault injection technique based on the virtual ISA of LLVM, and a second fault injection technique based on the C programming language. Both methods target fault in the data and the instruction of the software application running on the system under evaluation. In order to target at software level, the memory system components in modern microprocessor such as RAM, caches and register files, we developed a memory subsystem emulator that is independent of ISA of microprocessor. It only requires some hardware configurations that are easy to provide by the user. The proposed subsystem emulator allows to emulate the behavior of the RAM, the caches and the register files at software level. The developed emulator has been integrated within the proposed fault injection techniques in order to evaluate faults occurring in the RAM and the caches.

The fault injection performed at software level reduces the execution time of several order of magnitudes compared to hardware-based techniques. However, the fault evaluation might be costly in terms of time for complex applications. In this thesis, we proposed a novel analytical technique able to evaluate the fault effect without performing long fault injection campaigns. The proposed method is based on the analysis of both the lifetime and the residence of the variables and the instructions. It is based on the virtual ISA of LLVM. The comparison of the fault evaluation in the RAM and the caches to hardware-based fault injections proves the accuracy of the method with huge gain in time evaluation.

In order to demonstrate the application of the analytical method on complex systems, we used a real avionic application provided by the industrial project partner. The proposed ap-

proach is also used to perform fast and flexible memory-aware design space exploration methodology to evaluate the system reliability.

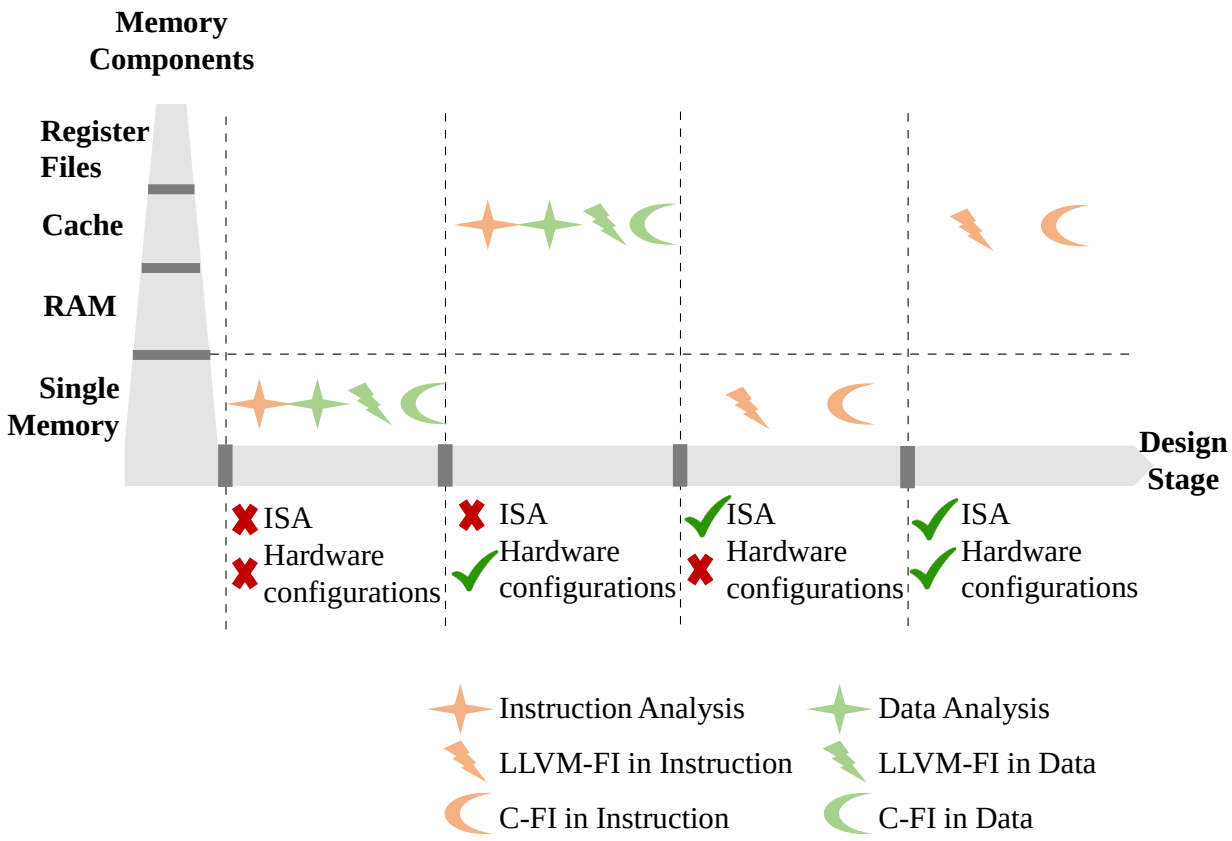

Figure 9.1: Usability of Fault Evaluation Methodologies in Different System Design Stages

In this thesis, we studied the role of the software stack to evaluate the system reliability at different design stages of the system. At advanced design stage, the ISA is known and therefore can be used to perform simulations to analyze how faults propagate through the system components. However, at early design stage of the system when the hardware architecture is not fully defined, the fault evaluation is a challenging task. The methodologies for reliability evaluation proposed in this thesis can be used in different design stages of the system. In fact, we divide the design stage into three intervals: (1) advanced design stage, *i.e.* the ISA is defined; (2) intermediate design stage, *i.e.* the ISA is not defined but the hardware configurations are available for the user, or the ISA is defined but not the hardware configurations; (3) early design stage *i.e.* neither the ISA nor the hardware configurations are defined. Given these design stage intervals, we provide in Figure 9.1, the possible application of each proposed methodology. For faults in instructions, the LLVM- and C-based fault injection can be used only when the ISA is defined. Before the definition of the ISA, the LLVM- and the C- based fault injection can not evaluate faults in the instructions but can be used for faults in the data. If the fault injection targets faults in precise data memory components, such as RAM (data segment or stack) and data caches, the LLVM- and the C- based fault injections can be used in the intermediate design stage. At this design stage, the analytical method can evaluate faults in both the data and the

instructions of the RAM and the caches. At earlier design stage, *i.e.* when neither the ISA nor the hardware configurations are defined, we consider a global fault evaluation of the data and the instruction of a software running on unspecified hardware architecture. The global data evaluation is performed using the analytical, the LLVM-based and the C-based fault injection techniques. The global instruction evaluation is performed using only the analytical approach.

At all design stages of the system, the accuracy of the fault evaluation using the three approaches has been validated through hardware-based fault injections. In addition, in comparison to the state-of-the-art fault injection techniques, the developed fault evaluation methodologies correct the drawbacks of each fault injection category while keeping the advantages, as modeled in Figure 3.3. Compared to software-based fault injections, we improved the fault evaluation accuracy and the precision in terms of hardware fault location thanks to the development of the memory subsystem emulator. Compared to simulation-based fault injections, we reduced the execution time first by performing fault injections at software level, and then by implementing analytical method. Compared to FPGA-based fault injections, our methodologies can be used at different design stages, as explained in Figure 9.1. Finally, compared to physical fault injections, our methodologies are performed at software level, thus they do not require any hardware platform and they provide good observability of software components.

9.2 Application in CLERECO Project

All the developed methodologies during this thesis are adopted by the CLERECO project. The proposed approaches to evaluate the effect of fault on the system reliability are used to develop the global methodology proposed in the project. The proposed method of the project consists on a scalable, cross-layer methodology, supporting suite of tools for accurate but fast estimations of computing systems reliability. The backbone of the methodology is a component-based Bayesian network model, which effectively calculates the system reliability driven by the masking probabilities of individual hardware and software components and considering their complex interactions. The model can be connected to any method and tool that measures the masking probabilities of the individual components of the network to feed it. We describe a complete realization of the model along with supporting tools, which separately provide the parameters of the underlying technology, circuit, micro-architecture and software layers.

The project methodology is based on Bayesian network model, where the application functions represent the nodes. Thus, we adapt the tools developed during this thesis by evaluating the reliability of the software layer function by function. In fact, the implemented fault injection and the analytical methods are configured to consider only the variables and the instructions of the target function. Therefore, we study the fault effect based on the function outputs. The results of the software reliability evaluation function by function represent the inputs of the global framework that evaluates the overall system reliability. This framework calculates the system reliability based on the masking probabilities of individual hardware and software components and by considering their complex interactions.

9.3 Future Work

As perspectives, we propose to apply the proposed fault evaluation methodologies in the development of significant soft error mitigation techniques. In fact, our methodologies provide accurate software and hardware components visibility, thus a better observability of the fault location. This allows to identify the most vulnerable data and instructions in the software applications. Therefore, we propose to apply protection mechanisms only on the most vulnerable data and instructions. The advantage of this method is to reduce the cost of the full program protection in terms of power and performance without loosing the reliability of the system.

In this thesis, we target single-core microprocessor systems. In future work, we propose to adapt the proposed fault evaluation methodologies in order to address faults in multi-core microprocessors. In fact, this requires further development of our memory subsystem emulator.

In addition, we propose to apply the analytical methodology on approximate computing. In fact, the development of instruments for a reliability analysis namely at early design stage, is key enabler of real applications on approximate computing systems. Studying the impact any change in the technology, circuits, micro-architecture and software on system reliability of, is most of the time a challenging task, because we have to take into consideration other crucial design attributes (or objectives) such as power, performance and cost.

Glossary

ACE: Architecturally Correct Execution
AVF: Architectural Vulnerability Factor
Bit-flip: Inversion of memory cell value
DSE: Design Space Exploration
ECC: Error Correcting Code
FPGA: Field-Programmable Gate Array
FMS: Flight Management System
IR: Intermediate Representation of LLVM code
ISA: Instruction Set Architecture
LEON3: Processor based on the SPARC v8 architecture
LLVM: Low Level Virtual Machine
MBU: Multiple Bit Upset
MCU: Multi-Cell Upset
RAM: Random Access Memory
SEU: Single Event Upset
SPARC: Scalable Processor ARChitecture
TMR: Triple Modular Redundancy
PVF: Program Vulnerability Factor

List of Figures

1.1 System Layers and Fault Propagation 9

2.1 The Dependability Tree [1] 14
2.2 The LLVM Compiler 17

3.1 Comparison between Existing Fault Injection Methodologies 25

4.1 Fault Propagation through System Layers 32
4.2 Single Bit Flip occurring on Opcode. 34
4.3 Variable Trace 35
4.4 Instruction Trace 36
4.5 Active Data and Instruction Memory according to the Program Execution . . . 37
4.6 Flow of the Fault Analysis Process 37
4.7 Flow of the Fault Injection Process. 38
4.8 Transient and Permanent Fault Injection in Data. 39
4.9 Transient and Permanent Fault Injection in Data. 40
4.10 Overview of the C-based Fault Injection. 43
4.11 Process of the C-based Fault Injection. 44
4.12 Data and Instruction Collection 45
4.13 Output of C-based Fault Injection 46
4.14 Simulation-based Fault Injector. 47
4.15 Communication between Host Computer and FPGA Board in SCFIT 48
4.16 Gaussian Distribution 48
4.17 Comparison between HW FI and SW FI. 50

5.1 Data Location in System 54
5.2 System Emulator 55
5.3 Example of Subsystem Emulator 55
5.4 Instructions' Trace 56
5.5 Algorithm for Cache Construction 57
5.6 Fully-Associative Cache 58
5.7 Direct-Mapped Cache 59
5.8 Algorithm of Register File Construction 59
5.9 Algorithm for Fault Injection in RAM 60
5.10 Algorithm for Fault Injection in Cache 61
5.11 Masked Faults Injected in Data Cache using LLVM- and FPGA-based Fault Injection 63

5.12 Masked Faults Injected in Instruction Cache using LLVM- and FPGA-based Fault Injection . . . 64
5.13 Masked Faults Injected in RAM using C- and FPGA-based Fault Injection . . . 65
5.14 Masked Faults Injected in Data Cache using C- and FPGA-based Fault Injection 65
5.15 Execution Time (in hours) of C- and FPGA-based Fault Injections for Faults in Data Cache . . . 66
5.16 Execution Time (in hours) of C- and FPGA-based Fault Injections for Faults in RAM . . . 66

7.1 Variable Lifetime . . . 78
7.2 Variable Lifetime Computation . . . 79
7.3 Instruction Lifetime Computation . . . 80
7.4 Classification of Faults Occurring in Data for the RAM and the cache. . . . 84
7.5 Classification of Faults Occurring in Instructions for the RAM and the cache. . 85
7.6 Masking Probabilities of LLVM-based Analysis and FPGA-based Fault Injection for Faults in RAM . . . 86
7.7 Masking Probabilities of LLVM-based Analysis and FPGA-based Fault Injection for Faults in Data Cache . . . 87
7.8 Lifetime Analysis for FMS Application . . . 89
7.9 Failure Probabilities of FMS Application using LLVM-based Analysis . . . 90

8.1 The Combinations of the Cache Configurations . . . 94
8.2 Failures Probabilities of Faults occurring in Data Cache and RAM, for 25 different Cache Configurations . . . 95
8.3 Cache Hit Rate depending on the Cache Size . . . 96
8.4 Failures Probabilities of Faults occurring in RAM and Instruction Cache, for 15 different Cache Configurations. . . . 99

9.1 Usability of Fault Evaluation Methodologies in Different System Design Stages 102

List of Tables

3.1 Fault Injection versus Mutation Testing 28

4.1 Wrong Data Fault Model. 38
4.2 Statistics (%) of Instruction Replacement for SPARC Architecture 41
4.3 Statistics (%) of Instruction Replacement for ARM Architecture 41
4.4 Statistics (%) of Instruction Replacement for Intel x86 Architecture. 42
4.5 Results of Single Transient Fault Injection in Data 49
4.6 Results of Single Transient Fault Injection in Opcode 50
4.7 Outcomes to Evaluate the Overall System Reliability. 50
4.8 Comparison of SW FI and HW FI Simulation Times. 51

5.1 Faults Evaluated with the Analytical Process. 63

7.1 Results of Data Lifetime Analysis compared with Simulation-based Fault Injection 82
7.2 Results of Instruction Lifetime Analysis compared with Simulation-based Fault Injection 82
7.3 Flight Plan Configuration Options 88

Appendix

- **LLVM Installation**

The version LLVM 3.4 used during this thesis is available in [116].

LLVM comes in three pieces. The first piece is the LLVM suite. This contains all of the tools, libraries, and header files needed to use LLVM. It contains an assembler, disassembler, bitcode analyzer and bitcode optimizer. It also contains basic regression tests that can be used to test the LLVM tools and the Clang front end.

The second piece is the Clang front end. This component compiles C, C++, Objective C, and Objective C++ code into LLVM bitcode. Once compiled into LLVM bitcode, a program can be manipulated with the LLVM tools from the LLVM suite.

There is a third, optional piece called Test Suite. It is a suite of programs with a testing harness that can be used to further test LLVM's functionality and performance.

The steps to install LLVM are the following:

1. Checkout LLVM:
   - cd where-llvm-will-live
   - svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
2. Checkout Clang:
   - cd where-llvm-will-live
   - cd llvm/tools
   - svn co http://llvm.org/svn/llvm-project/cfe/trunk clang
3. Checkout Compiler-RT (required to build the sanitizers) [Optional]:
   - cd where-llvm-will-live
   - cd llvm/projects
   - svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt
4. Checkout Libomp (required for OpenMP support) [Optional]:
   - cd where-llvm-will-live
   - cd llvm/projects
   - svn co http://llvm.org/svn/llvm-project/openmp/trunk openmp

5. Checkout libcxx and libcxxabi [Optional]:
   - cd where-llvm-will-live
   - cd llvm/projects
   - svn co http://llvm.org/svn/llvm-project/libcxx/trunk libcxx
   - svn co http://llvm.org/svn/llvm-project/libcxxabi/trunk libcxxabi
6. Get the Test Suite Source Code [Optional]
   - cd where-llvm-will-live
   - cd llvm/projects
   - svn co http://llvm.org/svn/llvm-project/test-suite/trunk test-suite
7. Configure and build LLVM and Clang: The build uses CMake. LLVM requires CMake 3.4.3 to build. The steps to build LLVM with CMake are available in [117].

- **Manipulation of LLVM IR**

1. Compile C/C++ source code to LLVM IR code. The LLVM code has *.ll* as extension.
   - clang -S -emit-llvm program.c
2. Compile the LLVM IR code. The output is *program.s* which is an assembly code. The assembler is the default machine architecture where the program is run. The assembly code is then compiled using gcc to an executable.
   - llc-3.4 program.ll
   - gcc program.s -o programExec

Bibliography

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[2] K. Rozier, *Dependability management - Part 1: Dependability management systems, International Electrotechnical Commission*, IEC Std., 2003.

[3] T. Boroomandnezhad and M. A. Azgomi, "An efficient control-flow checking technique for the detection of soft-errors in embedded software," *Computers and Electrical Engineering*, vol. 39, no. 4, pp. 1320–1332, 2013.

[4] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computer*, vol. 33, no. 6, pp. 518–528, Jun. 1984.

[5] M. Nicolaidis, *Soft errors in modern electronic systems*. Springer Science & Business Media, 2010, vol. 41.

[6] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*, ser. MICRO 36, pp. 29–42.

[7] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: Circuit-Level Correction of Timing Errors for Low-Power Operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[8] R. Vadlamani, J. Zhao, W. Burleson, and R. Tessier, "Multicore Soft Error Rate Stabilization Using Adaptive Dual Modular Redundancy," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE, 2010, pp. 27–32.

[9] A. Benso, S. D. Carlo, G. D. Natale, and P. Prinetto, "Static analysis of SEU effects on software applications." in *ITC*. IEEE Computer Society, 2002, pp. 500–508.

[10] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. (2013, 02) The Java Virtual Machine Specification. [Online]. Available: http://docs.oracle.com/javase/specs/jvms/se7/html/

[11] Microsoft Corporation, .Net Framework 4. [Online]. Available: http://msdn.microsoft.com/en-us/library/vstudio/w0x726c2%28v=vs.100%29.aspx

[12] Mono Project. [Online]. Available: http://www.mono-project.com

[13] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04, 2004, pp. 75–86.

[14] H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques," vol. 1, no. 2, pp. 171–186, July 2004.

[15] S. Duzellier and G. Berger, "Test facilities for see and dose testing," in *Radiation Effects on Embedded Systems*. Springer, 2007, pp. 201–232.

[16] R. Ecoffet, "In-flight anomalies on electronic devices," in *Radiation Effects on Embedded Systems*. Springer, 2007, pp. 31–68.

[17] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using heavy-ion radiation to validate fault-handling mechanisms," *IEEE Micro*, vol. 14, no. 1, pp. 8–11, 13–23, 1994.

[18] F. Miller, N. Buard, T. Carrière, R. Dufayel, R. Gaillard, P. Poirot, J.-M. Palau, B. Sagnes, and P. Fouillat, "Effects of beam spot size on the correlation between laser and heavy ion seu testing," *IEEE transactions on nuclear science*, vol. 51, no. 6, pp. 3708–3715, 2004.

[19] R. Velazco, B. Martinet, and G. Auvert, "Laser injection of spot defects on integrated circuits," in *First Asian Test Symposium, (ATS)*. IEEE, 1992, pp. 158–163.

[20] P. Fouillat, V. Pouget, D. Lewis, S. Buchner, and D. McMorrow, "Investigation of single-event transients in fast integrated circuits with a pulsed laser," *International journal of high speed electronics and systems*, vol. 14, no. 02, pp. 327–339, 2004.

[21] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 913–923, 1993.

[22] H. Madeira, M. Zenha-Rela, F. Moreira, and J. Silva, "Rifle: A general purpose pin-level fault injector," in *1st European Dependable Computing Conference (EDCC-1), (Berlin, Germany)*, vol. 852. Springer-Verlag, 1994, pp. 199–216.

[23] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[24] J. J. H. Pontes, N. Calazans, and P. Vivet, "An accurate single event effect digital design flow for reliable system level design." in *DATE*, W. Rosenstiel and L. Thiele, Eds. IEEE, 2012, pp. 224–229.

[25] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: the MEFISTO tool," in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*. IEEE, 1994, pp. 66–75.

[26] Lifting. [Online]. Available: http://gforge-lirmm.lirmm.fr/gf/project/lifting/

[27] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi, "A fast, flexible, and easy-to-develop FPGA-based fault injection technique," *Microelectronics Reliability*, vol. 54, no. 5, pp. 1000–1008, 2014.

[28] V. Sieh, O. Tschäche, and F. Balbach, "VERIFY: Evaluation of reliability using vhdl-models with embedded fault descriptions." in *FTCS*. IEEE Computer Society, 1997, pp. 32–36.

[29] A. Bosio and G. Di Natale, "Lifting: A flexible open-source fault simulator," in *Proceedings of the 2008 17th Asian Test Symposium*, ser. ATS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 35–40.

[30] M. S. Shirazi, B. Morris, and H. Selvaraj, "Fast FPGA-based fault injection tool for embedded processors," in *International Symposium on Quality Electronic Design, ISQED, Santa Clara, CA, USA, March 4-6, 2013*, 2013, pp. 476–480.

[31] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai, "Fault emulation: A new methodology for fault grading," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 10, pp. 1487–1495, 1999.

[32] L. Antoni, R. Leveugle, and B. Fehér, "Using run-time reconfiguration for fault injection applications," *IEEE Transactions on Instrumentation and Measurement*, vol. 52, no. 5, pp. 1468–1473, 2003.

[33] D. De Andrés, J. C. Ruiz, D. Gil, and P. Gil, "Fault emulation for dependability evaluation of vlsi systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 4, pp. 422–431, 2008.

[34] L. Sterpone and M. Violante, "A new partial reconfiguration-based fault-injection system to evaluate SEU effects in SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, no. 4, pp. 965–970, 2007.

[35] C. Lopez-Ongil, L. Entrena, M. Garcia-Valderas, M. Portela, M. Aguirre, J. Tombs, V. Baena, and F. Munoz, "A unified environment for fault injection at any design level based on emulation," *IEEE Transactions on Nuclear Science*, vol. 54, no. 4, pp. 946–950, 2007.

[36] S.-A. Hwang, J.-H. Hong, and C.-W. Wu, "Sequential circuit fault simulation using logic emulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 8, pp. 724–736, 1998.

[37] H. R. Zarandi, S. G. Miremadi, and A. Ejlali, "Dependability analysis using a fault injection tool based on synthesizability of HDL models," in *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2003.18th IEEE International Symposium on*. IEEE, 2003, pp. 485–492.

[38] A. Ejlali, S. G. Miremadi, H. Zarandi, G. Asadi, and S. B. Sarmadi, "A hybrid fault injection approach based on simulation and emulation co-operation," in *International Conference on Dependable Systems and Networks*. IEEE, 2003, p. 479.

[39] A. Mohammadi, M. Ebrahimi, A. Ejlali, and S. G. Miremadi, "SCFIT: A FPGA-based fault injection technique for SEU fault model," in *DATE*, 2012.

[40] Altera. [Online]. Available: www.altera.com

[41] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, Feb. 1995.

[42] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 125–136, Feb. 1998.

[43] (2003-2013) FAUmachine. [Online]. Available: www.FAUmachine.org/

[44] S. Potyra, V. Sieh, and M. D. Cin, "Evaluating fault-tolerant system designs using FAUmachine," in *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*, ser. EFTS '07. New York, NY, USA: ACM, 2007.

[45] Qemu. [Online]. Available: http://wiki.qemu.org

[46] Virtualbox. [Online]. Available: www.virtualbox.org/

[47] C. Lattner. The LLVM compiler infrastructure. [Online]. Available: http://www.llvm.org

[48] ——, "LLVM," in *The Architecture of Open Source Applications*, A. Brown and G. Wilson, Eds., vol. I, ch. 11.

[49] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic – System Effects (SELSE)*, 2013.

[50] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[51] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), Vancouver, BC, Canada, December 2-4*, 2013, pp. 41–50.

[52] R. L. de Oliveira Moraes and E. Martins, "Jaca - A software fault injection tool," in *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, 2003, p. 667.

[53] E. Martins, C. M. F. Rubira, and N. G. M. Leme, "Jaca: A reflective fault injection tool based on patterns," in *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, 2002, pp. 483–482.

[54] B. P. Sanches, T. Basso, and R. Moraes, "J-SWFIT: A java software fault injection tool," in *5th Latin-American Symposium on Dependable Computing, LADC 2011, São José dos Campos, Brazil, 25-29 April 2011*, pp. 106–115.

[55] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Transactions on Computer*, vol. 52, no. 9, pp. 1115–1133, Sep. 2003.

[56] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference, DAC*, 2013, pp. 101:1–101:10.

[57] G. Di Natale, "Software-implemented system dependability for safety critical applications," Ph.D. dissertation, Politecnico di Torino, 2003.

[58] A. Benso, S. Di Carlo, G. Di Natale, L. Tagliaferri, and P. Prinetto, "Validation of a software dependability tool via fault injection experiments," in *Proceedings of the Seventh International On-Line Testing Workshop*, ser. IOLTW '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–8.

[59] W. D. van Driel, M. Schuld, R. Wijgers, and W. E. J. van Kooten, "Software reliability and its interaction with hardware reliability," in *Thermal, mechanical and multi-physics simulation and experiments in microelectronics and microsystems (eurosime), 2014 15th international conference on*, 2014, pp. 1–8.

[60] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 80–96, Jan. 2013.

[61] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[62] R. A. P. Oliveira, L. B. R. Oliveira, B. B. P. Cafeo, and V. H. S. Durelli, "Evaluation and assessment of effects on exploring mutation testing in programming courses," in *2015 IEEE Frontiers in Education Conference, FIE 2015, El Paso, TX, USA, October 21-24, 2015*, pp. 1–9.

[63] D. Singh and B. Suri, "Mutation testing tools- an empirical study," in *Computational Intelligence and Information Technology, 2013. CIIT 2013. Third International Conference on, Mumbai, 18-19 Oct. 2013*, pp. 230 – 239.

[64] L. du Bousquet and M. Delaunay, "Towards mutation analysis for lustre programs," *Electr. Notes Theor. Comput. Sci.*, vol. 203, no. 4, pp. 35–48, 2008.

[65] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*, pp. 220–229.

[66] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, 2001.

[67] M. B. Bashir and A. Nadeem, "A state based fitness function for evolutionary testing of object-oriented programs," in *Software Engineering Research, Management and Applications 2009*. Springer, pp. 83–94.

[68] L. du Bousquet and M. Lévy, "Proof process evaluation with mutation analysis," in *Tests and Proofs, 4th International Conference, TAP 2010, Málaga, Spain, July 1-2, 2010*, pp. 55–60.

[69] F. C. Ferrari, A. Rashid, and J. C. Maldonado, "Towards the practical mutation testing of aspectj programs," *Sci. Comput. Program.*, vol. 78, no. 9, pp. 1639–1662, 2013.

[70] H. Kai, P. Zhu, R. Yan, and X. Yan, "Functional testbench qualification by mutation analysis." *VLSI Design*, vol. 2015, pp. 256 474:1–256 474:9, 2015.

[71] Y. Serrestou, V. Beroulle, and C. Robach, "Functional Verification of RTL Designs driven by Mutation Testing metrics," in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, DSD*, Lubeck, Germany, Aug. 2007, pp. 222–227.

[72] V. Guarnieri, G. D. Guglielmo, N. Bombieri, G. Pravadelli, F. Fummi, H. Hantson, J. Raik, M. Jenihhin, and R. Ubar, "On the reuse of TLM mutation analysis at RTL," *J. Electronic Testing*, vol. 28, no. 4, pp. 435–448, 2012.

[73] N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton, and F. Letombe, "Functional qualification of tlm verification," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 190–195.

[74] N. Bombieri, F. Fummi, and G. Pravadelli, "On the mutation analysis of systemc TLM-2.0 standard," in *10th International Workshop on Microprocessor Test and Verification, MTV, Austin, Texas, USA, 7-9 December 2009*, pp. 32–37.

[75] A. Sen and M. S. Abadir, "Coverage metrics for verification of concurrent systemc designs using mutation testing," *IEEE International High Level Design Validation and Test Workshop, HLDVT*, pp. 75–81, 2010.

[76] P. Lisherness and K.-T. T. Cheng, "SCEMIT: a systemc error and mutation injection tool," in *DAC*. ACM, 2010, pp. 228–233.

[77] H. M. Le, D. Große, and R. Drechsler, "Automatic TLM fault localization for systemc," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 31, no. 8, pp. 1249–1262, 2012.

[78] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013.

[79] S. Mukherjee, *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[80] N. J. Wang and S. J. Patel, "Restore: Symptom-based soft error detection in microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 188–201, 2006.

[81] S. Mirkhani, M. Lavasani, and Z. Navabi, "Hierarchical fault simulation using behavioral and gate level hardware models," in *Asian Test Symposium*, 2002, pp. 374–379.

[82] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2012, pp. 123–134.

[83] J. Wei, L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Comparing the effects of intermittent and transient hardware faults on programs," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, ser. DSNW '11, 2011, pp. 53–58.

[84] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 265–276, Mar. 2008.

[85] *The SPARC Architecture Manual*.

[86] *ARM Architechture Reference Manual*.

[87] Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manuals*.

[88] Ht-lab, "CPU86 8088 FPGA IP Core," www.ht-lab.com/freecores/cpu8086/cpu86.html, accessed: 2015-10-02.

[89] Leon3. [Online]. Available: www.gaisler.com/index.php/products/processors/leon3

[90] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09, 2009, pp. 502–506.

[91] The gem5 simulator. [Online]. Available: www.gem5.org

[92] S. LLC. (2004) Simplescalar LLC to serve and project. [Online]. Available: www.simplescalar.com

[93] Mibench. [Online]. Available: wwweb.eecs.umich.edu/mibench

[94] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing architectural vulnerability factors for address-based structures," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 532–543, May 2005.

[95] P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 286–296.

[96] A. Ma, Y. Cheng, and Z. Xing, "Accurate and simplified prediction of AVF for delay and energy efficient cache design," *J. Comput. Sci. Technol.*, vol. 26, no. 3, pp. 504–519, 2011.

[97] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach, "Transient fault models and AVF estimation revisited," in *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, 2010, pp. 477–486.

[98] M. Ebrahimi, L. Chen, H. Asadi, and M. B. Tahoori, "CLASS: combined logic and architectural soft error sensitivity analysis," in *18th Asia and South Pacific Design Automation Conference, ASP-DAC 2013, Yokohama, Japan, January 22-25, 2013*, 2013, pp. 601–607.

[99] V. Sridharan and D. R. Kaeli, "Quantifying software vulnerability," in *Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*, ser. WREFT '08, 2008, pp. 323–328.

[100] F. Restrepo-Calle, S. Cuenca-Asensi, A. Martínez-Álvarez, E. Chielle, and F. Kastensmidt, "Efficient metric for register file criticality in processor-based systems," in *2014 15th Latin American Test Workshop-LATW*. IEEE, 2014, pp. 1–6.

[101] S. Bergaoui, P. Vanhauwaert, and R. Leveugle, "A new critical variable analysis in processor-based systems," *IEEE Transactions on Nuclear Science*, vol. 57, no. 4, pp. 1992–1999, 2010.

[102] J. Lee and A. Shrivastava, "Static analysis to mitigate soft errors in register files," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 1367–1372.

[103] Y. Huang and P. Mishra, "Reliability and energy-aware cache reconfiguration for embedded systems," in *17th International Symposium on Quality Electronic Design, ISQED, Santa Clara, CA, USA, March 15-16, 2016*, pp. 313–318.

[104] J. Hiser, J. W. Davidson, and D. B. Whalley, "Fast, accurate design space exploration of embedded systems memory configurations," in *Proceedings of the 2007 ACM Symposium on Applied Computing SAC, Seoul, Korea, March 11-15, 2007*, pp. 699–706.

[105] A. Ghosh and T. Givargis, "Analytical design space exploration of caches for embedded systems," in *Design, Automation and Test in Europe Conference and Exposition DATE, 3-7 March 2003, Munich, Germany*, pp. 10 650–10 655.

[106] Y. Liang and T. Mitra, "Static analysis for fast and accurate design space exploration of caches," in *Proceedings of the 6th International Conference on Hardware/Software*

Codesign and System Synthesis, CODES+ISSS 2008, Atlanta, GA, USA, October 19-24, 2008, pp. 103–108.

[107] ——, "An analytical approach for fast and accurate design space exploration of instruction caches," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 3, pp. 43:1–43:29, Dec. 2013.

[108] M. Alipour, M. E. Salehi, and H. S. Baghini, "Design space exploration to find the optimum cache and register file size for embedded applications," *CoRR*, vol. abs/1205.1871, 2012.

[109] M. Alipour, H. Taghdisi, and S. H. Sadeghzadeh, "Multi objective design space exploration of cache for embedded applications," in *25th IEEE Canadian Conference on Electrical & Computer Engineering, CCECE*. IEEE, 2012, pp. 1–4.

[110] R. Patel and A. Rajawat, "Instruction cache design space exploration for embedded software applications," in *19th International Symposium on VLSI Design and Test, VDAT, Ahmedabad, India, June 26-29, 2015*, pp. 1–5.

[111] X. Li, H. S. Negi, T. Mitra, and A. Roychoudhury, "Design space exploration of caches using compressed traces," in *Proceedings of the 18th Annual International Conference on Supercomputing, ICS, Saint Malo, France, June 26 - July 01, 2004*, pp. 116–125.

[112] Y. Cai, M. T. Schmitz, A. Ejlali, B. M. Al-Hashimi, and S. M. Reddy, "Cache size selection for performance, energy and reliability of time-constrained systems," in *Proceedings of the Conference on Asia South Pacific Design Automation: ASP-DAC, Yokohama, Japan, January 24-27, 2006*, pp. 923–928.

[113] M. Maghsoudloo and H. R. Zarandi, "Design space exploration of non-uniform cache access for soft-error vulnerability mitigation," *Microelectronics Reliability, 2015*, vol. 55, no. 11, pp. 2439–2452.

[114] S. Wang, J. S. Hu, and S. G. Ziavras, "On the characterization and optimization of on-chip cache reliability against soft errors," *IEEE Trans. Computers*, vol. 58, no. 9, pp. 1171–1184, 2009.

[115] S. D. Carlo, P. Prinetto, and A. Savino, "Software-based self-test of set-associative cache memories," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 1030–1044, 2011.

[116] LLVM download page. [Online]. Available: www.llvm.org/releases/download.html

[117] Building LLVM with CMake. [Online]. Available: www.llvm.org/docs/CMake.html