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

FELIPE ROCHA DA ROSA

Early Evaluation of Multicore Systems Soft Error Reliability Using Virtual Platforms

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

Advisor: Prof. Dr. Ricardo A. da Luz Reis
Coadvisor: Dr. Luciano Ost

Porto Alegre
June 2018

CIP — CATALOGING-IN-PUBLICATION

Rocha da Rosa, Felipe

Early Evaluation of Multicore Systems Soft Error Reliability Using Virtual Platforms / Felipe Rocha da Rosa. – Porto Alegre: PGMICRO da UFRGS, 2018.

166 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2018. Advisor: Ricardo A. da Luz Reis; Coadvisor: Luciano Ost.

1. Multi/Manycore Systems. 2. Fault Tolerance. 3. Reliability. 4. Virtual Platforms. 5. Simulation. 6. ARM. 7. Soft Errors. 8. Machine Learning. I. Reis, Ricardo A. da Luz. II. Ost, Luciano. III. Título.

“Scientists investigate that which already is;
Engineers create that which has never been.”
— DR. ALBERT EINSTEIN

“Scientists dream about doing great things.
Engineers do them.”
— JAMES A. MICHENER

ABSTRACT

The increasing computing capacity of multicore components like processors and graphics processing unit (GPUs) offer new opportunities for embedded and high-performance computing (HPC) domains. The progressively growing computing capacity of multicore-based systems enables to efficiently perform complex application workloads at a lower power consumption compared to traditional single-core solutions. Such efficiency and the ever-increasing complexity of application workloads encourage industry to integrate more and more computing components into the same system. The number of computing components employed in large-scale HPC systems already exceeds a million cores, while 1000-cores on-chip platforms are available in the embedded community.

Beyond the massive number of cores, the increasing computing capacity, as well as the number of internal memory cells (e.g., registers, internal memory) inherent to emerging processor architectures, is making large-scale systems more vulnerable to both hard and soft errors. Moreover, to meet emerging performance and power requirements, the underlying processors usually run in aggressive clock frequencies and multiple voltage domains, increasing their susceptibility to soft errors, such as the ones caused by radiation effects. The occurrence of soft errors or Single Event Effects (SEEs) may cause critical failures in system behavior, which may lead to financial or human life losses. While a rate of 280 soft errors per day has been observed during the flight of a spacecraft, electronic computing systems working at ground level are expected to experience at least one soft error per day in near future. The increased susceptibility of multicore systems to SEEs necessarily calls for novel cost-effective tools to assess the soft error resilience of underlying multicore components with complex software stacks (operating system-OS, drivers) early in the design phase.

The primary goal addressed by this Thesis is to describe the proposal and development of a fault injection framework using state-of-the-art virtual platforms, propose set of novel fault injection techniques to direct the fault campaigns according to with the software stack characteristics, and an extensive framework validation with over a million of simulation hours. The second goal of this Thesis is to set the foundations for a new discipline in soft error reliability management for emerging multi/manycore systems using machine learning techniques. It will identify and propose techniques that can be used to provide different levels of reliability on the application workload and criticality.

Keywords: Multi/Manycore Systems. Fault Tolerance. Reliability. Virtual Platforms. Simulation. ARM. Soft Errors. Machine Learning.

Avaliação de sistema de larga escala sob à influência de falhas temporárias durante a exploração de inicial projetos através do uso de plataformas virtuais.

RESUMO

A crescente capacidade de computação dos componentes multiprocessados como processadores e unidades de processamento gráfico oferecem novas oportunidades para os campos de pesquisa relacionados computação embarcada e de alto desempenho (do inglês, high-performance computing). A crescente capacidade de computação progressivamente dos sistemas baseados em multicores permite executar eficientemente aplicações complexas com menor consumo de energia em comparação com soluções tradicionais de núcleo único. Essa eficiência e a crescente complexidade das cargas de trabalho das aplicações incentivam a indústria a integrar mais e mais componentes de processamento no mesmo sistema. O número de componentes de processamento empregados em sistemas grande escala já ultrapassa um milhão de núcleos, enquanto as plataformas embarcadas de 1000 núcleos estão disponíveis comercialmente.

Além do enorme número de núcleos, a crescente capacidade de processamento, bem como o número de elementos de memória interna (por exemplo, registradores, memória RAM) inerentes às arquiteturas de processadores emergentes, está tornando os sistemas em grande escala mais vulneráveis a erros transientes e permanentes. Além disso, para atender aos novos requisitos de desempenho e energia, os processadores geralmente executam com frequências de relógio agressivos e múltiplos domínios de tensão, aumentando sua susceptibilidade à erros transientes, como os causados por efeitos de radiação. A ocorrência de erros transientes pode causar falhas críticas no comportamento do sistema, o que pode acarretar em perdas de vidas financeiras ou humanas. Embora tenha sido observada uma taxa de 280 erros transientes por dia durante o voo de uma nave espacial, os sistemas de processamento que trabalham à nível do solo devem experimentar pelo menos um erro transiente por dia em um futuro próximo. A susceptibilidade crescente de sistemas multicore à erros transientes necessariamente exige novas ferramentas para avaliar a resiliência à erro transientes de componentes multiprocessados em conjunto com pilhas complexas de software (sistema operacional, drivers) durante o início da fase de projeto.

O objetivo principal abordado por esta Tese é desenvolver um conjunto de técnicas de injeção de falhas, que formam uma ferramenta de injeção de falha. O segundo objetivo desta Tese é estabelecer as bases para novas disciplinas de gerenciamento de confiabilidade considerando erro transientes em sistemas emergentes multi/manycore utilizando aprendizado de máquina. Este trabalho identifica multiplicas técnicas que podem ser usadas para fornecer diferentes níveis de confiabilidade na carga de trabalho e na criticidade do aplicativo.

Palavras-chave: Sistemas Multi/Manycore, Tolerância a Falhas, Confiabilidade, Plataformas Virtuais, Simulação, ARM, Falhas Transientes, Aprendizado de Máquina .

LIST OF ABBREVIATIONS AND ACRONYMS

ASIC	Application-specific Integrated Circuit
ABNT	Associação Brasileira de Normas Técnicas
COTS	Commercial Off-the-shelf
DSE	Design Space Exploration
DSE	Design Space Exploration
DBT	Dynamic Binary Translation
DRAM	Dynamic Random-Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
FIM	Fault Injection Module
GPU	Graphics Processing Unit
HPC	High Performance Computing
ISA	Instruction Set Architecture
JIT	Just-in-time
MPI	Message Passing Interface
MIPS	Million Instructions per Second
MBU	Multiple Bit Upset
MET	Multiple Event Transient
nm	Nanometer
OVP	Open Virtual Platforms
OVPsim	Open Virtual Platforms Simulator
OS	Operating System
PE	Processing Units
SDC	Silent Data Corruption
SMT	Simultaneous Multithreading

SEU	Single Event Upset
SER	Soft Error Rate
SMP	Symmetric Multi-Processor
KIPS	Thousand Instructions per Second
TMR	Triple Modular Redundancy
VA	Virtual Address
VP	Virtual Platform
VP-FIM	Virtual Platform Fault Injection Module

LIST OF FIGURES

LIST OF TABLES

CONTENTS

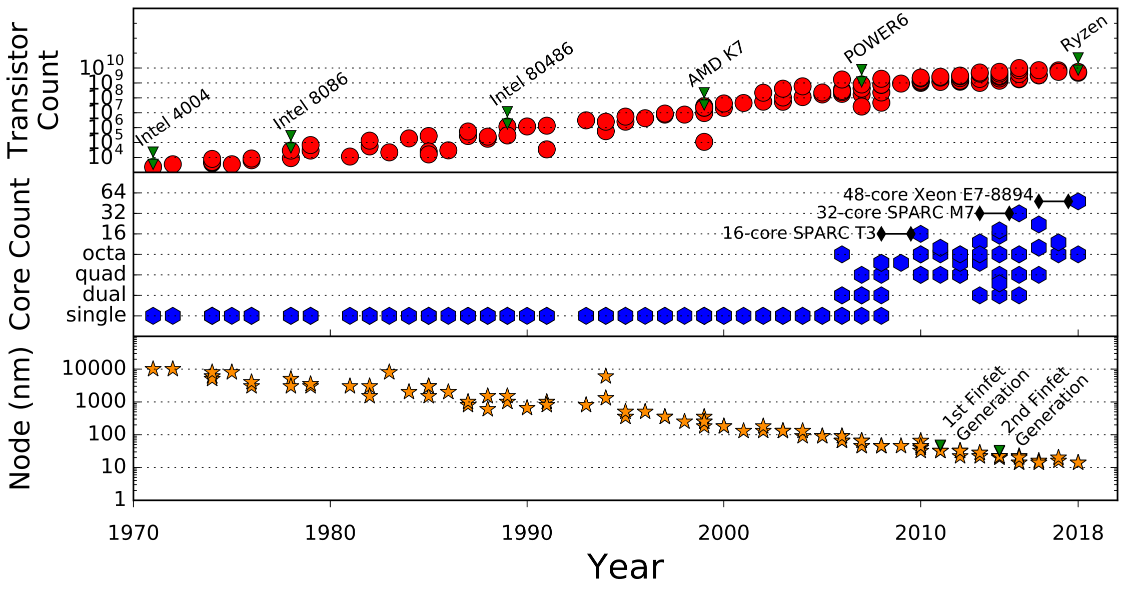
1 INTRODUCTION

Computers become ubiquitous in our modern society ranging from everyday life devices (e.g., televisions, vending machines, smartphones) to complex systems such as those used to weather forecast or search for microscopic subatomic particles (BUCHANAN, 2018; Inside HPC, 2018). The continuous technology scaling (BORKAR; CHIEN, 2011) and the advance of multicore components such as processors and graphics processing unit (GPUs) are driving the microelectronics industry forward. This evolution can be exemplified by the advanced driver-assistance systems (ADAS),(VLACIC; PARENT; HARASHIMA, 2001; MCCLUSKEY, 2017; JONES, 2002; BISHOP, 2005) enabling self-driven cars in the near future (GEIGER; LENZ; URTASUN, 2012a). The emerge of *The Internet of Things* (IoT) is another example(PERERA et al., 2014; ZANELLA et al., 2014) which is expected to integrate about 30 billion of devices by 2020 (NORDRUM, 2016). The semiconductor systems dissemination phenomenon was possible (or caused) by Gordon Moore's seminal work (MOORE, 1965) on transistor scaling. This work introduces the famous Moore's Law which states that the number of transistors per square inch doubles every 18 months. As a consequence, every new technology node had delivered increasing performance, lower power consumption, and smaller transistor cost. The top plot of Figure 1.1 displays the increasing number of transistors in microprocessors since 1970.

Single-thread processors have benefited from technological advancements to improve their performance by increasing clock frequency (BORKAR; CHIEN, 2011). However, in mid-2000, this trend reaches the physical limits due to the increased power consumption and the current density within the chip (ESMAEILZADEH et al., 2011; ESMAEILZADEH et al., 2012). The central plot of Figure 1.1 shows the growing number of cores in commercial processors over the last decades. Integrating modern multicore processors (e.g. big.LITTLE (ARM, 2017)) and GPUs in the same system is now commonplace in both embedded and high-performance computing (HPC) domains (DINECHIN et al., 2013; BORKAR; CHIEN, 2011). Such systems aim to perform complex software stacks (i.e., operating system OS, drivers, and applications) from diverse fields (e.g., spatial, avionics, automotive). However, the ever-increasing demand for performance, energy efficiency and high reliability of emerging systems is imposing a myriad of challenges to the design of underlying systems:

- *programmability*, ease of programming is a feature of paramount importance in

Figure 1.1: Evolution of commercial processors during the last decades considering the number of transistors (top), number of cores (middle), and associeted technology node (botton) from 1970 to 2018. Information gathered from multiple sources including the ITRS (https://www.itrsgroup.com/).



Source: The Authors

large-scale systems composed of different processors, resulting in different platform libraries (e.g., MPI, OpenMP), compilers, instruction set architecture (ISAs) (GARIBOTTI et al., 2013).

- *security*, with the increasing number of components and devices sending and receiving user sensitive data, providing a secure service is fundamental. The increasing system complexity introduces vulnerabilities to software and hardware architectures compromising the system behavior. Attackers can exploit such vulnerabilities to introduce malicious code, which may incur in undesired effects, security breaches, or damage to a system (MCCLUSKEY, 2017).

- *energy efficiency*, while the constant supply and threshold voltage scaling in transistors led to an exponential increase of leakage current. In addition to that, other physical restrictions related to device packaging, intra-chip current distribution, and cooler dissipation during power peaks further impact on the systems energy consumption (ZHANG et al., 2013). Energy-efficiency is becoming more critical than high-speed operation, and dark silicon era is imposing more power-oriented constraints to the design of such systems.

- *reliability and dependability*, the technology transistors reach the operation physical limits, thus becomes increasingly difficult for the hardware components to achieve reliable execution. The unreliability of multicore-based systems is emerging from several sources, e.g., electrical noise, cross-talk, radiation particles, aging, and variability (KARNIK; HAZUCHA, 2004).

Reliability is rapidly emerging as a significant design metric in both embedded and HPC domains. The increasing chip power densities allied to the continuous technology shrink is making emerging multicore-based systems more vulnerable to soft errors, such as the ones caused by radiation events (KARNIK; HAZUCHA, 2004). As illustrated in the bottom graph of Figure 1.1, commercial processors based on $10\,\mathrm{nm}$ process node are likely to be available in the market in the coming years. Until recently, radiation-induced faults issue was relegated to high-availability systems such as military applications and radiation-bounded systems as spatial and avionics. Nowadays, the occurrence of soft error appears as a primary concern of electronics systems working at ground level (MUKHERJEE, 2008). A transient error, also known as a soft error, induced by radiation particles can lead to financial or human life losses (YOSHIDA, 2015). For instance, the occurrence of a soft error in HPC systems could lead to the underutilization of resources, which results in extra cost and time wasted waiting for the re-execution of applications/jobs. Although, a supercomputer with 900 compute nodes registered a rate of 0.15 error per day, over a year of operation (BAUTISTA-GOMEZ et al., 2016), electronic computing systems working at ground level are expected to experience at least one soft error per day in near future (GRANLUND; GRANBOM; OLSSON, 2003).

The resulting growing susceptibility of multicore systems to soft errors necessarily calls for novel cost-effective tools to assess the soft error resilience of underlying systems early in the design phase. The preceding context provides the *motivation* for this Thesis, which aims at investigating novel fault injection techniques and tools that can be used to assess soft errors of multicore-based systems under fault campaigns at early design time.

1.1 Hypothesis to be demonstrated in Thesis

This Thesis relies on two hypothesis:

- Enhancing virtual platforms with fault injection capabilities enable to speed up the evaluation of more realistic multicore systems (i.e., real software stacks, state-of-the-art ISAs) at early design phases. The use of such fault injection frameworks increases the probability of generating soft errors and failures in those multicore systems, which allows to generate and collect more error/failure-related data in a reasonable time.

- With large error/failure-related data sets, engineers are more likely to identify meaningful relationships or associations between fault injection results, application characteristics, and platform parameters. When dealing with large failure-related data sets obtained from the fault campaigns, it is essential to filter out non-correlated features (i.e., Parameters without a direct relationship with the system reliability). In this regard, the second hypothesis of this Thesis is that the use of supervised and unsupervised machine learning techniques are appropriate to filter and identify the correlation between fault injection results and application and platform characteristics, enabling them to improve existing fault mitigation techniques, as well as to investigate and propose new and more efficient ones.

1.2 Thesis Goal

In order to address the hypothesis mentioned above, the strategic goal of this Thesis is first to combine existing and novel simulation and fault injection techniques into virtual platforms, targeting fast and detailed soft error reliability exploration of state-of-the-art multicore systems. The second strategic goal of this Thesis is investigate appropriete machine learning techniques to develop an automated engine capable of searching and identifying the individual (or combinations of) microarchitectural (e.g., instruction types, memory stats) and software parameters (e.g., number of branches, etc), which present the most substantial relation relationship with each detected soft error and failures.

To accomplish the first strategic goal, the following specific objectives should be fulfilled:

- Identify the most suitable and efficient virtual platforms to include fault injection capabilities, aiming to support the soft error analysis of state-of-the-art processor models;

- Port of several benchmarks from embedded and HPC domains, including the Rodinia and NASA NAS Parallel Benchmark (NPB) suites;

- Investigate the soft error analysis consistency between an instruction-accurate VP and a cycle-accurate full system simulator;

- Proposal and development of novel fault injection techniques and tools that enable to trace, evaluate, and identify particular source of errors (e.g., application functions, code structure).

The second goal requires the following task to be achieved:

- Support the use of machine learning techniques to enable the identification of individual (or combinations of) microarchitectural and software parameters that present the most substantial relation relationship with each detected soft error or failure.

- Employ proposed techniques and developed tools to investigate the impact of software and hardware parameters on soft error system resiliency, considering a significant number of fault injection campaigns.

1.3 Original Contributions of this Thesis

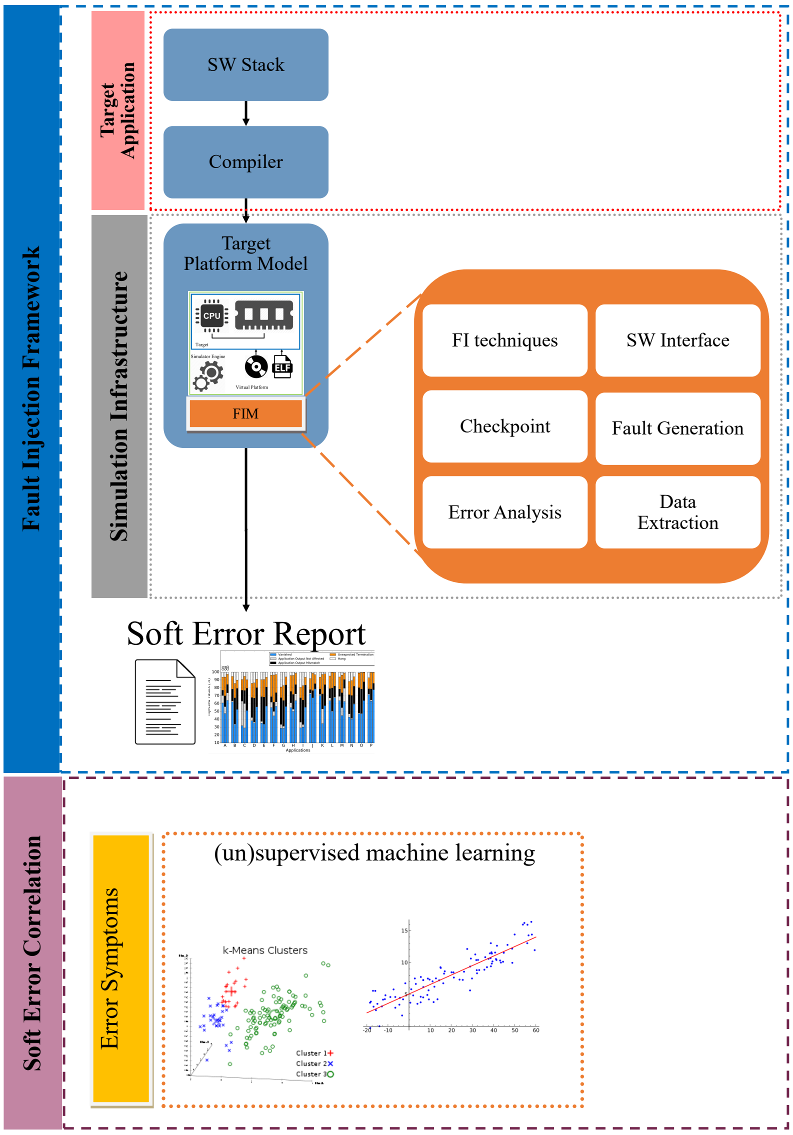
Figure 1.2 illustrates the main contributions of this work, which are joined into a soft error analysis flow that includes fault injection and soft error analysis extensions(fully described in Chapter 3), and automated soft error correlation using machine learning techniques (described in Chapter 5).

The main contributions of this Thesis are described as follows:

1.3.1 Early soft error evaluation

Proposal of two flexible fault injection (FI) frameworks: the OVPsim-FIM developed on top of the OVPsim (IMPERAS, 2017) and the gem5-FIM, which relies on the gem5 (BINKERT et al., 2011). Both frameworks integrate a set of fault injection techniques, allowing fast soft error susceptibility exploration considering state-of-the-art multicore processor architectures, such as ARMv7, ARMv8, and big.LITTLE.

Figure 1.2: Diagram with the design space exploration flow and this thesis contributions.



Source: The Authors

1.3.2 Novel non-intrusive fault injection techniques

Random fault injection homogeneously probes the application (i.e., every function has an equal fault probability), nevertheless, some code segments are more critical than others to the system reliability. The OVPsim-FIM was extended with four novel and non-intrusive FI techniques targeting: (i) the virtual memory, : (ii) variables, : (iii) function code, and : (iv) function execution. The underlying techniques offer flexibility and full control over the fault injection process when targeting complex software stacks. Further, this new module provides a more powerful fault inspection module, enabling the user to add custom verifications. In other words, it is possible to check data structures and execution patterns during or after the application simulation. For example, this new module reduces false-positive SDC (i.e., silent data corruption) detections by checking only a select group of variables instead of the entire memory. This tool provides software engineers with detailed and comprehensive fault injection capabilities to explore critical code sections in a quick and non-intrusive manner.

1.3.3 Instruction-accurate fault injection consistency

Instruction-accurate simulators provide a simulation performance of thousands of millions of instructions per second (MIPS), enabling quick explorations of large and complex scenarios. However, its accuracy regarding soft error assessment was never addressed and to investigate this matter, the proposed OVPsim-FIM (i.e., instruction-accurate simulator) was compared against the cycle-accurate gem5-FIM. This exploration comprises millions of fault injections. Results show that the unmodified OVPsim-FIM can achieve an average mismatch of less 25% when considering the gem5-FIM. This work further explores the OVPsim-FIM engine by investigating different configurations. With the proper simulator settings, the average mismatch can be reduced to less than 15%. More interesting, the worst-case mismatch can be reduced by fivefold while sustaining the high-performance simulation suitable to early design space explorations.

1.3.4 Extensive investigation of the software stack impact on the system reliability

This work uses the proposed FI framework scalability to explore early design decisions impact on the system reliability, e.g., architecture, number of cores, ISA, OS, parallelization library, among other possible configurations. Different from other works, the promoted frameworks use a realistic software stack comprising multiple unmodified operating systems (e.g., Linux 4.3, Linux 3.13, FreeRTOS) alongside parallelization libraries (e.g., OpenMP, MPI, PTHREADS, and OmpSs). Further, this work addresses another common issue of fault injection frameworks: *performance* and *scalability*. The developed fault injector adopt three simulation techniques to increases the soft error analysis performance: *(i)* host multicore parallelization, *(ii)* checkpoint and restore technique, and *(iii)* large-scale and distributed simulation, targeting its use on supercomputers.

This extensive evaluation considers more than three million fault injections (requiring up to three million of simulation hours) targeting 45 distinct benchmarks, considering serial, MPI, and OpenMP implementations from the NAS Parallel Benchmark (NPB) suite (BAILEY et al., 1991) and the Rodinia Benchmark suite (CHE et al., 2009) among others. This exploration targets single, dual, quad, and octa-core ARM Cortex-A9 and ARM Cortex-A72 processor models. The investigation shows distinct effects of the chosen parallelization (e.g., OpenMP vs. MPI) library on the system fault tolerance, also, how the ISA decision can impact the system behavior under fault influence.

1.3.5 Correlating Soft Errors and Microarchitectural Data

Converting fault injection explorations into actual system reliability improvements is not a straightforward process. This Thesis proposes a cross-layer investigation toolset that uses machine learning techniques to perform multi-variable and statistical analysis using the gem5 microarchitectural information (e.g., memory usage, application instruction composition) along with other software profiling tools (e.g., line coverage) that are combined with soft error vulnerability evaluation results (i.e., fault injection campaigns). Proposed toolset enables to reduce the number of fault injection campaigns required during early design space exploration by using software symptoms (e.g., execution time, number of branches) correlated with soft error vulnerabilities to improve the target application reliability. Developed toolset provides users with a flexible investigation, where several information sources can be easily included, selected, and conformed to different machine learning investigation techniques.

1.4 Thesis Outline

This Thesis is organized into six chapters. Chapter 1 introduces the reliability issues in modern system design and this Thesis contributions. The following paragraphs present a succinct Thesis summary chapter by chapter.

Chapter 2 - Background on Soft Errors: The first section of this chapter (Section 2.1) introduces the modern challenges to modern electronic devices then Section 2.2 presents a brief background on soft errors history and source mechanisms. Section 2.3 presents the state-of-the-art on soft error assessment using distinct approaches.

Chapter 3 - Simulation-based Fault Injection Using Virtual Platforms: First, Sections 3.1 and 3.2 provide a discussion on the available virtual platforms and their fault injection frameworks. Section 3.3 describes the adopted fault model while Section 3.4 presents the fault injection flow. The fault injection framework detailed development is shown by Sections 3.5 and 3.6 targeting the gem5 and OVPsim simulators respectively. Several features had been included in the fault injection framework including speed boosting techniques (Section 3.7). Section 3.8 provides extra tooling to help software developers to explore in deep the application effects under soft errors this chapter proposes two FIM extensions: (i) include new software-focused fault injection techniques to restrict the targeting area, e.g., a particular function or data representation. (ii) provide a non-intrusive library to customize the error analysis and detection to accommodate the user necessities.

Chapter 4 - Evaluation of Proposed Fault Injection Framework: This chapter presents results for several fault injection campaign using dozens of applications in hundred of distinct scenarios. Section 4.1 describes the 45 adopted applications from the Rodinia benchmarks and the NAS Parallel Benchmark. The chapter is divided in three main parts: Section 4.2; Performance and Accuracy Evaluation of proposed OVPsim-FIM. Section 4.3; Soft Error Evaluation Considering Multicore Design Metrics/Decisions. Section 4.4; Focused Fault Injection Preliminary Results.

Chapter 5 Machine Learning Applied to Soft Error Assessment in Multicore systems This chapter describes the promoted a cross-layer investigation tool which performs multivariable and statistical analyses. First, Section 5.1 debates the state-of-the-art of reliability using machine learning techniques. Sections 5.2 and 5.3 discusses, respectively, the problem of investigating large-scale fault injection campaigns and how this work mitigates this issue. The proposed tool requires multiple machine learning techniques which are described by Section 5.4, while Section 5.5 details the tool execution flow. Finally, Section 5.6 shows results related to multiple investigations using the proposed tool.

Chapter 6 - Conclusion and Future Works: This chapter summarizes this work contribution until this point. Also, it describes the future works related to this Thesis.

2 BACKGROUND ON SOFT ERRORS

This chapter details some necessary backgrounds and state-of-the-art works related to this Thesis exploration on soft error assessment. First, Section 2.1 enumerates several reliability challenges encountered by modern electronic devices, while Section 2.2 provides a brief history and background on radiation-induced soft errors. In particular, Section 2.2.1 explains the particle strike and the charge accumulation process, Section 2.2.2 shows several fault masking mechanisms, and Section 2.2.3 introduces useful metrics to evaluate the occurrence of soft errors. The Section 2.3 investigates the recent innovations in soft error vulnerability assessment.

2.1 Main Reliability Challenges in Electronic-based Systems

The semiconductor industry is facing significant reliability challenges to guarantee the correct functionally of electronic systems (KARNIK; HAZUCHA, 2004). Problems such as process variability, permanent faults, and transient faults are a significant issues for semiconductor-related industry sectors (HENKEL et al., 2013; SHIVAKUMAR et al., 2002). Three main fault groups comprehensive encompass the device reliability challenges for the future technologies:

(i) **Process Variability**: As the transistor's features scales down the chip variability grows during the fabrication process (PANDINI, 2009) impacting multiple parameters, e.g., channel length, doping concentration, oxide thickness. For comparison's sake, state-of-the-art fabrication processes 10 nm shapes are only four times larger than the diameter of a DNA strand. Consequently, devices with identical logical design have distinct physical and electrical characteristics from die to die (i.e., inter-die variations) and on the same die (i.e., intra-die variations). These issues result in yield, power, and performance reduction as the same design must withstand a wider parameter variation (BORKAR et al., 2003).

(ii) **Permanent faults**: are physical imperfections such as stuck-at-zero and timing violations in conjunction with aging problems. For instance, decreasing transistor sizes cause faster aging and eventually transistor wear out due to distinct phenomena such as Hot Carrier Injection (HCI), Bias Temperature Instability (BTI), Electromigration, and Time Dependent Dielectric Breakdown (TDDB) (ALAM et al., 2007). These problems reduce the chip average lifetime and impact on its meantime between failures (MTBF) by steadily increasing propagation delays.

(iii) **Transient faults**: or soft errors encompass all sort of malfunctions without permanent circuit damage. Such errors may occur due to the occurrence of electrical noise, electromagnetic interference, as well as exposition to radiation. Soft errors cause single event effects (SEEs) in a processor, which can be propagated through logical (e.g., logic gates) and memory elements (e.g., latches, registers). Whenever an SEE surpasses a specific charge threshold[1] it will induce an incorrect computation affecting logical and storage elements. In contrast to permanent faults and process variability, new data can still be correctly written and stored on the affected device. Due to the technology high-frequency, low voltage supply, and continuous technology shrink the transistor becomes more susceptible to soft errors as the minimal energy to provoke it is reducing (SEIFERT, 2010; BAUMANN, 2005).

While process variability increases the production cost and development time, permanent faults lead to premature wearing (i.e., reducing the system's lifetime). The technology fabrication process and new low-level system designs (i.e., gate-level) can partially mitigate its occurrence. Also, it affects appears during a prolonged time span enabling early detection and correction. In contrast, transient faults will introduce erroneous behavior in the system at random time without previous warning or at predictable rates. Soft errors may emerge either during intensive or idle working periods, affecting both critical and non-essential system functionalities. Among aforementioned reliability challenges, soft errors are the most prominent one in several sectors of the semiconductor industry. Processor-based systems working at sea level are expected to experience at least one soft error per day (GRANLUND; GRANBOM; OLSSON, 2003).

2.2 Radiation Induced Soft Errors

Soft Errors caused by radiation particles became in the last years a recurring research topic in both academia and industry (SLAYMAN, 2010; LI et al., 2016). Nevertheless, the radiation effects on semiconductor materials are well-known for more than half-century. In 1961, (WALLMARK; MARCUS, 1961) predicted the transistor scaling limits arround $10\,\mu\mathrm{m}$ for several reasons, including cosmic rays. A decade later in 1975,

[1]This critical threshold depends on the technology node, cell design, and neighboring gates.

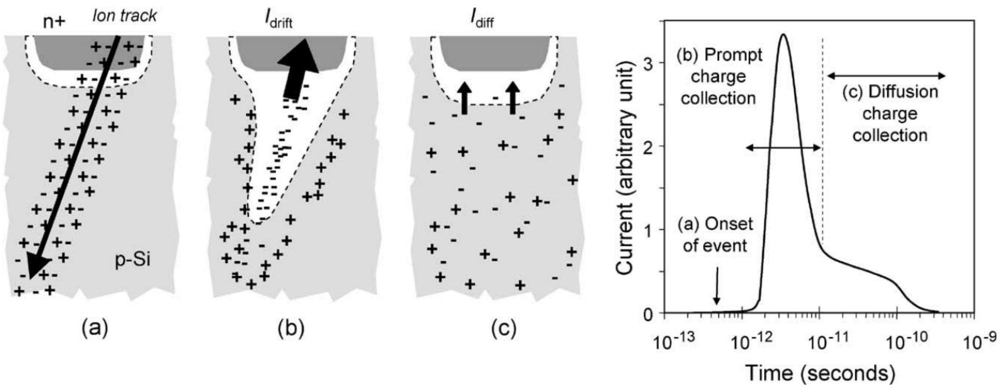
the misbehavior of digital circuits used in a communication satellite was identified and studied by (BINDER; SMITH; HOLMAN, 1975). Binder *et al.* reported galactic cosmic rays effects in the processor flip-flops as the principal error source. In a retrospective investigation, (NORMAND et al., 2010) found an unusual number of parity errors in the Los Alamos Cray-1 supercomputer during the year of 1976, pointing to ground-level high-energy neutrons as its cause. DRAM soft error vulnerability to heavily-ionizing radiation are discussed by (MAY; WOODS, 1978), and (ZIEGLER; LANFORD, 1979) investigate the silicon interactions at sea-level with cosmic-ray nucleons and muons. The next subsection presents examples of radiation strike mechanisms. Afterward, Section 2.2.2 discusses fault masking mechanisms during and fault propagation, Section 2.2.3 shows some soft errors quantification metrics, and Section 2.2.4 explores the trends of soft errors in future systems.

2.2.1 Radiation Source and Soft Errors Mechanisms

According to Baumann *et al.* (BAUMANN, 2005) soft errors induced by radiation originate from three primary sources: (*i*) the emissions of **alpha particles** due to the presence of radiative impurities on the chip packaging; (*ii*) an alpha particle (i.e., two neutrons and two protons) traveling through a semiconductor material loses kinetic energy leaving an ionization trail behind; and (*iii*)**high-energy cosmic rays** originating from outer space, which produce a complex cascade of secondary particles in earth's atmosphere (e.g., muons, protons, neutrons, and pions). For example, a neutron collision with one Si atom emits lighter ions and sub-particles (e.g., protons, alpha particles). **Low-energy cosmic rays** (i.e., up to 1 MeV) create ionizing particles in electronic devices from the interaction of neutrons and borons atoms (i.e., p-type dopant).

The collision of a sub-atomic particle induces a single event transient (SET) by generating secondary particles capable of ionizing the n-p junctions of sensitive transistors causing a voltage charge or discharge in the stroke node. Modern and smaller technology can also suffer from multiple event transients (MET) as the same particle induces a charge in several neighboring transistors. For simplicity's sake, this subsection focuses on single event effects. Figure 2.1 (a-c) shows a high-energy ion path through a reverse-biased junction, i.e., the most charge-sensitive circuit node (BAUMANN, 2005; LI et al., 2016). The Figure 2.1a displays the corresponding current pulse resulting from the following three phases:

Figure 2.1: Charge generation and collection phases in a reverse-biased junction and the resultant current pulse caused by the passage of a high-energy ion.



Source: Baumann, R.C., figure adapted from (BAUMANN, 2005).

(i) The high-energy particle interaction with the silicon transfers kinetic energy to the semiconductor material creating a track of electron-hole pairs and forming a conical high carrier concentration in the wake of the energetic ion's passage (Figure 2.1a).

(ii) The electric field in the depletion region collects the closest charge carriers creating a transient current/voltage at the target device node (Figure 2.1b). During this phenomenon, a temporary channel may be formed for a short period, around few picoseconds.

(iii) A nanosecond later the diffusion begins to dominate the collection process Figure 2.1c, inducing additional current formation.

The radiation event deposits a certain amount of charge (Q_{all}) due to the hole-pair formation (IBE et al., 2010; HUBERT; ARTOLA, 2013). The collection mechanism described above has a maximum efficiency coefficient (L_{max}) and depends on the ion track length (x_c). For instance, the total amount of collected charge by a SRAM is giving by Equation (2.1) (IBE et al., 2010):

$$Q_{collected} = Q_{all}\left(\frac{x_c}{L_{max}}\right) \tag{2.1}$$

$Q_{collected}$ values range from one to several hundred pC, and the precise $Q_{collected}$ estimation involves the ion strike angle, path, energy, mass, and point of impact considering the nearest reverse-biased junction. The device geometry and electrical characteristics have additional influence on the collection process. The displacement of charge carriers in the time creates an electrical current in the target devices modeled by Equation (2.2):

$$I(t) = \frac{Q_{collected}}{\tau_\alpha - \tau_\beta}\left(e^{-\frac{t}{\tau_\alpha}} - e^{-\frac{t}{\tau_\beta}}\right) \tag{2.2}$$

τ_α and τ_β are process dependent constants denoting the collection time and ion-track establishment time. The literature reports a typical value of $164\,\mathrm{ps}$ for the τ_α and $50\,\mathrm{ps}$ for the τ_β (PALAU et al., 2001; LI et al., 2016). The charge collection in the stroke node may lead to a single event upset (SEU), in other words, introducing an incorrect bit in the memory cell. The quantity of $Q_{collected}$ required to create an SEU in a device is denoted by the critical charge ($Q_{critical}$) and expressed by:

$$Q_{critical} = \int_0^{T_F} I_D(t)\,\mathrm{d}t \tag{2.3}$$

In the Equation (2.3), $I_D(t)$ represents the time-dependent drain transient current, and (PALAU et al., 2001) defines the flipping time (T_F) as the time instant when the struck transistor drain voltage is equal to the gate voltage after the radiation event. In simple circuits such as DRAMs, an error only occurs whenever the $Q_{critical}$ is greater than $Q_{collected}$ masking otherwise the radiation event. The SRAM feedback loop can restore the original value if the recovery time[2] does not exceed the feedback time[3] (DODD; SEXTON, 1995).

2.2.2 Fault Propagation and Masking

The transient errors create unintentional electrical signals that need to travel through many design abstraction layers from the transistor until reaching the application variables and control flow. Several mechanisms can mask this fault: First, a particle strike needs to generate enough charge collection to create a noticeable electrical signal, which depends on the transistor electrical characteristics. The signal propagation also is attenuated by the circuit resistance (*electrical masking*) The internal logic structure leads to *logical masking* when the fault propagation path is blocked by another dominating data path. The circuit timing requirements (e.g., setup and hold times) constrain all electrical signals, including faults. Whenever the faulty signal violates one delay constraints (i.e., the signal arrives either too early or too late to be captured during the clock edge) results in a *temporal masking*.

[2]Time taken for the struck node voltage to return to its pre-strike value.
[3]The time taken for the struck node voltage to become latched as incorrect data.

Digital systems present two main circuit components: Combinational (e.g., AND, OR, XOR) and sequential circuits (e.g., SRAM cells, latches, flip-flops). Combinational circuits are regarded as less prone to soft errors due to the above-described masking mechanisms and the absence of feedback loop in the underlying circuit. In turn, the sequential circuits are more vulnerable to radiation events as a single strike has enough energy to reverse the stored data as result of the feedback loop. Also, memory elements are susceptible to bit-flips during extended periods of time when holding data, in contrast, the sequential circuit switches (i.e., changes the value) more often. For example, an SRAM cell may be affected by SEUs during almost the entire clock cycle (SEIFERT, 2010). Also, the clock network under the influence of soft errors results in memory elements incorrect operation.

At *architecture level* (e.g., program counter, pipeline registers, register-file, arithmetic, and logic unit), an erroneous bit can be further masked due to the write and the read operations. For example, a fault present in a register can be overwritten by a write before a read, and thus eliminate the incorrect bit. An error is a fault that propagated inside the application (or OS) before being perceived by the user. At the software level, SEUs are incorrect variable values or wrong control flow executions. An algorithm can overwrite the variable before its value is consumed masking the fault. Even with all those masking processes, the incidence of soft errors is increasing due to the technology susceptibility to radiation events. Well defined metrics and methods are necessary to analyze the impact of transient faults under different conditions considering system architecture, application, and compiler.

2.2.3 Soft Error Metrics

This subsection describes some useful soft error metrics: The transient errors occurrence per time are quantified by the *Soft Error Rate* (SER) and measured by the *Failure-In-Time* (FIT) parameter. The FIT is equivalent to the number of failures (i.e., soft errors in this context) in one billion of operation hours (BAUMANN, 2005).

A well-established metric is the architectural vulnerability factor (AVF) (MUKHERJEE et al., 2003) that estimates a particular bit susceptibility to create a visible error in the application. The architectural bits can be divided into two subgroups: (1) The bits required for an *architecturally correct execution* (ACE) and (2) the un-ACE bits. While an ACE bit propagates faults to the final output, un-ACE bit does not create a visible error. The used of AVF enables the search for the most vulnerable architectural state bits. The AVF focus on the correct result, and thus bit fault can change the intermediary computations and still have an AVF of 0%. For instance, a transient fault in a branch predictor generates a miss-prediction, which potentially requires the re-execution of some instructions without altering the final application.

The AVF lacks an explicit masking rate model that varies according to the hardware component, which may lead to an over-estimation the number of errors. The register vulnerability factor (RVF) (YAN; ZHANG, 2005) is a metric explicitly used to measure the register file susceptibility to soft errors. The RVF accounts for the vulnerability factor by calculating the intervals between two vulnerable register operations (i.e., write/read and read/read). Write operations mask any fault previously propagated, and thus, the interval between writes is the most vulnerable register operations.

The AVF shows a larger granularity than desired to measure the instructions interaction. To address this issue, (BORODIN; JUURLINK, 2010) create an instruction-based criticality assessment metric called Instruction Vulnerability Factor (IVF). This metric uses distinct fault injection techniques to probe every instruction in the code. Given the involved complexity, complete coverage can be difficult to achieve. Another instruction-oriented approach is the Instruction Vulnerability Index (IVI) (REHMAN et al., 2011). Its composition includes the ACE-bits, the area, and specific pipeline components vulnerability. This analytical approach avoids the exhaustive simulation by using a fault probability in each pipeline component. Also, the IVI metric allows the extension to register file IVI by incorporating the vulnerability window of each register in the instruction[4].

2.2.4 Soft Error Trends in Electronic Systems

In the last decades, the Moore's and the Dennard scaling laws coupled both transistor features and V_{dd} scaling down. Nevertheless, the V_{dd} has a direct impact on memory cells sensitivity to radiation events due to the charge reduction to achieve the $Q_{critical}$. On larger designs as flip-flops[5] the area reduction plays a significant role to improve the cell reliability as V_{dd} reduction impacts on SRAM cells reliability. The scaling process reduces the cell sensible areas, consequently, the total collected charge (DIXIT; WOOD, 2011;

[4]The vulnerability window is the time between a register write and the last read in the value. During this period, the register is susceptible to propagate soft errors.

[5]When compared with SRAMs.

EBRAHIMI et al., 2016). Nevertheless, the transistor density creates a new phenomenon: a single particle strike can induce charge generation in several neighboring cells (i.e., MET - multiple event transients), which may result in Multiple-Bit Upset (MBU). Also, MET events in sequential logic reduce the electrical masking probability due to the multiple concurrent propagations paths.

The difficulties in downscaling the planar CMOS technology lead the major semiconductor foundries, such as Intel, TSMC, and GlobalFoundries, to adopt the multi-gate nonplanar transistor technology also known as Fin Field-Effect Transistor(FinFET). (DOYLE et al., 2003; RUSS, 2008). Radiation effects as SETs and SEUs have been studied and analyzed in bulk CMOS (HUBERT; ARTOLA, 2013), and more recent FinFET has been proven equally susceptible to neutron and alpha particles strikes (HUBERT; ARTOLA; REGIS, 2015; ARTOLA; HUBERT; SCHRIMPF, 2013).

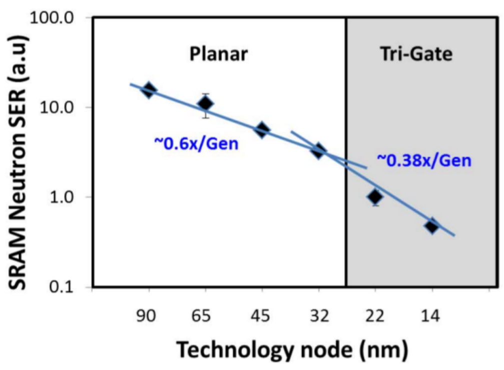
The FinFET technology improves the system soft error reliability by reducing the gate width in favor of the height forming a tri-gate shape (SEIFERT et al., 2015). The tri-gate architecture reduces the drain/source area (i.e., reverse-biased junctions), and by consequence, diminishing the charge collection process. This phenomenon reduces the total collected charge and the critical charge[6] as shown in Figure 2.2b. Figure 2.2a shows an SRAM cell SER rate transition from planar nodes to FinFET tri-gate nodes. Seifert *et al.* (SEIFERT et al., 2015) reports a 23 times SER improvement on a second generation 14-nm SRAM cell when compared to a 32-nm planar technology at nominal voltage. The amelioration from the first FinFET generation (i.e., 22-nm) to the second (i.e., 14-nm) achieves up to 8 times for the same cell. The newer generation is taller and slimmer than the previous one, and so further reducing the device sensitive areas.

The ITRS (KAHNG, 2013) reports soft errors as one major design issues to the sub-22nm nodes. Although FinFET-based systems improvements, the trend points to more vulnerable architectures (SEIFERT et al., 2015; LI et al., 2016) as the memory integrated volume grows at each new architecture (e.g., larger caches and more complex pipelines). Further, most works on circuit reliability consider only the system execution at nominal voltage and temperature, which does not reflect the reality and can affect directly the FinFET reliability (ROSA et al., 2015a; ROSA et al., 2015b). The high-energy neutrons are becoming the primary source of soft error at ground-level, with up to 77% (Figure 2.2a), surpassing the alpha particles. To ensure the system's reliability or at least fail-safe functionality, the designer should be able to identify soft errors during the ini-

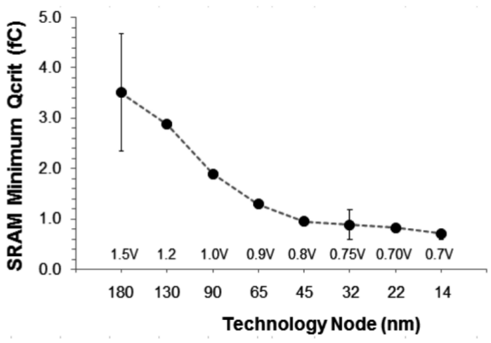
[6]Amount of induced charge in an SRAM node to change it data status.

Figure 2.2: FinFET reliability future trend across new technology nodes.

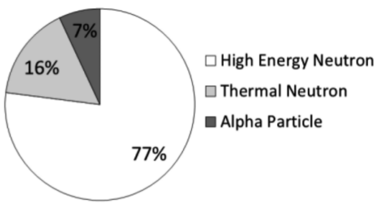
(a) SER rate from planar to tri-gate nodes.



(b) Critical charge accros differente technologies.



(c) Radiation-induced soft errors sources into 14-nm FinFET technology.



Source: Seifert *et al.* (SEIFERT et al., 2015)

tial design cycle. Aiming at accelerating the fault injection evaluation at early design phases, researchers are investigating fast, and efficient simulation-based fault injection approaches to enable complex soft error resilience analysis regarding different system configurations at an acceptable time.

2.3 Soft Error Assessment

This section resumes the most widely adopted approaches to assess soft error effects on embedded systems. The development of techniques for soft error injection in processors has been studied to evaluate processor architecture, organization, and applications running on those processors in early product design phase. Fault injection can be performed on board-oriented approaches by interrupting the processors and forcing the processor to execute corrupted data that is modeling the fault. However, this approach is also very time consuming when considering complex applications executing under millions of fault injection experiments. Another alternative is exposing the board to neutron radiation. Although radiation tests produce the most accurate results, they have a very high cost. Especially when looking for a large number of error events under neutron fluence, each test may easily take several days to build a desirable confident statistic.

The first group relies on detailed fault injection estimation at register or gate level employing commercial tools, such as ModelSim from Mentor. For instance, a tool called VFIT integrates a series of VHDL-based elements designed around the ModelSim to support fault injection evaluation (BARAZA et al., 2000). Authors considered 3,000 faults per experiment. A similar work extends Modelsim capabilities in order to inject faults in RTL descriptions using Perl and Tcl scripts (RUANO et al., 2007). Authors in (VALDERAS et al., 2007) use Foreign Language Interface (FLI) available in ModelSim to diminish control overheads during simulation. FLI enables C-Written modules to monitor and modify any signal in simulation time. Results include a compiled CORDIC processor in a 10,000 faults scenario reaching 2.33 faults per second in the best simulation case. Due to the number of modeled aspects (BARAZA et al., 2000; RUANO et al., 2007; VALDERAS et al., 2007), the detailed evaluation produces accurate soft error results although it is time-consuming, and the amount of memory required by these approaches is too high. Consequentially restricting the experiments to few thousands faults (e.g.(BARAZA et al., 2000; VALDERAS et al., 2007)) considering a single target processor or specific ISA. Other drawbacks of such approaches are the poor fault access, and design modifications are usually highly intrusive, which increase the design space exploration cost.

To speed up the fault injection simulation while improving modeling capabilities the second group emphasizes the use of SystemC (RAMACHANDRAN et al., 2008). The work proposed in (EBNENASIR; HAJISHEYKHI; KULKARNI, 2012) explores how faults can be modeled and injected at different SystemC abstraction levels (e.g., RTL, TLM). The authors in (BELTRAME; BOLCHINI; MIELE, 2009) focus on soft error evaluation and a NoC model was used as case study. In (SHAFIK; ROSINGER; AL-HASHIMI, 2008), a behavioral SystemC description of an MPEG-2 is used to validate a fault injection technique developed on the SystemC API basis. Authors in (MISERA et al., 2006), coupled VHDL and SystemC in a hierarchical design fault simulation process based on SystemC, which can be employed at different abstraction levels. A Python-based framework, called ResSP is proposed in (BELTRAME; FOSSATI, 2008; BELTRAME et al., 2007). ResSP provides wrappers to assemble SystemC components and processor models (e.g., PowerPC, Leon2, and ARM7) described in ArchC (RIGO et al., 2004). Experiments comprise 10,800 faults injections in a Leon processor model connected to a memory through a bus. As reported in (BELTRAME; FOSSATI, 2008), ReSP simulation achieves 2-3 MIPS on a 2-core host. Although SystemC reduces the simulation cost, the lack of processor/ISA models and the still inadequate performance of SystemC kernel limit its adoption when exploring large fault experiments and complex systems.

Fault injection analysis based on simulation is widely used and accepted as an efficient way to perform soft errors assessment, enabling different system configurations explorations during early DSEs (CHO et al., 2013; KOOLI; NATALE, 2014). Some state-of-the-art fault injectors use high-level behavioral models, and others are based on hardware description language level (HDL) or gate-level models. Although HDL-level and gate-level models are more precise than a high behavioral model, there are two main problems when using those models: First, commercial processors are rarely available to users in HDL or gate-level descriptions. Second, the simulation time in such abstraction levels is exceptionally high. Thus, thousands of simulations may take several months, which is not suitable to evaluate systems under soft errors during the early design phases. Given the ever-increasing complexity of both processor and software architectures, researchers have been investigating the use of virtual platforms as an alternative means to assess soft error resilience. The next chapter details the proposed use of virtual platforms as fault injection frameworks.

3 PROPOSED FAULT INJECTION FRAMEWORK

This chapter describes the first *contribution* of this Thesis, the development of two fault injection frameworks designed on the basis of two virtual platform simulators: OVP-sim (Section 3.5) and gem5 (Section 3.6). The Section 3.1 presents a brief description of most well-know display state-of-the-arts on virtual platforms (VPs). Next, in Section 3.2, fault injectors implemented on the top of VP are discussed. After, Section 3.3 presents the adopted fault model and the Section 3.4 describes the proposed fault injection flow integrated into both fault injection frameworks developed under the frame of this Thesis. Section 3.7 presents a set of modeling and simulation techniques developed to improve the performance and the design space exploration capabilities of both frameworks. Section 3.8 another important contribution of this Thesis is presented: the proposal of four novel fault injection techniques.

3.1 Virtual Platforms

A virtual platform is a full-system simulator that emulates hardware components (e.g., CPUs, memories), and the execution of real software stacks, on the same machine, as it is running on real physical hardware. Such simulators usually offer a set of processor architectures, peripherals, and memory models, allowing a fast and flexible software development at early design stages. VPs differ concerning modeling, flexibility, and simulation performance. While event-driven VPs such as gem5 target microarchitecture exploration, just-in-time (JIT) simulators (e.g., OVPsim (IMPERAS, 2017)) are devoted to software development. JIT-based simulators can achieve speeds approaching actual execution time, e.g., thousands MIPS at the cost of limited accuracy. In turn, event-driven simulators typically report best-case simulation performances of 2-3 MIPS.

It is hard to cover all modeling aspects (e.g., flexibility, debuggability) and simulation (e.g., accuracy, scalability) requirements in one single simulator. This section starts by providing an extension of the surveys (BUTKO et al., 2012; GUTIERREZ et al., 2014) considering the most popular virtual platforms. Such virtual platform simulators are compared according to different criteria: (i) accuracy, (ii) flexibility regarding supported processor architectures, (iii) licensing, and (iv) support activity. Table 3.1 summarizes the reviewed work according to such four criteria.

The wind river Simics (MAGNUSSON et al., 2002) is a simulator that enables unmodified target software (e.g., operating system, applications) to run on top of a platform

Table 3.1: Most recognizable virtual platforms simulators.

Simulator	*ISA(s)*	*Guest OS*	*Accuracy*	*License*
Simics	Alpha, ARM, MIPS, PowerPC, SPARC, x86	Linux, Solaris, Windows, and others	Functional	Closed
PTLsim	x86	Linux	Cycle	GPL
Flexus	SPARC and x86	Linux and Solaris	Cycle	BSD
SimpleScalar	Alpha, ARM, PowerPC, and x86	Linux	Cycle	none
MARSS	x86	Linux	Cycle and Instruction	GPL
gem5	ARM, MIPS, SPARC, and x86	Linux, Android, Solaris, and others	Cycle	GPL
QEMU	Alpha, ARC, ARM, PowerPC, MicroBraze, and others	Linux, Android, Solaris, and others	Instruction	GPL
OVP	Alpha, ARC, ARM, PowerPC, MicroBraze, and others	Linux, Android, Solaris, FreeRTOS, and others	Instruction	Modified Apache 2.0

Source: The Authors

model. A wide range of processor architectures (e.g., ARM, MIPS, PowerPC), as well as operating systems (e.g., Linux, VxWorks, Solaris, FreeBSD, QNX, RTEMS), can be adapted to model the desired systems. This simulator includes SystemC interoperability, debuggers, software and hardware analysis views, as well. Simics is not open source, and thus users require a commercial license, which is one disadvantage.

The PTLsim (YOURST, 2007) is a cycle-accurate simulator, offering the complete cache hierarchy, different processor architectures, memory subsystem and hardware devices. This tool presents two main drawbacks: it only supports x86-64 architectures, and it has no active maintained. SimpleScalar (AUSTIN; LARSON; ERNST, 2002) is an open source infrastructure for simulation and architectural modeling. Similar to previous simulators, software engineers can use SimpleScalar to develop applications and execute them onto a range of processor architectures, which varies from simple unpipelined processors to detailed microarchitectures with multiple-level memory hierarchies. However, SimpleScalar is not actively maintained anymore (the last update was in March 2011), and other faster solutions, like gem5, are available.

QEMU is an instruction-accurate and open source simulator that relies on dynamic binary translation supporting several CPUs (e.g., x86, PowerPC, ARM, and SPARC). Similar to QEMU, the Open Virtual Platform Simulator (OVPsim) (IMPERAS, 2017) engine also employs a just-in-time dynamic binary translation, i.e., target instructions (e.g., ARM, MIPS) are morphed into x86-64 host instructions through a translation mechanism. The OVPsim API provides several component models, including processors, memories, uarts, among others. The processor architectures and variants sum more than 170, including ARMv7, ARMv8, MIPS, Renesas, and MicroBlaze.

gem5 (BINKERT et al., 2011) is a modular discrete event simulator, which has an open-source code and supports a rich set of models including processor cores, memories, caches, and interconnections. Among the available instruction set architectures (ISAs) are x86, MIPS, Alpha, SPARC, and primarily ARM, which is the subject of this work. The source code and license usage are open, thus enabling any component addition or modification. Further, the gem5 simulator is described in C++ and Python, and it has an active development and support team. Its target microarchitectural explorations, which incurs in substantial simulation overheads due to the number of modeled aspects (e.g., memories, caches, and interconnections). Further, the amount of memory required by the gem5 simulator is very high, making its use infeasible to explore large-scale system models. MARSS (PATEL et al., 2011) is cycle-accurate full system simulation of the x86-64 architecture, which uses a hybrid approach through the PTLsim (YOURST, 2007), as the basis of its CPU simulation environment on top of the QEMU (BELLARD, 2005).

QEMU was initially designed for single-processor platforms and virtualization purposes, and more recently, researchers are extending its capability to multicore explorations as well. However, the lack of documentation on the APIs or standardized methodology for creating manycore platform models limits its use. Excluding PTLSim and MARSS that only supports x86, reviewed simulators provide support to several processor architectures. Cycle-accurate simulators such as SimpleScalar and gem5 entail high-simulation time, thereby limiting their applicability to more complex or large-scale systems explorations. Simics has a private license while the others are free to use. Further, SimpleScalar does not provide support or development anymore.

After careful selection, this work adopted OVPsim (IMPERAS, 2017) and gem5 (BINKERT et al., 2011) as as means to develop the fault injection frameworks. The OVPsim (IMPERAS, 2017) is highly deployed in the industry and the academic communities under the frame of several research projects. The OVPsim can achieve speeds approaching thousands MIPS and its supports several component models), including 170 processor variants (e.g., ARMv7, ARMv8, MIPS, Renesas, MicroBlaze), memories, uarts, among other components. Beyond the high simulation performance, two main other reasons justify the adoption of OVPsim as means to develop the fault injection framework. First, among available VPs, OVPsim has the most substantial number of processor models and thus enabling a broader initial space exploration. Second, Imperas Software Ltd.[1], is directly involved in the work conducted in this Thesis. One expected outcome of this collaboration is to promote the first commercial fault injection framework in the market. This work also adopts the state-of-the-art gem5 simulator (BINKERT et al., 2011) for three main reasons: (1) the gem5 source code is open and several extensions have been proposed in the past (ALIAN; KIM; KIM, 2016; SHAO et al., 2016), (2) the gem5 enables *microarchitectural cycle-accurate simulation* in an acceptable time (i.e., 0.4–2 MIPS depending on the application workload), and (3) it supports the current ARM Cortex-A architectures (i.e., ARMv7, ARMv8, and big.LITTLE).

3.2 Related Work on Fault Injection Approaches using Virtual Platforms

Virtual platform simulators facilitate fault injection implementation and analyses due to their design flexibility (e.g., several processor models available), and debugging capabilities (e.g., GDB support) are shown in Table 3.2. Authors in (HARI et al., 2012) present the Relyzer, a hybrid simulation framework for SPARC core using Simics (MAGNUSSON et al., 2002) and GEMS (MARTIN et al., 2005) simulators coupled with a pruning technique to reduce injected faults. Also, this framework target architectural integer registers and in the output latches of the address generation units. In this work, a 200-cores cluster was employed and approximately 11 days were required to inject around 32 million faults, resulting in an average of 33 injected faults per second. The low number of injected faults is due the high-cost simulation time of simics+GEMS simulator, which can achieve few hundred KIPS. This framework employs 12 benchmarks, four from each suite Parsec 2 (BIENIA et al., 2008), Splash-2 (WOO et al., 1995), and SPEC-Int (HENNING, 2006). In (HARI et al., 2014), Relyzer is extended to more aggressive pruning, reducing the number of faults that must be simulated. With the pruning and analysis techniques embedded in the GangES the authors further reduce this simulation time from

[1]http://www.imperas.com/

15,600 CPU-hours to 8,200 CPU-hours.

In (GEISSLER; KASTENSMIDT; SOUZA, 2014), a fault injection framework based on QEMU (BELLARD, 2005) is proposed. Faults are injected in an x86 architecture running applications in a Real-Time Operating System (RTEMS). The experimental setup accounts for four applications developed in-house During the experiment, 8,000 faults were injected in 8.7 hours, given an average of less than one fault per second. The authors in (KALIORAKIS et al., 2015) propose two tools: the GeFIN tool, a gem5-based fault injection framework and MaFIN, a MARSS-based (PATEL et al., 2011) fault injection framework. In this work, faults were injected, randomly in time, in general-purpose registers, caches control registers, and other microarchitectural components. The experimental setup includes only the execution of 10 bare metal benchmarks selected from the MiBench (GUTHAUS et al., 2001). The use of an operational system is unknown.

Recently, in (TANIKELLA et al., 2016) the authors introduces the gemV, a gem5-based fault injection framework for micro-architectural elements such as instruction queue, reorder buffer, load-store queue, pipeline queue, renaming unit, and register file. The experimental setup includes ten application from the MiBench and SPEC-Int 2006 benchmark suites for eleven microarchitectural elements. Each element is subject to a 300-long fault campaign for each application, totaling 33,000.

The work (GUAN et al., 2016) presents the P-FSEFI tool construct around the QEMU (BELLARD, 2005) simulator. This tool injects faults in the CPU logic units, registers, caches, and memory. The experimental setup consists of seven applications from the NAS Parallel Benchmark (NPB) (BAILEY et al., 1991), each one in a sequential and a parallel version; injecting 10 thousand faults in each setup. Thus totaling 140 thousand faults. In (DIDEHBAN; SHRIVASTAVA, 2016) introduces a gem5-based fault injection capable of flipping random register file bits, pipeline registers, functional units, and load-store queue. This work employs ten MiBench applications in a total of 72 thousand faults.

Most reviewed approaches consider only small scenarios and only single-core processors. Exploration of soft error reliability of single-core architectures has been successfully supported over the last decades. However, the assessment of multicore architecture soft error resilience strongly requires complementary modeling and simulation mechanisms to manage other aspects such as resource sharing, memory allocation, and data dependencies. Further, such works typically report best-case simulation performances of 2-3 MIPS, allowing 33 fault injections per second considering a supercomputer (HARI et al., 2012).

Table 3.2: Most recognizable virtual platform fault injection simulators.

Year	*Author*	*Simulator*	*ISA(s)*	*Applications*	*Guest OS*	*Accuracy*	*Faults*	*Features*
2014	Hari et al. Relyzer	Simics GEMS	SPARC V9	12: Parsec 2, Splash-2, and SPEC-Int 2006	OpenSolaris	Cycle and Instruction	32M (15K hours)	Architectural integer registers and in the output latches of the address generation units
2014	Geissler et al.	QEMU	x86	Four in-house applications	RTEMS	Instruction	32K	8 general-purpose registers, 6 segment registers, and instruction pointer
2015	Kaliorakis et al. MaFIN and GeFIN	MARSS gem5	x86 ARMv7	10 MiBench	None or Unknown	Cycle	300K	Architectural integer registers, L1 and L2 cache, and Load/Store Queue
2016	Tanikella et al. gemV	gem5	x86 ARMv7 SPARC ALPHA	10: MiBench and SPEC-Int 2006	None or Unknown	Cycle	33K	Eleven Microarchitectural components
2016	Didehban et al.	gem5	ARMv8	10 MiBench	gem5 Syscall Mode	Instruction	72K	The register file, pipeline registers, functional units, and load-store queue
2016	Guan et al. P-FSEFI	QEMU	ARMv7 x86	14 serial NPB	None or Unknown	Instruction	140K	CPU logic units, registers, caches, and memory
2018	**This work**	**gem5 OVPsim**	**ARMv7 ARMv8**	**26 OpenMP; 10 Serial, 9 MPI-Based and others**	**Linux OSs, FreeR-TOS, and Baremetal**	**Cycle and Instruction**	**3.3 M (1.2M hours)**	**Register file and physical address space in both simulator. Virtual Memory, Variable data, function liveness, function instruction code in the OVPsim-based fault injection**

Source: The Authors

3.3 Fault Modeling

This Thesis reproduces the soft errors behavior using single-bit-upsets (SBUs) due to its higher probability of occurrence in electronic systems operating(JOHANSSON et al., 1999) at sea level. Our SBU model consists of a single bit-flip generated randomly in one general-purpose register (e.g., r0-13, sp, pc) or memory address during the application execution. Additionally, this model does not target the operational system (OS) explicitly (i.e., the boot process). Nevertheless, the OS calls arising in the application lifetime could be affected. This approach analyses the application behavior also considering the execution environment, thus exposing unforeseen consequences when compared with standalone implementation.

3.4 Fault Injection Flow

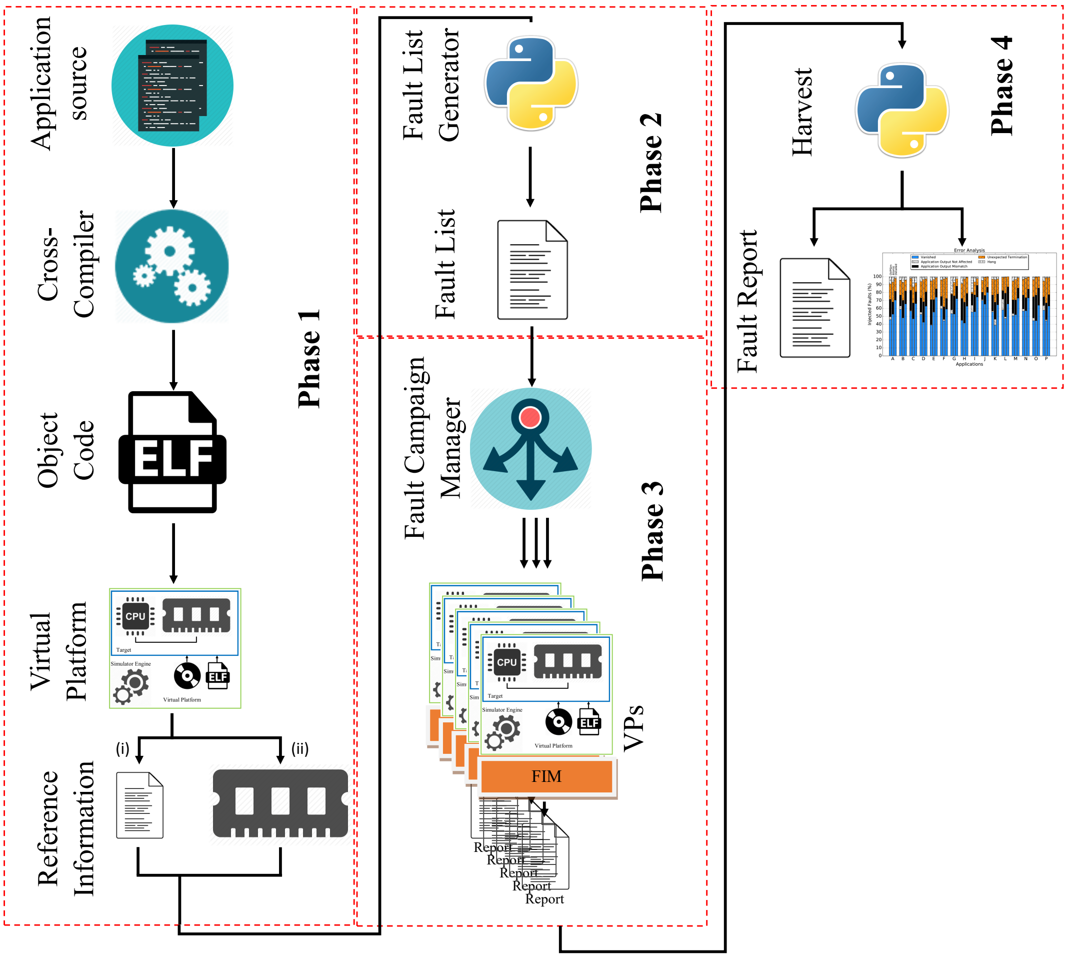
A group of fault injections targeting a particular application scenario including all related steps is here defined as a *fault campaign*. These steps are divided into four phases: the reference (or faultless) phase (1), fault generation (2), fault injection management(3), and final report generation phase (4). The fault campaign flow described by Figure 3.1 is independent of the simulator. The simulation flow is implemented by the *simulation infrastructure (SI)*, which was mostly developed using shell and python scripts, thus being compatible with Linux, Windows, and MacOS OS distributions.

In phase one, the SI cross-compiles the application source targeting a given architecture resulting in an elf object file. After that, the SI simulates the application in an unmodified virtual platform (i.e., OVPsim or gem5) to verify its correctness and also to extract information using a gold standard and for fault creation. The register's contexts (i, Figure 3.1) and final memory state (ii) composes this reference information. However, it can be easily adjusted to contain other types of fault injection (e.g., bus, network-on-chip, or cache information).

Phase 2 creates register or memory fault patterns consisting of injection time, a target, and a fault mask (i.e., the target bit). A random generation scheme selects the insertion time, the location, and the register bit since it covers the majority of faults at a low computational cost. The injection time ranges from one to the final instruction count extracted during phase 1 (Figure 3.1). A bit-flip injection requires a bitmask with all bits set to '0', excepting the target bit. For instance, to change the second least significant bit of a given 32-bit register requires the 0x00000002 mask. This fault generator can be extended effortlessly to include new types of fault or more sophisticated selection methods. Afterward, the complete fault list is available to the next phase from a simple plain text file.

The fault injections (phase 3), the most complex one, has two main components: single fault injection and simulation speed boost. This flow description considers a single fault injection as performance boost techniques will be covered in Section 3.7.4. The simulation infrastructure maps one fault injection to one VP-FIM execution, in other words, ten fault injections require ten independent VP-FIMs. Each VP-FIM starts by reading the fault list and then schedule an event targeting insertion time (i.e., the number of executed instructions). At the fault injection moment, the FIM combines a fault bit mask with the current register value using an exclusive OR (XOR) operation. Supposing a 32-bit value of 0x00000009 and a fault targeting the fourth least significant bit, the mod-

Figure 3.1: Fault injection campaign simulation flow.



Source: The Authors

ule performs an XOR operation between 0x00000009 and 0x00000008, which generates 0x00000001. Subsequently, the FIM writes the new value in the target register and the simulation restarts.

Each VP-FIM instance performs an error analysis where the application behavior under fault injection is compared with the reference run (phase 1). The proposed flow supports custom error analysis and this work adopts the Cho *et al.* (CHO et al., 2013) five groups error classification:

- **Vanished**, no fault traces are left;
- **Application Output Not Affected** (ONA), the resulting memory is not modified. However, one or more remaining bits of the architectural state is incorrect;
- **Application output mismatch** (OMM), the application terminates without any error indication. However, the resulting memory is affected;
- **Unexpected termination** (UT), the application terminates abnormally with an error indication;
- **Hang**, the application does not finish requiring a preemptive remove.

Lastly, phase four assembles all FIMs individual reports to create a single database and graphics.

3.5 OVPsim-FIM

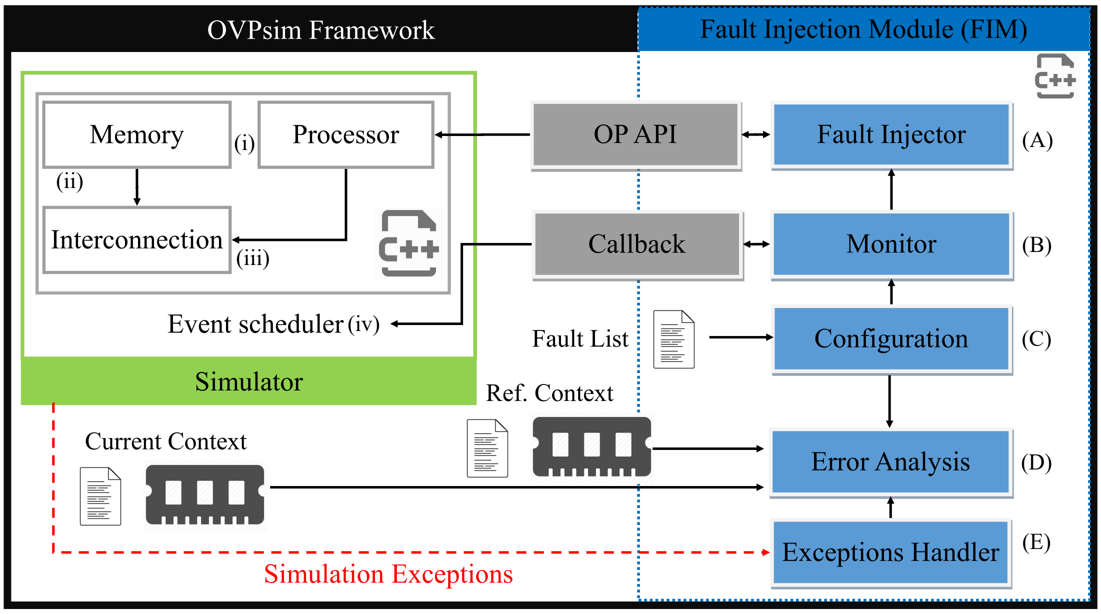
The Open Virtual Platform simulator (OVPsim) (IMPERAS, 2017) is an instruction accurate simulator framework market by Imperas under an open license. The OVPsim relies on a dynamic binary translation (DBT) engine, which enables simulation running real application code at the speed of hundreds of MIPS. The DBT mechanism sequentially fetches each instruction from a target processor (or core), morphing it into new x86_64 micro-operations. Further, it uses a micro-operations dictionary to hold the previous translations aiming to speed-up the simulation process. The OVPsim multicore simulation algorithm creates fixed length blocks of instructions belonging to each core. Each of those blocks is sequentially simulated. Note that in such simulation scenarios, each processor/core advances in fixed-length instruction steps. The OVPsim main limitation is the lack of detailed microarchitectural modeling (e.g., pipeline, decoder, reorder buffer) or cache interconnections. Reads and writes are always guaranteed to be atomic in an instruction-accurate simulator (i.e., the previous instruction always completes before the next starts), thus removing any data hazards originated from the pipeline access to information before the write-back stage update.

The fault injection module or just FIM is a series of components embedded in OVPsim, and it is responsible for:

- Monitoring the target processor;
- Accessing resources as memories and registers;
- Injecting the faults;
- Capturing unexpected events arising from the simulator;
- Extracting information;
- Analyzing errors.

The developed FIM minimizes the intrusion in the simulator engine, thus enabling any researcher in possession of the original simulator to use, modify, or extend its functionalities. In a matter of fact, the OVPsim provides a complete set of APIs written in C/C++, which easily enables the simulator extension without any source code access. Figure 3.2 shows the OVPsim-FIM main components: (A) fault injector, (B) fault monitor, (C) configuration, (D) error analysis, and (E) exception handler.

Figure 3.2: OVPsim-FIM main components.



Source: The Authors

The configuration component (C) reads the fault list, setups the monitor component (B), triggers the error analysis (D), and forward the unexpected events to the exception handler (E), invoking each one at the appropriated time. We restrict this section to the essential features regarding the fault injection, and other available functions (e.g., speed boost techniques) are further discussed in the following sections. The monitor component (B) controls the internal simulator flow (i.e., starts and restarts the simulator), as the fault injection need to stop the simulator. This module schedules an event associated with the number of executed instructions to call the fault injector (A) whenever the injection time arrives. The fault injector component (A) access to the registers or memory through the OVPsim APIs enabling the modification of any available microarchitectural element. After the FI conclusion, it setups the hang (i.e., infinite loop) detector through a similar event targeting the stipulated threshold and resumes the simulation.

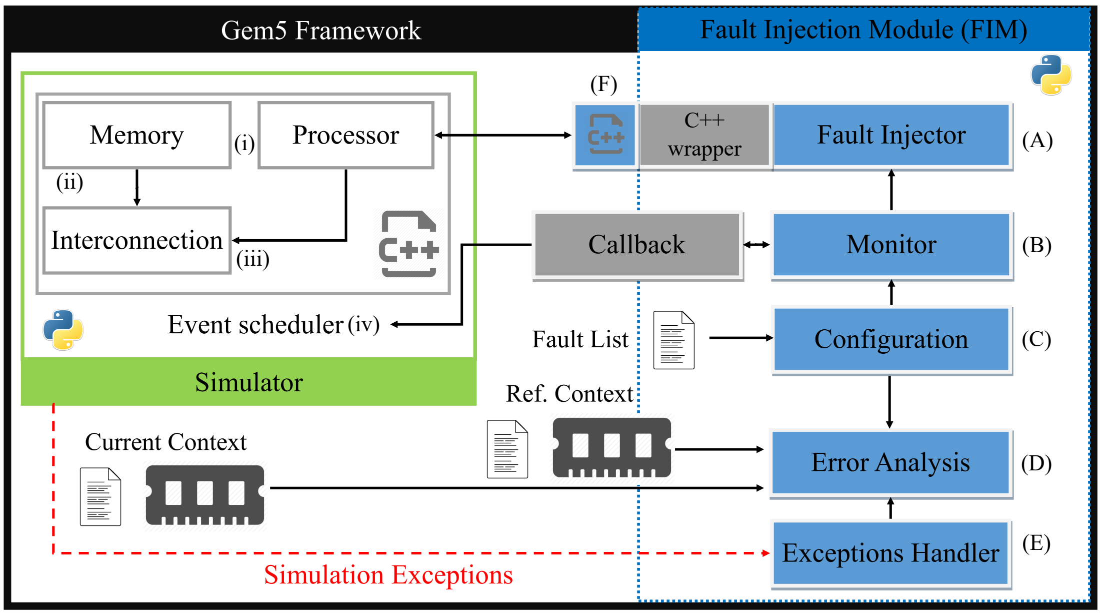
Injecting faults can lead to abnormal application behavior such as OS malfunctions, segmentation faults, or hard faults, and the exception handler (E) is responsible for captures these events. Finally, the error detection component (D) verifies the simulator context for any mismatches considering the reference (faultless) execution. For this purpose, it extracts the current memory state, the register's context including the program counter, and the number of executed instructions. Thus, the error detection classifies the application under fault influence using the information acquired in the faultless execution. Also, it consolidates exceptions captured by the exception handler (E) to create a final report.

3.6 gem5-FIM

The gem5 simulator offers few accuracy levels depending on the exploration suitability, ranging from the simple atomic mode to a detailed one which includes an Out-of-Order (OoO) pipeline. The gem5 detailed mode provides a sophisticated memory timing and cache coherency protocols while the atomic mode emulates the memory and cache using a single cycle access mechanism. Therefore, the gem5 modes can slightly differ regarding the execution time and cache activity as additional cache misses or incorrect branch speculations can cause a pipeline flush. The gem5 simulator possesses an intermediary model with the atomic internal microarchitectural elements and enhanced memory access timing.

The unmodified gem5 simulator precision accuracy varies from 1.39% to 17.94% when considering the execution time against a prototyping board (BUTKO et al., 2012). The microarchitectural elements do not always correlate to a particular hardware implementation due to speed overheads or to provide a more generic model. Thus, to address this issue, in (GUTIERREZ et al., 2014) the gem5 was strictly modified to match the ARM Versatile Express board through the addition or modification of microarchitectural elements such as cache memory, branch predictor, and fetch buffer. This setup can achieve a mean percentage error smaller than 5% across several benchmarks.

Figure 3.3: gem5-FIM main components.



Source: The Authors

The gem5 offers the complete source code and no proper extension API, and to maintain a non-intrusiveness extension, the gem5 fault injection module (gem5-FIM) independently extends the source code requiring only the access to some classes attributes/methods. The gem5 simulator employs Python scripts to control the simulation flow and C++ modules to model the microarchitectural components. Figure 3.3 displays the main gem5-FIM components, where white boxes represent the original gem5 components (i.e., (i) processor, (ii) memory, (iii) interconnection), and the blue the developed ones. The gem5-FIM incorporates five main components developed using Python: (A, Figure 3.3) fault injector, (B) fault monitor, (C) configuration, (D) error analysis, and (E) exception handle and one C++ extension to access the microarchitectural components (F).

The gem5 has both a Python and C++ objects representations of components where the python configures and triggers the C++ objects through a series of events. Consequently, the gem5-FIM requires a C++ module alongside a Python counterpart. Conveniently the gem5 framework adopts the swig (i.e., Simplified Wrapper and Interface Generator) tool, which interfaces booth programming languages through wrappers. The Python side (A Figure 3.3) gathers the processor and memory handlers from the gem5 configuration and invokes the C++ wrapper (F) to perform the bit flip in the system description (e.g., memory, register file, pipeline)

3.7 Extensions

State-of-the-art software stacks, including OSs, compilers, and application workloads leading to several extensions were conducted in both FIM simulators.

3.7.1 Targeting Complex SW Stacks

The proposed fault injection module and surrounding simulation infrastructure until this point presented a general methodology to inject faults in the register file of a given processor. Beyond controlling the fault campaign flow as described in Section 3.4, the simulation infrastructure requires other modifications to include an operating system. First, any OS needs a bootloader[2], kernel image, file system image and a different cross-compiler. The simulation infrastructure automatically appends the application binary file in the file system image, selects the bootloader and kernel files. Indeed, the development time spends on the development and improvement of the simulation infrastructure is comparatively more substantial than the FIM by itself.

The VP-FIM should be able to capture the exact moment where the application finish (i.e., *main* function return statement). Also, the FIM captures application starting point (i.e., faults are only injected during the application execution) and abnormal events (i.e., attempt to execute an illegal instruction or out of range memory address). In bare-metal systems this can be easily accomplished, for example, surveying physical memory access. However, OS abstracts and protects the hardware components from the guest applications requiring a new solution.

These issues lead to the development of an OS/VP-FIM communication API called *FIM-API*, enabling the application inside the guest OS to call different VP-FIM services. Among the available services are: the application begin and end point, a segmentation fault events, and others. The gem5-FIM version makes use and extends the gem5 built-in artificial instructions to deploy these necessary services. Note that the artificial instructions are implemented using a library linked to the application, and thus, requiring

[2]Some OSs require more than one bootloader

no modification in the compiler. The OVPsim provides an ISA-independent and more generic solution as it is capable of creating virtual memory callbacks on-the-fly. The OVPsim-FIM intercepts special function symbols by name during the system execution. The simulation infrastructure links an FIM-API header file together with the application while the Linux file system image includes an FIM-API standalone executable.

3.7.2 Injecting Faults on Multicore systems

The adopted VP-FIM handles an arbitrary number of cores/processors limited only by the simulator supported architectures. To inject and collect the results from multiple cores, few modifications are necessary: During the reference phase (1), the FIM collects individual core information such as instruction count and register state. The fault creation takes into account the target architecture core count, as it distributes the faults evenly through available cores. For instance, when creating 8,000 fault campaign targeting a quad-core ARM processor, the fault creator will assign 2,000 faults for each core. Also, it holds specific information concerning each core instruction count necessary for the hang threshold. After the simulation end, it retrieves the information to compare with the fault-less execution for each core. The complete simulation infrastructure automatically adjusts the FIM, fault generator, and other components according to the user core selection.

3.7.3 ARMv8 Architecture extension

The development of more complex applications in the mobile domain and the user demand for high-performance devices lead to the adoption of 64-bit architectures (WATT, 2011; KANTER, 2012). This new architecture enables a larger virtual memory addressing alongside other microarchitectural enhancements. The ARM company developed a series of processors (e.g., Cortex-A53, Cortex-A72) fitted with a new AARCH64 instruction set. This new ISA had 33 general purpose registers and fixed 32-bit instruction width, restricted conditional execution instructions to branches, alongside new float-pointing, and encryption support. The AARCH64 reduces and simplifies the previous ARMv7-A ISA, removing old legacy instruction and executions modes particularly suitable for embedded controllers.

The VP-FIM extension to support the new ARM architecture requires modifications in the FIM and simulation infrastructure. This ISA has 17 new integer register, totaling 33 64-bit wide registers. The reference phase and fault injection capture this new registers for error analysis. Also, the fault generator has 64-bits extensions to target the entire architecture. The simulation infrastructure requires additional Linux kernels, file system images, makefiles, and support libraries.

3.7.4 Fault Campaign Speed Enhancement

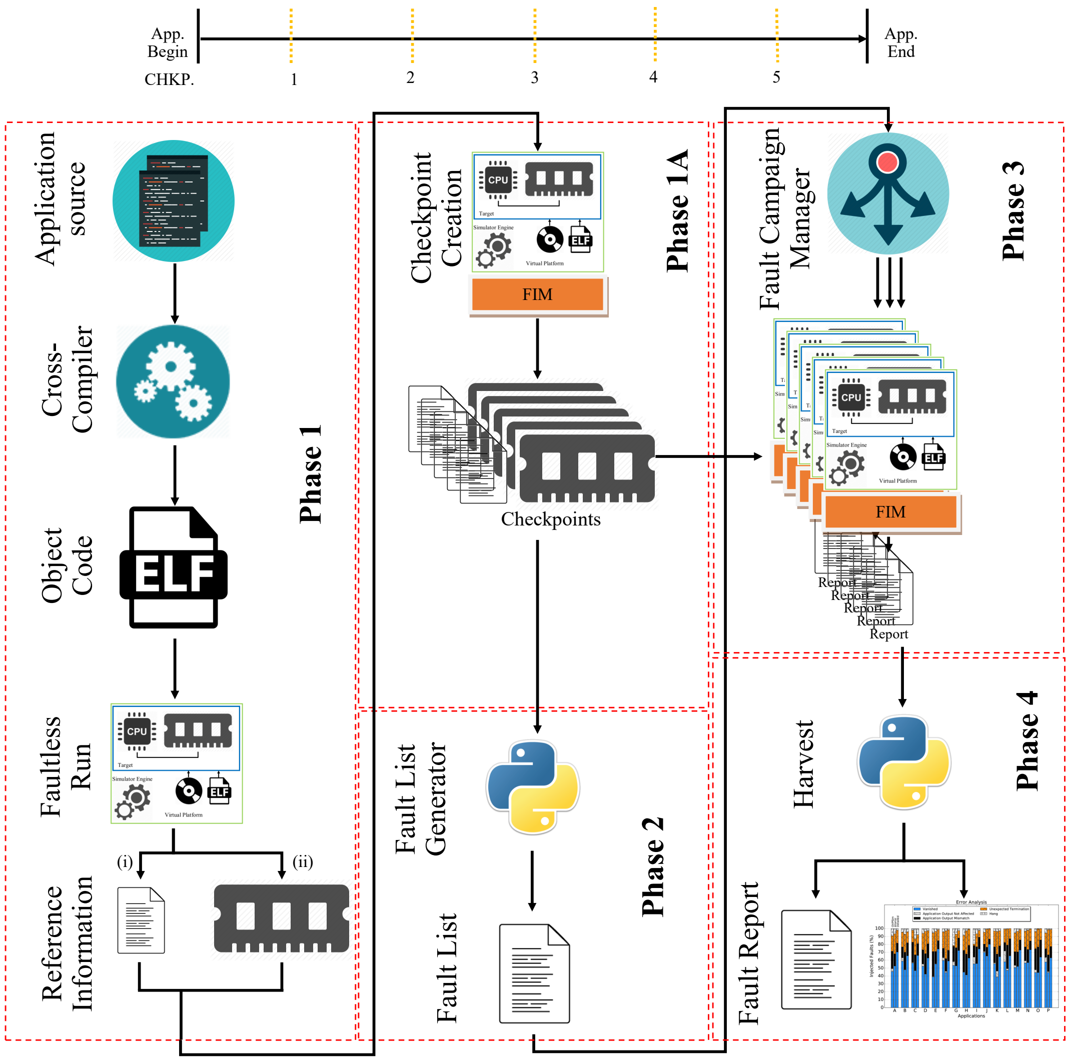
A fault injection campaign comprises thousands of simulations, demanding a significant computational effort, limiting most investigations to small or simpler scenarios. This Thesis investigates the virtual platforms as a technique to speed up early design space explorations regarding reliability assessment of commercial processors. Although the simulation of some VPs reaches hundreds of MIPS simulating complex software stacks is real challenge. Two characteristics are noticeable by observing the fault injection campaign: (i) One fault injection is independent of the other. (ii) The application executes for a significant time without the influence of faults (i.e., equal to gold execution) to provide an appropriated context for the fault injection. This code executed before the fault injection is unnecessary and do not influence the final fault result. The proposed fault campaign deploys techniques to reduce the simulation computation cost at different levels of granularity, which are discussed in the next subsections.

3.7.4.1 Checkpoint and Restore Technique

The checkpoint technique consists of periodically saving the system context during a faultless execution and later restoring the appropriate context for each fault injection. During faultless execution, the VP-FIM stores periodically (i.e., according to a predefined instruction interval) the application context covering processor and memory models. At the fault campaign, each FIM identifies the closest checkpoint before the fault injection time to be restored. Additionally, the fault injection event trigger adjusts the injection time considering the fast for the number of forwarded instructions. The user can specify this interval or assign some checkpoints, and thus, the simulation infrastructure automatically estimates the interval between checkpoints.

The OVPsim-FIM creates the checkpoints concomitantly with the faultless execution. For this purpose, the OVPsim-FIM includes a checkpoint component responsible

Figure 3.4: gem5 atomic checkpoint modified simulation flow.

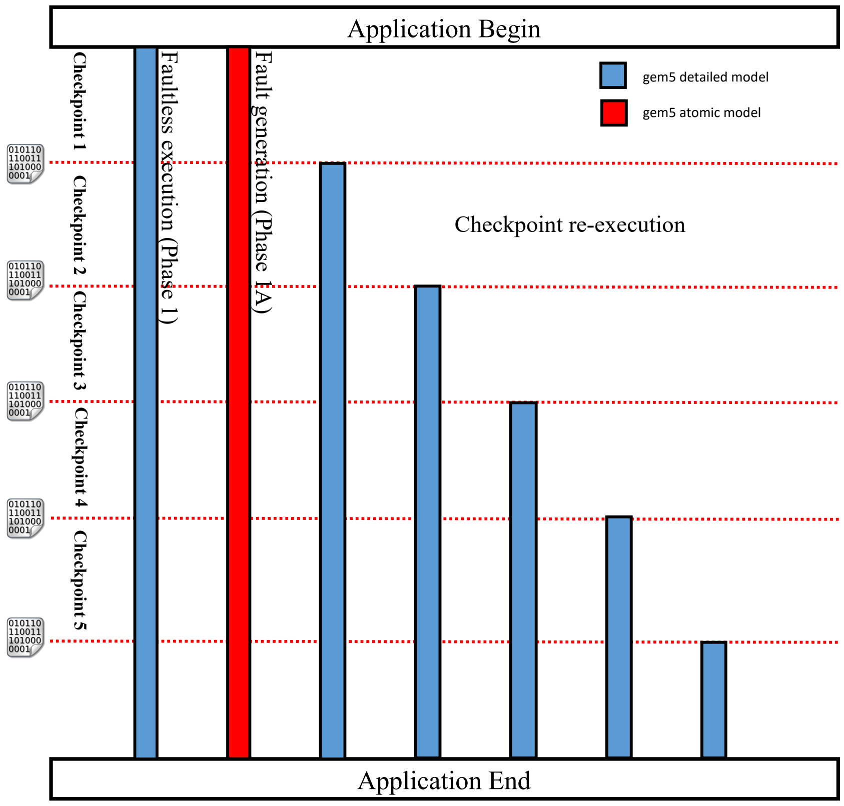


Source: The Authors

for stopping and saving the context according to the specified interval by the user. During the fault campaign, the FIM selects the best matching checkpoint to restore (i.e., the first checkpoint before the fault injection time). From this point, the injection process resembles the version without checkpoints. While the OVPsim provides memory and processor checkpoints, it does no offers the same functionalities for other modules (e.g., uarts, timers). Consequently, the deployed checkpoint mechanism current works only in bare-metal platforms.

The gem5 simulator supplies save and restore functionalities, which allows restoring processor and memory context from a binary file. A restored processor model should execute identically to an unmodified execution, nevertheless, the gem5 does not always behave as expected. The gem5 checkpoint function has a significant limitation, as it auto-

Figure 3.5: gem5 detailed mode checkpoint scheme.



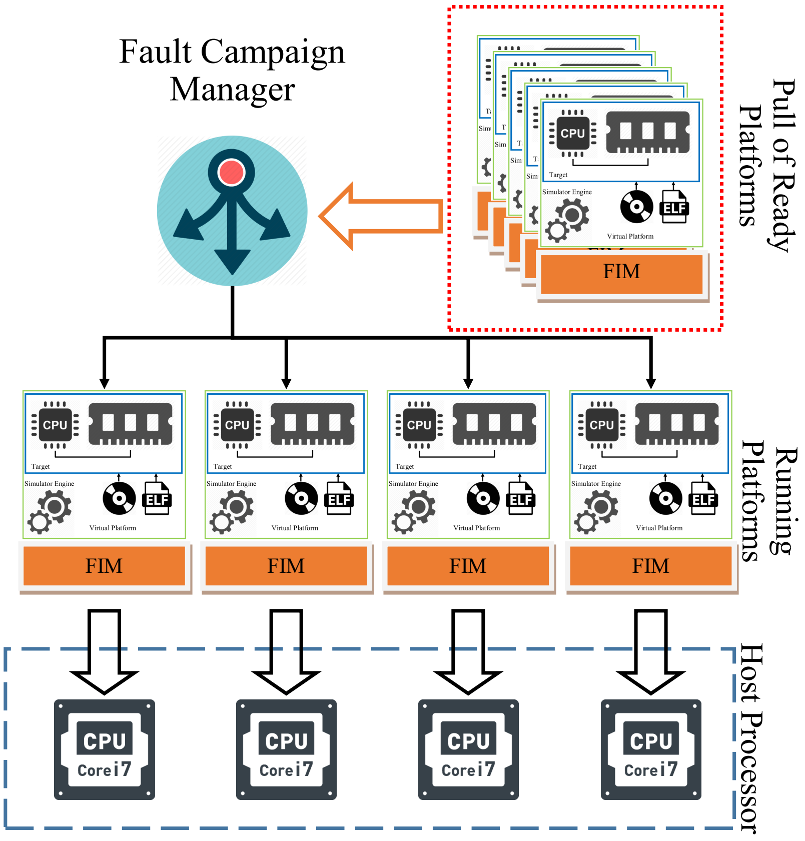
Source: The Authors

matically restarts the simulation engine to process pending events. This action introduces few hundreds of ticks[3] and in some occasions, increases the simulation in a few instructions. Resuming, a simulation without any checkpoint finishes with a slightly smaller number of instructions (and ticks) when compared with a simulation target of checkpoints. This behavior does not impact the application behavior, however, to avoid any mistaken comparison it is necessary to extract the exact information. The simulation flow was modified to incorporate a checkpoint profiling run (see Phase 1A Figure 3.4) to overcome this problem. This phase extracts the checkpoints and at the end updates the reference information (i.e., faultless). At this point, the simulation flow has two reference data sets: One without checkpoints and another after the checkpoint process. Consequently, the FIM can compare the correct set during the error analysis.

The gem5 detailed mode has another peculiarity (see Figure 3.5), the simulator until the present moment does not create executable checkpoints (i.e., which cannot be restored). The new checkpoint scheme first creates the checkpoints using the gem5 atomic mode, as previously described (Phase 1A). Later each checkpoint will be simulated using the detailed mode until the application end to acquire the reference information. Assum-

[3]one tick is the minimum granularity inside the gem5, and usually, each clock cycle has 500 ticks.

Figure 3.6: Host multicore fault injection campaign.



Source: The Authors

ing one application with five checkpoints, for instance, first the faultless execution (Phase 1), then a checkpoint generation using the atomic (one simulation). Finally, the gem5-FIM detailed simulates the five checkpoints At this point, we have six possible references: One without a checkpoint and five for each checkpoint re-execution. Additionally, this phase occurs in parallel to reduce the time overhead.

3.7.4.2 Shared Memory Multicore Parallelization

The fault injection campaign is naturally a parallel process as the fault injections are independent of each other. To exploit this characteristic, the simulation infrastructure deploys multiple VP-FIMs across a shared memory multicore processor, nowadays found in any workstation. The simulation infrastructure automatically limits the maximum number of platforms running in parallel to match the number of host cores as shown in Figure 3.6. Each platform match one Linux process with an overall core utilization of 100%, thus in a quad-core, the simulation infrastructure allows four platforms in parallel. The simulation infrastructure surveys the platform process identification (PID) number and dispatches the next platform shortly after the completion of any running VP-FIM.

3.7.4.3 Distribute Fault Injection Campaigns using HPCs

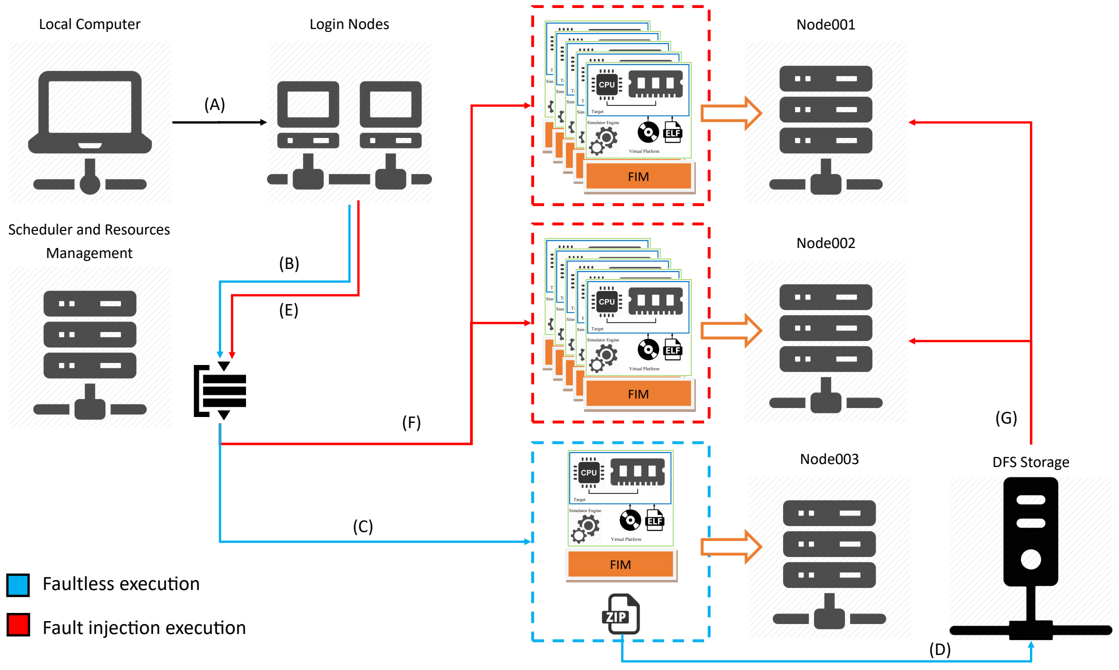
High-performance computers had been used for many decades by researchers of many domains such as quantum mechanics, weather forecast, environmental research, and oil and gas explorations to speed-up simulations. HPC system can be divided into two distinct classes: Extremely complex applications which require a considerable amount of resources (e.g., RAM, storage, cores) not available in standard workstations. The other type consists of more straightforward and smaller applications easily executable in ordinary computers, however, which requires a significant number of single executions to obtain meaningful results.

OVPsim is suitable for few thousand faults when considering the contemporary workloads with hundreds of billions of instructions. For example, a single execution of the largest application (i.e., NPB EP) requires 12 hours or 4000 days in an 8000-fault scenario using the gem5 atomic. This Thesis proposes a simulation infrastructure extension to distribute fault injections across an HPC system, in this work the University of Leicester ALICE supercomputer. The ALICE has 170 standard nodes; each one has two 14-core Intel Xeon Skylake, 128 GB of RAM, and a local storage disk. Further, this HPC extension uses the Portable Batch System (PBS) framework, which is used in most larger scale systems to provide a generic solution.

An HPC environment deploys a job scheduler to maximize the overall hardware utilization where a job (i.e., shell scripts that executes a specific work) describes the necessary resources (i.e., memory, the number of cores, the number of nodes, and wall time precisely) and commands. The walltime (i.e., the maximum execution time) is the job most significant attribute, and usually, the job scheduler deploys a First-come, First-served service policy with some modifications. HPC systems aim for higher hardware utilization and not always the fairest resource division, consequently smaller jobs jump ahead to fill gaps in the scheduling. Longer jobs will wait in the queue longer, and to reduce the starvation problem the scheduler also deploys a priority inversion policy.

The simulation flow follows the previous sections and Figure 3.7 illustrates the same flow across multiple computer nodes. The first phase is performed only once (A, in Figure 3.7), where the application and disk image are compiled in a local computer and then transferred to the supercomputer due to some environmental limitations. The reference phase (B, Figure 3.7) collects the reference information and generates the fault

Figure 3.7: Distributed fault injection campaign flow.



Source: The Authors

list. Additionally, this job estimates the total simulation time for the scenario (i.e., the number of faults times the execution time of one fault injection) and selects the number of jobs to be submitted.

The simulation infrastructure does not match one fault per job as it would create a substantial management overhead, and instead, it agglomerates the fault blocks in approximately 24-hour jobs[4]. Considering an 8,000-fault campaign and an application simulation time of one hour, the optimum arrangement in this scenario requires 24 applications per job (i.e., a walltime of 24 hours) requiring 333.33 jobs, which is unfeasible. The simulation infrastructure selects the best fit between the number of faults per job and walltime, where for the previous example is 400 jobs with 20 hours walltime. Therefore, the simulation infrastructure creates job templates (i.e., bash scripts) for each fault injection scenario.

Developing an application to HPC systems present several challenges, for example, the number of jobs running in parallel is limited by the distributed file system performance. The simulation flow has three options to manage scenarios source files (e.g., simulator executable, checkpoint files, application binaries, fault list):

(i) The easiest (i.e., no significant modification is needed) solution executes remotely

[4]Jobs with walltime up to 24h are classified as short jobs by the ALICE scheduler and thus increasing the execution possibility.

over the distributed file system (DFS) where the HPC remotely access files in the storage using the network. This option only fits small scenarios (up to 25 jobs) due to the bottleneck created in the DFS by the multiple access to the same memory region.

(ii) The next option transfers all simulation-related files across the network to the node local storage during each job startup. Nevertheless, the initial copies may overlap with other jobs reducing the overall transference speed and congesting the DFS buffers due to the number of files required (i.e., thousands of files). Using the local storage improves the flow scalability to approximately 500 parallel jobs, not satisfying our requirements.

(iii) The last solution compresses the simulation source files in a single zip file during the phase 1, copying this file during each job startup to the node storage where they are locally decompressed. This approach enables the management of thousands of concurrent jobs by simulation infrastructure, and it is the adopted in this work.

The jobs responsible for the fault injection are submitted to the scheduler and queued (E, Figure 3.7) to later be assigned to nodes as the resources are made available (F). During the job initialization, it copies the compressed file (G) (simulation source files), locally extracts the contents, and the node executes the designated platforms. The individual reports are later merged into a final fault injection report.

3.8 Focused Fault Injection Extension

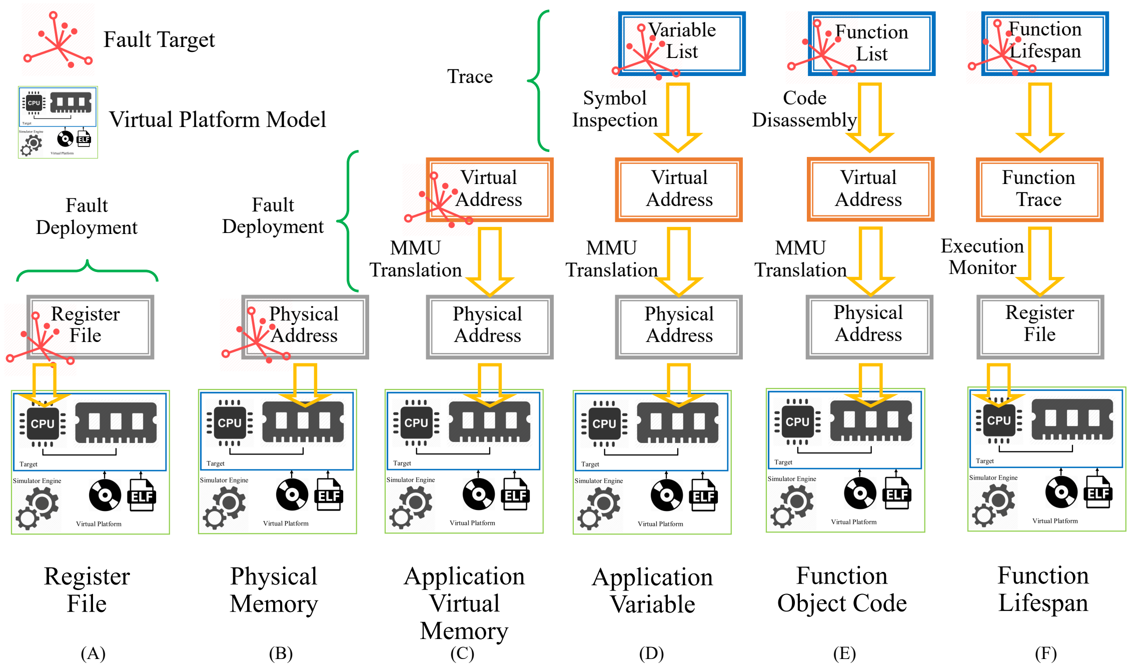
Considering the works reviewed by Section 3.2, authors in (HARI et al., 2012) present the Relyzer, a hybrid simulation framework for SPARC core using Simics (MAGNUSSON et al., 2002) and GEMS (MARTIN et al., 2005) simulators coupled with a pruning technique to reduce the number of injected faults. The Relyzer is capable of injecting faults into architectural integer registers, and output latches of the address generation unit. In (GEISSLER; KASTENSMIDT; SOUZA, 2014), a QEMU-based fault injection framework is proposed targeting general-purpose registers. Fault injection campaigns in this work consider an X86 architecture running four in-house applications on the top of RTEMS kernel. Another fault injection framework, called F-SEFI that relies on QEMU is described in (GUAN et al., 2016; GUAN et al., 2014). The F-SEFI employs the QEMU using a hypervisor mode, i.e., it does not emulate the complete target system, which reduces both its fault injection and soft error analysis capabilities.

The authors in (KALIORAKIS et al., 2015) propose the GeFIN and the MaFIN tools, which support the injection of faults in microarchitectural components such as general-purpose and cache control registers. Conducted experiments consider the execution of 10 bare metal benchmarks. Authors in (TANIKELLA et al., 2016) propose a gem5-based framework that allows injecting faults in different microarchitecture elements (e.g., reorder buffer, load-store queue, register file). In this work, each element is subject to small 300-long fault campaign for each of the ten applications collected from both MiBench and SPEC-Int 2006 benchmark suites. A similar gem5-based fault injection framework is described in (DIDEHBAN; SHRIVASTAVA, 2016).

The reviewed frameworks only support the injection of bit-flips in memory and general single-core processor components, including registers, load/store queue, among others. Another drawback of such approaches is the lack of detailed and customizable post-simulation analysis. Reviewed works classify the detected soft errors according to the inspection of the processor architecture context (i.e., memory and registers), disregarding the impact of software components (e.g., functions and variables) on the system reliability. Further, such approaches typically report low simulation performances of up to 3 MIPS (HARI et al., 2012), which restricts the number and the complexity of fault injection campaigns. While some works consider a single ISA (HARI et al., 2012), others use only in-house applications (GEISSLER; KASTENSMIDT; SOUZA, 2014) or bare-metal implementations (TANIKELLA et al., 2016; KALIORAKIS et al., 2015; DIDEHBAN; SHRIVASTAVA, 2016).

Different from the above works, the OVPsim-FIM extension called SOFIA (Soft error Fault Injection Analysis) offers four novel non-intrusive fault injection techniques that provide software engineers with flexibility and full control over the fault injection process, allowing to disentangling the cause and effect relationship between a injected fault and the occurrence of possible soft errors, targeting an specific critical application, operating system or API structure/function. This tool also differs from all previous works by allowing users to define bespoke fault injection analysis and soft error vulnerability classifications, taking into account both software and hardware component particularities and the system requirements.

Figure 3.8: OVPsim-FIM fault injection techniques according to with fault location and injection time.



Source: The Authors

3.8.1 SOFIA

The SOFIA framework was developed on the basis of M*DEV simulator, a more advanced multicore and commercial version of the OVPsim. The M*DEV provides intercept libraries, and multicore debug primitives used to develop the SOFIA fault injection techniques. The SOFIA framework supports six fault injection techniques (A to F) making it suitable for fast and detailed soft error vulnerability analysis at an early design space explorations. Note that the *six* techniques target the register file or physical memory *without altering the target software stack* (i.e., application, OS, and related libraries). This extension proposes four new fault injection techniques besides the already available in the original OVPsim-FIM, Figure 3.8 displays the six fault techniques supported by the module. These two techniques already embedded on the OVPsim-FIM randomly assign fault injections to deploy bit-flips targeting the Register File (A) (e.g., sixteen integer registers from r0 to r15) and Physical Memory (B) (e.g., one bit in a one-gigabyte memory) respectively.

3.8.1.1 Application Virtual Memory

The technique C targets the application virtual address space (VAS) to enhance the fault injection controllability. Operating systems abstract the physical hardware implementation from the user by making available a set of virtual address ranges while using a translation table to connect both virtual and physical ranges. The promoted technique C automatically extracts the virtual addressing ranges from the target application object code, including different segment addresses (e.g., data, code, read-only, debug) during the phase 1 in order to create the fault lists (phase 2). For each fault injection, the SOFIA accesses the target OS virtual memory translation table, acquires the correspondent physical address from the target virtual address, and injects the bit-flip in the system physical memory. The advantage of this technique over the purely physical memory fault injection relies on the fact that it targets the application virtual address space (VAS) without affecting the OS, other applications, or libraries, reducing the number of necessary faults campaigns since soft errors manifest much quicker. Additionally, this approach enables the user to target a particular application running in a complex environment with multiple applications and libraries.

3.8.1.2 Application Variables and Data Structures

To precisely evaluate an application vulnerability to soft errors the fault injection infrastructure should provide efficient means to correlate errors with particular application blocks or data structures. Technique D (Application Variable) enables the software engineer to direct bit-flip injections into particular data structures, enabling to isolate and identify the most vulnerable ones with a lower number of fault campaigns and higher precision. Further, this approach allows evaluating the impact of specific application variables on the soft error reliability without affecting the application main control flow. For this purpose, the user is asked to inform the target variable name, and the SOFIA framework automatically captures the variable virtual address to create a set of faults targeting the data structure virtual addressing. During the application execution, the variable will suffer a single bit-flip on its physical memory representation using the aforementioned translation table.

3.8.1.3 Function Code

To explore the criticality of function codes, this work proposes the technique E (*Function Object Code*) that limits the injection spectrum to the memory region, which holds the target function code (i.e., instructions), including local variables, etc. The probability of function code to be hit by a transient fault depends on its relative size when compared to the complete memory range. This technique enables the user to investigate the soft error reliability of a particular function independent of its size or execution time.

3.8.1.4 Function Lifespan

In state-of-the-art frameworks, the fault injection time follows a random generation scheme where faults are scattered over the entire application and OS execution. Consequently, the number of faults per function depends on its execution time and not on its criticality for the system reliability. *Function Lifespan* (F) technique enables to reduce the fault injection spectrum by limiting the insertion time to those small intervals where the target function is active. During the simulation, the fault monitor component (B, Section 3.2) traces the function execution at the instruction level and thus create a list of active ranges, including the processor core(s) that executed the underlying function. In this work, the lifespan technique implementation targets the general purposed registers (r0-r15), including the program counter (PC) and the stack pointer (SP). However, this technique can be combined along with any other fault injection technique (e.g. (C, D, E) whenever necessary.

3.8.1.5 Fault Inspection

The OVPsim-FIM extension provides a flexible soft error assessment module, which enables the creation of customizable error classifications. The software engineer can alter the classification order, add new classes, change their criteria, or include new parameters. This module is capable of extracting one or more variables value on the fly during the simulation, and compare then with pre-characterized data. For instance, a target variable can be compared to another variable in the same application. New classifications are easily embedded in the fault campaign flow, enabling custom fault injection scenarios according to the application requirements. Injecting a bit-flip in the physical memory (i.e., B, C, D, and E) produces dirty memory in the majority of the cases. In other words, the targeted bit remains untouched until the simulation conclusion, resulting in silent data corruption (SDC). Nevertheless, if the mismatch against the faultless execution is the target bit (i.e., no additional control flow or memory errors), it can be considered benign depending on the system constraints. The SOFIA tool is capable of differencing dirty memories from other types of errors.

3.9 Closing Remarks

This chapter presented a fault injection framework targeting commercial multicore system executing complex software stacks. The promoted framework provides a fast and flexible soft error assessment tool, especially for early design space explorations. During this phase, multiple software stacks and hardware configurations can be tested in feasible time using the two VP-FIMs developed in this chapter. After exploring randomly assigned fault injections this work investigated new techniques to target/expose the system critical software segments. In this way enabling a more substantial fault coverage in a complex software stack by prioritizing the critical portions of the code. Next, Chapter 4 will use this FI framework to evaluated hundreds of distinct fault injection scenarios considering commercial multicore systems and real workloads.

4 EVALUATION OF PROPOSED FAULT INJECTION FRAMEWORK

This Chapter addresses the three additional contributions of this Thesis: (i) An extensive comparison between two distinct virtual platforms (i.e., instruction vs. cycle-accurate) regarding simulation performance, soft error analysis credibility/accuracy, and fault injection flexibility (Section 4.2); (ii) Explore the impact of different software stacks including OS, parallelization library, ISA using more than 50 distinct embedded and high-performance applications with up to 85 billion of object code instructions (Section 4.3); (iii) Demonstrates the benefits of the novel fault injection techniques and error inspection described in the Section 4.4.

Table 4.1: Fault injection campaigns summary.

VP	*Description*	*Faults*	*Total of faults*	*Simulation time (hours)*
OV	16 Rodinia Benchmarks for one, two, and four cores	8,000	384,000	545.82
GA	16 Rodinia Benchmarks for one, two, and four cores	8,000	384,000	5,855.72
GD	16 Rodinia Benchmarks for one, two, and four cores	8,000	384,000	44,117.78
OV	10 NPB OpenMP version for one, two, and four cores	8,000	240,000	1,599.22
OV	10 NPB Serial version for a single core	8,000	80,000	490.69
OV	9 NPB MPI version for one, two, and four cores	8,000	200,000	1,160.11
GA	10 NPB OpenMP version for one, two, and four cores	8,000	240,000	179,137.78
GA	10 NPB Serial version for a single core	8,000	80,000	394,468.89
GA	9 NPB MPI version for one, two, and four cores	8,000	200,000	578,555.56
OV	16 Rodinia Benchmarks for one, two, and four cores; time-slice 0.00001	8,000	384,000	615.53
OV	16 Rodinia Benchmarks for one, two, and four cores; time-slice 0.000001	8,000	384,000	782.86
OV	16 Rodinia Benchmarks for one, two, and four cores; time-slice 0.0000001	8,000	384,000	815.00
		Total	3,344,000	1,208,144.96

OVPsim-FIM (OV) - gem5-FIM atomic (GA) - gem5-FIM detailed (GD)
Source: The Authors

The remaining results in this Chapter comprise **3,344,000** fault injections which require up to **2 million** simulation hours. Considering a single-thread sequential computer, this workload requires approximately more than **150 years**. Table 4.1 summarizes the fault campaign presented in this chapter. Section 4.1 summarizes the adopted benchmark suites with their a description and characteristics.

4.1 Experimental Setup

This Thesis performs thousands of fault injections using distinct application. Configurations including single, dual, quad, and octa-core ARM Cortex-A9 processors (ARMv7 Architecture) or Cortex-A72 (ARMv8). The gem5-FIM includes a two-level cache memory model where the detailed mode provides a more accurate timing model while the OVPsim does not account for a cache memory model, and thus, accessing the RAM directly. To avoid external influences and assure the closest comparison, the software stack uses the same compilation environment regarding compiler, flags, libraries, and target an identical Linux kernel. Note that the OVPsim does not currently possess a checkpoint in place for the Linux platform, and instead, it boots the Linux kernel for each fault injection. However, the checkpoint load process would be less efficient than simulating the entire boot due to the high OVPsim simulation speed. Consequently, the OVPsim simulation time accounts for the kernel startup, which increases the simulation to approximately 1.3 billion of instructions. Also, the OVPsim-FIM has a signal flag to control the application begins, which enables it to inject fault only after the application start. Table 4.2 summarizes the VP-FIMs experimental setup which is identical to perform a fair comparison between multiple scenarios. Further, MPI applications require a local communication library, which should be compiled and included in the Linux virtual disk. This work uses the MPICH, a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard (GROPP; THAKUR; LUSK, 1999).

This work adopts two distinct workloads[1]: the Rodinia benchmarks (CHE et al., 2009) and the Nasa NAS Parallel Benchmarks (NPB) (BAILEY et al., 1991). The Rodinia is a set of applications well-known developed by the University of Virginia aiming the high-performance computing domain. This suite assembles 24 parallel applications using three different programming APIs (i.e., OpenMP, CUDA, and OpenCL) from distinct domains such as Medical, Biological, Physical, Data Mining, and Image Processing. Nevertheless, CUDA and OpenCL are GPU-based programming languages, and so requiring a distinct simulator, which does not belong to the scope of this work. For this

[1]This work adopts benchmark and application as similar terms.

Table 4.2: Virtual platforms experimental setup.

Parameter		*OVPsim*	*gem5 atomic*	*gem5 detailed*
Architecture	ARMv7	ARM Cortex-A9 multicore		
	ARMv8	ARM Cortex-A72 multicore		
Memory	RAM	One Gigabyte of RAM		
	Cache	None	L1 Inst 32kB 4-Way Associative L1 data 32kB 4-Way Associative L2 512kB 8-Way Associative	
Cross-Compiler	ARMv7	arm-linux-gnueabi-gcc Ubuntu 6.2.0-5ubuntu12		
	ARMv8	aarch64-linux-gnu-gcc Ubuntu 6.2.0-5ubuntu12		
Compilation Flags	ARMv7	-O3 -g -w -gdwarf-2 -mcpu=cortex-a9 -mlittle-endian -DUNIX -static -fopenmp -pthread		
	ARMv8	-O3 -g -w -gdwarf-2 -mcpu=cortex-a72 -mlittle-endian -DUNIX -static -fopenmp -pthread		
Linking Flags	ARMv7	-static -fopenmp -lm -lstdc++ -lm5		
	ARMv8	-static -fopenmp -lm -lstdc++ -lm5		
OS	ARMv7	Linux Kernel 3.13.0-rc2		
	ARMv8	Linux Kernel 4.3.0+		

Source: The Authors

experimental setup, we select 16 OpenMP benchmarks as shown in Table 4.3 from A to P. Table 4.4 shows the simulation time in seconds (i.e., Intel Core I7-7700K 4.20GHz with 16 GB DDR4 2400 MHz) for the Rodinia applications varying the number of target cores and VP-FIM: OVPsim-FIM, gem5-FIM atomic, and gem5-FIM detailed.

The Rodinia simulation time using the OVPsim-FIM is around six seconds independently the number of target cores due to the just-in-time engine. In contrast, the gem5 has several interconnect models for different hardware components, and therefore, each application behavior (e.g., the number of instructions, type of instructions, memory accesses pattern, cache misses, and branch predictor misses) impacts simulation time. Some application (e.g., J, L) are visibly impacted by the number of target cores, for example, the application J simulation time using the gem5 detailed mode varies from 359 to 1,410 seconds, in this case, due to its memory access pattern (data mining algorithm)

The NAS Parallel Benchmark is developed by the NASA Advanced Supercomputing Division (BAILEY et al., 1991) as a set of programs designed to evaluate the performance of parallel supercomputers. With constant support, these applications suffered several revisions during the last years to correct errors and improve its performance. Further, this suite has a unique feature among the benchmark suites: some parallel applications (i.e., OpenMP and MPI) derive from a common serial version. Table 4.5 shows the application name, description, and parallelization paradigm. The NPB aims modern high-performance computers leading to a larger workload when compared with the Rodinia. For example, The Rodinia applications simulation time range from 4 to 104 seconds (i.e., considering the gem5 atomic only) while the NPB varies between 176 to 42,370 seconds (i.e., 12 hours) for a single simulation. Due to the more extended applications, the NAS exploration uses only the gem5-FIM atomic mode because the detailed mode has unacceptable simulation times (i.e., more than a day per simulation). We chose to execute the serial version (Table 4.6) using a single-core ARM Cortex-A9, as its lack any explicit parallelization. Table 4.8 and Table 4.7 display the simulation time for both MPI and OpenMP variants in one, two, and four cores systems.

Table 4.3: Selected Rodinia applications.

#	*Name*	*Domain*	#	*Name*	*Domain*
A	backprop	Pattern Recognition	I	myocyte	Biological Simulation
B	bfs (Breadth-First Search)	Graph Algorithms	J	nn (k-Nearest Neighbours)	Data Mining
C	heartwall	Medical Imaging	K	nw (Needleman-Wunsch)	Bioinformatics
D	hotspot	Physics Simulation	L	particlefilter	Medical Imaging
E	hotspot3d	Physics Simulation	M	pathfinder	Grid Traversal
F	kmeans	Data Mining	N	srad v1	Image Processing
G	lavaMD	Molecular Dynamics	O	srad v2	Image Processing
H	lud	Linear Algebra	P	streamcluster	Data Mining

Source: The Authors

Table 4.4: Rodinia applications simulation time varying the VP and number of cores.

#	*Simulation Time (Seconds)*								
	OVP *Cores*			*gem5 atomic* *Cores*			*gem5 detailed* *Cores*		
	1	2	4	1	2	4	1	2	4
A	4	5	4	11	11	11	30	33	37
B	6	6	6	24	26	27	124	136	150
C	7	5	5	17	17	17	93	103	116
D	5	9	5	22	22	25	111	120	137
E	6	6	6	73	73	75	536	500	526
F	4	4	4	14	14	15	56	60	74
G	4	6	5	20	19	21	123	123	147
H	5	6	5	13	12	14	36	40	60
I	5	4	4	14	15	17	66	71	97
J	6	6	8	57	70	104	359	645	1,410
K	4	5	4	13	14	15	48	59	79
L	4	4	4	39	42	42	282	313	506
M	4	4	4	14	15	15	73	74	76
N	4	5	5	76	71	73	467	507	566
O	4	4	4	40	40	42	260	284	313
P	5	5	5	53	66	59	445	516	746

Source: The Authors

Table 4.5: NAS parallel benchmarks.

Name	*Description*	*Serial*	*OpenMP*	*MPI*
BT	Block Tri-diagonal solver	x	x	x
CG	Conjugate Gradient, irregular memory access and communication	x	x	x
DC	Data Cube	x	x	
DT	Data Traffic			x
EP	Embarrassingly Parallel	x	x	x
FT	Discrete 3D fast Fourier Transform, all-to-all communication	x	x	x
IS	Integer Sort, random memory access	x	x	x
LU	Lower-Upper Gauss-Seidel solver	x	x	x
MG	Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive	x	x	x
SP	Scalar Penta-diagonal solver	x	x	x
UA	Unstructured Adaptive mesh, dynamic and irregular memory access	x	x	

Source: The Authors

Table 4.6: NPB serial benchmarks simulation time.

#	*Simulation Time (Seconds)*	
	OVP Cores	*gem5 Atomic Cores*
	1	1
BT	36.7	5.12×10^3
CG	19.7	1.92×10^3
DC	13.2	182
EP	157	39.5×10^3
FT	31.9	5.22×10^3
IS	13.3	163
LU	22.9	2.21×10^3
MG	13.9	312
SP	21.7	2.67×10^3
UA	114	23.3×10^3

Source: The Authors

Table 4.7: NPB openMP-based benchmarks simulation time.

#	*Simulation Time (Seconds)*					
	OVP Cores			*gem5 Atomic Cores*		
	1	2	4	1	2	4
BT	47.5	39.6	43.8	5.10×10^3	5.48×10^3	6.28×10^3
CG	20.1	21.8	24.5	1.85×10^3	2.06×10^3	2.52×10^3
DC	25.1	15.0	16.5	176	234	225
EP	185	181	179	38.1×10^3	42.4×10^3	42.4×10^3
FT	33.1	34.5	43.3	5.55×10^3	5.93×10^3	6.05×10^3
IS	13.6	14.7	16.1	171	186	202
LU	23.6	24.0	26.5	2.28×10^3	2.49×10^3	3.26×10^3
MG	14.3	15.5	18.3	318	340	357
SP	22.1	24.0	34.4	2.75×10^3	2.83×10^3	3.48×10^3
UA	124	126	138	24.9×10^3	26.5×10^3	27.2×10^3

Source: The Authors

Table 4.8: NPB MPI-based benchmarks simulation time.

#	*Simulation Time (Seconds)*					
	OVP *Cores*			*gem5 Atomic* *Cores*		
	1	2	4	1	2	4
BT	38.1	$*$	51.5	5.00×10^3	$*$	6.26×10^3
CG	21.3	22.9	29.2	1.84×10^3	2.06×10^3	2.49×10^3
DT	23.9	15.1	18.5	440	417	466
EP	151	155	168	39.0×10^3	41.3×10^3	42.4×10^3
FT	33.9	36.2	47.4	5.48×10^3	5.85×10^3	6.39×10^3
IS	14.0	21.3	29.6	176	225	472
LU	26.0	26.1	29.7	2.66×10^3	2.99×10^3	3.59×10^3
MG	14.7	16.8	19.4	332	391	619
SP	23.4	$*$	29.5	2.90×10^3	$*$	3.87×10^3

*MPI BT and SP are not available for dual-core processors
Source: The Authors

4.2 Performance and Accuracy Evaluation of Instruction-Accurate Virtual Platforms

This section compares the OVPsim-FIM (instruction-accurate) precision against the gem5-FIM (cycle-accurate) considering three main aspects: accuracy (Section 4.2.1), OVPsim engine configuration (Section 4.2.2), and simulation speed (Section 4.2.3). For this purpose, this study employs the ARM Cortex-A9 ISA configured in the OVPsim-FIM and gem5-FIM using atomic and detailed modes. Recapping, the deployed soft error model consists of randomly generated single bit-flips injected in any available general-purpose register (i.e., r0-15) during the software stack execution (i.e., OS, drivers, and applications). The OS reliability is not the primary focus of this chapter, and thus fault injections only occur during the application lifespan (i.e., the OS startup is not subject to faults). Nevertheless, OS system calls arising during this period (i.e., application execution time) are susceptible to fault injections as part of the application behavior.

4.2.1 Accuracy

This subsection explores the impact on the soft error vulnerability assessment using different software simulation approaches: discrete event-driven simulation (e.g., gem5) and a just-in-time dynamic binary translation engines (e.g., OVPsim). Figures 4.1 and 4.2 show 8,000 randomly assigned register fault injections for each scenario using 10

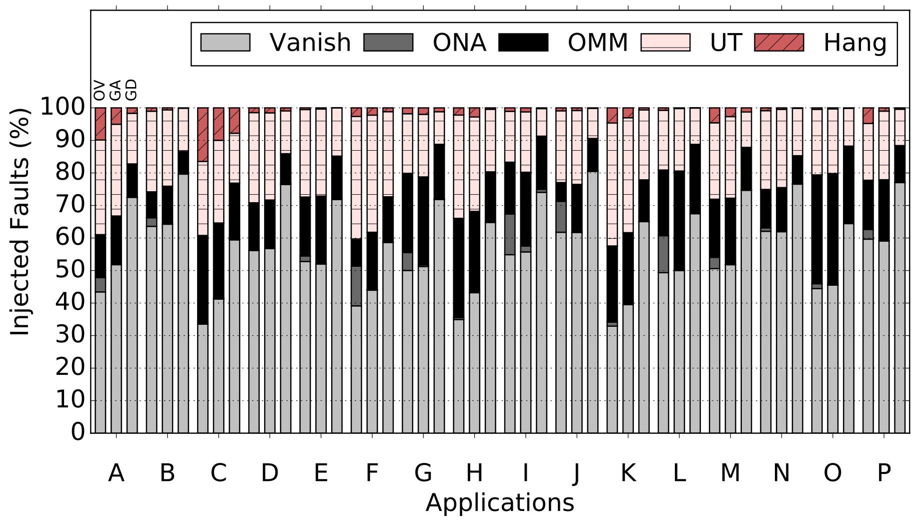
NPB Serial, 10 NPB OpenMP, 9 NPB MPI, and 16 Rodinia OpenMP applications.

The gem5 has a higher vanished percentage (i.e., no trace emerges from the fault injection), in particular, the detailed mode when compared against the OVPSim. This difference can be traced to components only available in the detailed mode leading to microarchitectural masking, i.e., some hardware component overwrites the target register/memory bit before the fault propagation to other memory elements. For instance, the gem5 detailed mode has a more precise cache coherency model, increasing the cache miss rate slightly, and by consequence, re-fetching some cache lines. The register renaming module is component exclusive to the detailed mode that protects the register-file from pipeline data hazards by mapping logical registers (e.g., r0-15) on physical ones. Both OVPsim and gem5 (atomic mode) do not emulate these two microarchitectural elements, then using single-cycle read and write operations resulting in the similar masking rate. In contrast, applications G, I, and N (Figure 4.1a) are exceptions of this behavior, in these cases, the OVPsim and gem5 (detailed mode) have more similar outcomes than the gem5 atomic versus the detailed. Due to its instruction-accurate engine the OVPsim simulation time (i.e., number of executed instructions) is affected by the running application characteristics (Section 4.2.2 details this behavior). In turn, the vanish errors results collected from gem5 detailed show that a more substantial occurrence of ONA error when compared to gem5 atomic and OVPSim-FIM, as illustrated in Figure 4.1c by applications such as C and M executing on a quad-core processor.

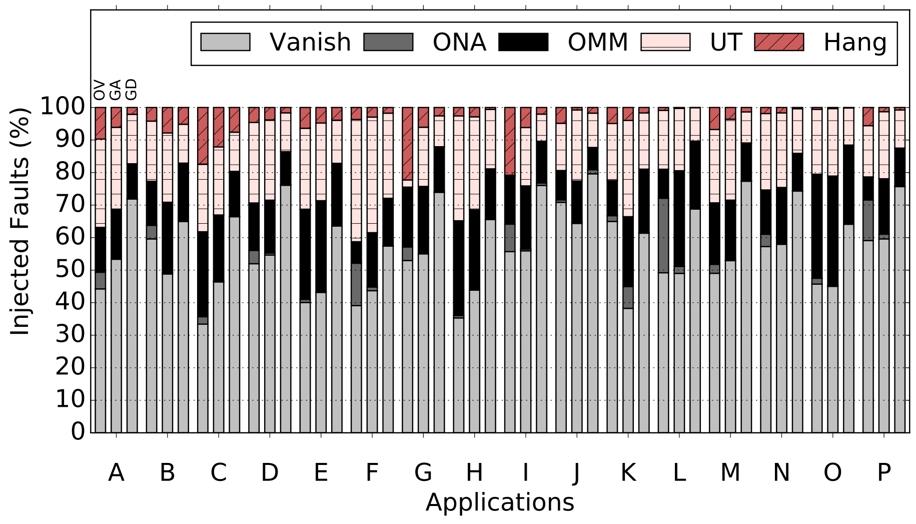
The Rodinia benchmarks have a more significant presence of ONA errors then the NPB, in other words, at least one incorrect register bit (e.g., r1, PC, SP) differs from the faultless execution. The NPB longer execution reduces the probability of dirty bits (i.e., lower ONA presence) in the simulation outcome due to a higher likelihood of a bit masking when comparing with the Rodinia applications. For example, the NPB applications BT and EP (Figure 4.2a) or the Rodinia benchmarks A, F, and L (Figure 4.1b) when using the OVPsim. Further, the Rodinia OpenMP applications have a higher number of hangs then the NPB ones, notably increasing the number of cores (Figure 4.1c). A *Hang* occurs when the application execution time exceeds the double of expected time (i.e., time compared to the faultless executions). The two leading causes to explain the higher hang presence in the Rodina: (i) the fault affected a loop statement (e.g., while, for) wherein these cases a more extended execution translates to more significant recovery time. For comparison's sake, the average Rodinia application executes 80 million instructions while NPB applications execute on average around 17 billion of instructions (i.e., 212x larger).

Figure 4.1: Rodinia benchmarks 8000-fault injection campaign for a multicore ARM Cortex-A9 processor.

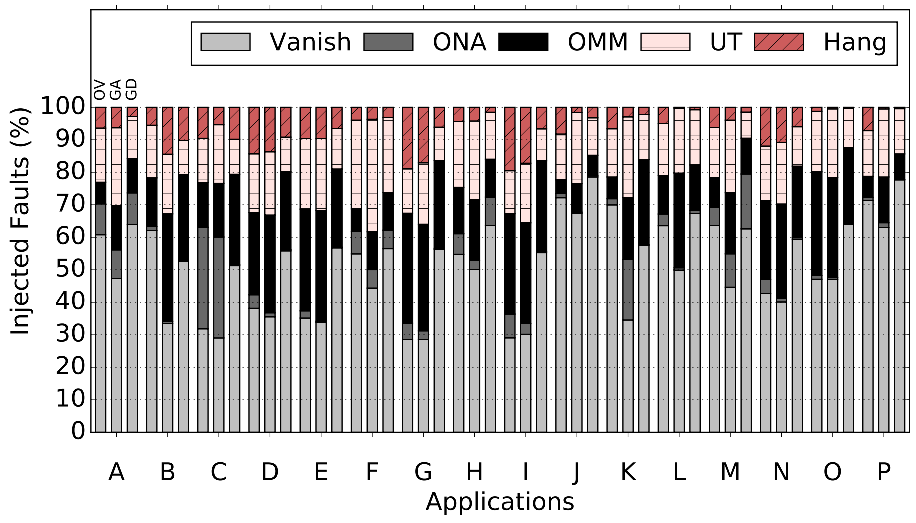
(a) Single-core ARM Cortex-A9 processor.



(b) Dual-core ARM Cortex-A9 processor.



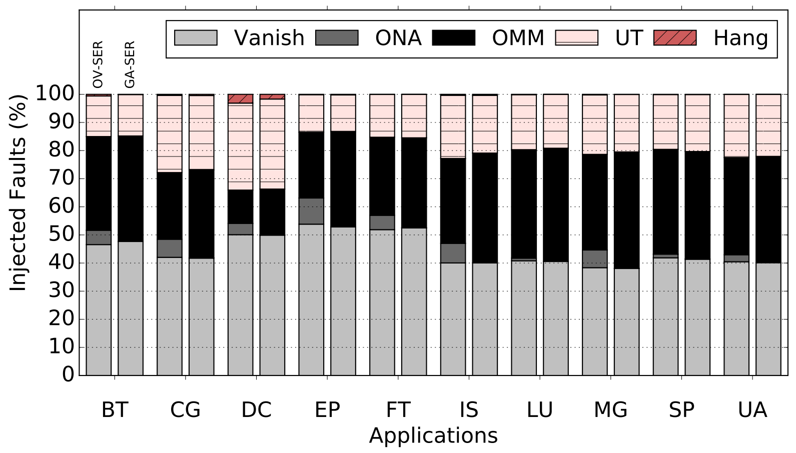
(c) Quad-core ARM Cortex-A9 processor.



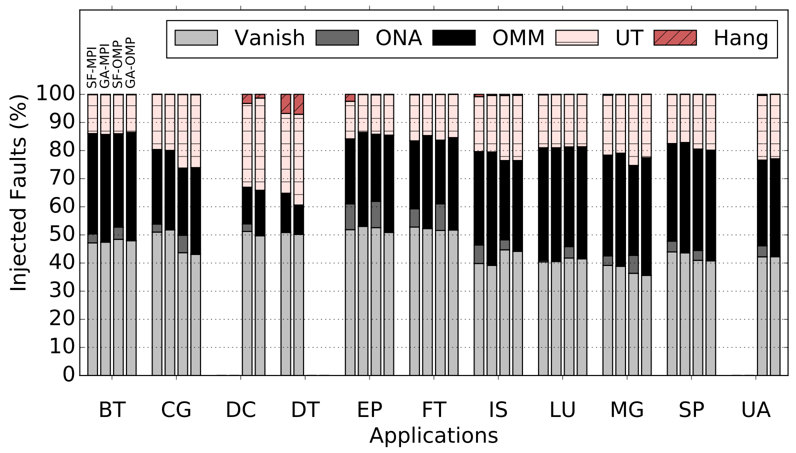
OVPsim-FIM (OV) - gem5-FIM atomic (GA) - gem5-FIM detailed (GD)
Source: The Authors

Figure 4.2: NPB applications 8000-fault injection campaign for a multicore ARM Cortex-A9 processor.

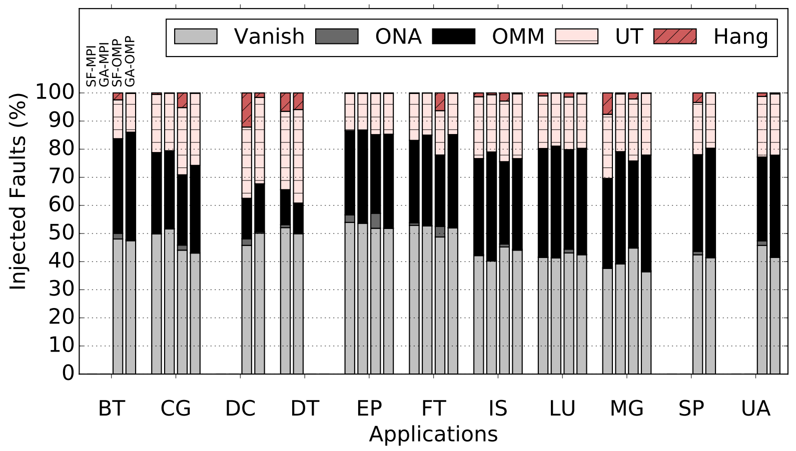
(a) Serial applications.



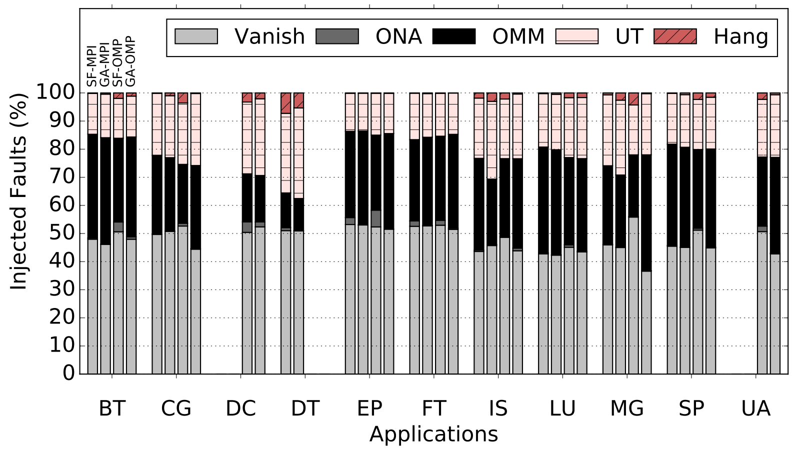
(b) MPI and OpenMP applications using the single-core processor.



(c) MPI and OpenMP applications using the dual-core processor.



(d) MPI and OpenMP applications using the quad-core processor.



MPI BT and SP are not available for dual-core processors
OVPsim-FIM (OV) - gem5-FIM atomic (GA)
Source: The Authors

(ii) kernel malfunctions: the fault injection leads to unrecoverable kernel perturbations (e.g., a thread scheduler error). A longer execution time reduces the Linux kernel exposure time (i.e., the probability of kernel function be stroke by a fault). In other words, the more prolonged the applications, the less kernel functions execute proportionally.

When considering a multicore processor, increasing the core count results on more thread context switching and combined with sub-linear scalability (i.e., underutilized cores in this context) from the Rodinia applications leads to further the kernel errors. During CPU inactivity moments the OS executes the scheduler algorithm[2] and then moves to a sleep mode *wait for interruption*. Faults striking during this waiting period will remain in the core register file until its wake-up, affecting the Linux kernel thread dispatcher and system control flow.

The gem5 suffers event scheduler malfunctions in some specific cases due to unforeseen application behavior resulting in a simulator crash (classified as UTs). The gem5 simplistic memory representation as a single binary vector, and whenever the Linux MMU translates an out of range address, the simulator reaches a segmentation fault in the host system. On the other hand, the OVPsim withstand better to unexpected application behaviors and continues to simulate the application. Thus, it exceeds the predefined threshold to be considered in an infinite loop, and the FIM pronounces it as a hang error.

To facilitate the data comprehension, we introduce the *Accumulated Classification Mismatch* (ACM), which is defined as the sum of absolute differences between classes divided by the total number of faults. The two VP-FIM in a hypothetical three classes case study (X, Y, and Z) under comparison as presented on Table 4.9. The difference between classes is (i.e., 5, 5, and 10,) 20 from 150 fault injections. Thus, the accumulated classification mismatch for this scenarios ten divide by 150 equals to 6.66%.

Table 4.9: Accumulated classification mismatch hypothetical 150-faults scenario.

Class	*VP-FIM 1*	*VP-FIM 2*	*Absolute Difference*
X	40	35	5
Y	60	55	5
Z	50	60	10
Total Absolute Difference			20
Total Accumulated Mismatch			10
Total Relative Difference			6.66%

Source: The Authors

[2]This scenario executes one application per time, and thus, there is no other thread to run.

Table 4.10: Rodinia benchmarks ACM summary comparing three distinct VP-FIMs.

#	*Comparison*	*One Core (%)*	*Two Cores (%)*	*Four Cores (%)*
Worst Case	OV vs GA	14.34	27.64	39.07
	OV vs GD	32.12	33.02	33.79
	GA vs GD	25.50	24.45	38.01
Best Case	OV vs GA	0.94	2.43	2.46
	OV vs GD	14.49	8.70	6.28
	GA vs GD	14.65	13.71	12.15
Average	OV vs GA	6.95	12.55	13.17
	OV vs GD	22.32	22.91	16.08
	GA vs GD	19.11	19.20	21.17

OVPsim-FIM (OV) - gem5-FIM atomic (GA) - gem5-FIM detailed (GD)
Source: The Authors

Table 4.11: NPB ACM summary considering the OVPsim-FIM against the gem5-FIM atomic.

#	*API*	*One Core (%)*	*Two Cores (%)*	*Four Cores (%)*
Worst Case	Serial	10.55	*	*
	MPI	11.55	9.72	9.49
	OpenMP	11.15	12.36	23.25
Best Case	Serial	1.70	*	*
	MPI	0.68	1.36	0.96
	OpenMP	3.69	3.05	2.38
Average	Serial	5.73	*	*
	MPI	5.32	3.21	3.17
	OpenMP	6.62	7.11	8.39

OVPsim-FIM (OV) - gem5-FIM atomic (GA)
Source: The Authors

In order to analyze the differences among the VP-FIMs Figures 4.3 and 4.4 explore the accumulated classification mismatch in three comparisons: gem5-FIM atomic versus gem5-FIM detailed, OVPsim-FIM versus gem5-FIM atomic, and OVPsim-FIM versus gem5-FIM detailed. As aforementioned, the NPB experimental setup does not include the gem5-FIM detailed due to it's higher simulation time, and thus the NPB mismatch considers only the OVPsim-FIM versus gem5-FIM atomic. Tables 4.10 and 4.11 summarize the ACM information in average, worst-case, and best-case.

Figure 4.3a compares the gem5 atomic and detailed modes using the Rodinia applications in a multicore system (i.e., one, two and four cores). Excluding the application C (heartwall) where the mismatch increases from 25% to 38%, the number of cores has little impact. For example, the application A (backprop) mismatch reduces with a growing number of cores, while the benchmark I (myocyte) has the oposite behavior. In other words, the atomic mode lack of some microarchitectural components imposes an almost constant difference when compared with the detailed mode.

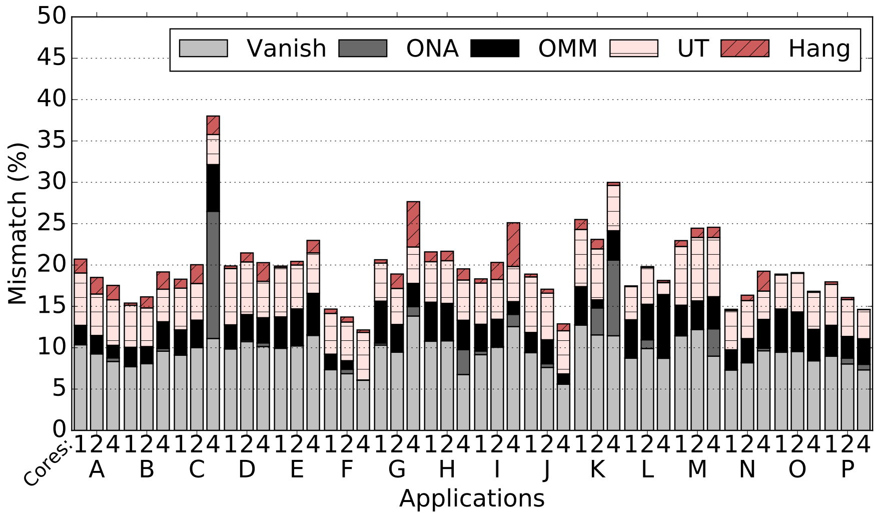
The OVPsim-FIM mismatch against the gem5-FIM detailed is around 22% (i.e., single and dual-core), while 6.95% and 12.55% when compared with the gem5-FIM atomic (Table 4.10). Expected behavior when comparing the instruction-accurate OVPsim against a microarchitectural simulator due to the already discussed microarchitectural masking mechanisms. However, the quad-core scenario shows a lower average mismatch from the OVPsim vs. detailed then atomic vs. detailed. In this case, both OVPsim and gem5 detailed execute more instructions then the gem5 atomic, however, for distinct reasons. The gem5 detailed mode has a better memory timing model adding cycles to the execution time, and thus, delaying the OpenMP synchronization events. The OVPsim engine uses a block-based approach to serialize the multicore simulation adding waiting times between inter-core synchronizations used by the OpenMP.

Figures 4.3b and 4.4a to 4.4c show several comparison scenarios between the OVPsim-FIM and gem5-FIM atomic. OpenMP applications show a mismatch worsening while increasing the number of cores, for example, the Rodinia experiments B, D, and K (Figure 4.3b) alongside the NPB application MG and CG (Figure 4.4c). The Rodinia mismatch increases using quad-core comparing with single-core processors with the worst case jumping from 14.34% to 39.07%s. In the same context, the average error grows from 6.95% to 12.55% considering one and two cores and remains stable for four cores with a mismatch of 13.17%. The ACM has a more diverse distribution across the categories than the gem5 atomic versus gem5 detailed, for example, in some applications, the vanish classification has a more significant mismatch and in others the ONA.

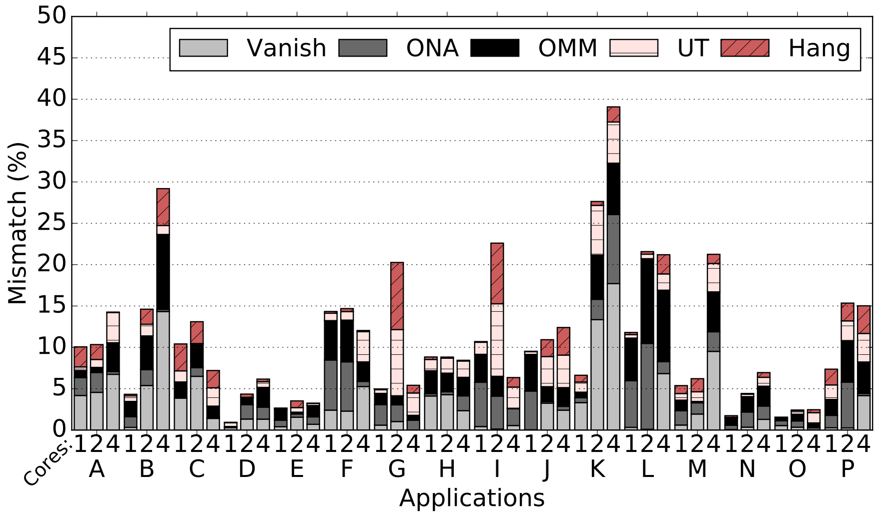
Also, NPB longer workloads reduce the ACM in general when compared with the Rodinia suite. NPB OpenMP applications average mismatch varies from 6.12% to 8.51% in contrast with the Rodinia figures of 6.95% and 13.17%. The worst-case mismatch between the gem5 atomic and OVPsim reduces up to 60% when using the NPB applications compared with the Rodinia benchmarks. MPI applications have in general a smaller mismatch for all experiments, and for instance, the worst case reduces from 23.25% to

Figure 4.3: Rodinia benchmarks ACM considering a multicore ARM Cortex-A9.

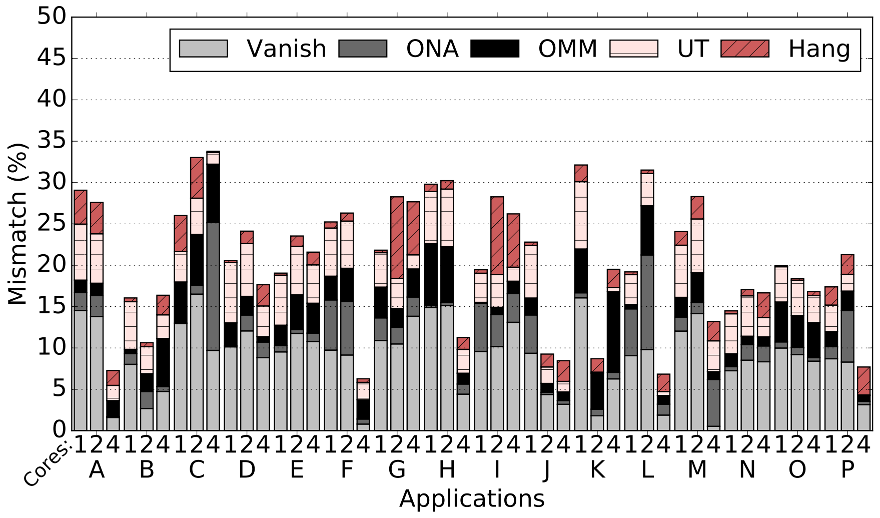
(a) gem5-FIM atomic vs gem5-FIM detailed.



(b) OVPsim-FIM vs gem5-FIM atomic.



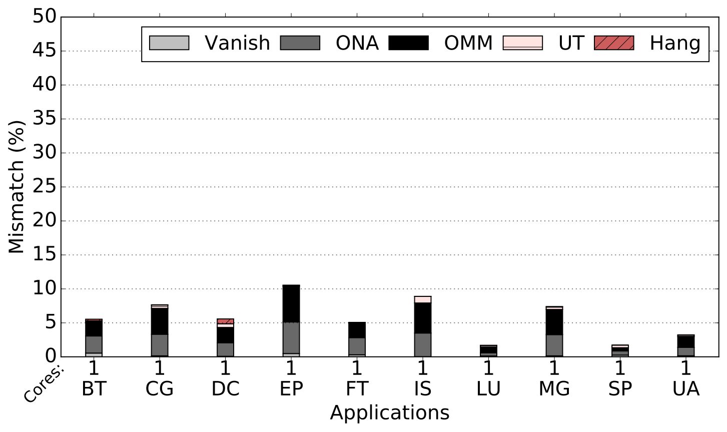
(c) OVPsim-FIM vs gem5-FIM detailed.



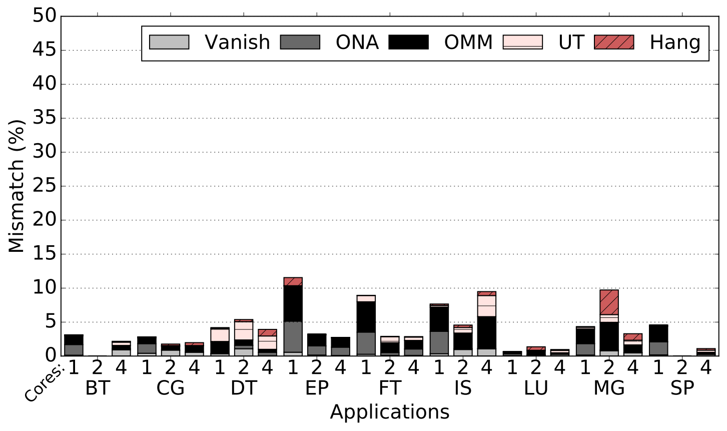
Source: The Authors

Figure 4.4: NPB applications ACM between the OVPsim-FIM and gem5-FIM atomic considering a multicore ARM Cortex-A9.

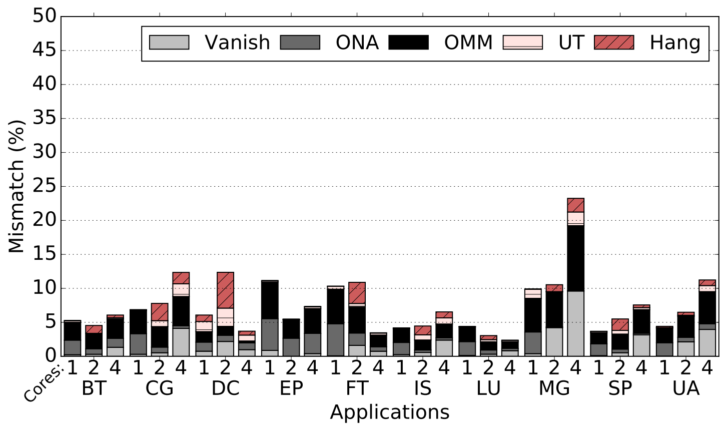
(a) Serial Applications



(b) MPI Applications



(c) OpenMP Applications



MPI BT and SP are not available for dual-core processors
Source: The Authors

9.49% considering a quad-core processor. Serial, MPI and OpenMP differences will be further explored in Section 4.3.2. Next Section 4.2.2 explores different OVPsim engine configurations under fault injection.

4.2.2 Instruction-accurate Simulation Engine Parameters Impact On Soft Error Assessment

The previous subsection explored the soft error analysis accuracy of the OVPsim-FIM instruction-accurate framework against cycle-accurate gem5. Results show an average error of 11% in all cases (considering the gem5 atomic), with the worst-case achieving up to 40In especial, the OpenMP applications displayed a more significant mismatch then the MPI or serial ones. This subsection investigates the origin of such mismatch and analyzes some solutions to improve the OVPsim-FIM accuracy. First, it is necessary to understand how the gem5 and OVPsim software simulation approaches behave under fault presence. The gem5 describes the target microarchitecture as components (i.e., register-file, pipeline, cache) interconnected by a series of events. A scheduler in the gem5 engine executes these events at each simulation tick, updating the whole system state including multiple cores, memories, and other subsystems. Events are executed at a rate of 500 ticks per CPU cycle in the simulated system, and consequently, a complete instruction takes a couple of thousands of ticks.

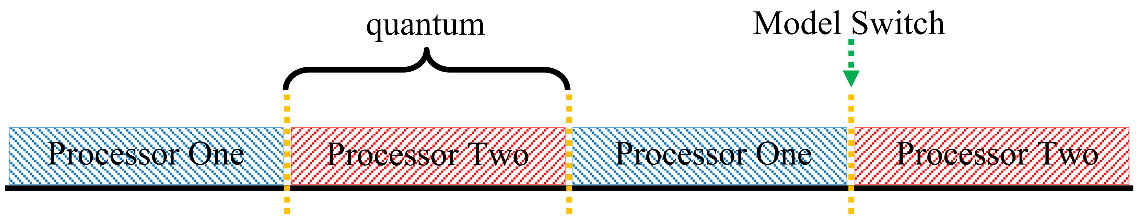
The OVPsim relies on just-in-time (JIT) dynamic instruction translation engine, which translates the target ISA (e.g., ARMv7, ARMv8) to host x86-64 instructions, providing its higher simulation speed. Further, a complete instruction is the OVPsim minimal simulation granularity, in other words, the simulation always advances one instruction. Similarly to an OS scheduler where several processes share the same CPU time, the OVPsim engine simulates each model instance (i.e., processor, core, peripheral) for a fixed-length block of instructions called *Quantum*. The quantum size is configurable using a variable, *time-slice*, representing a time in seconds[3]. The quantum size is given by the following equation where by default the time-slice is 0.001 seconds (one millisecond):

$$BlockSize = (processor\ nominal\ MIPS\ rate)\ x\ 1E6\ x\ (time\ slice\ duration) \quad (4.1)$$

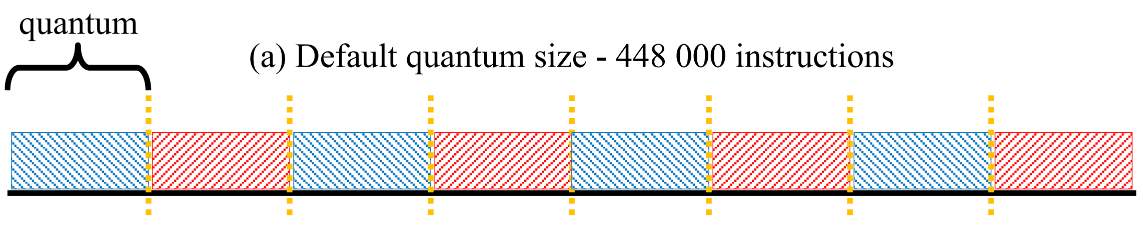
The target processor nominal MIPS rate is 448 MIPS resulting on a quantum size equal to 448 x 1E6 x 0.01 = 448,000 instructions. The OVPsim also deploys a scheduling policy to manage the processors and other components simulation. The simulator selects the first processor, after the first processor (or core) has simulated during 448,000

[3]The time-slice value in seconds refers to an internal configuration parameter and not to the simulation, host, or real-time.

Figure 4.5: OVPsim scheduling policy varying the quantum size for a dual-core processor executing the same workload.



(a) Default quantum size - 448 000 instructions



(b) Smaller quantum size - 224 000 instructions

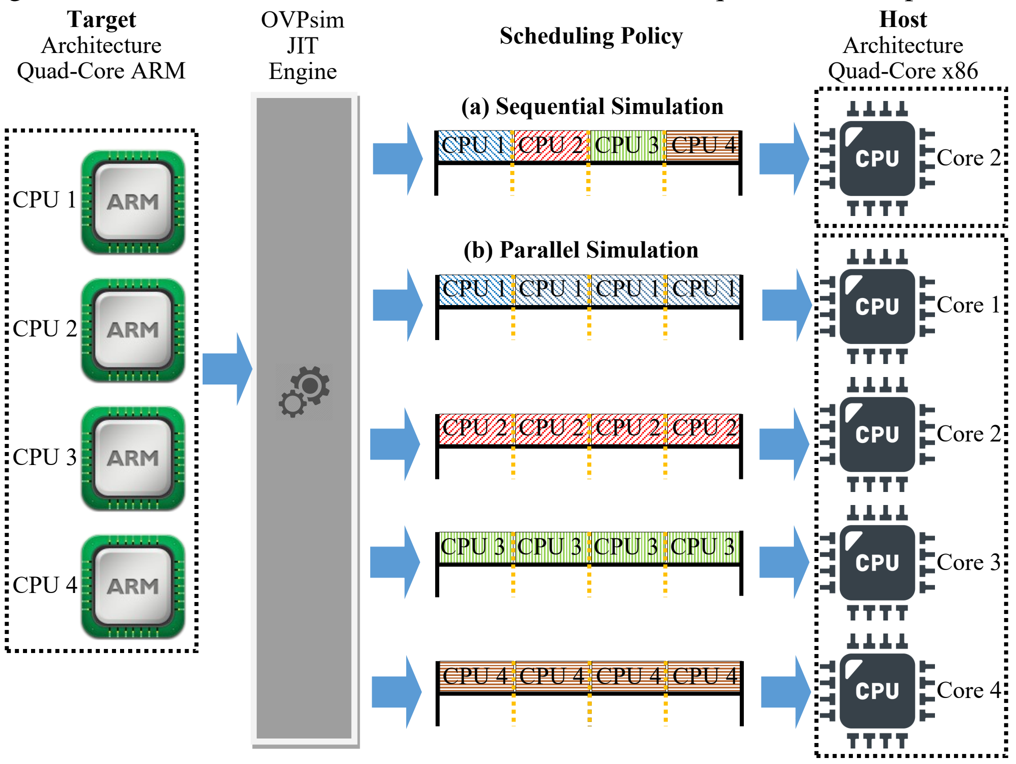
Source: The Authors

instructions, it is suspended, and the next processor assumes. In the case of multicore processors such as the ARM Cortex-A9x2, each processor core receives a separate quantum, and it is scheduling accordingly. Figure 4.5 displays two simulation scenarios regarding a dual-core processor, one with the default quantum and another using half of its size (i.e., 224,000 instructions), which increases the number of model switches for an identical workload. This block simulation approach delays inter-core communication (or synchronization events), as the sender and receiver cores cannot execute simultaneously. For instance, communications between the 1st and the 4th cores (i.e., considering a quad-core processor) take at least two full quantums because the OVPsim needs to simulate 2nd and 3rd quantum blocks, before simulating the target 4th core.

To diminish the soft error mismatch between the gem5 and the OVPsim is necessary to reduce the time required to complete the inter-core communications. Considering the JIT-engine characteristics, there are two main solutions: reduce the time-slice variable or parallelize the quantum simulation. In this context, the OVPsim provides an acceleration feature called *quantum-leap* (QL), which enables mapping processor models (e.g., Quad-Core ARM) to physical host cores (i.e., x86_64 multicore). For example, considering a quad-core ARM processor model, in the sequential simulation mode (Figure 4.6a), four target cores (CPU 1-4) share a single x86 host core (Core 2). In turn, in the quantum-leap mode (Figure 4.6b) each ARM model (CPU 1-4) is individually simulated in an x86 host core (Cores 1-4).

First, this work explores the impact of using distinct quantum sizes (i.e., 448,000,

Figure 4.6: OVPsim simulation workload division into a quad-core host processor.



Source: The Authors

4,480, 448, and 44 instructions per block) and quantum-leap configurations on the OVPsim-FIM accuracy to assess soft error reliability, considering gem5-FIM as the reference. Further, the two quantum-leap scenarios (i.e., QL-1 and QL-2), where each one employs a distinct thread allocation scheme restricted to quad-core processors. Table 4.13 shows the six proposed scenarios mismatch between OVPsim-FIM and gem5-FIM.

4.2.2.1 Quantum-leap impact on soft error assessment

The first scenario (QL-1, Table 4.13) uses a greedy allocation algorithm and affinity thread, i.e., children threads can only execute in a fixed physical core to avoid host caches synchronizing costs. In this context, the OVPsim QL children threads compete with other system threads depending on the system current workload. In the second scenario (QL-2, Table 4.13) the thread affinity is relaxed, enabling children threads to execute in any host core and apply a less aggressive thread allocation technique, which reduces the resource competition with other system threads.

QL-1 and QL-2 scenarios show a worse accuracy for all applications when compared to the single-core OVPsim-FIM execution, highlighting that the only distinction between both scenarios is the thread allocation scheme. The number of quantums ex-

Table 4.12: Rodinia benchmarks time-slice explorations worst, best, and average cases considering the gem5-FIM atomic as reference.

#	*Time-Slice*	*One Core (%)*	*Two Cores (%)*	*Four Cores (%)*
Worst Case	0.001	14.33	27.63	39.07
	0.00001	12.32	19.86	13.40
	0.000001	12.02	19.68	11.97
	0.0000001	11.31	18.53	11.42
	QL-1	*	*	71.24
	QL-2	*	*	38.69
Best Case	0.001	0.93	2.42	2.46
	0.00001	1.12	1.91	1.30
	0.000001	0.52	1.10	1.72
	0.0000001	0.60	1.55	1.23
	QL-1	*	*	4.00
	QL-2	*	*	3.84
Average	0.001	6.95	12.55	13.16
	0.00001	5.79	9.17	5.54
	0.000001	4.96	8.95	5.23
	0.0000001	4.09	7.62	5.39
	QL-1	*	*	17.56
	QL-2	*	*	13.38

Source: The Authors

ecuting in parallel varies according to the target application and host system workload leading to a simulation performance improvement of 200% on average when using a host quad-core processor. The quantum-leap provides a higher performance, however, its disadvantages outweigh the simulation performance gain considering the soft error analysis. First, soft error analysis requires deterministic simulation of thousands of FI campaigns and the QL execution can be affected by the host OS thread allocation mechanism and current workload. Second, the FI flow enables other better speedup techniques as discussed in the previous chapter. From this point, this thesis will focus on the sequential execution of the OVPsim only.

4.2.2.2 Mismatch considering the quantum size

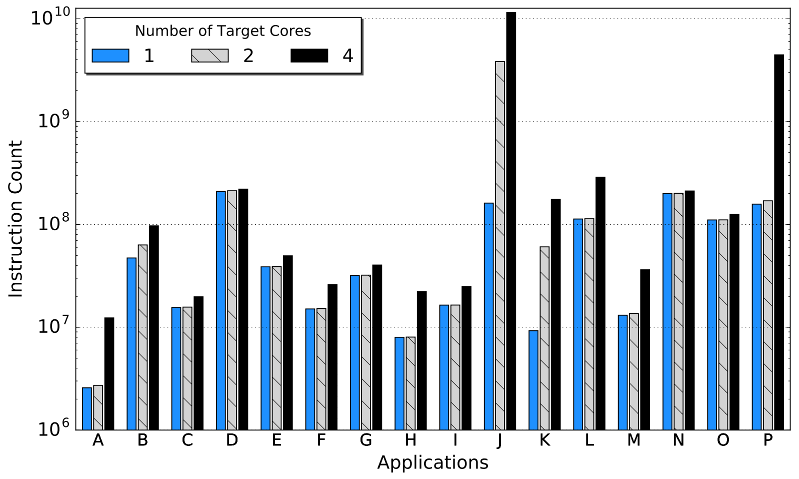
Besides the Table 4.13, Figures 4.8a, 4.9a and 4.10a show the reference gem5-FIM atomic (ψ) and four quantum sizes 448,000 (λ), 4,480 (γ), 448 (β), and 44 (δ) instructions per block for one, two, and four cores on an ARM cortex-A9 respectively. The experiments show that the quantum reduction has a significant impact on the soft error analysis of OVPsim-FIM. For example, the quad-core processor model presents an average accuracy improvement of up to 40% when using the smallest block (i.e., 44 instructions), while some benchmarks reach a five-fold accuracy gain (e.g., F and I). The quantum ten thousand smaller (i.e., 44 instructions) cuts the average error from 6.95% to 4.09% in the quad-core scenario (Table 4.13) while the *worst case* reduces from 39.07% to 11.42%. Noticeably, reducing the quantum size decreases the communication cycles between cores approximating the OVPsim-FIM and gem5-FIM behaviors. While the smallest block size presents the best accuracy, using a slightly larger quantum of 4480 instructions leads to 86% of worst-case improvement and 91% in the average mismatch.

The resulting mismatch can be traced back to its block-based simulation engine, as previously discussed, each core executes a fixed amount of instructions before changing to the next one. Note that inter-core communications are completed during the core switch, leading to temporally unsynchronized cores. Inter-core communication is necessary to synchronize events across multiple cores, for instance, in a parallelization library. The Rodinia OpenMP-based applications use a fork-join parallelization paradigm where synchronization barriers coordinate multiple children threads execution. One synchronization event that requires all cores to reach the same statement (i.e., a barrier) requires multiple quantum executions until completion. Delaying these communication events lead to some extra instructions executed by the OVPsim due to other cores waiting.

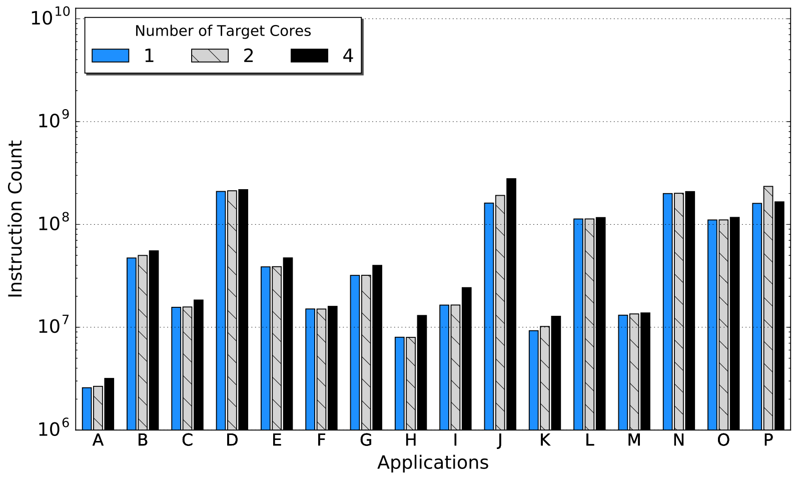
It is possible to observe this behavior by comparing the instruction count of different VP-FIMs executions. Figure 4.7 displays the number of executed instructions for a single faultless execution considering the OVPsim with the default quantum (a), a ten thousand smaller quantum (b), the gem5 atomic (c), and the gem5 detailed (d). Note that in the quad-core scenario the applications *nn, nw, streamcluster, and backprop* present a more substantial variance in the number of executed instructions at the same time as they present the worst soft error mismatch. In contrast, when the overall OVPsim using the 44-instructions quantum follows closely the gem5 atomic comportment (see Figure 4.7c). For instance, in the *nw* scenario, the OVPsim-FIM executes nine times more instructions than gem5-FIM atomic with a 78.15% soft error mismatch. The same scenario using the smaller quantum results on only 28% more executed instructions along with a five-fold soft error accuracy improvement. Reducing the quantum size diminishes this inter-core communication gap (i.e., where one core waits for another) and approaches the gem5-FIM atomic moded behavior under FI, especially when targeting multicore architectures.

Figure 4.7: Instruction count for a single faultless application execution considering four VP-FIMs using one, two, and four cores.

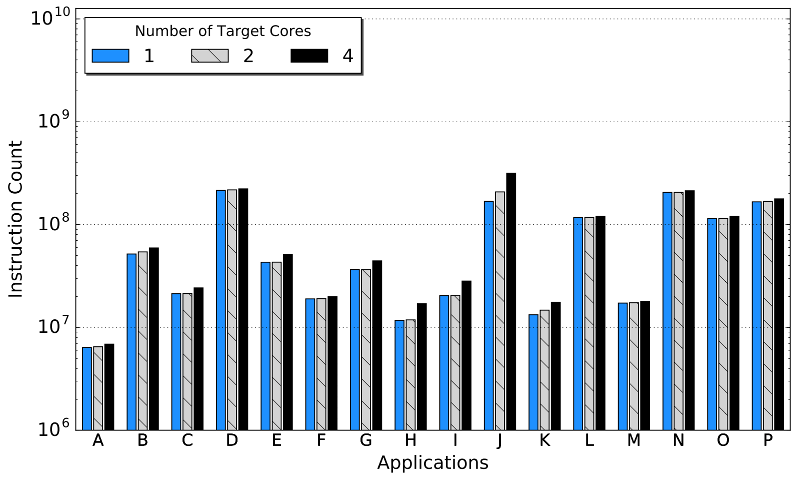
(a) OVPsim default quantum (448 000 instructions).



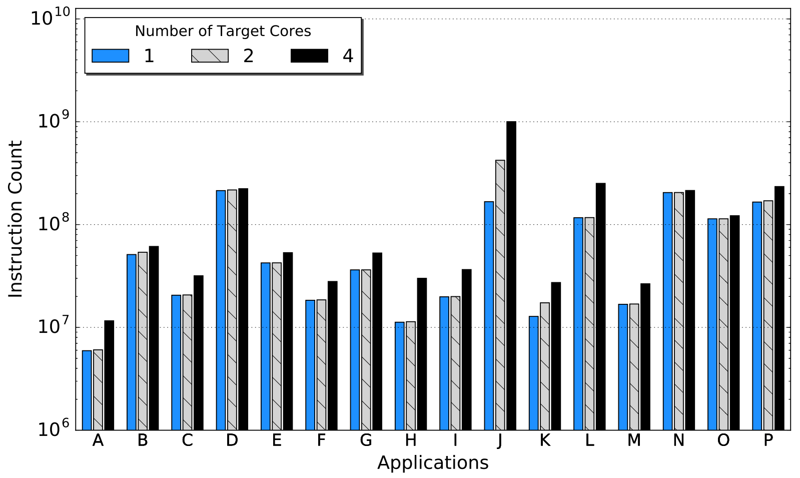
(b) OVPsim 44-instruction quantum.



(c) gem5 atomic.



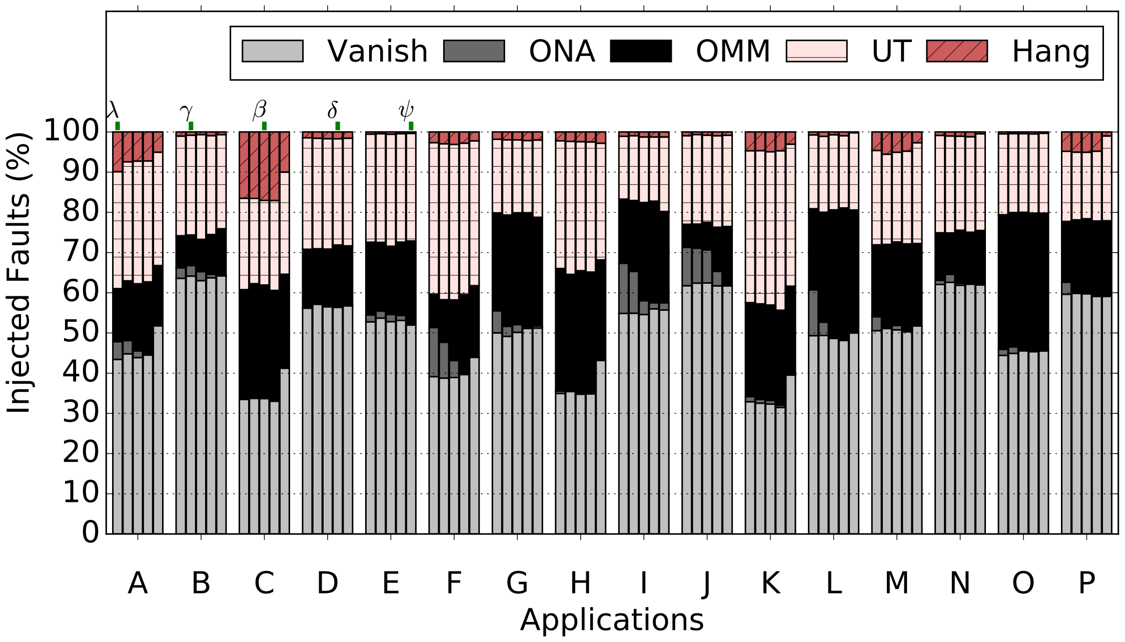
(d) gem5 detailed.



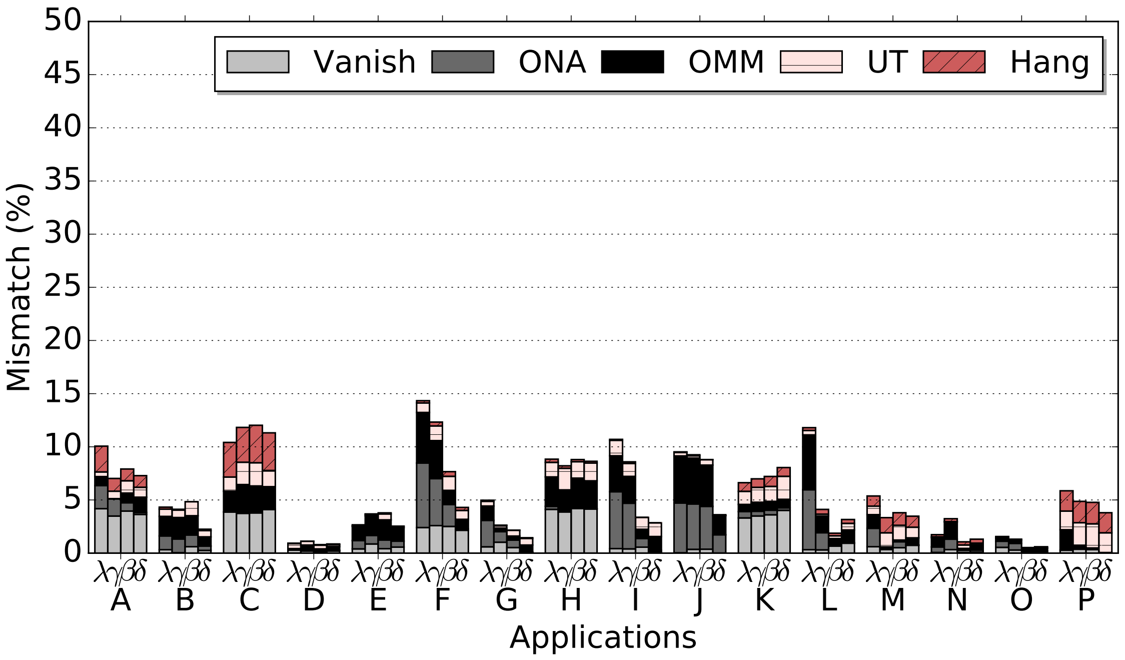
Source: The Authors

Figure 4.8: 8000-fault injection campaign deploying the OVPsim-FIM for a single-core ARM Cortex-A9 processor varying the time-slice and the reference gem5-FIM atomic.

(a) Fault campaign.



(b) Accumulated classification mismatch for each OVPsim-FIM time-slice value.



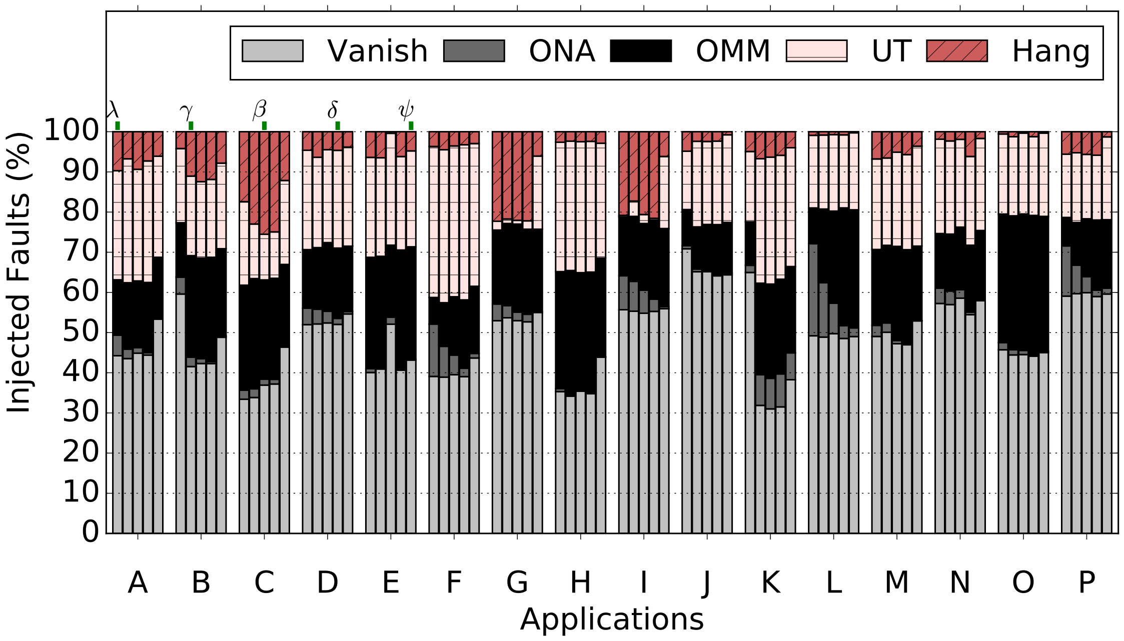
OVPsim-FIM time-slices values: 0.001 (λ), 0.00001 (γ), 0.000001 (β), and 0.0000001 (δ)
gem5-FIM atomic (ψ)
Source: The Authors

While the optimal quantum size varies according to the application behavior and how its synchronization primitives are defined, its reduction improves the overall simulation accuracy.

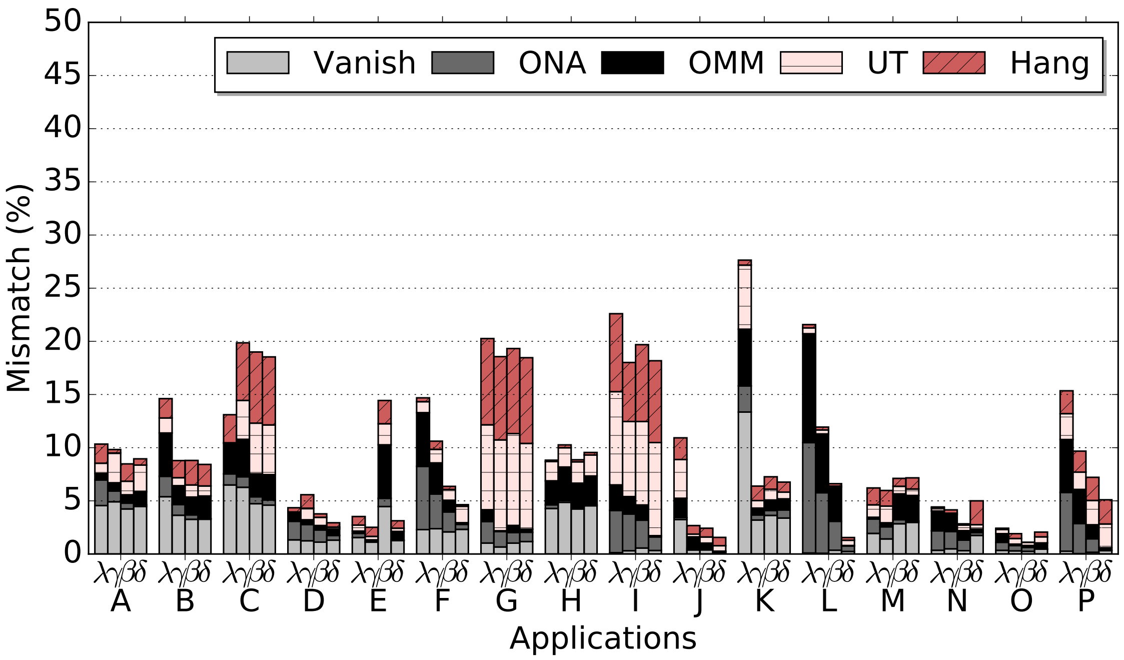
We select the smallest quantum (i.e., a 44 instructions block size) to expand our

Figure 4.9: 8000-fault injection campaign deploying the OVPsim-FIM for a dual-core ARM Cortex-A9 processor varying the time-slice and the reference gem5-FIM atomic.

(a) Fault campaign.



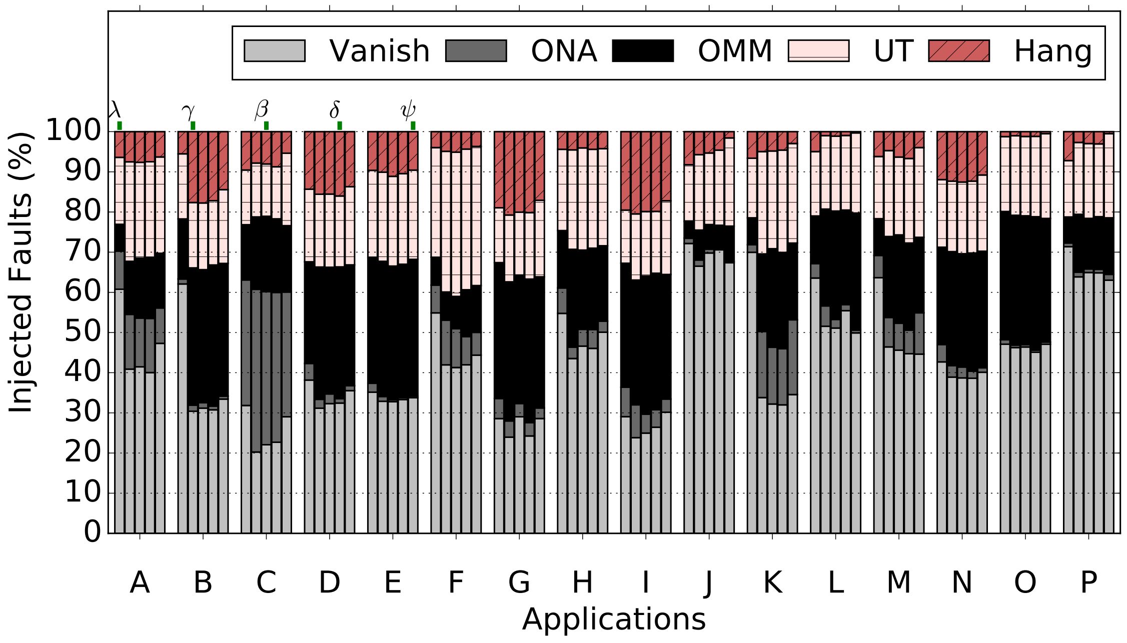
(b) Accumulated classification mismatch for each OVPsim-FIM time-slice value.



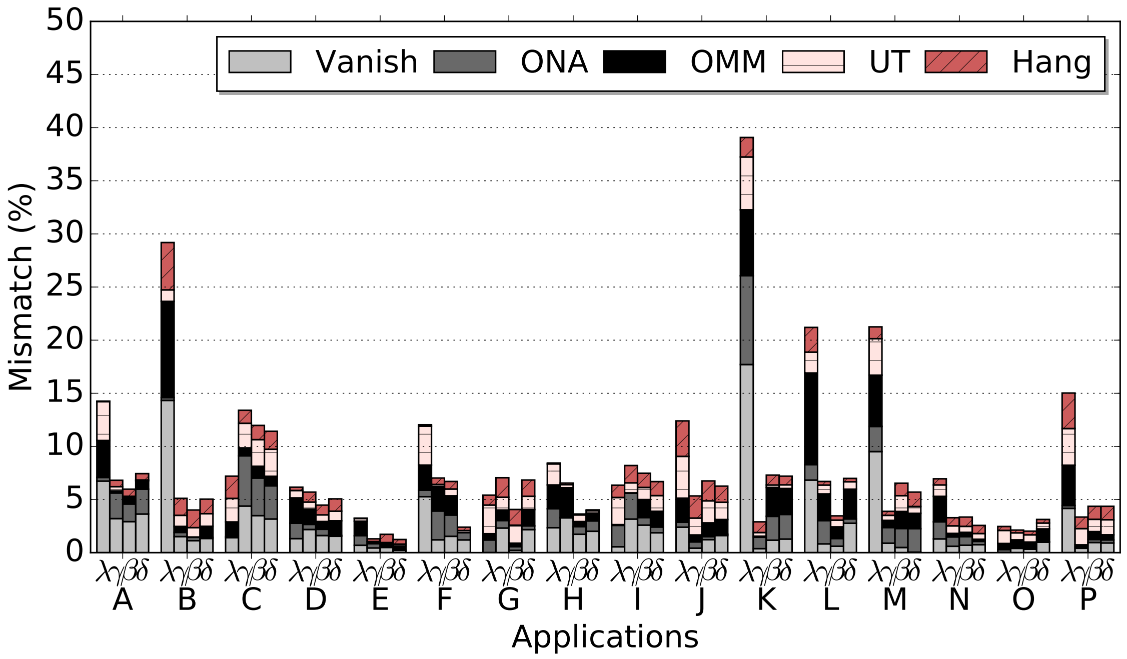
OVPsim-FIM time-slices values: 0.001 (λ), 0.00001 (γ), 0.000001 (β), and 0.0000001 (δ)
gem5-FIM atomic (ψ)
Source: The Authors

Figure 4.10: 8000-fault injection campaign deploying the OVPsim-FIM for a quad-core ARM Cortex-A9 processor varying the time-slice and the reference gem5-FIM atomic.

(a) Fault campaign.



(b) Accumulated classification mismatch for different OVPsim-FIM time-slices for a quad-core ARM Cortex-A9 processor in comparison with the gem5 atomic.



OVPsim-FIM time-slices values: 0.001 (λ), 0.00001 (γ), 0.000001 (β), and 0.0000001 (δ) gem5-FIM atomic (ψ)

Source: The Authors

Table 4.13: Rodinia mismatch benchmarks comparison considering the OVPsim-FIM with default (DF) and 44-instruction quantum (Q) against the gem5-FIM atomic.

#	*Comparison*	*One Core (%)*		*Two Cores (%)*		*Four Cores (%)*	
		DF	*Q*	*DF*	*Q*	*DF*	*Q*
Worst Case	OV vs GA	14.34	11.31	27.64	18.54	39.07	11.43
	OV vs GD	32.12	33.55	33.02	30.74	33.79	39.16
	GA vs GD	25.50		24.45		38.01	
Best Case	OV vs GA	0.94	0.60	2.43	1.55	2.46	1.24
	OV vs GD	14.49	14.41	8.70	16.57	6.28	8.54
	GA vs GD	14.65		13.71		12.15	
Average	OV vs GA	6.95	4.10	12.55	7.62	13.17	5.40
	OV vs GD	22.32	21.54	22.91	24.18	16.08	22.38
	GA vs GD	19.11		19.20		21.17	

Source: The Authors

Table 4.14: NPB mismatch comparison considering the OVPsim-FIM with default (DF) and 44-instruction quantum (Q) against the gem5-FIM atomic.

#	*API*	*One Core (%)*		*Two Cores (%)*		*Four Cores (%)*	
		DF	*Q*	*DF*	*Q*	*DF*	*Q*
Worst Case	Serial	8.90	7.34	*	*	*	*
	MPI	11.55	3.45	9.72	3.29	9.49	12.50
	OpenMP	10.32	4.38	12.36	4.21	23.25	4.50
Best Case	Serial	1.70	0.79	*	*	*	*
	MPI	0.68	0.16	1.36	0.55	0.96	1.34
	OpenMP	3.69	1.04	3.05	0.84	2.38	0.24
Average	Serial	5.20	2.73	*	*	*	*
	MPI	5.32	1.30	3.21	1.36	3.17	4.18
	OpenMP	6.12	2.61	7.29	2.07	8.51	2.10

Source: The Authors

investigation using the NPB suite. Figure 4.11 shows the Rodinia benchmarks FI using the OVPsim-FIM with reduced quantum (OV-Q), the gem5-FIM atomic (GA), the and gem5-FIM detailed (GD) while Figure 4.12 replicate the experimental setup using the NPB. Tables 4.13 and 4.14 display the mismatch between gem5-FIM and OVPsim-FIM using the default quantum size (DF) of 448,000 instructions and a smaller quantum (Q) value of 44 instructions, considering four distinct workloads: NPB OpenMP (OMP), NPB MPI, and NPB Serial (SER) alongside the Rodinia OpenMP applications.

Notice, the OVPsim-Q (i.e., OVPsim with the 44-instructions quantum) decreases the mismatch whenever compared with the gem5-FIM atomic for the Rodinia suite, and more accentuated in the quad-core system due to the instruction count reduction as previously mentioned. For example, the average ACM reduces by 58% (from 13.17% to 5.4%) as shown in Table 4.13. In the same scenario, the worst-case scenario diminishes from 39.07% to only 11.43%. Note the migration of ONA to OMM by decreasing the quantum size, in other words, the incorrect content previously restricted to the register file migrate to the final memory. For instance, note application A, E, F, and L in Figure 4.11a. Notice, the campaigns simulated with the OVPsim-Q presents a mismatch reduction in 28 out of 30 scenarios with a significant (five-times) improvement in the worst-case of OpenMP-based benchmarks, which is justified by the impact of synchronization barriers between children threads.

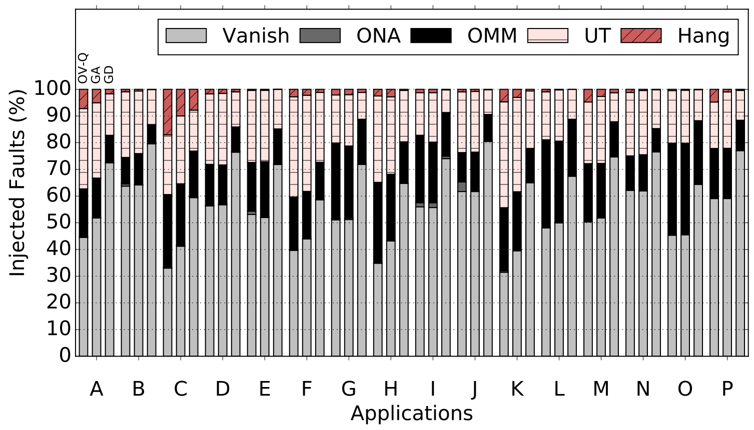
While the Rodinia benchmarks include applications with up to 220 *million*, NPB benchmark applications vary from 16 to 87 *billion* instructions. By consequence, NPB benchmarks have more extended computations between synchronization points than the Rodinia, which impacts on the soft error analysis. NPB benchmarks also have a better workload distribution and scalability, which means in conjunction with the more prolonged execution that children threads have enough instructions to complete one or more quantums between OpenMP barriers. In contrast, Rodinia applications have a smaller computation between synchronization points, sometimes shorten then one quantum execution leading the OpenMP barriers to execute extra instructions while waiting for other threads. The Rodinia behavior magnifies the mismatch originated due to the OVPsim simulation policy using fixed-length instructions blocks, and by consequence, these applications benefit the most when reducing the quantum, achieving a five-fold accuracy gain. Applications using the MPI library are based on independent threads, which leads to a smaller number of synchronization points, and thus resulting in a lower mismatch. The quad-core MPI workload has two NPB applications (*IS and MG*) in which the scenarios with RQ lead to a mismatch worsening. As discussed before, some application may suffer from over reduced quantum size, for instance by selecting the best case for these two applications; the average mismatch reduces to 3.48% and worst-case of 5.85%. *IS and MG* are the smallest NPB applications that might contribute to the precision worsening by reducing the number of instruction per quantum.

The OVPsim-Q decreases the mismatch considering the gem5 atomic while increasing when compared with gem5 detailed. For example, the OVPsim-Q quad-core scenario average ACM is (22.38%) is higher than previously 16.08% with the unmodified OVPsim engine (Table 4.13). Even though the unmodified OVPsim-FIM more accurately emulates the gem5-FIM detailed in some cases, this setup increases the mismatch. The

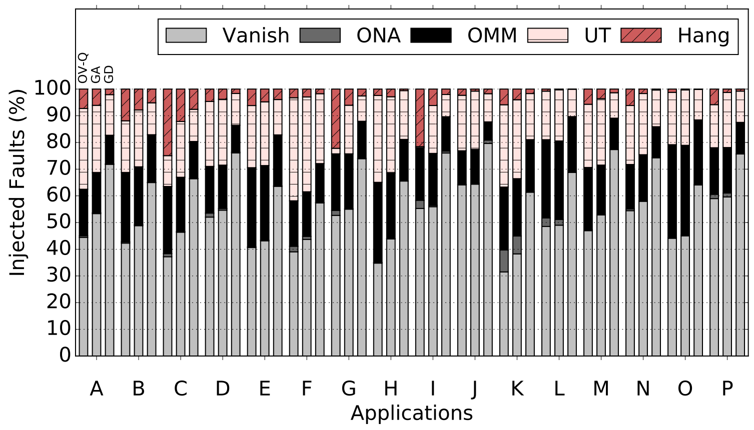
OVPsim-Q has a more similar relationship when compared with the gem5 atomic, and by consequence, a better average figure can be extracted.

Figure 4.11: Rodinia benchmarks the OVPsim-FIM with reduced quantum (OV-Q), the gem5-FIM atomic (GA), the and gem5-FIM detailed (GD) for a multicore ARM Cortex-A9 processor.

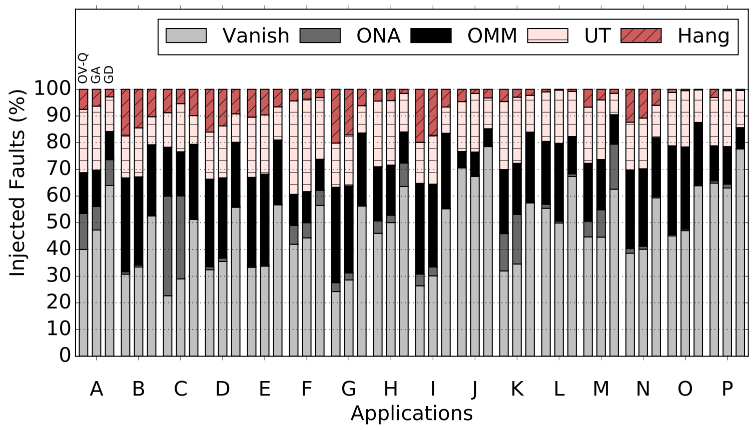
(a) Single-core ARM Cortex-A9 processor.



(b) Dual-core ARM Cortex-A9 processor.



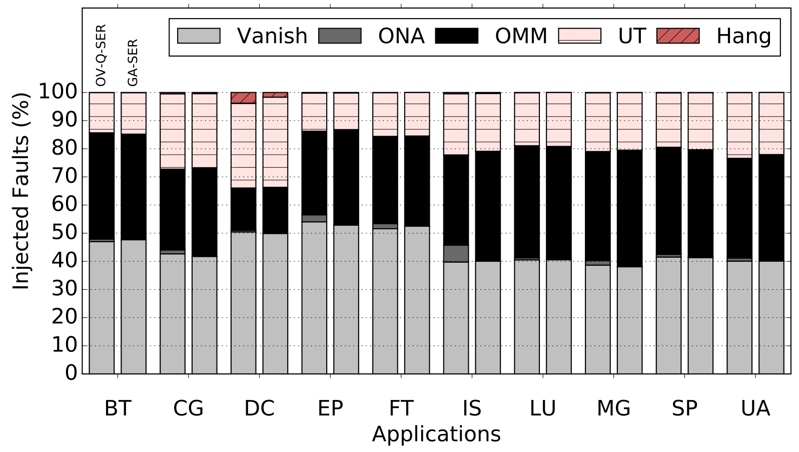
(c) Quad-core ARM Cortex-A9 processor.



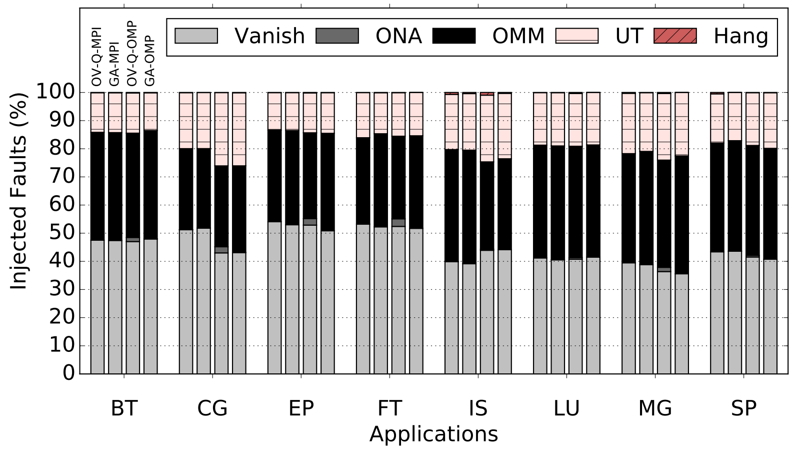
Source: The Authors

Figure 4.12: NPB applications using the OVPsim with smaller quantum (OV-Q) and gem5-FIM atomic (GA) for a multicore ARM Cortex-A9 processor.

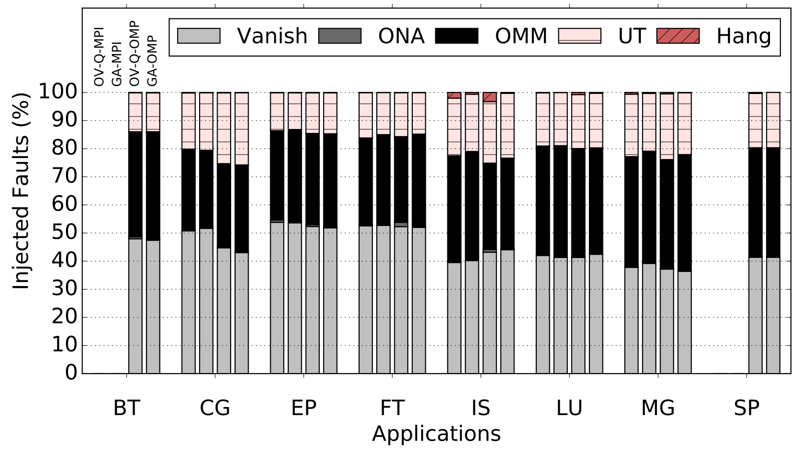
(a) Serial applications.



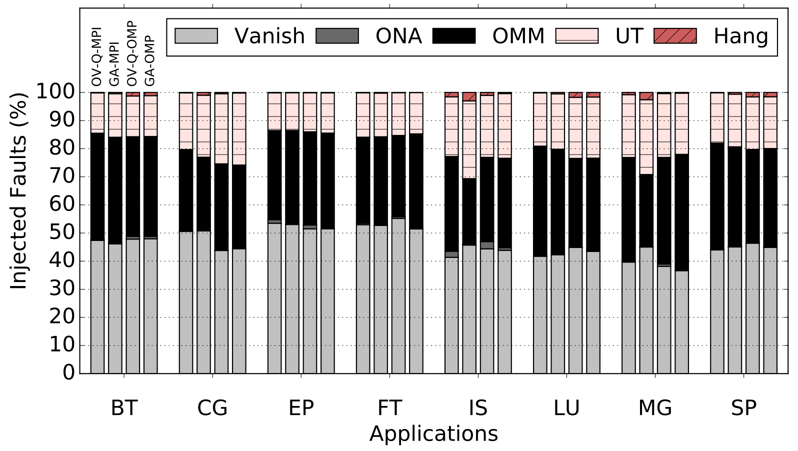
(b) MPI and OpenMP applications using the single-core processor.



(c) MPI and OpenMP applications using the dual-core processor.



(d) MPI and OpenMP applications using the quad-core processor.



MPI BT and SP are not available for dual-core processors

Source: The Authors

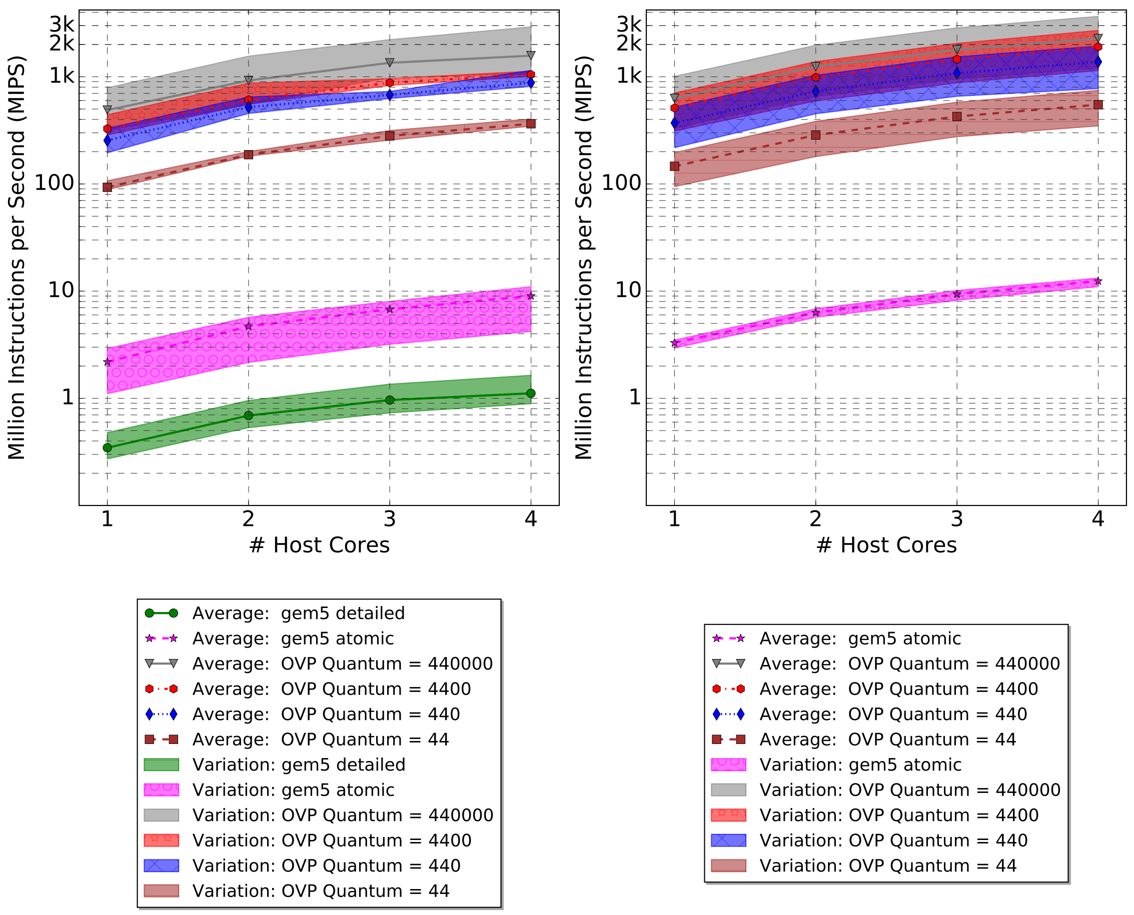
4.2.3 Performance and Speedup

The OVPsim has two main advantages when comparing with other frameworks: modeling flexibility and simulation speed. This subsection compares the simulation speed regarding MIPS considering the gem5 and OVPsim in multiple configurations and workloads. The experiments were conducted in a Quad-core Intel Core I7-7700K 4.20GHz with 16 GB DDR4 2400 MHz. Figure 4.13 presents the simulation performance as we increase the host cores from 1 to 4 where both the OVPsim-FIM and gem5-FIM can perform and manage parallel simulations.

Figure 4.13a displays the first set of simulation considering the Rodinia applications. The gem5-FIM atomic simulation speed ranges from 4.2 to 11 MIPS while the detailed mode achieves 0.89 to 1.65 MIPS. In turn, the OVPsim varies from 345 to 2,921 MIPS depending on quantum configuration and application. Modifying the quantum size

Figure 4.13: Multiple VP-FIMs simulation speed and scalability using a quad-core host processor in terms of MIPS.

(a) Rodinia Benchmarks. (b) NPB Applications.



Source: The Authors

reduces the number of executed instructions per block impacting simulator performance directly due to the increasing switching between the models (i.e., cores). The unmodified OVPsim-FIM (in gray Figure 4.13a) has an average performance of 1,500 MIPS, the 4,400 (red) and 440 (blue) reduce the average performance in 30% and 42% respectively while the smallest quantum reaches 360 MIPS. The second experiment, Figure 4.13b, deploys the NPB larger applications (i.e., up to 87 billion of instructions) shows a better OVPsim performance in all configurations while the gem5 atomic remains stable With four host cores, the longest workload achieves up to 3,910.86 MIPS when deploying the OVPsim-FIM. In a similar scenario, the gem5 atomic achieves 12.52 MIPS, approximately 325 times faster. The OVPsim speed increases as the application grows due to the just-in-time engine algorithm, and thus benefiting from larger applications. For example, by comparing the larger and smaller applications, the simulation speed ranges from 1,190 to 3,910 MIPS (i.e., a 3.28 increase) where the gem5-FIM atomic difference is less than 20%, varying from 12 to 10 MIPS.

4.3 Soft Error Evaluation Considering Multicore Design Metrics/Decisions

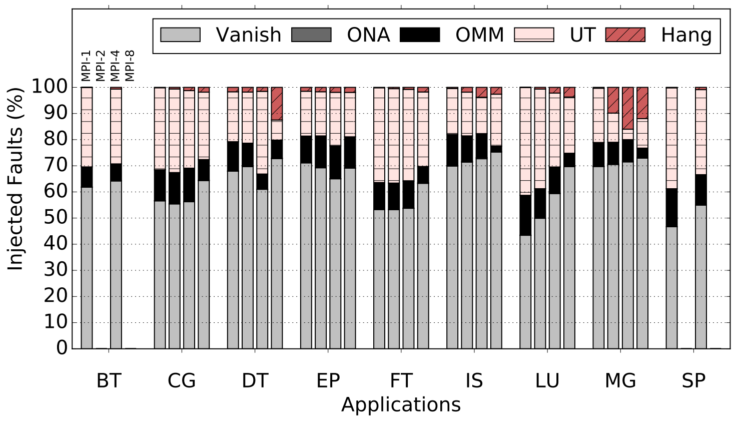
The emerging use of multicore processors requires specialized libraries, in this way, including an additional complexity to the system reliability assessment. Section 4.3.2 extensive explores the use of OpenMP and MPI-based applications reliability. Different ISAs are available during early design space explorations, for example, ARMv7 32 bist and ARMv8 64 bits. Thus, Section 4.3.1 investigates the impact of such distinct ISAs on the system behavior under fault injection is crucial because simulating it at lower levels is unfeasible.

4.3.1 ISA Reliability Assessment

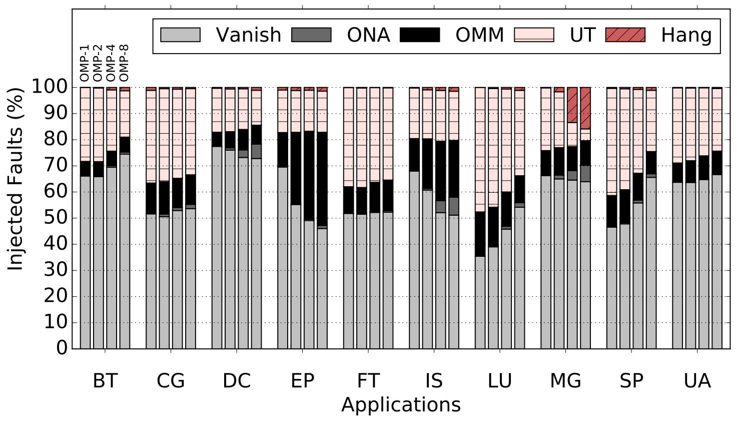
4.3.1.1 Execution Time and Workload

The ARMv7 workload for a single faultless execution has an instruction count that ranges from 299 million to 87 billion, with an average of 16 billion of instructions. In contrast, the 64-bit architecture applications execute in average 654 million instructions, varying from 41 million to 3 billion. Table 4.15 summarizes the workload regarding simulation time and the number of executed instructions with average, smaller, and larger

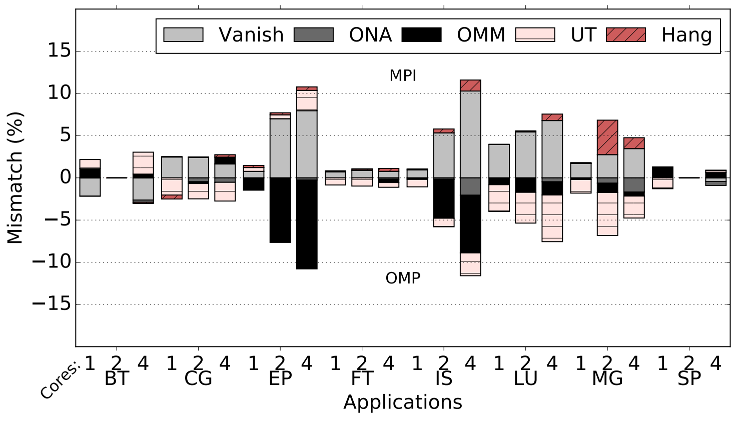
Figure 4.14: Fault injections using a multicore ARM Cortex-A72 processor (ARMv8).



(a) MPI benchmarks.



(b) OMP benchmarks.



(c) Mismatch.

cases.

Applications executed using the ARMv8 ISA present a significant performance improvement when compared to the ARMv7. In some cases, the speedup reaches up to 10 times. This performance gain can be pinpointed to the removal of several legacies features (e.g., fast and multilevel interruptions, conditional instructions) and to significant improvements made in the floating-point (FP) unit by adding new specialized instructions

Table 4.15: NPB workload summary.

Description		*Minimum*	*Average*	*Maximum*
Executed Instructions	ARMv8	41.1×10^6	654×10^6	3.08×10^9
	ARMv7	299×10^6	16.5×10^9	87.4×10^9
Simulation Time Single Run (sec.)	ARMv8	35	437	2,134
	ARMv7	163	7,929	42,763
Single Campaign Run (hours)	ARMv8	77	971	4,742
	ARMv7	363	17,620	95,028
Total Fault Campaign (hours)				
ARMv8	82,820		ARMv7	1,152,160

and increasing the number of FP registers. The ARMv7 often resorts to the ARM software FP library to perform some operations and thus increasing execution time. This choice was made automatically by the compiler. The evaluated workload employs HPC scientific applications with some of them heavily depending on FP computation, leading to a significant performance boost. The executed instruction count for each application where the average value reduces from 16 billion (ARMv7) to 654 million (ARMv8) instructions (Table 4.15). A shorter execution time improves the ARMv8 mean time between failures (MTBF) as it has a smaller probability of being stroke by a radiation event for a given particle fluence

4.3.1.2 Register File Size

The new 64-bit ISA also enlarges the integer register-file, from 16 to 33 registers, increasing the number of possible targets for fault injection by a factor of four. However, the compiler algorithm uses a reduced fraction of the available registers for load/store and control flow operations leaving other registers for global variables or unused. As in this experiment, each register suffers an identical number of fault injections, critical registers (e.g., program counter, stack pointer, those used on load/store and control flow operations) are less likely to face faults in the ARMv8 rather than in the ARMv7.

4.3.1.3 Branches and Function calls

The Hang error occurs when the target application control flow is severely affected by transient faults, in most cases, leaving the algorithm in an infinite loop. Analyzing in-

Table 4.16: Hang occurrence compared with the normalized function calls multiplied branches (F*B).

Scenario	*Parameter*	*Number of Cores*		
		Single	*Dual*	*Quad*
IS MPI V7	Hang (%)	0.413	0.625	3.000
	Branches	56.0×10^6	58.0×10^6	196×10^6
	F. Calls	22.6×10^6	23.1×10^6	26.9×10^6
	Index F*B	1.000	1.024	1.700
IS OMP V7	Hang (%)	0.288	0.313	0.400
	Branches	54.1×10^6	54.3×10^6	54.7×10^6
	F. Calls	21.7×10^6	21.7×10^6	21.7×10^6
	Index F*B	1.000	1.001	1.002
IS MPI V8	Hang (%)	0.438	1.850	3.800
	Branches	11.2×10^6	15.9×10^6	17.6×10^6
	F. Calls	2.85×10^6	3.35×10^6	4.84×10^6
	Index F*B	1.000	1.302	1.799
IS OMP V8	Hang (%)	0.225	0.925	1.175
	Branches	7.99×10^6	9.05×10^6	9.50×10^6
	F. Calls	1.81×10^6	2.05×10^6	2.06×10^6
	Index F*B	1.000	1.172	1.194

dividual parameters not always expose direct relationships between profiling data and fault injection campaigns. For example, the mean branch composition from the total executed instructions is 19.24% ($\sigma = 0.21$), 14.08% ($\sigma = 0.56$), 17.65% ($\sigma = 0.03$), and 12.01% ($\sigma = 0.36$) considering the four macro scenarios MPI V7, OpenMP V7, MPI V8, and OpenMP V8, where σ is the standard deviation. While the ARMv8 displays a 2% decrease in the mean branch occurrence compared with the 32-bit architecture, the application behavior under fault influence does not show any meaningful impact. Additionally, function calls variation also does not display any distinctive link with Hangs incidence. By combining both figures, nonetheless, is possible to uncover a correlation between this new index value (i.e., number of function calls times number of branches) with the Hang incidence after comparing the 130 scenarios. Table 4.16 exemplifies this behavior using the *IS* application as a case study, note that this new index value and the Hang percentage increases simultaneously, an observable behavior through several scenarios. The ARM ISA(i.e., ARMv7 and ARMv8) use distinct instructions to compare the conditional statement (e.g., cmp) and another to perform the control flow branching, while function calls use unconditional branches (e.g., jumps) in conjunction with argument registers.

Table 4.17: ARMv7 Memory transactions and soft error classification for selected scenarios.

	Scenario	*Vanish +OMM +ONA*	*UT*	*Mem. Inst. (%)*	$^{RD}/_{WR}$ *Ratio*
1	MG MPIx1	78	22	15.8	1.18
2	MG MPIx2	78	22	16.3	1.12
3	MG MPIx4	70	30	22.5	2.83
4	IS MPIx1	80	20	18.0	0.85
5	IS MPIx2	80	20	19.0	0.83
6	IS MPIx4	70	31	26.0	2.73

4.3.1.4 Memory Transactions

Table 4.18: ARMv8 Memory transactions and soft error classification for selected scenarios.

	Scenario	*Vanish +OMM +ONA*	*UT*	*Mem. Inst. (%)*	$^{RD}/_{WR}$ *Ratio*
A	LU OMPx1	57	48	29	1.9
B	LU OMPx2	59	45	27	1.9
C	LU OMPx4	67	40	22	1.9
D	SP OMPx1	57	42	35.1	1.5
E	SP OMPx2	59	40	34.0	1.5
F	SP OMPx4	70	32	28.5	1.5
G	FT MPIx1	62	37	25.7	1.00
H	FT MPIx2	62	37	24.6	0.95
I	FT MPIx4	62	36	23.7	0.95

UTs (i.e., unexpected terminations) originates from OS segmentation fault exceptions, which means that the program has attempted to access an area of memory outside its permissions. At instruction level, the address generation of memory access operations (e.g., load and stores) is compromised by transient faults in the source registers to lead to wrong address calculations. The reduced number of ARMv7 registers to perform address calculations leads to the use of load/store templates by the compiler to diminish the computational cost of register recycling. In other words, the ARMv7 compiler continuously utilizes the same register to perform memory transactions (e.g., R0–3 and SP). As consequence of this behavior, increasing the number load/store operations can lead to a more significant UT occurrence in the target application using an OS on top of the ARMv7 processor. Table 4.17 shows the soft error results (e.g., Vanish, UT, Hangs) alongside the memory access figures for some examples of the behavior mentioned above. By increasing the percentage of memory transactions (i.e., load and stores instructions) in applications such as MG and IS increases the UT ratio. For example, MG application memory-oriented operations for single and quad-core processors are 15%, and 22% while the UT occurrence increases from 22% to 30%. Further, increasing the core count alone does not reduce the UT percentage as is possible to note by comparing scenarios (1, Table 4.17) against (2) where both have similar memory instruction occurrence.

The 64-bit architecture exhibits a similar behavior considering FP memory transactions, supporting the claim above that wrong address calculation related to memory access, as FP instructions are exclusively used for computation and not for control flow operations (e.g., branches and jumps). Table 4.18 displays nine scenarios (A-I) of soft error analysis and FP memory figures. Reducing the memory transactions participation from the total number of executed instructions for LU (A-C) and SP (D-F) applications show a UT occurrence reduction trend. Scenarios (G-I) reinforce this hypothesis by demonstrating that a constant memory-oriented instruction incidence leads to a regular UT percentage.

4.3.2 Parallelization API

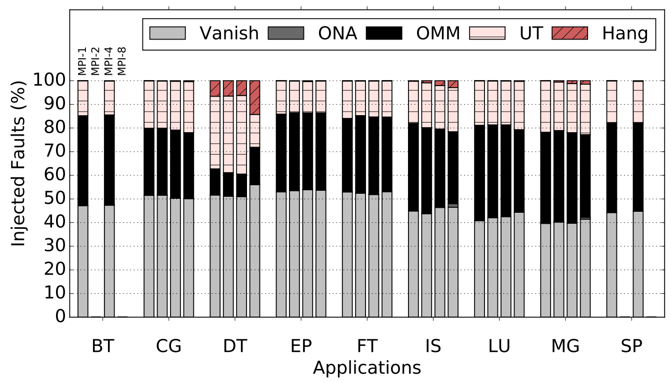
The OpenMP library uses a series of the fork and joins approach to parallelize loop statements (e.g., *for*, *while*) where the API automatically create children threads, being suitable for shared memory. In contrast, the MPI standard is adequate to distribute systems due to the use of a message-oriented parallelization technique, which requires the direct user parallelization regarding thread creation and communication. Figure 4.15 display fault injection campaigns and mismatches comparing the MPI and OpenMP applications.

4.3.2.1 Serial vs APIs

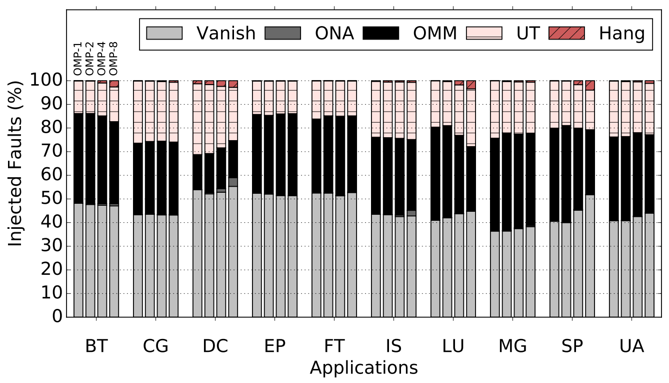
When we compare the serial implementation with either parallelization libraries on both architectures, some patterns can be observed. In ARMv7 MPI, only CG has a small improvement in the number of UTs, while in IS and MG the number of UTs and Hangs increases. Considering the OpenMP versions, no significant variation can

Figure 4.15: NPB fault campaigns using distinct parallelization paradigms.

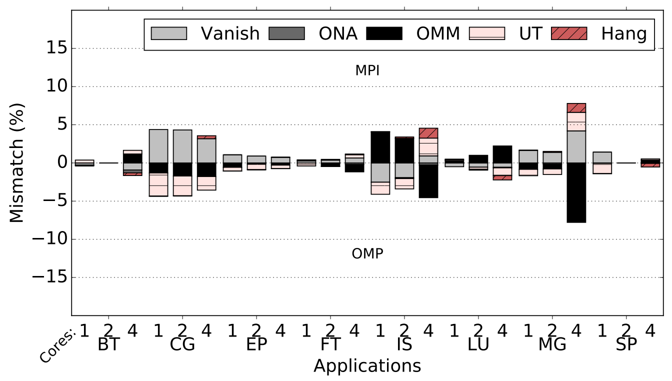
(a) gem5 atomic with MPI applications.



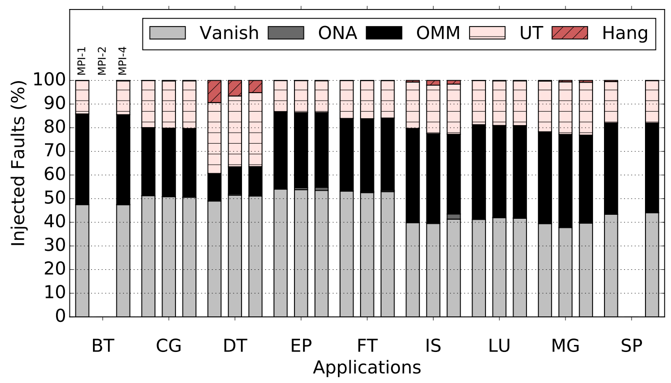
(b) gem5 atomic with OpenMP applications.



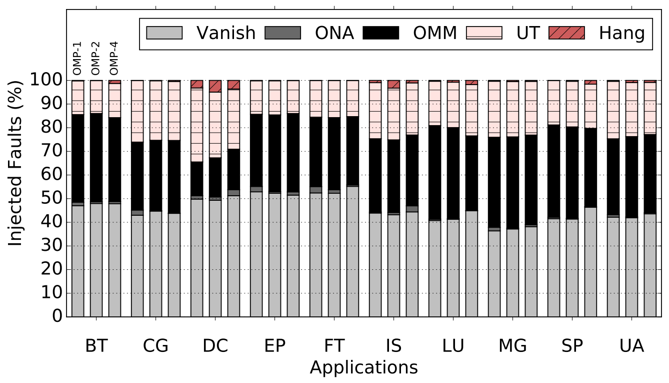
(c) gem5 atomic mismatch MPI vs OpenMP.



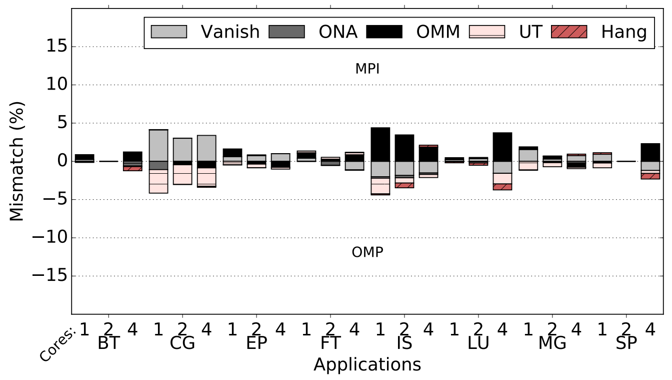
(d) OVPsim with MPI applications.



(e) OVPsim with OpenMP applications.



(f) OVPsim mismatch MPI vs OpenMP.



MPI BT and SP are not available for dual-core processors
Source: The Authors

be found. For the 64-bit application set, *CG, LU, MG, SP, and UA* the number of UTs diminishes. Further, CG application maintains the number of UTs when the number of cores increases. The same cannot be said about the other application, where the number of UTs diminishes with the increase in core count. Other applications have negligible variations.

4.3.2.2 Vulnerability Window

Within a software stack, some components are more critical to the system correct behavior. For example, targeting a thread scheduling function with faults has a potentially more hazardous effect on the system reliability than a purely arithmetic code portion. By comparing these critical functions active periods against application execution time, it is possible to define a time interval called *vulnerability window*, which varies with the number of calls and executions of the function. Using the NBP benchmark suite provides a real high-performance workload, enabling a more accurate evaluation of the OpenMP and MPI libraries impact on the system reliability. Due to its reduced vulnerability window, the parallelization mechanism has a limited effect on the final reliability assessment, less than 23% in the worst case.

From the 44 possible comparisons between the MPI and OpenMP scenarios, in 38 the MPI has a higher masking rate (i.e., executions without any errors) due to two main reasons: *First*, MPI applications have a better workload balance among the used cores, in other words, the number of executed instructions per core is very similar. For instance, the average difference concerning executed instructions per core is around 4% for both ARMv7 and ARMv8 considering MPI applications, while the OpenMP variation reaches up 16%. As the OpenMP does not fully utilize the available cores due to the fork/join parallelization approach where a loop statement executes in parallel and other code portions hastily. By contrast, the MPI has individual and independent working threads for each running core providing a better workload balance during its execution. Whenever a core is sub-utilized, it executes a thread scheduling policy and when no thread is suitable the core waits in a sleep mode. By consequence the kernel probability to suffer a transient fault increases, as the scheduling is more often executed. *Second*, OpenMP benchmarks have a smaller execution time, 16% on average, compared against the MPI applications. By consequence, diminishing the vulnerability window of the MPI inner-functions when comparing against the OpenMP. Further, the longer execution increases the chance of the injected fault being erased due to software and microarchitectural masking mechanisms.

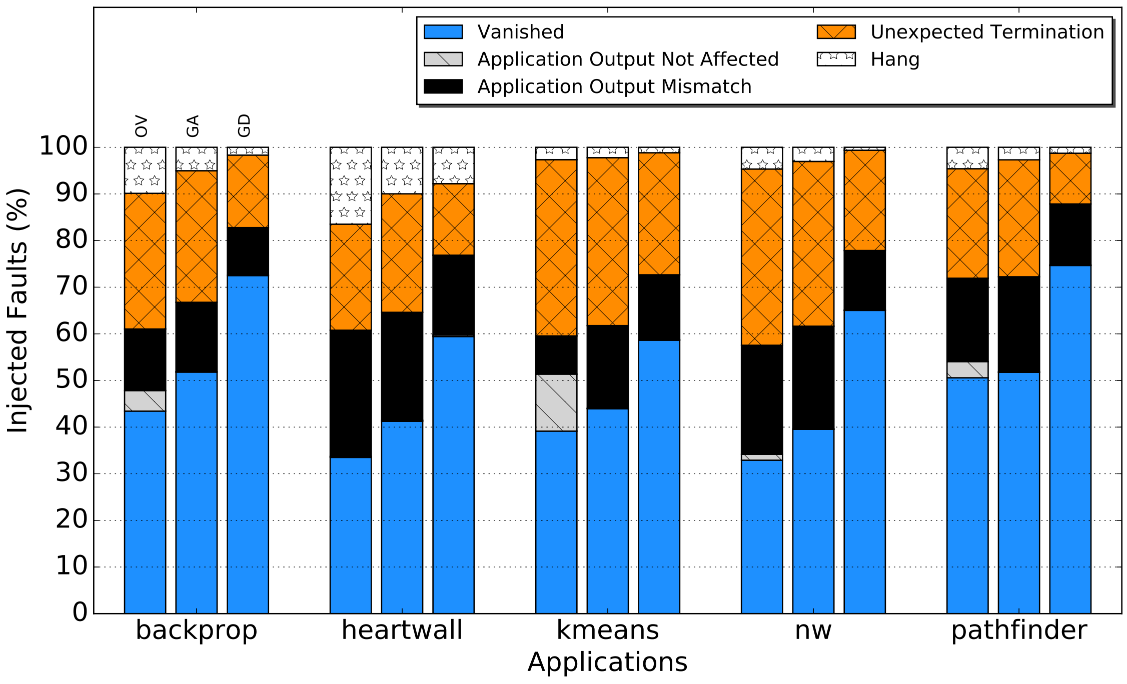
4.4 Focused Fault Injection Results

The traditional fault injection flow focuses on error rate estimation for a hypothetical radiation fluence through a probabilistic figure. Nevertheless, it does not stress all possible fault injection outcomes satisfactorily. In other words, it exposes the average error rate, ignoring that application domains have different reliability constraints. For example, considering the scientific applications, a silent memory corruption (SDC) denotes a higher reliability issue than an unexpected termination (UT). A UT is easily detectable, and the HPC environments already offer solutions as periodical checkpoints and restartable jobs to deal with this issue. In contrast, SDC detection or correction techniques are computational costly (DIDEHBAN; SHRIVASTAVA, 2016) and the incorrect detection may affect the final application result (BAUTISTA-GOMEZ et al., 2016). The incorrect SDC treatment can jeopardize the scientific investigation. In the automotive domain, an unexpected termination leads to harmful consequences (YOSHIDA, 2015), and in contrast, SDCs are mostly benign in a real-time application.

The set of experiments presented in this section aims to demonstrate the OVPsim-FIM extension usability during early design space explorations stages concerning the software reliability. The target architecture comprises an ARM Cortex-A9 interconnect through a bus on a dedicated one-gigabyte memory. The first experiment goal is to provide a reference for register targeting faults in comparison with the promoted novel fault injections techniques. It software stack included a Linux 3.13 kernel and selected Rodinia OpenMP applications (i.e., backprop, heartwall, kmeans, nw, and pathfinder) both compiled with the cross-compiler *arm-linux-gnueabi-gcc 6.2.0 20161005*. The following experiment, Figure 4.16, assess the selected benchmarks reliability under an eight thousand register-file fault injections aiming to estimate the percentage of errors that are not masked by each benchmark. Further, this setup includes three different VP-FIMs: the OVPsim-FIM (OV), gem5-FIM atomic (GA), and gem5-FIM detailed (GD). As shown in previous sections, the gem5 detailed presents more vanished faults due to the masking mechanisms from the microarchitecture representation.

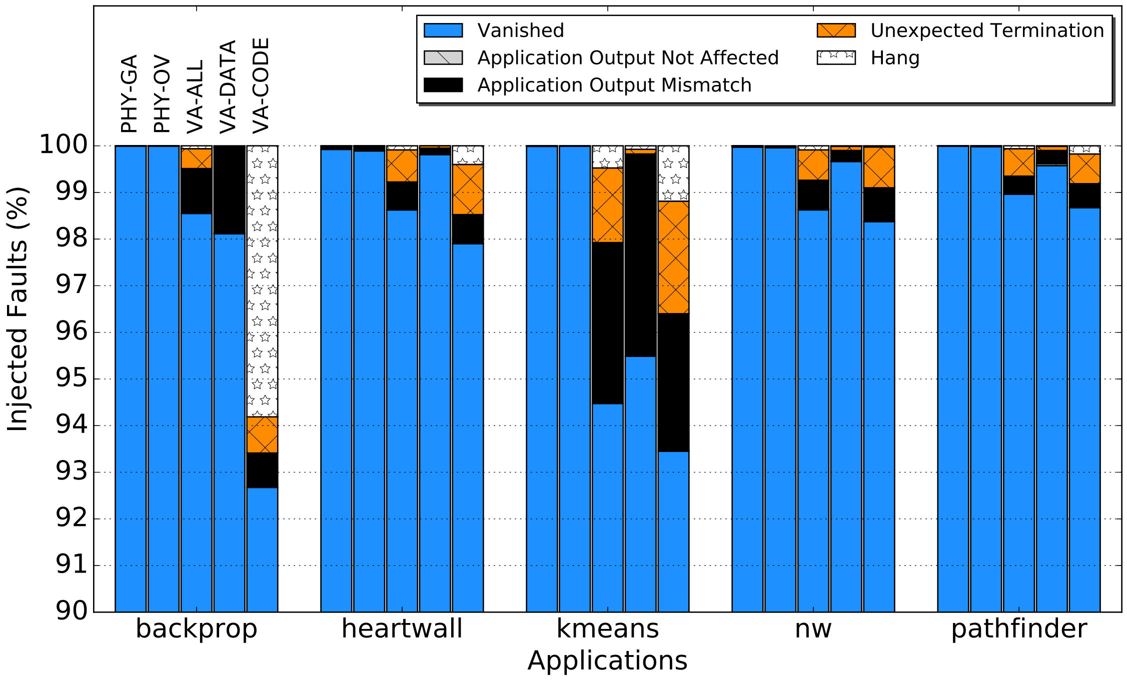
The second case study targets the application virtual memory primarily and to demonstrate the OVPsim-FIM extension necessity and to establish comparative results it also shows the outcome of a physical memory fault injection campaign. Figure 4.17 displays the selected Rodinia applications in five different fault injection scenarios: 80,000 physical memory faults considering the *(1)* gem5-FIM (PHY-GA) atomic and the *(2)*

Figure 4.16: Soft error classification according to the system behavior under register fault injection considering 5 Rodinia benchmarks executing onto single core ARM Cortex-A9, regarding the three fault injection module (FIM) implementations: gem5-FIM atomic (GA), gem5-FIM detailed (GD), and OVPsim-FIM (OV).



Source: The Authors

Figure 4.17: Memory-based fault injection campaigns targeting physical memory gem5-FIM atomic (PHY-GA) and OVPsim-FIM (PHY-OV), comparing with OVPsim-FIM extension VA the entire range (VA-ALL), the code section (VA-CODE), and data section (VA-DATA).



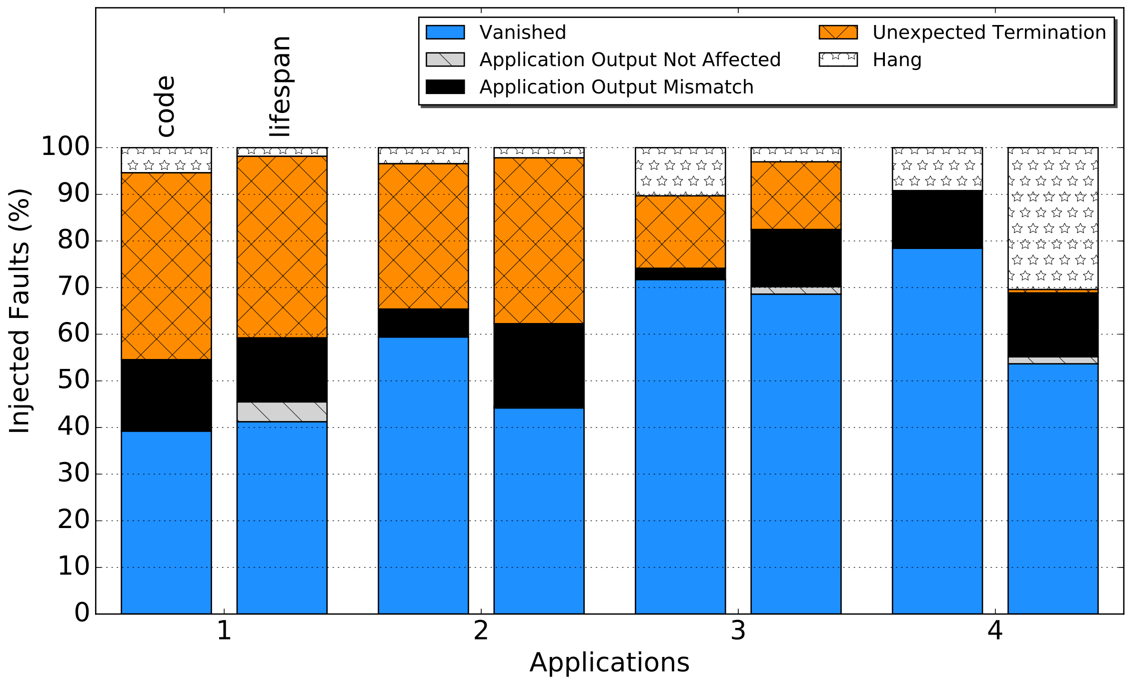
Source: The Authors

OVPsim-FIM (PHY-OV), and three 8,000 faults targeting the application *(3)* entire VA space (VA-ALL), *(4)* only data sections (VA-DATA), and *(5)* the code section (VA-CODE). Physical memory fault injection campaigns faithfully represent the memory behavior under the influence of soft errors the information extracted has little or none relevance. To achieve meaningfully statistical figures the experimental setup was lengthened by ten times when compared with the virtual memory experiments; where 99.9% of the total injected fault. This small physical memory scenario (i.e., 80,000 faults for five applications) requires approximately 4,400 simulation-hours, and after this long simulation, only 200 from 80,000 fault injections do not present a masked outcome in average. The one-gigabyte memory has 137,438,953,472 available bits multiplied by the execution time in which each intersection is a possible target. At any time, the application accesses a limited memory range (i.e., few kilobytes) during its execution, even the data-intensive algorithms process small memory blocks per access. Consequently, the chance of physical memory bit-flip effect the application behavior is minimal, also, this event timing should be precise and occur before reading access.

The virtual address fault injection technique reduces the target range, thus focusing on the application code and data. Faults targeting the code segment exclusively (i.e., VA-CODE) exhibits a larger occurrence of control flow related errors (i.e., UT and Hangs) due to the instruction representation changes. As a possible utilization of this technique, several mitigation techniques replicate or introduce a new instruction in the original application code after assuming a fault-free code. With this promoted fault injection technique is possible to explore the SWIFT-based mitigation methods and improve its coverage by adding a new approach angle. In contrast, the data section faults incur in memory corruptions, and reduced number of control flow errors as the applications data structures are affected. Even when reducing the memory range from the full memory to the virtual memory range a significant portion of the fault injections still leads to a vanished outcome. The compiler includes hundreds of functions and data structures from adjacent Linux and C libraries alongside the own application source. Some of them only run during the application startup, and the majority never execute during the simulation and consequently, a large part of the virtual memory, either code or data, does not have any influence over the application behavior. This technique may be used to target a particular process (i.e., an application with a distinct virtual address space) in a sophisticated software stack for example.

The fault injection scope is further reduced in the third case study by using the

Figure 4.18: Function-based fault injection techniques targeting four different functions: *bpnn_adjust_weights._omp_fn.1, bpnn_layerforward, gomp_barrier_wait_end, and pick_next_task_fair.*



Source: The Authors

function code and lifespan techniques which use targets four backprop functions:

(i) The most timing consuming function (bpnn_adjust_weights._omp_fn.1), this function represents the application kernel OpenMP parallelization;

(ii) A sequential portion of the application (bpnn_layerforward);

(iii) The OpenMP synchronization barrier (gomp_barrier_wait_end);

(iv) The Linux kernel next process selection algorithm (pick_next_task_fair).

Figure 4.18 shows the four 8000-fault campaigns for both function code and lifespan techniques. The lifespan technique targets the 16 general purpose registers (r0-r15) including the program counter (PC) and stack pointer (SP) assigning each one 500 faults from the total 8,000. The most time-consuming function bpnn_adjust_weights._omp_fn.1 is the application kernel parallel phase and data intensive. This function behavior under both fault techniques (lifespan and instruction code) resembles the compartment when targeting the complete application as it accounts for approximately 20% of the total execution time. The sequential function (2) has a smaller number of hangs due to it execution being limited to the application initialization, as the subsequent code execution masks most of the faults.

The OpenMP gomp_barrier_wait_end is a short (i.e., few lines) function that acts as a thread synchronization barrier. Therefore, the code and register fault injections have a similar behavior causing UT (i.e., Linux segmentation) errors due to the wrong address calculation. Code faults show a more significant number of hangs in the first three cases, as the small function code increases the probability of a control flow error leading to an infinite loop. The registers are continuously overwritten at each function invocation and in contrast, code faults remain incorrect for more extended periods. The Linux kernel function (4) has a complex chain of control flow statements (i.e., if and else). Whenever a fault strikes an addressing register (i.e., r3, r4, and r5 for this particular function), the control flow is severely affected leading to kernel malfunction and consequently, infinite loops (Hang) and therefore other registers, beside the PC and SP, have little or none impact. In several cases, it causes an SDC error as the target register remains in the function context stored in the stack. In contrast, the code targeting in this particular function show as smaller impact, as large portions of the function are not executed due the control flow statements.

4.4.1 Case Study

To demonstrate the soft error analysis capabilities of proposed fault injection techniques, we select a matrix multiplication (MM) kernel as a case study due to its application in several fields and branches of science. During this case study, we will subject the MM to the fault injection techniques presented in Section 3.8, targeting distinct software components alongside a customized error classification module. Each technique covers different aspects of the MM considering its variables and critical functions in an isolated manner, demonstrating the importance of providing software engines with appropriate means that can lead to application reliability improvements. Experimental setup comprises an ARM Cortex-A9 processor, executing Linux kernel (3.13) and the MM kernel using 300-wide 32 bits integer square matrices as inputs and output.

A. **Sequential and Parallel MM**

The first fault injection campaign deploys the MM kernel in two versions: (i) sequential implementation that uses a simple iteration to perform the multiplication, and (ii) a parallel MM using the Pthreads library to create two working threads. Figure 4.19 shows the two MM implementations subjected to 8 fault campaigns of 8,000 fault injections each

Table 4.19: Deployed fault injection techniques into the TMR-Based Matrix Multiplication reliability exploration.

#	*Target*	#	*Target*
A	Register file	E1	Function Code 1st replication
B	Physical Memory	E2	Function Code 2nd replication
C1	VA Complete	E3	Function Code 3rd replication
C2	VA Code Section	E4	Function Code Voter
C3	VA Data sections	F1	Function Lifespan 1st replication
D1	Variable Matrix 1	F2	Function Lifespan 2nd replication
D2	Variable Matrix 2	F3	Function Lifespan 3rd replication
D3	Variable Matrix 3	F4	Function Lifespan Voter
D4	Variable Matrix Result		

Source: The Authors

(i.e., totaling 128,00 simulations), considering the six fault injection techniques:

(i) random registers (A),

(ii) physical memory (B),

(iii) virtual memory (VM) entire range (C1),

(iv) VM code section (C2),

(v) VM data sections (C3),

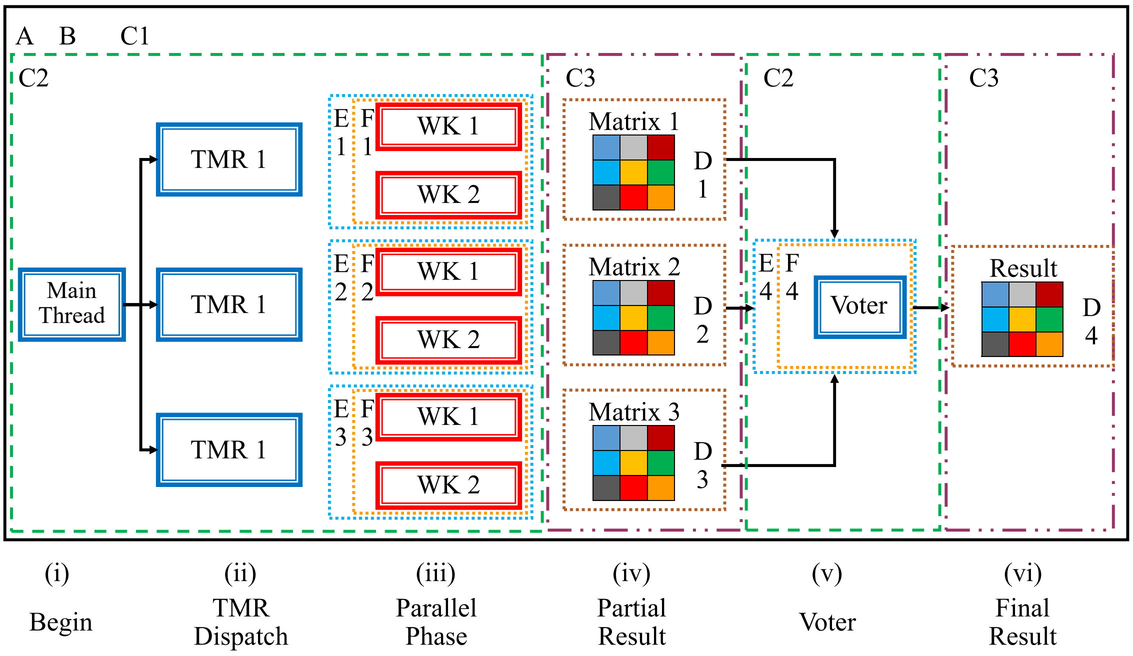
(vi) result matrix (D1),

(vii) multiplication function object code (E1),

(viii) multiplication function lifespan (F1).

The sequential MM presents a larger OMM (e.g., silent data corruption) due to the high number of memory reads and writes with a small number of operations for each matrix cell. Resulting scenario leads to a significant number of silent data corruptions in the target system memory as consequence of dirt registers used for the MM. Underlying implementations are susceptive to UTs (i.e., unexpected terminations) due to the incorrect memory address computation caused by registers under fault influence, which may lead to errors such as *segmentation fault*. However, the parallel MM presents a substantially more significant number of UT when compared to the sequential version. This behavior is explained because Pthreads scheduling algorithm increases the application control

Figure 4.19: TMR-Based Matrix Multiplication execution flow and fault target locations according Table 4.19. WK refers to working thread.



Source: The Authors

flow complexity, which might incur in more wrong address computation during the MM execution.

4.4.1.1 Triple Modular Redundancy

As identified in the previous Section, the injection of bit-flips severely impacts the matrix multiplication kernel operation due to its simplicity and small code. Aiming to reduce the amount of detected silent data corruptions, we propose the use of the MM-TMR to improve its reliability by using a well-known fault mitigation technique the *Triple Modular Redundancy* (TMR). This technique adopts three independent *parallel MM* instances (i.e., six working threads in parallel) enabling one incorrect execution to be masked by a voting process at the end of MM execution (i.e., a function vote the majority from the partial results). The TMR version corrects most of the errors originated from the input and output matrices. Nevertheless, the TMR implementation using the Pthread library increases the occurrence of UT.

To improve the experiment, we included a custom soft error analysis step, which compares the four matrices (i.e., each TMR replica and the voter) alongside two additional error classifications considering three possible outcomes:

(i) All matrices are identical, in this case, the SOFIA classifies the error according to one of the five default classes (e.g., Vanish, UT, Hang). Note that OMM and ONA only occur if the result matrix is correct, and thus being considered benign errors in this context.

(ii) If one TMR matrix does not match the other replicas, the voter will mask the error and produce the correct result. Nevertheless, in this case, the simulation diverges concerning the number of executed instructions from the faultless run, which leads to a false-positive error (i.e., control flow error with incorrect memory) in traditional fault injection flows. The SOFIA classifies this execution context as **Corrected** to signal the appropriate behavior, (i.e., even with the context mismatching the reference execution of the final matrix is correct.

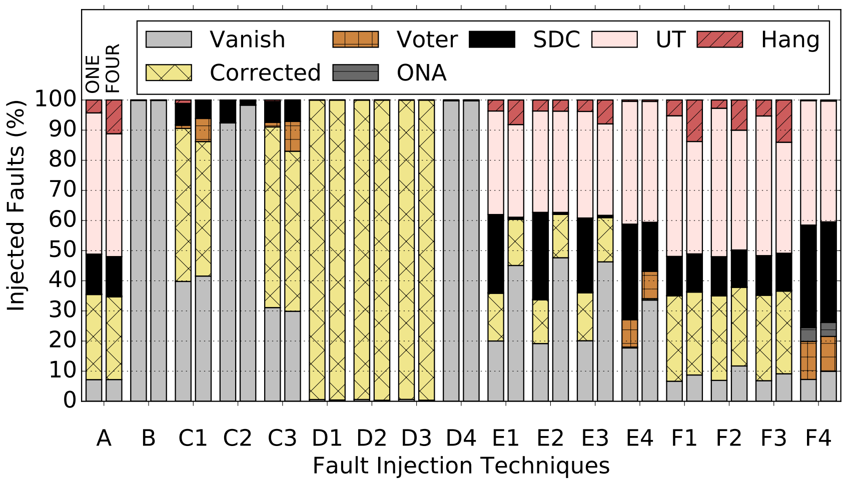
(iii) The third possible outcome originates from an incorrect voter execution (i.e., the three TMR matrices are identical and differ from the voter matrix) due to the fault injection being classified as **Voter Error**.

Table 4.19 describes 17 distinct fault injection scenarios targeting the MM-TMR, while Figure 4.20a shows the results considering a single and quad-core ARM Cortex-A9 processor where each fault injection scenario comprises 8,000 faults. Register-based fault injection (A, E, and F) displays a considerable amount of UT (i.e., Linux OS segmentation faults in this context), around 40% due to the wrong address computation using registers under fault influence. In contrast, the memory-based technique errors depend on the stroke region, for example, targeting the 1 Gb physical memory (using technique B) would result to a minimal number of errors (i.e., masking rate of 99.95%) as the benchmark accesses a limited memory range (i.e., few dozen kilobytes). The complete VM range (C1) and data sections (C3) present a similar behavior as most of the faults hit the application 300-wide square matrices due to its size (i.e., each one possessing 360 kilobytes or 20% of application size). The code section (C2) contains, besides the application code, hundreds of Linux and C libraries unused functions added by the compiler, leading to higher masking rate. By individually targeting the matrix replicas (D1-3) we exercise the TMR main functionality resulting in an almost complete error coverage. The fault campaign D4 leads to a 99.9% masking rate as the final result is composed of the voter function at the application end, which incurs in a narrow sensitive window (i.e., any faults previously present in this matrix are overwritten).

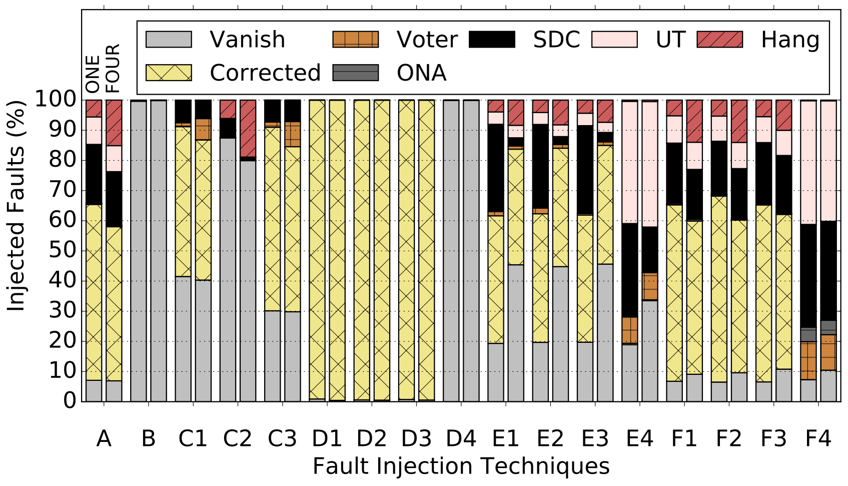
Passing the focus to function criticality, groups E (E1-4) targets the function object

Figure 4.20: MM soft error vulnerability analysis considering a single and quad-core ARM Cortex-A9 processor.

(a) MM-TMR



(b) Improved MM-TMR



Source: The Authors

code while the F (F1-4) injects bit-flips into the processor register-file during the target function execution (i.e., the function lifespan). The matrix multiplication kernel has a compact code of few dozens of lines executing for extended periods, which composes most of the application execution time. Single and quad-core processors show a similar rate of correct results (i.e., vanish, SDC, and corrected) when targeting the function object code (E1-4). However, their composition diverges while the single-core processor presents a more significant SDC rate (i.e., the MM result is correct with silent data corruptions on the memory) the multicore system displays a larger masking rate. Further, the multicore system reveals a higher number of *Hangs* due to the longer and more significant executions of the PTHREAD scheduling policy leading to unrecoverable control flow. Random register (A) and Lifespan (F1-3) techniques show similar behaviors under fault injection as the MM application spends 95% on those multiplication functions. Directly targeting the voter function show a behavior not seen when targeting the complete application with random faults due to its short execution time, and thus, demonstrating the necessity of more detailed fault injection framework. Subjecting the voter code (E4) and lifespan (F4) to fault injection causes an erroneous matrix voting, which is a severe error in this context.

4.4.1.2 Improving the Triple Modular Redundancy

The initial MM-TMR solution provides complete coverage to fault injections for the replicated data (i.e., the partial matrices) while the control flows still prone to unexpected terminations. By using the promoted framework, it is possible to pinpoint the major UT cause as OS segmentation faults in one of the thread replicas that terminates the complete application even if the other replicas had not experienced any errors. To mitigate this issue, we modified the application algorithm to include a segmentation handler for each replication, and consequently, the improved MM-TMR (MM-TMR-I) finishes correctly even if one of the replicas generates an OS segmentation fault. The experiments displayed in Figure 4.20b reproduce the 17 fault injection scenarios mentioned above for the MM-TMR-I version using the single and quad-core processors. The MM-TMR-I improves the MM kernel reliability by achieving of up to 90% of coverage (i.e., with correct final results) in contrast to the 50% of the traditional TMR considering register-based fault injections targeting the replicas working threads. Fault injection techniques (D4, E4, and F4) targeting the voter function and data remains unchanged without any modification being made in its code.

4.5 Closing Remarks

This chapter presented three additional contributions contribution thesis. First, Section 4.2 compares the accuracy between instruction-accurate (i.e., OVPsim) and cycle-accurate (i.e., gem5) virtual platforms. Further, Section 4.2.2 explores the relationship between the instruction-accurate engine and mismatch with the gem5. Performance is fundamental to early design space explorations, for this purpose this work shows the OVPsim-FIM peak of simulation speed around 4,000 MIPS considering a quad-core host processor, Section 4.2.3. The OVPsim improves the simulation performance with larger workloads in the range of hundreds of billions of instructions, in contrast, both gem5-FIM atomic and detailed do not show any considerable speed variation by varying the workload size.

The second portion demonstrates the promoted fault injection framework to evaluate distinct design decisions during the initial development phase. This work explores the novel 64 bits ARM architecture and assesses its reliability under a soft error in Section 4.3.1. The larger register file from the new ISA increases the masking rate as the application fails in taking advantage of the extra registers. Further, this work (Section 4.4) investigates the impact of parallelization APIs in the overall software stack reliability by comparing serial, OpenMP, and MPI implementations of the same benchmarks. Fault campaigns show a smaller incidence of the parallelization API in the overall system reliability due to the limited time ratio of those libraries in comparison with the total execution time.

The third contribution involves the validation/use of proposed novel fault injections techniques and error analysis methodologies. In this Section 4.4 we demonstrate the adaptability to different aspects of the fault injections campaigns scenarios. With the proposed framework, software engineers can focus the resiliency exploration to specific software stack components.

5 MACHINE LEARNING APPLIED TO SOFT ERROR ASSESSMENT IN MULTICORE SYSTEMS

Since the computer science origins, researchers are investigating different manners to program computers. Usually, by defining the system behavior under any given inputs and current state (i.e., rule-based programming) as a set of predefined rules. In this way, the system correct execution depends on its designer ability to transfer the desired behavior into a group of rules. This technique has several limitations: (i) unforeseen events lead to unexpected behaviors; (ii) new requirements can be not quickly introduced in the system description. Searching for another approach computer scientists started investigating how to give computers the ability to learn without being explicitly programmed (SAMUEL, 1959). More precisely, machine learning (ML) is the field of study that uses the target system past (knowledge) to predict its future behavior (intelligence). In other words, machine learning is an algorithm or model capable of learning from a collection of inputs without requiring rules-based programming (i.e., if-else)(AKERKAR; SAJJA, 2016; UNPINGCO, 2016) by transforming this information into actions or predictions (UNPINGCO, 2016).

Machine learning provides a robust method to solve a broad range of complex problems, such as weather forecast or oil exploration. ML has already several applications in our daily life already: For example, a video-on-demand distribution algorithm such as the employed by Netflix, Amazon, or Youtube can recommend for a user new films and series based on its past interests. Oncologists are using ML algorithms to uncover early cancer symptoms by analysis millions of breast image reviews (KOUROU et al., 2015). In this case, by reviewing an extensive set of images, the ML algorithm is capable of finding complex patterns, which otherwise would be impossible to achieve by humans. Machine learning algorithms can be categorized into three broad groups:

- **Supervised Learning** goal is to predict the effect of one set of observations (i.e., input, features, attributes) has on another dependent variable (i.e., output, label, class). In other words, supervised learning algorithms make predictions based on a set of training examples using two main approaches: A regression technique models the target system using mathematical equations producing a continuous value output (e.g., linear regression) to approximate the target system behavior. Classification algorithms divide a data set into smaller subsets by evaluating its attributes (e.g., decision tree).

- **Unsupervised Learning** searches for patterns on unlabeled datasets (i.e., without a dependent variable output). While supervised learning correlates two groups of observations, unsupervised learning describes the system features into more abstract levels of representations. For example, K-means is a well-known example of an unsupervised learning algorithm to subdivide *n* observations into *k* clusters where each observation belongs to the cluster with the nearest mean.

- **Reinforced Learning** gains experience (i.e., knowledge) through a series of trial-and-error training sessions where a cost function calculates reward or punishment value depending on its prediction. By minimizing the target cost function, the reinforced learning algorithm improves the system prediction until reaching a pre-defined quality threshold.

5.1 Machine Learning Applied to System Reliability

Machine Learning has been employed in different domains in recent years to recognize patterns and predict the future system behavior. For example, (HASHIMOTO; LIAO; HIROKAWA, 2017) trains a random forest ML algorithm with multiple TCAD simulations to estimate the SER of an SRAM Cells. Giurgiu *et al.* (GIURGIU et al., 2017) uses field information as a training set for a random forest algorithm to predict a DRAM number of errors. Considering the challenges involved in soft error assessment, some researchers are studying the applicability of ML techniques to speed-up its simulation, prediction, or mitigation. Table 5.1 shows a review of the state-of-the-art on soft error investigation on multi/manycore system using machine learning approaches.

Vishnu *et al.* (VISHNU et al., 2016) analysis the impact of multi-bit memory errors targeting both permanent and transient faults on large scale applications. The training sets consist of fault injections targeting the applications primary data structures. This work tests eight different ML algorithms and compares their predictions (i.e., the application error probability) with the ground-truth (fault injection). This paper shows an average error detection of 90% considering three applications under the influence of permanent and transient faults. The fault injection and analysis are tightly coupled with the application, requiring an excellent understanding of the application execution.

The work (SUBASI et al., 2017) proposes an online adaptive SDC detection algorithm using machine learning. First, it investigates different supervised ML algorithm

Table 5.1: Most recognizable virtual platform fault injection simulators.

Year	*Author*	*Machine Learning*	*Features*
2015	(ASHRAF et al., 2015)	Supervised Learning	Linear Regression
2016	(DITOMASO et al., 2016)	Supervised Learning	Decision Tree
2016	(VISHNU et al., 2016)	Supervised and Unsupervised Learning	SVM, k-nearest Neighbors, Decision Tree, and other four algorithms
2017	(SUBASI et al., 2017)	Supervised Learning	Linear Regression, SVM, k-nearest Neighbors, Decision Tree, and Ada boost regressor
2017	(NIE et al., 2017)	Supervised Learning	Artificial Neural Network

Source: The Authors

execution time to select the best suitable for an online detector. The proposed SDC detector runs after each application iteration, applying one ML algorithm (from five), predicting the error impact on the system. The system has a high percentage of correct predictions, 99%, over eleven HPC applications.

Nie *et al.* (NIE et al., 2017) collects several parameters (e.g., power, GPU error logs, and temperature) of a supercomputer during four months. This work correlates these pieces of information for each node searching for error patterns hidden in this big data problem. After an initial statistical analysis, this work employs an artificial neural network to predict the GPU error probability in different conditions.

(ASHRAF et al., 2015) studies the propagation of transient errors on large-scale MPI application (i.e., up to 1000 cores). This work introduces additional instructions in the application code using the LLVM Intermediate Representation (IR) to inject faults and check for errors. Further, this instrumentation tracks the "error" through several MPI process by monitoring communication messages, load-store operations, and function calls. With this information, the authors create fault propagation models to estimate the number of corrupted memory locations.

(DITOMASO et al., 2016) measures several NoC parameters such as temperature and wearing to feed an error prediction model called VARIUS. The VARIUS provides the error probability for different types of faults depending on the input parameters. All this information is supplied to a decision tree which predicts on-the-fly if a specific link will suffer *No Error*, *Few Errors*, or *Many Errors* before any package transmission. According to the model output, the proposed system adjusts the packet mitigation level using CRC. SECDED, or retransmission. This mitigation technique requires a two CRC, four

SECDED, the decision tree, and control modules in every system router.

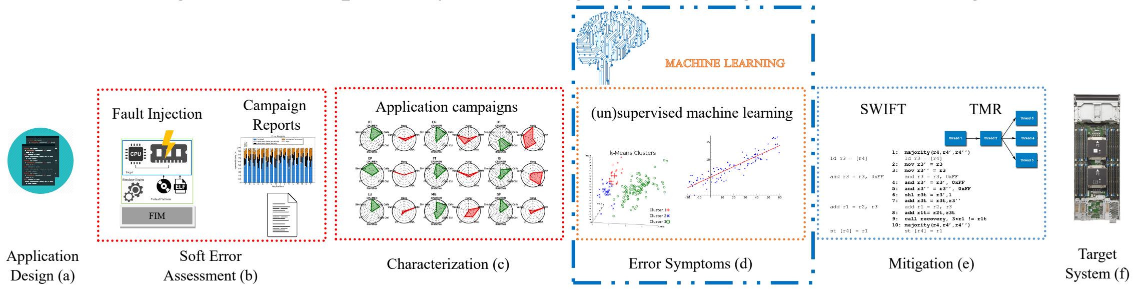
5.2 Problem Description

Soft errors reliability assessment is a time-consuming process which requires extensive fault injection campaigns, in some cases, taking thousands of simulation hours. Accelerating assessment of soft errors impact on the system design during early design space explorations (DSE), particularity, in the software stack, is one these Thesis goals. For this purpose, it was two virtual platforms to provide a flexible and fast fault injection framework, adequate for such explorations. Nevertheless, even virtual platforms cannot provide adequate fault coverage when executing more realistic workloads. For example, this work longer simulation (i.e., EP) takes 12h hours of simulation for a single run to execute only 50 seconds of the real system time. Note that this simulation targets a single-core ARMv7 processor, memory, and cache using the gem5 atomic mode. During early DSEs, the target application undergoes several cycles of reliability optimizations to meet the desired system constraints. How can we reduce the number of fault injections (i.e., computational cost) and aid the software engineer to improve the system reliability?

5.3 Proposed Solution

Machine learning techniques are being employed to predict and model systems behavior considering multi-parameter optimizations. However, the reviewed works target distribute HPC applications using the ML approach to overcome the inherent overhead from such scenarios simulation. This work proposes the utilization of multiple machine learning techniques to improve multicore systems software stack design during early explorations. Figure 5.1 shows the traditional application development cycle without any modifications (a,b,d and e) and the proposed solution. The software engineer describes the application and system in the first step (a), e.g., architecture, compiler, libraries, # of cores. The second phase (Figure 5.1b) display the traditional soft error vulnerability compressing a random fault injection campaign. Any fault injection flow can be adapted in the promoted solution, such as the ones based on FPGAs, VHDL simulation, and virtual platforms. For this work, we explore cycle and instruction-accurate virtual platforms in this role. Using this information, the software engineer recommends a mitigation technique or application modification (Figure 5.1e) according to the target system (f) requirements.

Figure 5.1: Proposed system design cycle using machine learning.



Source: The Authors

The proposed solution adds two new development phases in the original flow (Figure 5.1a,b,e,f): Characterization (c) and error symptoms detection (d). The characterization phase facilitates the access to raw gem5 microarchitectural parameters for an initial exploration. Software engineers can interactively search for parameters of interest (e.g., # of loads and stores) and compare with previous iterations or workloads. The application characteristics vary during the development cycle by algorithms or system modifications. This phase enables the user to evaluate the software adjustments impact on the system reliability over multiple iterations.

This work proposes an exploratory flow (Figure 5.1d) to find the soft error correlations with multiple application characteristics. This flow applies supervised and unsupervised machine learning techniques to investigate the correlations between the fault injection results (i.e., vanish, hang, ONA, OMM, UT) and the application characteristics (e.g., cache statics, # of branches) without the presence of faults. This phase goal is to reduce soft error assessment time during early design stages by examining the impact of these characteristics on the application reliability. This work proposes a fully automated and standalone tool capable of searching and identifying the individual (or combinations of) parameters which present the most substantial relationship with each detected soft error. Initially, the investigation considers the microarchitectural information provided by the gem5 simulator (e.g., memory usage, application instruction composition). This tool collects over one million single fault injection raw outcomes alongside gem5 simulation reports to create statistical figures (e.g., the percentage of Vanish, Hangs) for each one of the explored scenarios. In possession of this database, the proposed framework applies supervised and unsupervised learning techniques to produce a model of each feature impact on the system reliability. With this observations, the software engineer may choose to alter the application or optimize the system parameters.

5.4 The Promoted ML Investigation Tool

This section provides an overview of multiple ML-related techniques developed in the promoted tool. The tool provides a generic exploration framework using machine learning supervised and unsupervised techniques to highlight the most relevant features. Initially, this framework combines two data sources: the gem5 microarchitectural statistics and the soft error vulnerability. However, its interface enables adoption of other information sources according to the investigation goals and direction. Further, exploration flows and other techniques can be created or modified because of the tool data structure parametrization on intermediate steps.

The proposed framework was developed using Python, taking advantage of available ML frameworks, in particular, the Scikit-learn module. Why Scikit-learn? Since its release in 2007, Scikit-learn (PEDREGOSA et al., 2011) has become the most widely-used, general-purpose, open-source machine learning modules that are popular in both industry and academia. Further, this work adopts the pandas (MCKINNEY, 2011) module which provides a data structure designed for large-scale explorations. Besides this two modules, several other libraries where employed, such as matplotlib, numpy, scipy, multiprocessing. Sections 5.4.1 to 5.4.3 display multiple techniques to acquire and process the features embedded in the proposed tool.

5.4.1 Feature Acquisition

Feature acquisition comprises the data preparation phase of any ML framework, where the relevant information should be extracted from the raw input files. The promoted framework collects features from different sources requiring precise knowledge of the file format. The tool splits each file line into sub-strings considering the format parameters arrangement, dropping unnecessary information to reduce the memory usage.

5.4.1.1 Pandas DataFrames

Machine learning algorithms demand large amounts of raw data requiring an optimize storage method to reduce the memory utilization and increase its execution performance. Pandas is a popular python module which provides robust, expressive, and flexible data structures for data science applications. For 2D problems, the Pandas supports a rectangular grid data structure called dataframes (DFs). Unlike a matrix, it supports hierarchical multi-level index enabling sophisticated data analysis and manipulation. Further, each cell holds an object pointer enabling the combination of different data types numeric, character, logical in the same structure. Dataframes can be merged, concatenated, divided, and manipulated in multiple ways facilitating the access to data subsets data.

The raw input is transferred to a single dataframe where columns represent features (i.e., soft errors and microarchitectural) while rows represent the fault injection scenarios. This dataframe has several missing cells (represented as NaN) because not all scenarios have the same microarchitectural elements (e.g., number of cores). The missing data are replaced by zeros when necessary to enable the correct algorithm execution. Another dataframe functionality is the possibility to export and import data structures from *comma-separated values* (CSV) files. The promoted framework enables pre-processing or fast-forwarding some exploration steps, reducing the investigation time.

5.4.2 Feature Transformation

The features original representation may be not suitable to be used as input in the ML algorithms. For example, some algorithm requires float-point or integer values or the target information may be scattered across several columns. In this context, feature transformation is a set of techniques which creates new features using the already existing ones.

5.4.2.1 Rescaling

Some machine learning techniques perform complex equations using high-dimension equations where the data magnitude impacts on the algorithm performance. For example, a nominal value of 1000 performs worst than the value 10. For this purpose, this work adopts a Min-Max Scaler method that transforms features by re-scaling the features. In this work, we rescale all features to a range between 0 and 10, which yield the best performance. This range improves the average performance of the scikit-learn methods due to the smaller range from 0 to 10 instead of 0 to billions.

5.4.2.2 Normalization

The normalization function is another popular resize adopted method which scales the input vector individually to the unit norm (vector length) or another value. Each resizing function impacts the target data differently (e.g., flatting the values), the proposed tool support distinct methods according to the target technique.

5.4.2.3 Merging Similar Features

Multiple features may contain similar information, and merging them is possible to highlight this behavior. For instance, the gem5 provides microarchitectural information subdivided by core, and consequently, the number of parameters depends on the target architecture. This feature transformation merges 'similar' gem5 parameters to get a statistical figure representing the entire processor which can have multiple cores. For example, merging the *number of branches* for each core creates a new feature describing the processor behavior.

5.4.2.4 Feature Combination

The feature transformation goal is to increase the data information gain by highlighting second-order correlations between features. By multiplying or adding arbitrary features, it is possible to emphasize any hidden relationships between variables. This function multiplies or adds two features (columns) from a dataframe using all possible column combinations. The generated columns are concatenated with the original DF.

5.4.3 Feature Selection

Selecting the relevant feature subset is crucial in any machine learning algorithms. Reducing the dataset improves the problem readability, shortens the training time, reduces the exploration dimensional space, and decreases the overfitting.

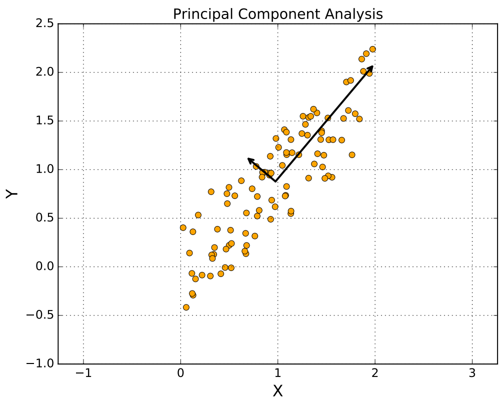
5.4.3.1 Variance Threshold

This technique removes features with low variability from the dataset (e.g., removing the features constant values). Further, the variance threshold method targets only the features (i.e., microarchitectural values) and not the labels (i.e., soft error analysis). If the feature has a lower variation, usually a constant value, it carries a reduced amount of information and can be removed.

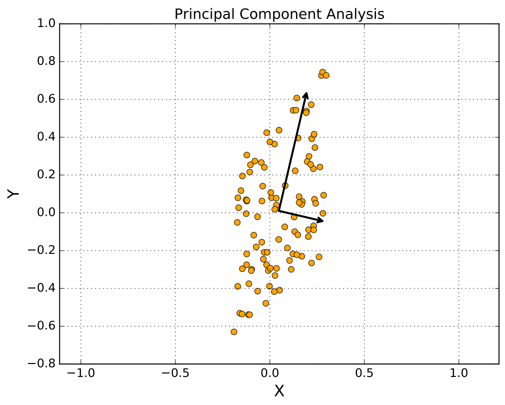
5.4.3.2 Principal Component Analysis

Figure 5.2: Principal Component Analysis.

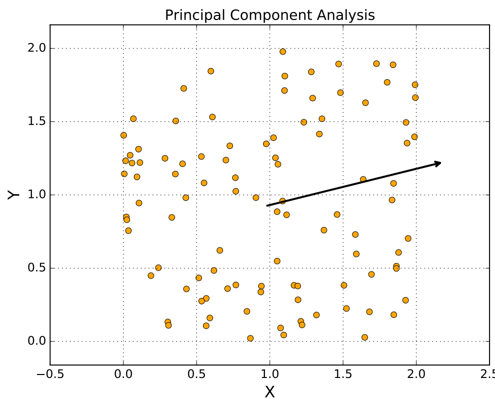
(a) Linearly Dependent Data.



(b) Linearly Dependent Data.



(c) Random Data.



Source: The Authors

Principal component analysis (PCA) is a mathematical method to reduce a dataset dimensionality by providing a set of orthogonal vectors indicating the maximum variance direction. Figure 5.2 shows random collections of points in orange and the black arrows represent the PCA components. The arrows indicate the direction and magnitude of the dataset variation when linearly dependent (Figures 5.2a and 5.2b) while a random dataset provides an arbitrary direction (Figure 5.2c). This work searches for the microarchitectural parameters with a significant impact on the application reliability. In this context, the PCA provides a fast approach to find linearly dependent variables that maximize a particular direction. By observing the components strength and direction (i.e., angle) it is possible to search the ones highlighting a strong correlation with the fault injection outcome. A greater X-axis variation (i.e., soft error) has more relevant information (Figure 5.2a) than a group with higher correlation (Figure 5.2b) with a high angle principal component (i.e., close to $90°$ or $0°$).

5.4.3.3 Linear Regression

Linear regression models the relationship between two scalar variables, one dependent variable, and one explanatory variable. This technique searches for the best-fitting straight line through the data points usually using a least squares approach. By analyzing the model angle and accumulated error it is possible to rank the features (Y) with a stronger impact on the soft error assessment (X).

$$Y = \alpha X + \beta \tag{5.1}$$

5.4.3.4 Correlation Coefficient

A correlation coefficient measures statistical dependence between two variables. In other words, this function describes two variables relationship through a score number between +1 and -1, where 0 represents the absence of correlation, 1 a complete correlation, and the sign gives the correlation direction. Pearson's is the most widely adopted correlation coefficient, and it measures linear relationships between two variables. Pearson's correlation coefficient is the covariance of the two variables divided by the product of their standard deviations.

$$\rho = \frac{\text{cov}(X,Y)}{\sigma_x \sigma_y} \qquad \text{Pearson's correlation coefficient}$$

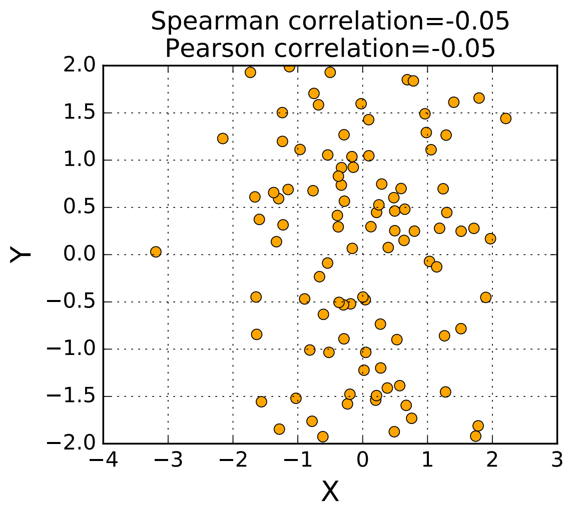
Further, it can be affected by strong outliers in a given dataset. For this purpose, Charles Spearman proposed a nonparametric measure of rank correlation to assess two variables monotonic relationship. A monotonic relationship means that two variables tend to towards the same relative direction, not necessary at an equal rate. By using ranked variables, the Spearman's coefficient presents a more robust solution to skewed points.

$$\rho_s = 1 - \frac{6\sum d_i^2}{n(n^2-1)} \qquad \text{Spearman's correlation coefficient}$$

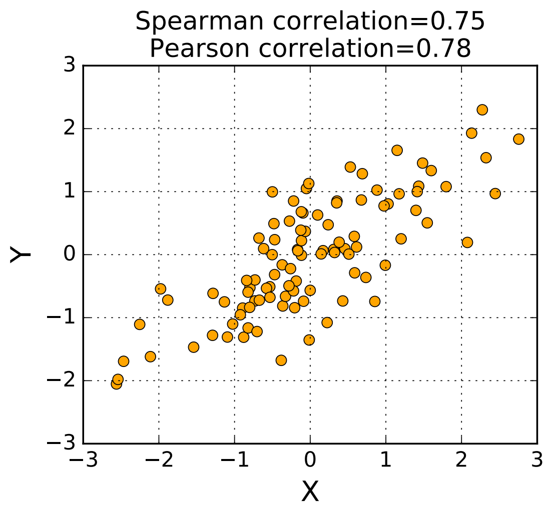
The coefficient selection is dependent on the raw dataset and its distribution. It is possible to filter features from a dataset by analyzing the ones with higher coefficients. Figure 5.3 shows four distinct datasets of 100 points considering the Spearman's and the Pearson's coefficient. First, Figure 5.3a shows random points with no dependence. Both coefficients have a similar performance in linear dependence scenarios (Figure 5.3b), while the Pearson's suffers from strong outliers (Figure 5.3c). Figure 5.3d displays a more monotonic dataset, where Spearman's has a better score.

Figure 5.3: Principal Component Analysis.

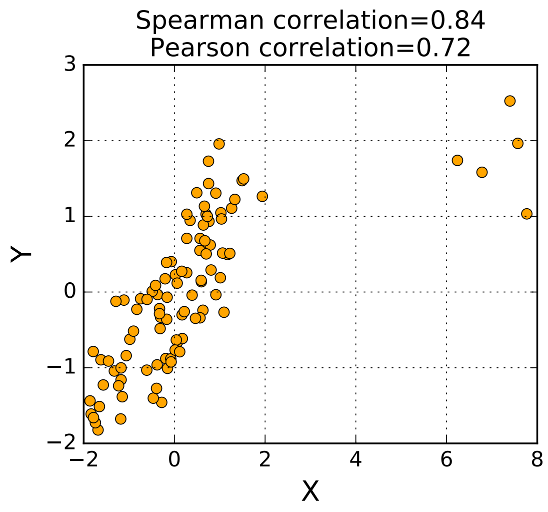
(a) Random.



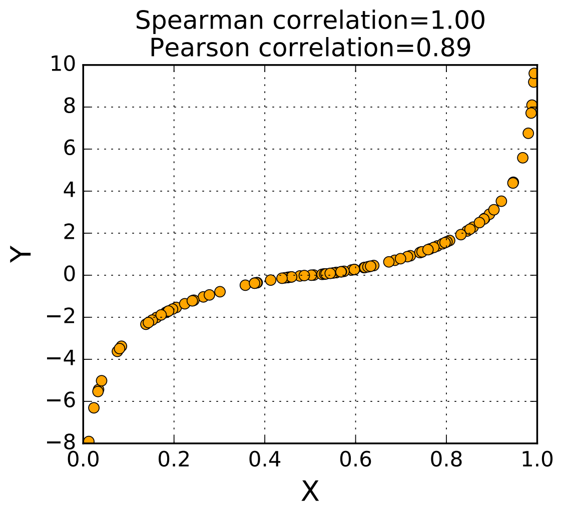
(b) Linearly Dependent.



(c) Linearly Dependent with outliers.



(d) Non-linear Relationship.



Source: The Authors

5.4.3.5 Recursive Feature Elimination

Recursive Feature Elimination (RFE) uses a given external estimator to prune a dataset until a target number of features. It employs the estimator coefficient to recursively remove features, creates a model, and calculates its accuracy. The Scikit-learn provides an

RFE wrapper method for an arbitrary estimator. This work explored a linear regression, Huber Regressor, and among other estimators. The quality of the selection depends on the estimator capability to classify different features relevance for the accuracy model.

5.4.3.6 Euclidean Distance

In the original data or due to transformations columns in the dataframe may have (almost) identical values. Removing such features improve the training time without reducing the dataset information. This method computes the Euclidean Distance between two features, removing it if they are closer then a predefined threshold.

5.4.3.7 Soft Error Score

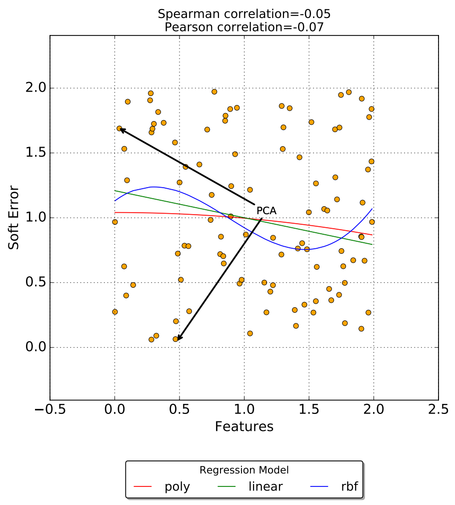
Microarchitectural information (i.e., features) comes in multiple ways and they represent a wide range of parameters such as cache misses, # of float instructions, or virtual memory page size. Each one has a different distribution, and for example, the virtual memory page size has few possible constant values while the # of float instructions ranges from zero to billions. Sometimes, parameters are constant for one application while fluctuates in another benchmark execution. Figure 5.4 shows six common feature arrangements found in the raw data from the gem5 microarchitectural statistics, for example, Figure 5.4a display a random value distribution. Figures 5.4b and 5.4c exhibit two linear relationships being the first one stronger while Figures 5.4d and 5.4e display constant features (i.e., independent from the soft error). Finally, Figure 5.4f demonstrates the average behavior of raw features, mixing some linear behavior, outliers, and constant values.

Applying one feature filtering algorithm does not provide the target results (i.e., the microarchitectural features with higher impact on the system soft error reliability). For instance, the correlation coefficients perform poorly on noise dataset (Figure 5.4f) where the relationships exist among other types of data (e.g., constant values). The PCA can find the maximum growth direction in complex features. However, it can result in false-positives depending on the data distribution (Figure 5.4e) Regression model provides a more robust solution without presenting a general solution. Further, applying multiple filters in sequence results on a small dataset with a significant amount of false-positive solutions.

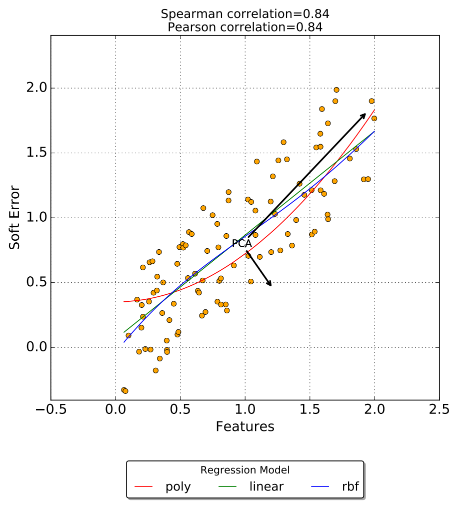
For this purpose, this work proposes a filter score (i.e., from 0 to 1) to measure

Figure 5.4: Different of feature combination extracted from the gem5.

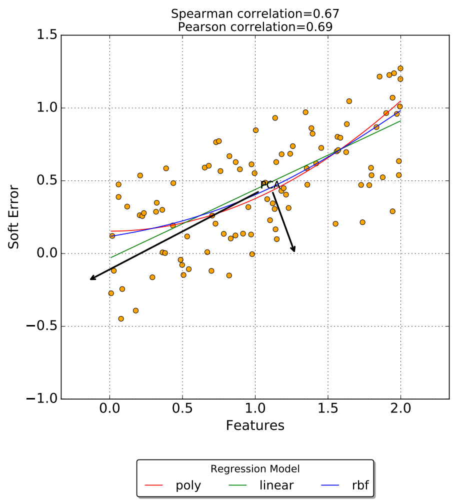
(a) Random distribution.



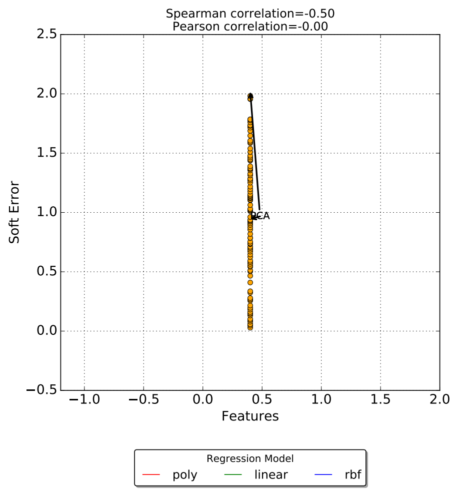
(b) Strong linear dependency.



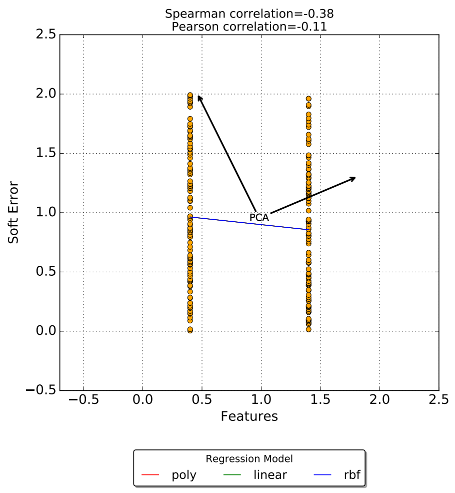
(c) Weak linear dependency.



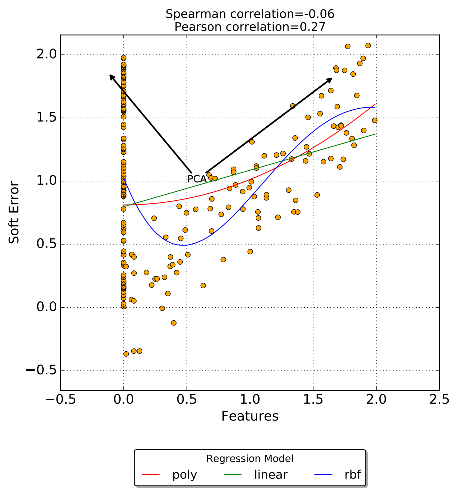
(d) Constant value.



(e) Multiple constant values.



(f) Usual feature shape.



Source: The Authors

the feature quality according to the target soft error problem. This score results from several algorithms simultaneous execution, in other words, the PCA, regression, and other techniques are computed in a single step. These outcomes are standardized in values between 0 and 1 to remove the parameter magnitude from the equation. The score computes the PCA first component, the Pearson's correlation coefficient, variance, dispersion score, and linear regression. Pearson's is preferable due to its smaller sensibility to false-positive linear relationships. Linear regression models with a $45°$ (from the origin) represent the optimal correlation between two dependent variables, increasing or decreasing this angle shows a weaker correlation.

The selection process computes the Soft Error Score for all columns in the target dataframe. The tool ranks the features by the score selecting the first *N* most significant values. This technique provides both filtering and ranking in a single technique, reducing the number of feature selection steps.

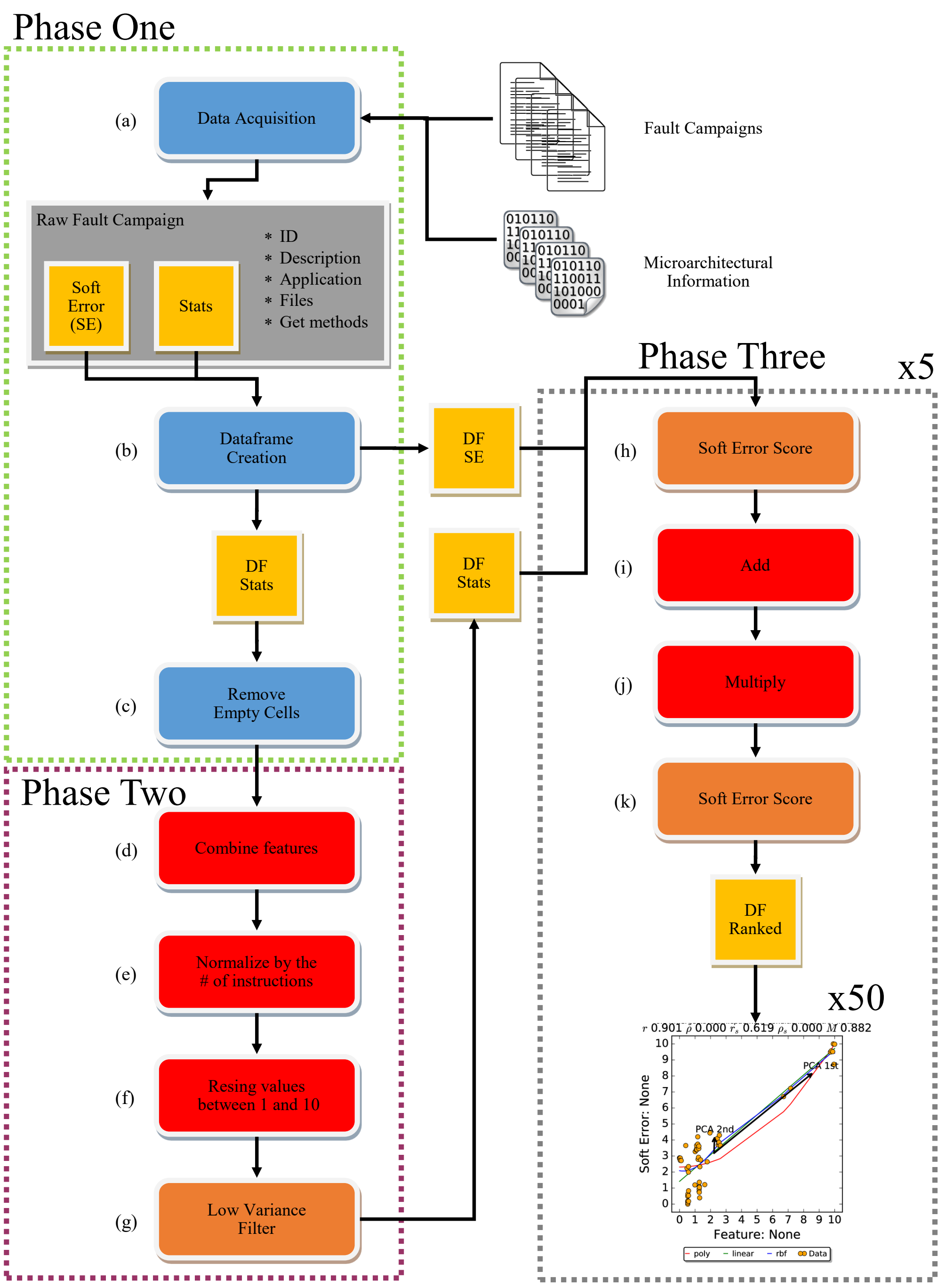
5.5 Exploration Flow

Considering this Thesis first exploration objective (i.e., to find gem5 microarchitectural statistics that affect the most the fault injections outcome) an automated exploration was proposed. Figure 5.5 shows this exploration execution flow which can be divided into three main phases.

5.5.1 Phase 1 - Feature acquisition and data homogenization

The tool searches for the input files and reads one by one, extracting the relevant information (Figure 5.5a). Thus, two dataframes are created (b) where lines represent individual fault campaign scenarios (e.g., varying the # of cores, kernel, application) and the columns store the soft error (labels) or the microarchitectural statistics (features). The soft error DF is guaranteed to be complete (i.e., every cell has a value), however, the features depends on the scenario leaving unfilled data. Empty data points are replaced with NaN values (i.e., not a number) in the dataframe rectangular grid to retain a regular shape. Machine learning methods do not handle correctly abstract values, for this purpose, NaNs are replaced by zero in the features DF (c).

Figure 5.5: Proposed Automated Flow.



Source: The Authors

5.5.2 Phase 2 - Unidimensional Feature Transformation and Selection

The second phase creates a new set of features by combining parameters from multiple cores. For instance, the gem5 report summarizes the # of float instructions by individual core (e.g., system.cpu0.num_fp_insts, system.cpu1.num_fp_insts, system.cpu2.num_-

fp_insts). By merging the multiple related features (Figure 5.5d), it is possible to have the cores and the complete processor behavior. These reports provide most parameters as nominal values (e.g., system.cpu0.num_fp_insts, system.cpu0.op_class::MemWrite) which depends on the target application execution time (i.e., # of instructions). A direct comparison between two applications may no result in an evident relationship due to the distinct data magnitudes. For instance, if an application A has 1,000 branches and another B has 5,000 may leads to a distinct conclusion then analyzing the branch participation (i.e., divided by the # of instructions). To expand the exploration, the tool creates a second features DF where all columns are divided by the application # of instructions (e). Similarly, it is necessary to remove the magnitude variance from the features by resizing their range between 1 to 10 (f), enabling a fair comparison between features. This transformation also improves several machine learning techniques performance. Finally, the tool reduces the total number of features (i.e., independently from the soft error) by eliminating the ones with lower variance (g).

5.5.3 Phase 3 - Multidimensional Feature Transformation and Selection

Until this point, the exploration flow was restricted to a single dimension (features). When considering the adopted soft error classification (i.e., vanish, hang, ONA, OMM, UT) the exploration becomes a five-dimensional problem. To reduce the computational cost, this flow performs five bi-dimensional investigations, each class against the features DF. First, the tool prunes the features from few thousand to 50 using the proposed Soft Error Score employing both features and labels (i.e., error classification) (Figure 5.5h). Then, this reduced dataframe suffers two feature combination transformations: (i) addition and (j) multiplication. In other words, the tool adds and multiplies the fifty columns in every possible combination leading to a total of 5000 new features. Again, this significant dataset is pruned to the 50 most relevant features using the Soft Error Score which also provides the ranking of features (k). The tool automatically plots each feature and label with its Spearman's, Pearson's, and Soft Error Scores alongside the PCA and three regression models.

5.6 Results

5.6.1 Training Set Selection and Bias

Selecting a representative training set is paramount to have a meaningful result from any machine learning technique, i.e., the input dataset caries a bias leading to different outcomes. The algorithm accuracy depends on the acquired data, in other words, by judiciously reducing or increasing the training set it is possible to achieve different goals. This work collects inputs from multiple applications representing a large number of algorithms during a broad investigation. At the same time, the data has serial, OpenMP, and MPI benchmarks which can be broken down into distinct explorations with different results. Table 5.2 display the available training sets.

Table 5.2: Available training sets.

Arch	*Short*	*Description*	*Scenarios*	*Fault Injections*
V7	WCET	WCET Serial	35	280,000
V7	MIB	Mibench Serial	19	152,000
V7	SER	Nasa NPB Serial	10 10	160,000
V7	ROD	Rodinia OpenMP	16	128,000
V7	OMP	Nasa NPB OpenMP	30 40	560,000
V7	MPI	Nasa NPB MPI	25 32	456,000
V8	OMP	Nasa NPB OpenMP	40	320,000
V8	MPI	Nasa NPB MPI	32	256,000
		Total	289	2,312,000

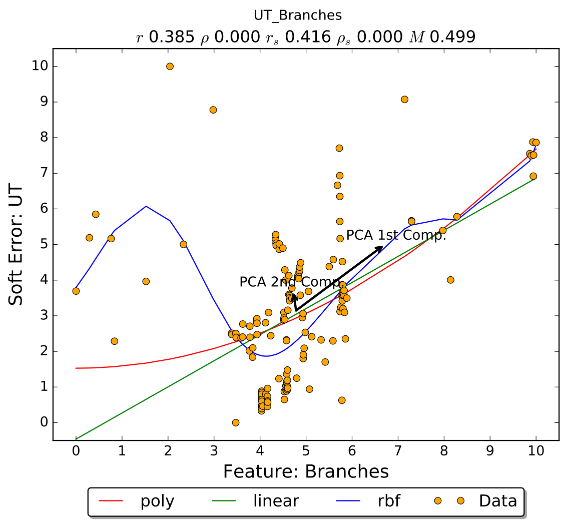
Source: The Authors

To exemplify this behavior this work compares one parameter exploration by varying the training dataset. Additionally, this exploration targets the *Unexpected Termination* (UT) soft error class. First, Figure 5.6 shows the branch parameter using distinct inputs: All scenarios (i.e., serial and parallel) (Figure 5.6a), Parallel (i.e., OpenMP and MPI) (Figure 5.6b), only MPI (Figure 5.6d), and only OpenMP (Figure 5.6c). Considering every possible scenario, the # of branches has a not negligible relationship with the UT occurrence. However, both Spearman's and Pearson's describe a weak to medium correlation due to the prevalence of random data points. This interrelationship increases when restricting the ML inputs to parallel applications which are composed of MPI and OpenMP applications. Further, Figures 5.6c and 5.6d breaks down the individual components of

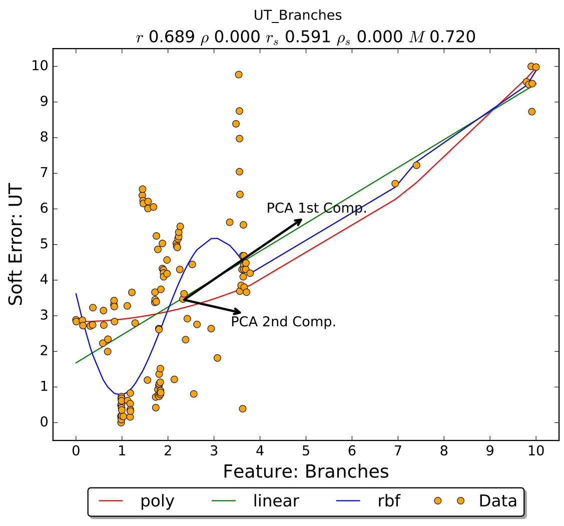
Figure 5.6b, where the MPI applications visibly demonstrate a strong interaction between the number of branches and UTs. The three adopted coefficients (i.e., Spearman, Pearson, and soft error scores) exhibit this pattern by increasing 25% on average. By selection, the most appropriate training set is possible to reduce the error or focus on a smaller target population.

Figure 5.6: Example of training set bias.

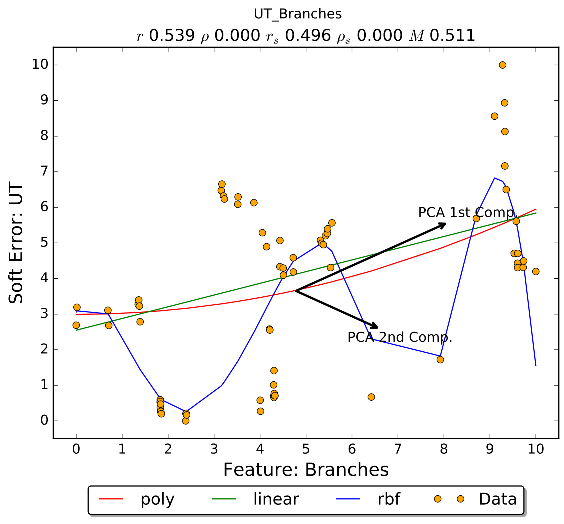
(a) All Scenarios.



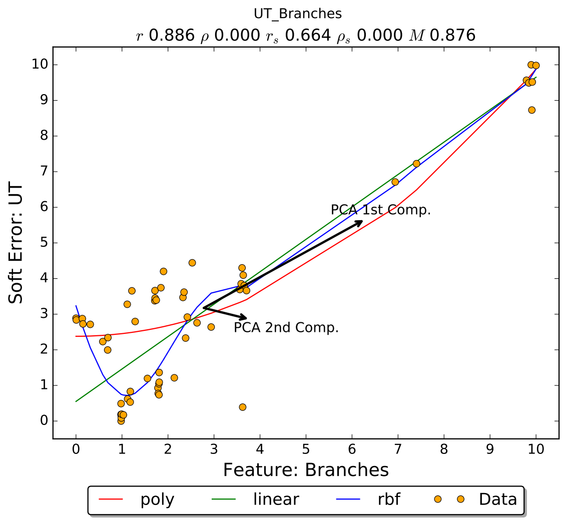
(b) MPI and OpenMP.



(c) OpenMP.



(d) MPI.



Source: The Authors

5.6.2 Characterization

The following subsection discusses the characterization phase (Figure 5.1c). We select the Nasa NPB OpenMP and MPI benchmarks because they display a more interesting and realistic scenario considering multicore systems. The characterization precedes the machine learning execution, enabling the access to every parameter available. This phase enables the comparison between multiple applications in order to create subgroups. Also, it is possible to profile the target application in multiple execution scenarios (e.g., distinct compilation flags).m Figure 5.7 shows these benchmarks soft error (red) and microarchitectural parameters (blue) divided by application (e.g., EP, LU). For this example, we selected six relevant parameters:

(i) **CPU/MEM** : The ratio between the number of ALU operations and the number of memory accesses, both write and read. A larger number means a CPU-bound application and a smaller one a Memory-bound.

(ii) **Func Calls** : The concentration (i.e., this instruction occurrence divided by the number of simulated instructions) of function call instructions.

(iii) **Cond Ctrl** : The concentration of instructions with conditional control.

(iv) **Branches** : The concentration of branch instructions.

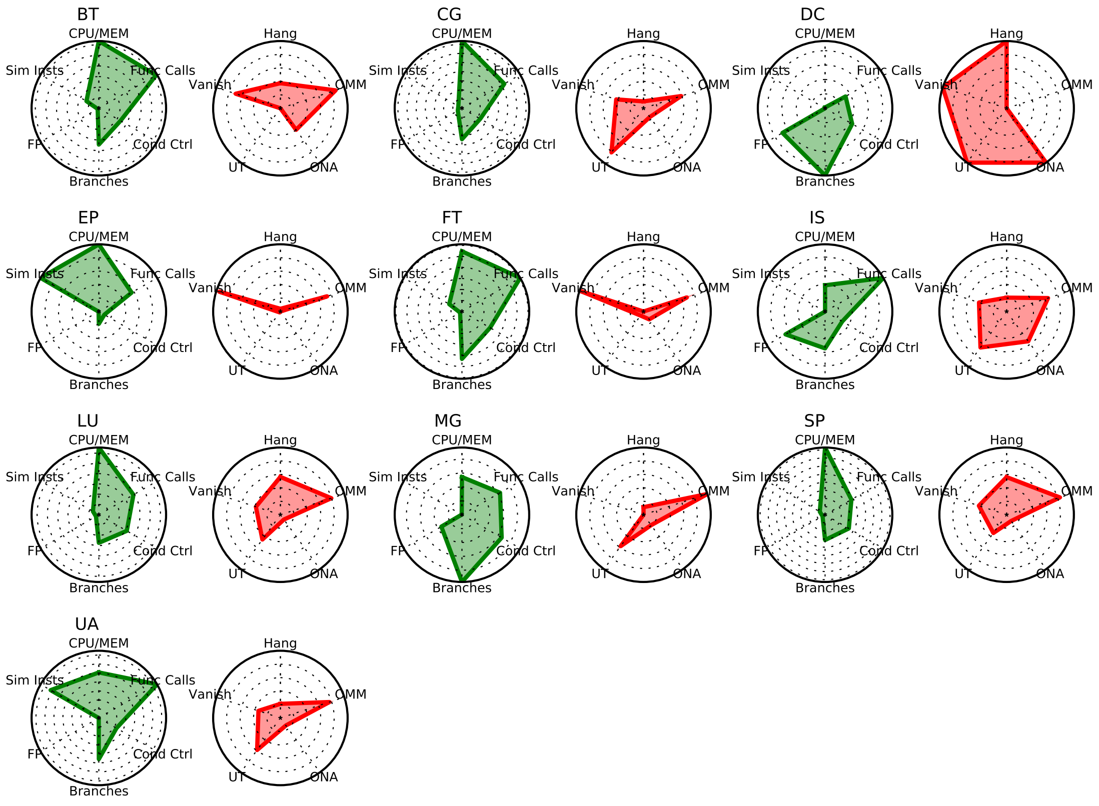
(v) **FP** : The concentration of float-pointing instructions.

(vi) **Sim Insts** : Total number of simulated instructions.

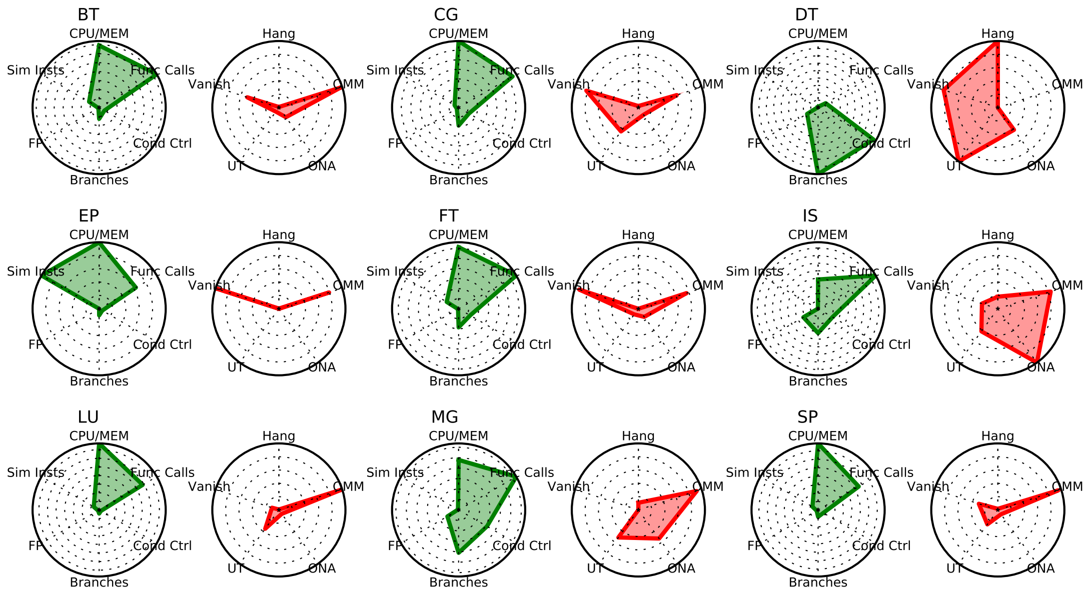
This initial investigation phase provides some useful insights in training sets (e.g., OpenMP, MPI). It is visible the large concentration of branch instructions in the OpenMP applications lead to UTs and hangs due to its parallelization nature as previously discussed. More significantly, it is possible to divide the application into subgroups among the datasets to explore similarities among the benchmarks. For instance, LU, MG, and SP are CPU-bounded with a high concentration of function calls. Further, the combination of both parameters correlates with the OMM (i.e., memory silent data corruptions) presence during fault campaigns. Next subsections explore the results extracted using the proposed tool also considering this characterization phase possible insights.

Figure 5.7: NPB characterization.

(a) OpenMP.



(b) MPI.

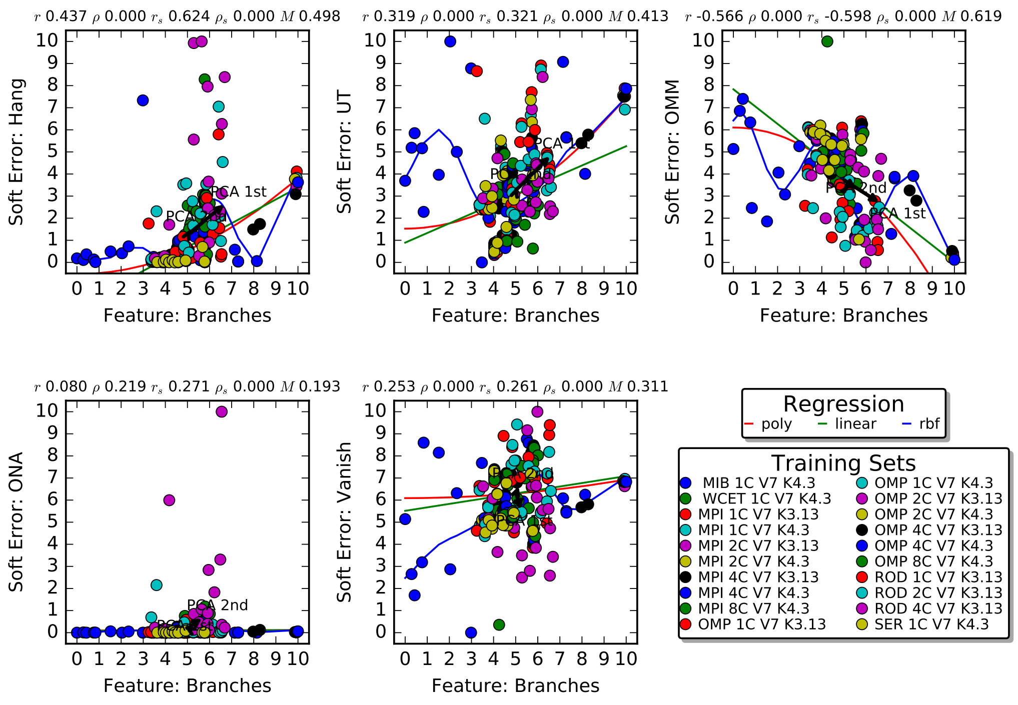


Source: The Authors

5.6.3 Branches and Function calls

Control flow statements (e.g., for, if, else, while) are the fundamental bedrock of any programming language. The ability of conditionally executing code portions enables software engineers to create more complex algorithms where jump, branch, and function call instructions fulfill the control flow function at the assembly level. Any fault in such instructions typically causes unrecoverable errors (e.g., segmentation faults). The tool provides preliminary results for investigating the impact of control flow instruction in the application reliability. Figure 5.8 shows the effect of the branch instructions concentration (i.e., the number of branches divided by the total of instructions) on the soft error vulnerability considering multiple training sets. None of the fault injection classes demonstrate a strong correlation with the branch concentration due to the diversity of input scenarios such as serial and parallel applications, with execution time varying from 10 million to 87 billion instructions, from one to eight cores.

Figure 5.8: Branch instruction impact on the soft error classification.



Source: The Authors

The training sets wide range conceals several individual relationships, in particular, how multicore applications behave under this conditions. For this purpose, Figure 5.9 display the branch instruction concentration focusing on the NPB MPI (Figure 5.9b) and OpenMP-based (Figure 5.9a) applications which presents a more extensive workload and better scalability. The execution time of the NAS suite ranges from 300 million to 87 billion instructions, from one to eight cores. Note that the host Linux kernel (i.e., 4.3 or 3.13) shows no direct impact on the application reliability for this particular set of applications. Due to the NPB has a very long execution time (i.e., 16 billion instructions on average) only a fraction of this total is dedicated to the kernel. Consequently, direct faults injections in the Linux inner workings are extremely rare.

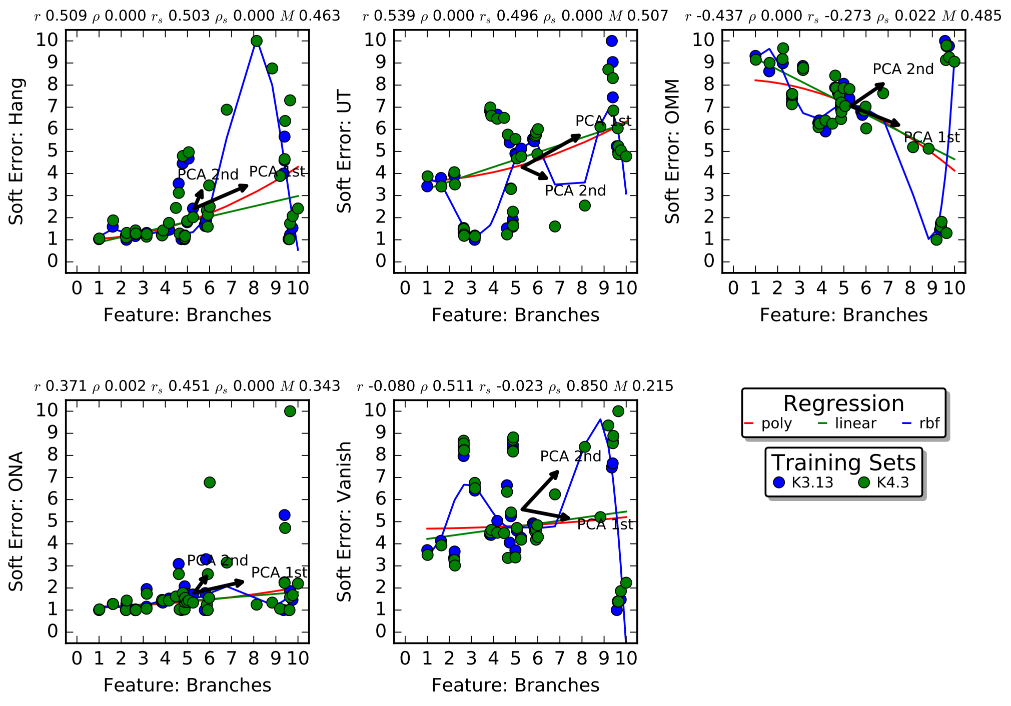
Considering the impact of the number of branches on the software stack reliability (Figure 5.9), note how the OpenMP benchmarks exhibit a weaker relationship than the MPI ones. OpenMP and MPI have distinct behaviors under fault injection, for example, the OpenMP relies on loop (i.e., for-while) parallelizations leading to a greater branch presence. In contrast, MPI applications show a more considerable influence of the number of branches on its soft error vulnerability, especially, when considering the Hang, UT, and OMM classes. Hang errors arise due to severe control flow errors (e.g., incorrect iteration counter), UT is an unexpected application termination (e.g., wrong address calculation), and an OMM results from an application finishing with an incorrect memory. Both Hang and UT show a direct and positive correlation while OMM a negative correlation, in other words, the occurrence of Hang and UT increases at the same time that the OMM reduces considering MPI applications. In this case, branch errors lead to Hangs or UTs which in other circumstance would finish with an erroneous memory.

Function calls are a particular type of control flow instruction which simultaneously changes the program counter and stack pointer. Observing the Hang error (see Figure 5.10), both MPI and OpenMP applications reliability improves when increasing the # of function call until a saturation point. The UT class follows the Hang behavior (Figure 5.10b) while the OMM presents an inverse correlation. Additionally, the distinct effect of the OMM versus Hang and UT is similar to the one displayed by the branch on MPI-based benchmarks (Figure 5.9b). Analyzing individual parameters not always expose direct relationships between profiling data and fault injection campaigns. By combining both figures, nonetheless, is possible to uncover strong correlations between this new index value (i.e., number of function calls times number of branches). Figure 5.11 exemplifies this behavior, note that this new index value and the Hang percentage increases simultaneously, an observable behavior through several scenarios for both MPI and OpenMP. The tool main ability is to combine multiple features to highlight previously unknown correlations leading to a profound understatement of the scenarios behavior.

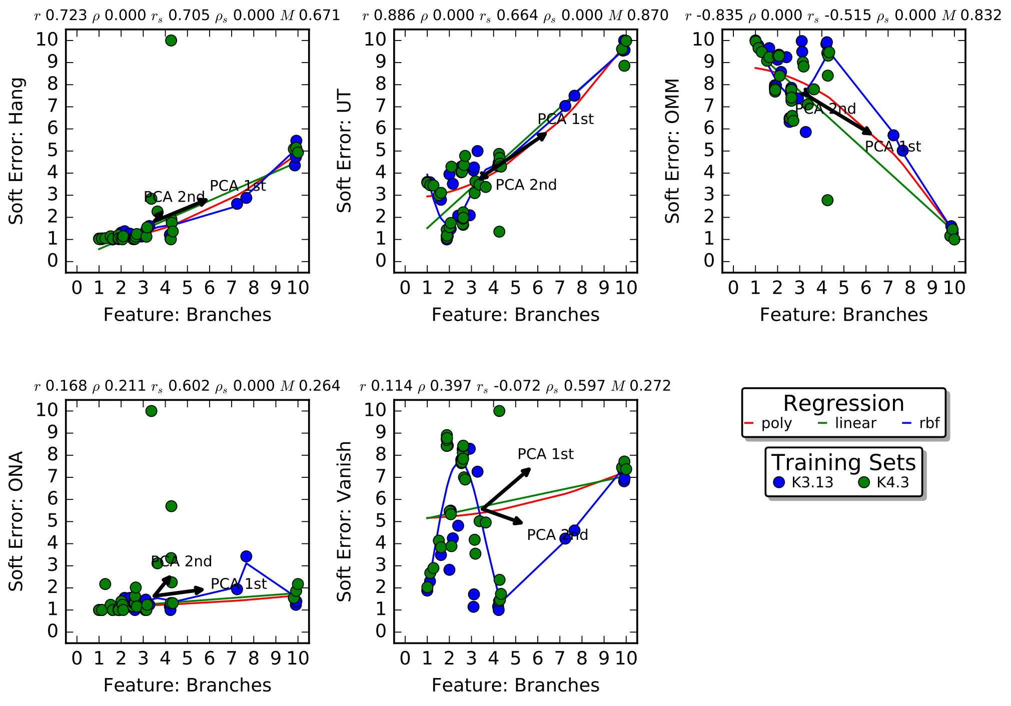
As previously mentioned, particular applications can be grouped into subgroups

Figure 5.9: Branch instruction impact on the soft error classification targeting parallel applications.

(a) NPB OpenMP.



(b) NPB MPI.

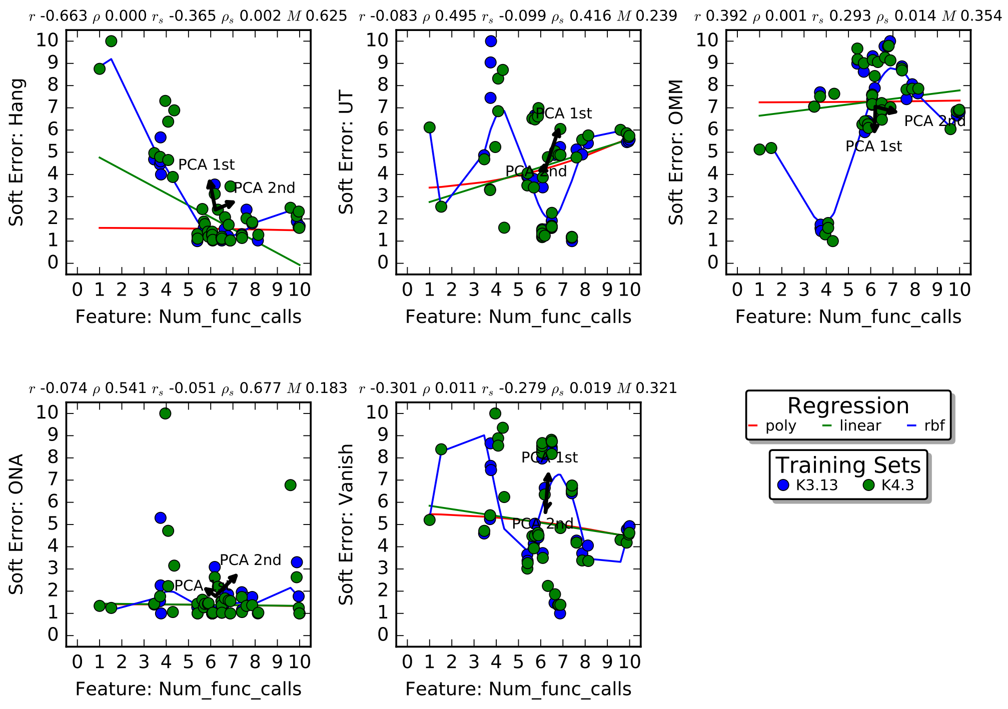


Source: The Authors

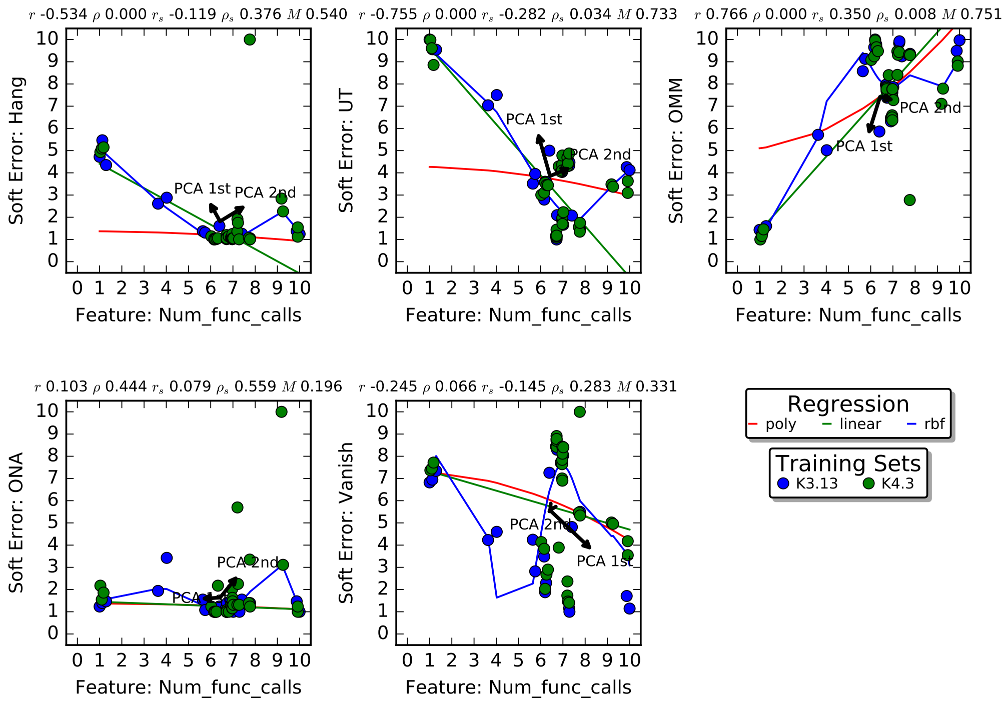
according to their similarity. LU, MG, and SP form one of such collections as they are CPU-bounded with a high concentration of function calls. The three applications reliability (i.e., vanish rate) improves as the as the index (i.e., function calls times the branches)

Figure 5.10: Function calls impact on the soft error classification targeting parallel applications.

(a) NPB OpenMP.



(b) NPB MPI.

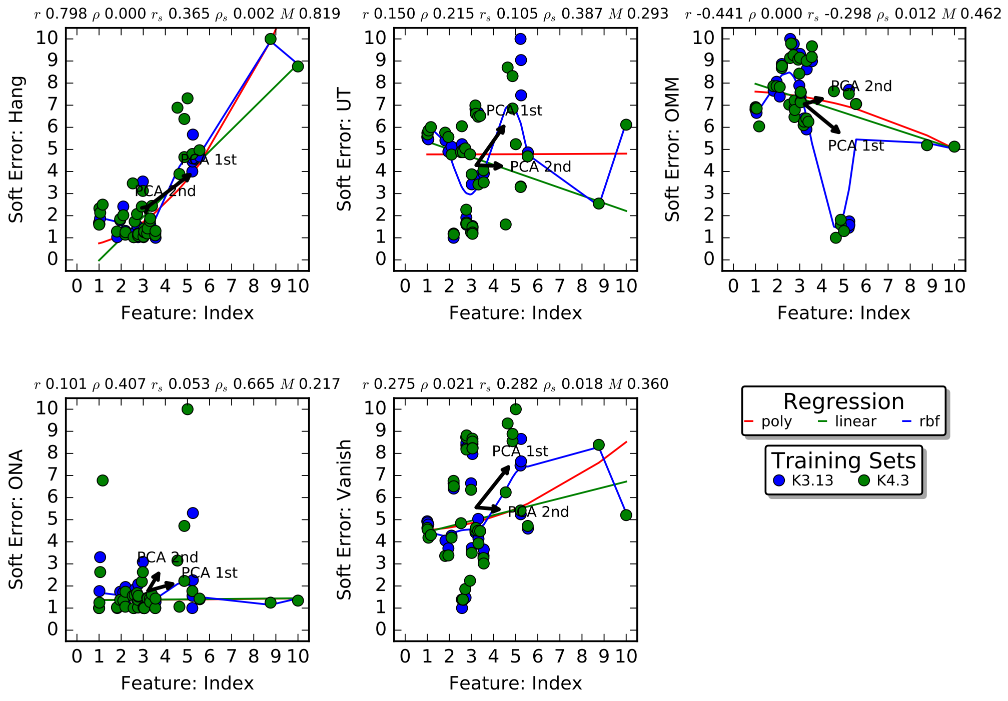


Source: The Authors

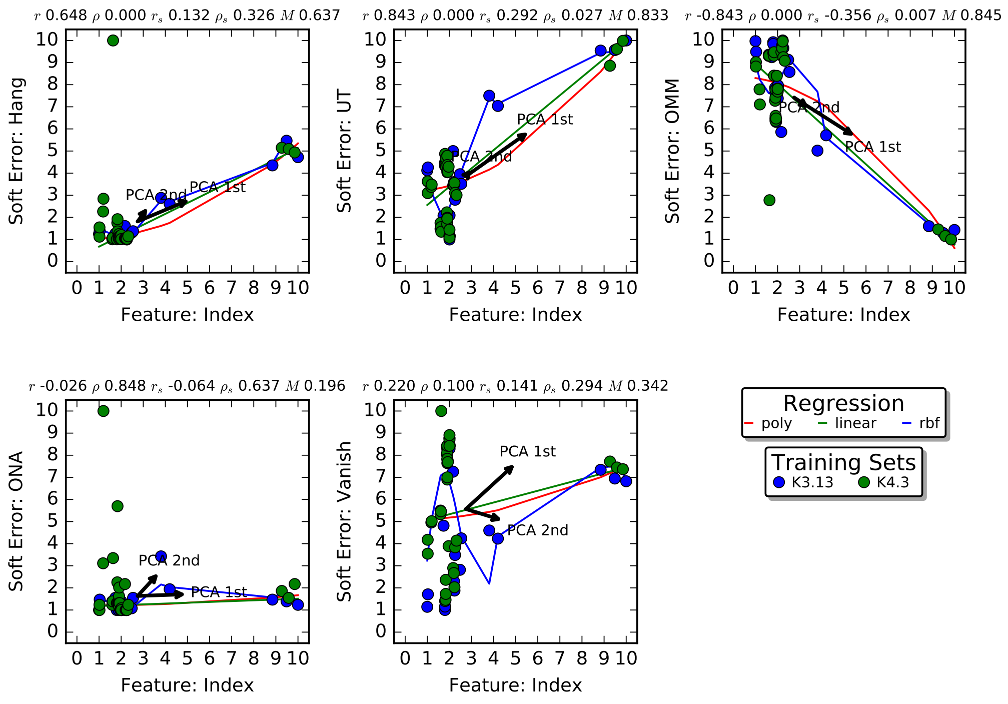
grows, also noting how the OMM occurrence also decreases in Figure 5.12a. Reads and write operations in memory-bounded algorithms usually move these fault values to the memory. Instead, their CPU-bounded nature leads to longer vulnerability windows, i.e.,

Figure 5.11: Proposed index impact on the soft error classification targeting parallel applications.

(a) NPB OpenMP.



(b) NPB MPI.

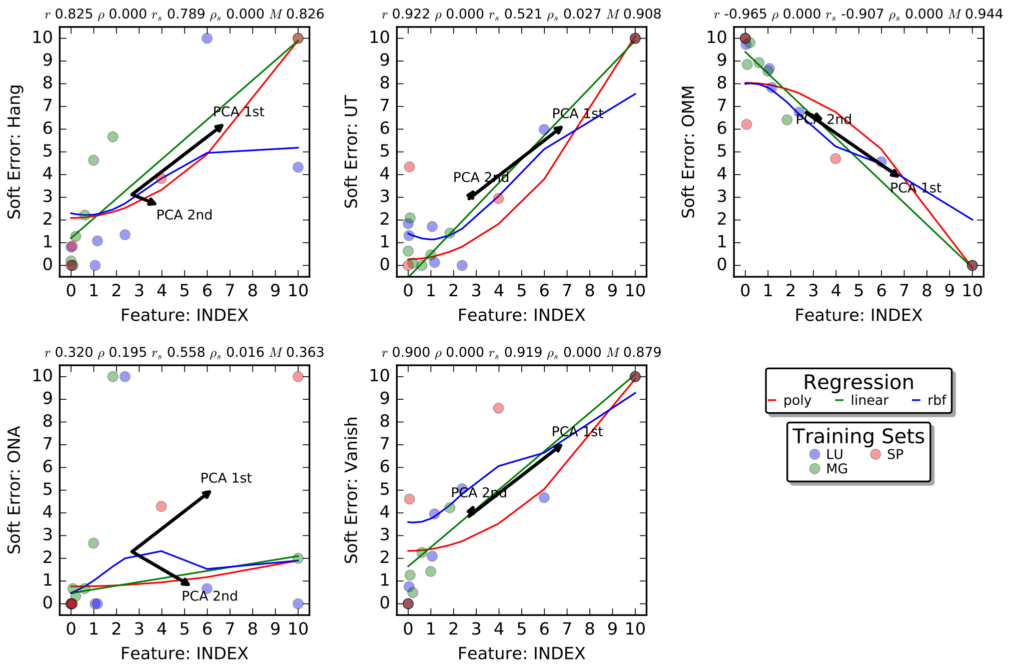


Source: The Authors

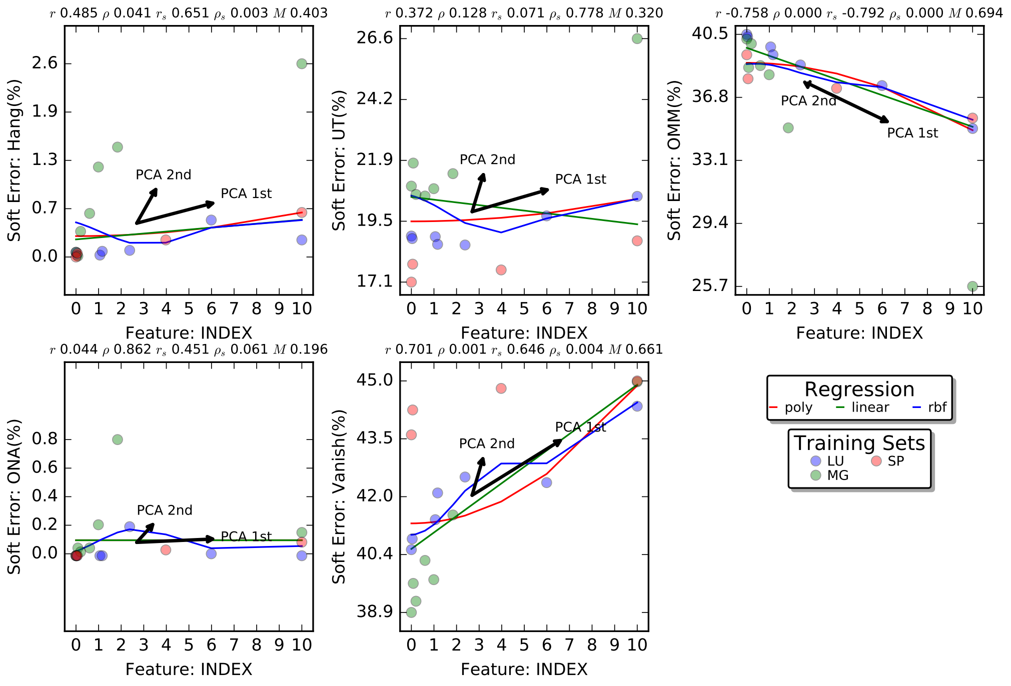
the data remains in the exposed register-file longer. In this particular applications, function calls have a rejuvenating effect by restoring old registers values saved during earlier iterations. This behavior is more explicit in Figure 5.12b by visualizing the same data

Figure 5.12: Proposed index targeting the LU,SP, and MG applications.

(a) Resizing both axis.



(b) Unmodified soft error values.



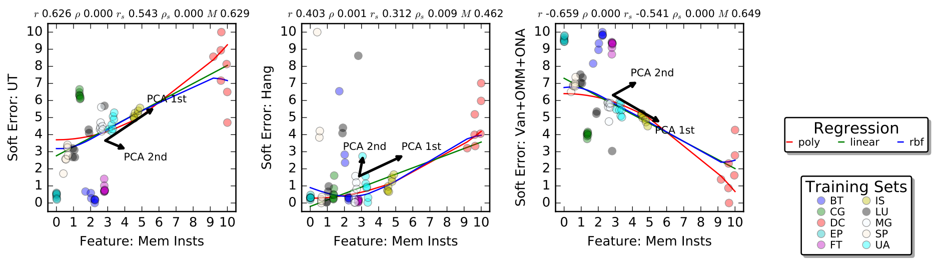
Source: The Authors

without resizing the soft error (Y) axis.

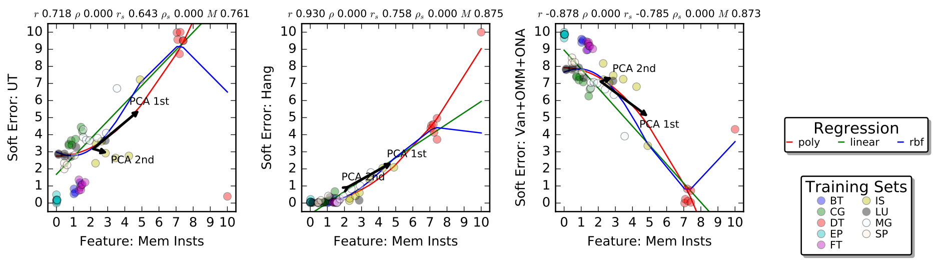
5.6.4 Memory Transactions

Figure 5.13: Integer and Float-pointing (FP) memory access instructions impact on the soft error classification targeting parallel applications.

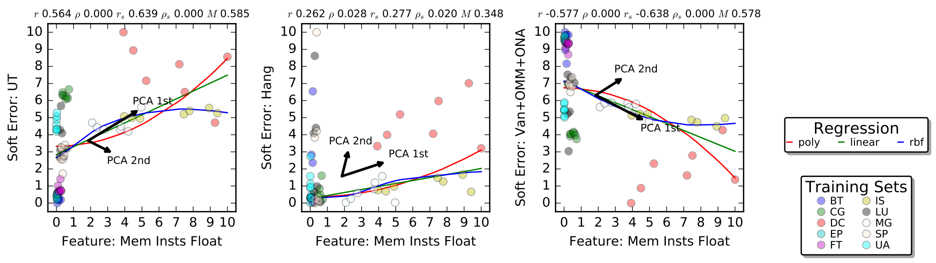
(a) NPB OpenMP Integer Mem. Access



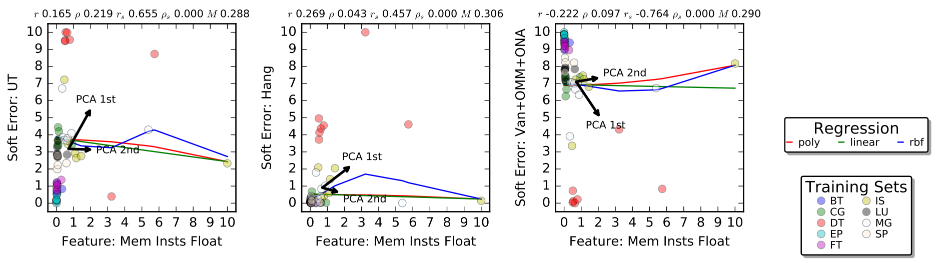
(b) NPB MPI Integer Mem. Access



(c) NPB OpenMP FP Mem. Access



(d) NPB MPI FP Mem. Access



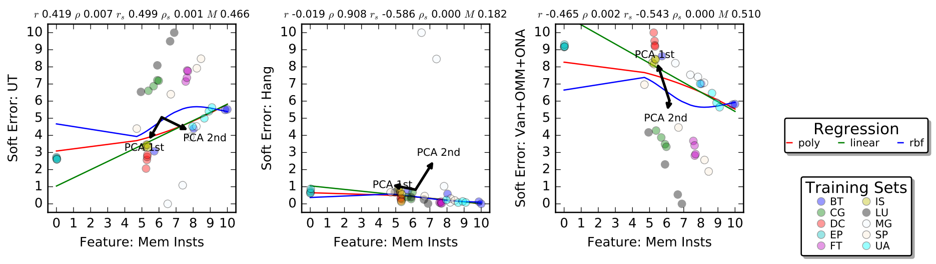
Source: The Authors

At instruction level on a RISC processor, the address generation of memory access operations (e.g., load and stores) can be compromised by soft errors as the source register faults lead to wrong address calculations. The reduced number of ARMv7 registers to perform address calculations leads to the use of load/store templates by the compiler to diminish the computational cost of register recycling. In other words, the ARMv7 compiler continuously utilizes the same register to perform memory transactions (e.g., R0–3 and SP). Figure 5.13 shows the soft error results (e.g., Vanish+OMM+ONA, UT, Hangs) alongside the memory access figures both integer and float-pointing (FP). Vanish, OMM, and ONA where clustered into a single group to investigate the UT and Hang incidence. Increasing the number of load/store operations can lead to a more significant UT and Hang occurrence in MPI applications using an OS on top of the ARMv7 processor (Figure 5.13d). In applications such as MG and IS the increasing percentage of memory transactions (i.e., load and stores instructions) display a growing occurrence of hang and UT. For example, MG MPI application memory-oriented operations for single and quad-core processors are 15%, and 22% while the UT occurrence increases from 22% to 30%. DT and DC have a less pronounced behavior due to its limited execution time approximately 500 million instructions while the NPB average floats around 16 billion. In contrast, the OpenMP-based applications exhibit no direct impact of the # of load/stores in the Hang/UT incidence. The OpenMP library parallelizes the *for* statements and to achieve a higher performance the loop index remains on the register-file (i.e., reducing the memory access for branch executions). In Figure 5.13c varying the number of memory accesses has no apparent correlation with the UT and Hang occurrence on OpenMP benchmarks.

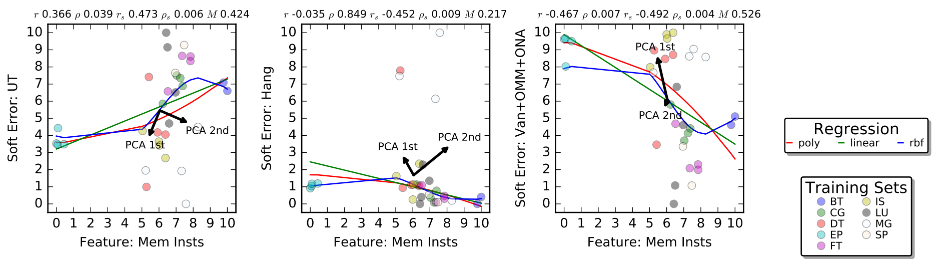
The ARMv7 workload for a single faultless execution has an instruction count that ranges from 299 million to 87 billion, with an average of 16 billion of instructions. In contrast, the 64-bit architecture applications execute in average 654 million instructions, varying from 41 million to 3 billion. Applications executed using the ARMv8 ISA present a significant performance improvement when compared to the ARMv7. This performance gain can be pinpointed to the removal of several legacy features (e.g., fast and multilevel interruptions, conditional instructions) and to significant improvements in the floating-point unit by adding new specialized instructions and increasing the FP register file. The ARMv7 often resorts to the ARM software FP library to perform some operations and thus increasing execution time due to automatically compiler decisions. The 64-bit architecture exhibits a similar behavior considering FP memory transactions, supporting the claim above that wrong address calculation related to memory access, as FP instructions are exclusively used for computation and not for control flow operations (e.g., branches and jumps). The ARMv8 architecture has a higher percentage of float-pointing instructions than the ARMv7. Figure 5.14 displays several scenarios of soft error analysis and FP memory figures. Reducing the memory transactions participation from the total num-

Figure 5.14: Integer and Float-pointing (FP) memory access instructions impact on the soft error classification using a 64-bit ARMv8 architecture.

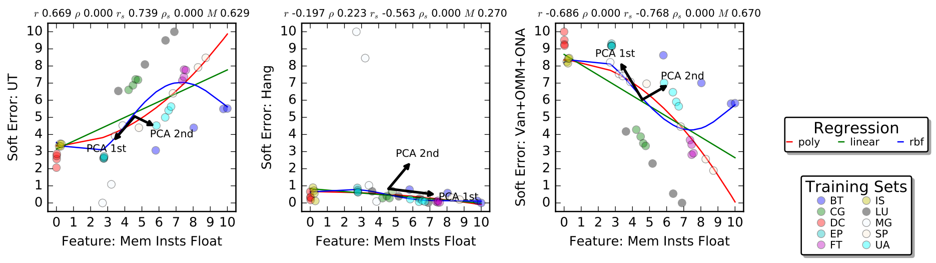
(a) NPB OpenMP Integer Mem. Access



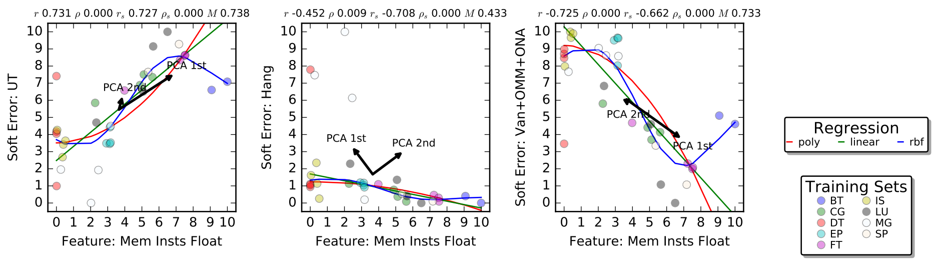
(b) NPB MPI Integer Mem. Access



(c) NPB OpenMP FP Mem. Access



(d) NPB MPI FP Mem. Access



Source: The Authors

ber of executed instructions for LU and SP applications show a UT occurrence reduction trend. FT and UA MPI applications reinforce this hypothesis by demonstrating that a constant memory-oriented instruction incidence leads to a regular UT percentage.

5.7 Case Study

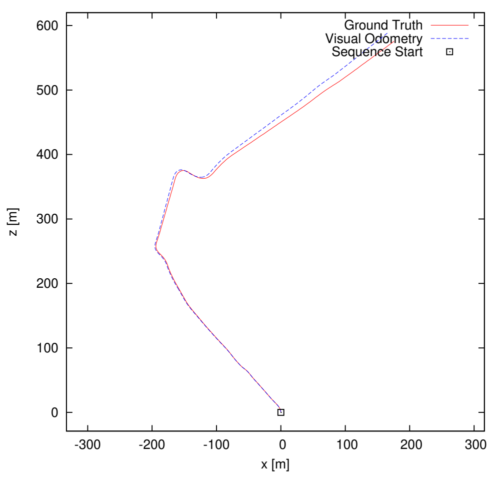
Following Figure 5.1, this case study first selects and design the target application. Nowadays self-driven cars capable of automatically steer and guide the vehicle is being tested. Such systems should be able to analyze the real word, make decisions, and perform actions resulting in a complex software stack with multiple algorithms. Visual odometry (VO) is one of the most critical subsystems, which is the process of determining the position and orientation of a robot by analyzing the associated camera images. In other words, the VO system uses a sequence of images to determine the vehicle traveled path. This work applies the acquired knowledge with multiple characterized benchmarks and the proposed machine learning tool in a real-life scenario. For this case study, we selected the LIBVISO2 (GEIGER; ZIEGLER; STILLER, 2011) visual odometry library developed by the Karlsruhe Institute of Technology (KIT). The library and dependencies are compiled using the arm GCC cross-compiler with hard float-pointing and single instruction, multiple data (SIMD) flags enabled. Our LIBVISO2 setup uses the KITTI Vision Benchmark Suite (GEIGER; LENZ; URTASUN, 2012b) which is composed of 22 stereo odometry sequences from real-life vehicle trajectories. Figure 5.15 shows two input frames of the benchmark 11 from the KITTI suite alongside the algorithm predicted path and ground truth.

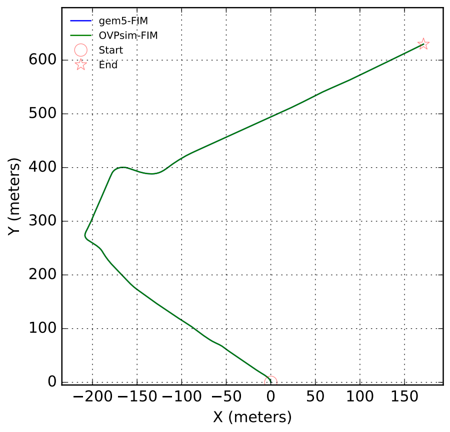
The initial exploration comprises four fault injection campaigns targeting the benchmark 11, each one compile with a different optimization flag. Further, this first investigation is restricted to 100 frames from a total of 920 where the fault campaigns are performed by the OVPsim-FIM as described in Section 3.5. The OVPsim-FIM presents a flexible and fast fault injection framework enabling more complex soft error vulnerability explorations. While the characterization phase deploys the gem5-FIM (Section 3.5) to extract the microarchitectural parameters, which also is necessary for the machine learning investigation. The chosen VPs have differences in the simulation engine without affecting the target algorithm produced output. Figure 5.15d shows the benchmark 11 execution using the OVPsim-FIM and gem5-FIM which are identical. Figure 5.16 shows this first exploration fault injection (a) and characterization (b) results. Note how the application without any optimization (i.e., O0) has an execution 4.7x times larger than any optimization flag (i.e., O1-3), from 150 to 32 billion instructions respectively. As result of this extended execution time results on a more significant number of register operations (ALU) as shown by Figure 5.16b. Because the compiler optimization reduces the

Figure 5.15: Visual Odometry benchmark 11.

(a) Input Frame. (b) Input Frame.

(c) Predicted path and ground truth. (d) Simulation using Virtual Platforms.



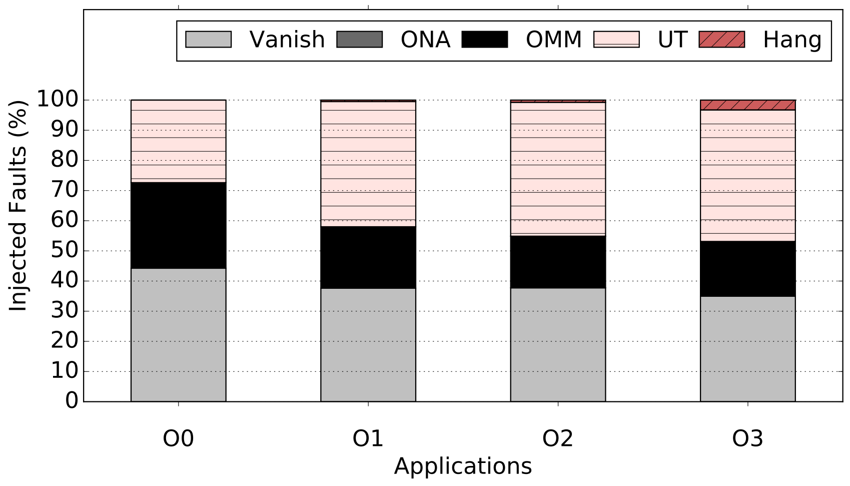


Images extracted from (GEIGER; ZIEGLER; STILLER, 2011; GEIGER; LENZ; URTASUN, 2012b).

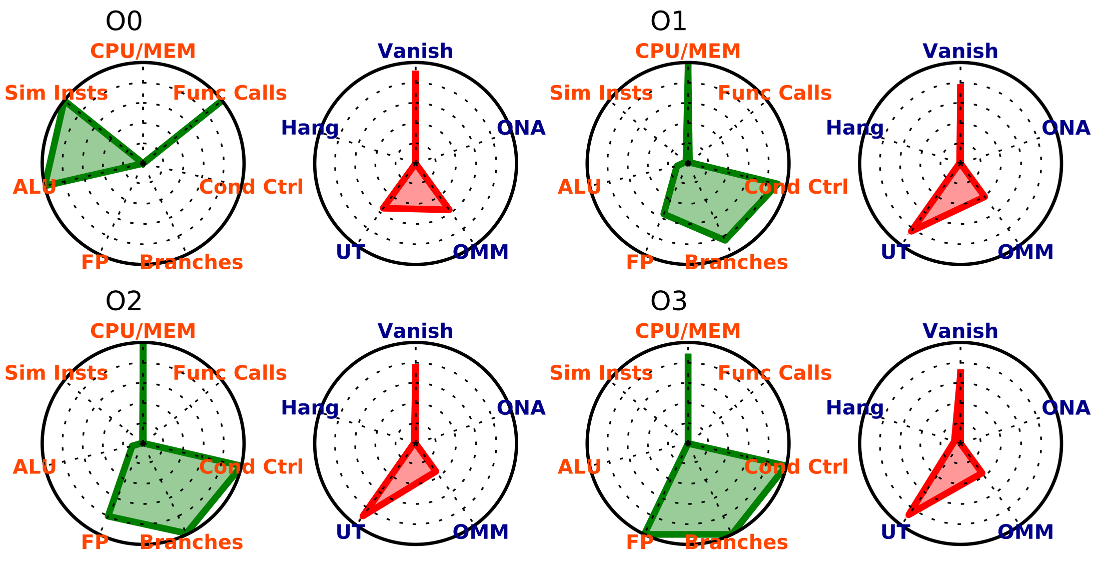
number of instructions between control flow statements (e.g., if) it increases the probability of Hangs and Unexpected Termination. When comparing the application with code optimizations (i.e., O1-3) is noticeable a growing number of hang occurrences due to increasing modification in the loop-based snippets. For example, observe the growing # of branches with more aggressive GCC optimizations.

The visual odometry benchmark was chosen due to its real-life applicability in self-driven cars which enables a physical representation of the system reliability (i.e., the traveled path). Figure 5.17 depicts the traveled way (A), # of evaluated frames (B), and the deviation between the correct and predicted stop point (C) for each algorithm simulation under fault injection. Lastly, Figure 5.17 (D) shows the scenario (C) restricted to completed executions (i.e., all frames were evaluated by the algorithm). Figure 5.17a exhibits these four plots considering no GCC optimization (O0). In this scenario, for instance, 73.12% of the fault injections terminate (i.e., evaluates 100% of the frames), from whom only four are incorrect. In contrast, when using compiler optimizations, the number of completed simulations decreases from 73.12% to 55%. In this example (Figure 5.17) the vehicle travels around 70 meters after processing 100 frames. The error reaches up to 70, where larger values are due to algorithm halts (i.e., UT or Hang) stopping far from

Figure 5.16: Benchmark 11 initial exploration.
(a) Fault Injection.



(b) Characterization.



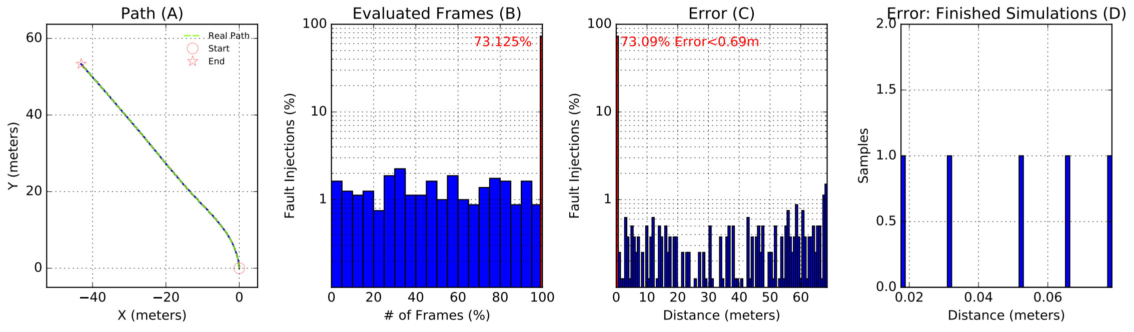
Source: The Authors

the correct point. The error considering only completed executions achieves up to 0.50m when compiled with optimizations while the O0 flag has no substantial deviation from the correct path.

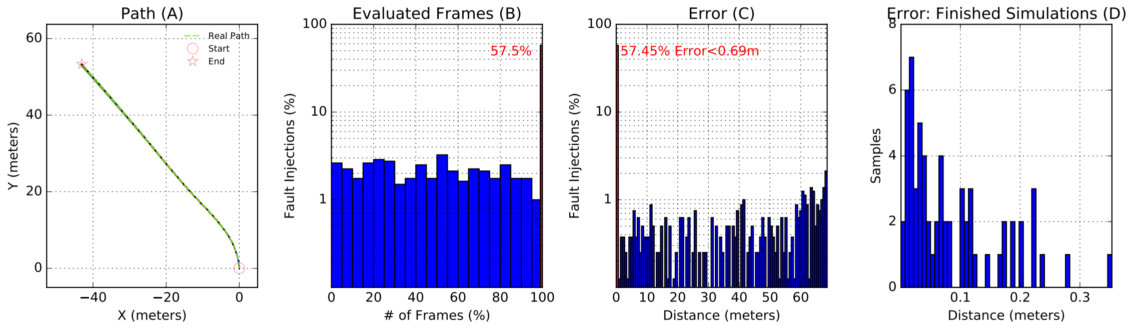
Altering the GCC optimization flags during the compilation is one possible solution to change the application characteristics. Each flag should change the compiler algorithm to improve assembly code in a specific manner. For example, the unroll loops option reduces the number of loop iterations, consequently, reducing the number of branch instructions. The next experiments explore the capabilities of the GCC to influence the application characteristics aiming to improve its reliability. Compilers, such as the GCC, have optimization flags enabled by default even when using the argument "-O0". The

Figure 5.17: Benchmark 11 initial exploration traveled route.

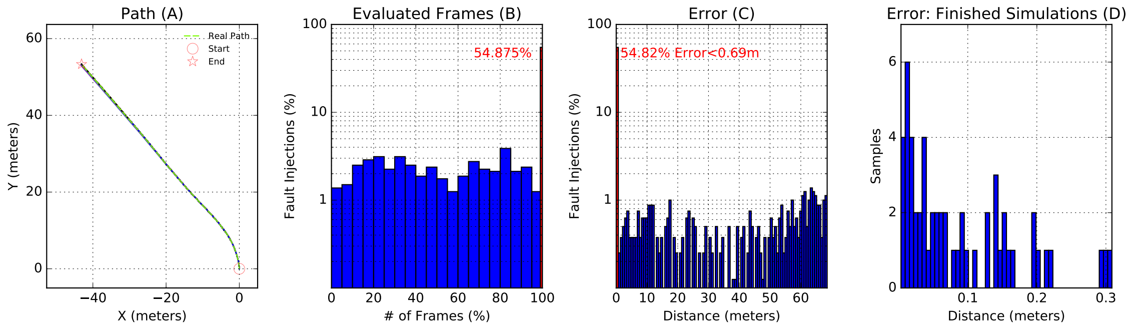
(a) O0.



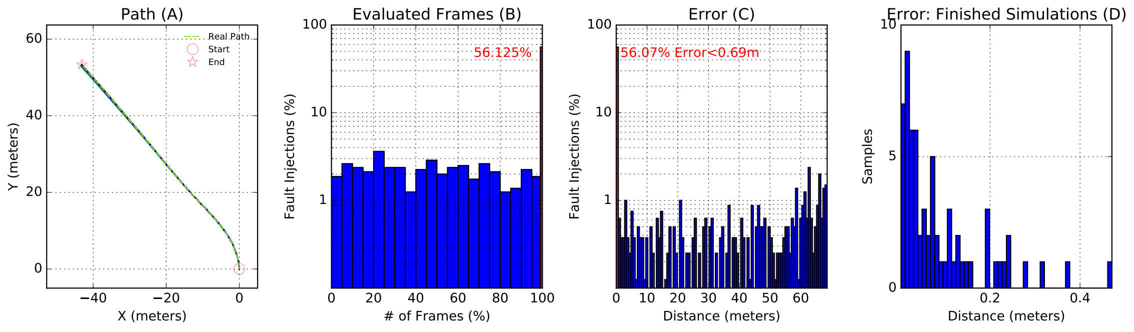
(b) O1.



(c) O2.



(d) O3.

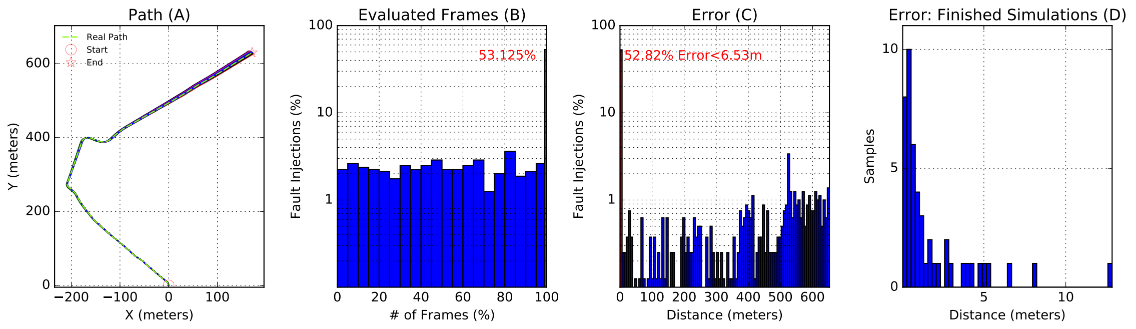


Source: The Authors

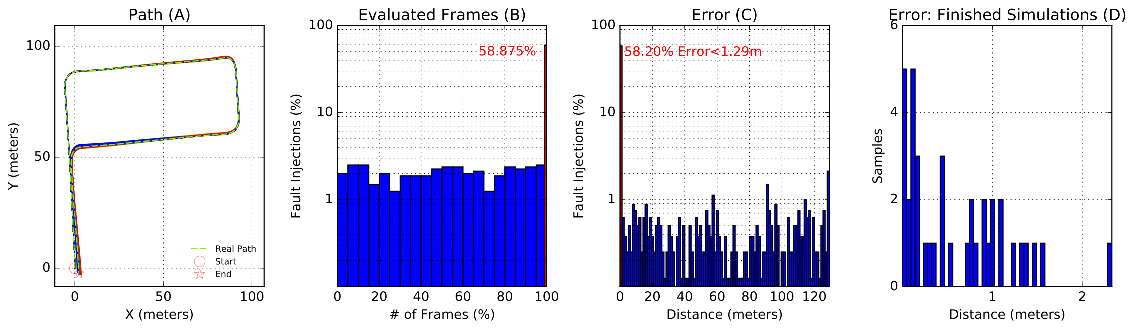
first fault campaign as depicted in Figure 5.19a shows the benchmark 11 compiled without any optimization flag under the influence of fault injections. Between the O0 flag and no optimization at all the final compiled code has not meaningful changes leading

Figure 5.18: Benchmarks complete execution.

(a) Benchmark 11.



(b) Benchmark 14.



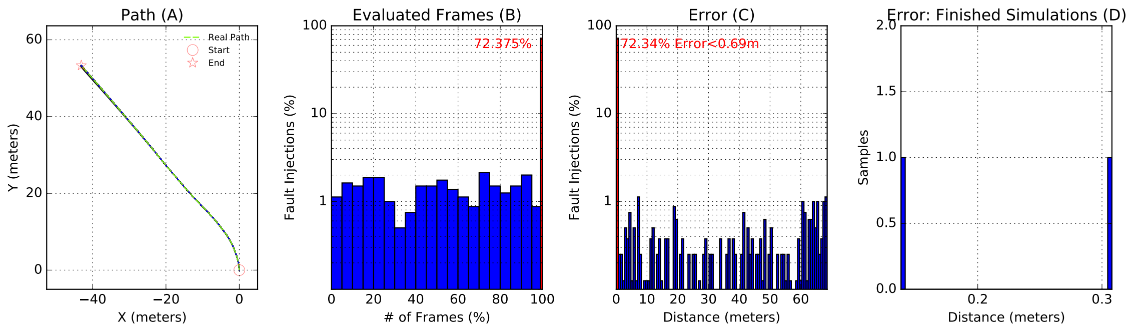
Source: The Authors

to almost identical simulations (i.e., both application execute 150 billion instructions). Figures 5.17a and 5.19a display this similarity considering the four parameters A-D.

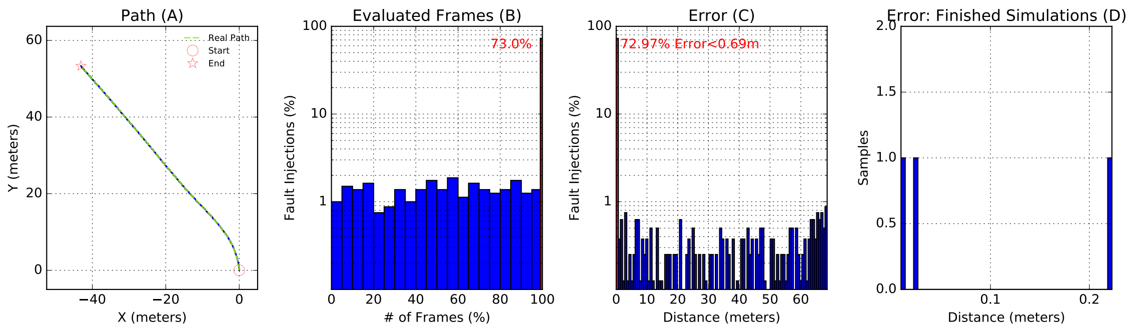
Previous results show a considerable reliability degradation when compiling with any other compilation (i.e., O1-3) while the O0 has a notable speed reduction (i.e., 4.7x). Further, the most significative effect occurs when transitioning from O0 to O1, and thus, we investigate how to improve the O0 performance by adding new compilation flags. The target application heavenly depends on loops to analyze the images, for this purpose, some unrolled loops options have been added to its compilation (Figure 5.19b). Also, the next experiment (Figure 5.19c) mimics the O3 option by using individual optimization flags instead (e.g., -falign-jumps -falign-loops -falign-labels -fcaller-saves -fcrossjumping). This investigation aims to replicate the O3 effect on the code without directly adding the "-O3" flag. Individual optimization flags have minimal impact on the final system reliability/performance as they have a minor impact on the compiler behavior. The main optimization flags (e.g., O0-3) are not just group alias, and they have a direct impact on the compiler algorithm. In other words, the GCC does not provide full control of these optimizations leaving the most relevant options hardcoded in the compiler source code. For example, the benchmark 11 without any optimization simulates 150 billion instructions while the O3 recreation reduces this amount by only 30 billion when considering 100 frames, still

Figure 5.19: Exploring the gcc optimization flags.

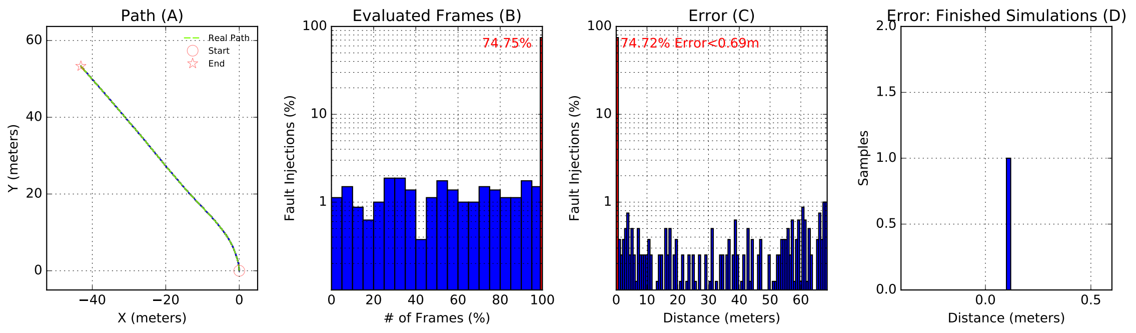
(a) No optimizations (i.e., removing all possible optimizations flags including the default ones).



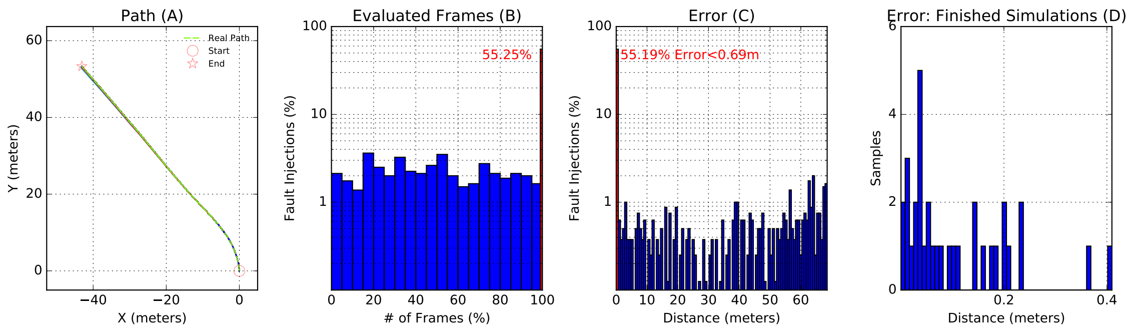
(b) O0 plus multiple unrolling loop options.



(c) Fabricated O3.



(d) O3 plus multiple unrolling loop options.



Source: The Authors

four times longer than a similar scenario compiled with the O1 flag.

After analyzing the impact of multiple GCC flags on the visual odometry application, we extended this exploration to the effects of the traveled path on the accumulated error. This exploration uses two input sets from the KITTI suite, the 11 featuring 920 frames and the 14 with 630 frames, and both compiled with the O3 optimization flag. Figure 5.18 shows the traveled path for each fault injection scenario (A) with a zoom on the final stopping point (B). The benchmark 11 follows a more straight path leading to horizontal errors, in other words, the vehicle diverges either to the right or the left (Figure 5.18a. In contrast, the benchmark 14 after each curve the traveled path leans towards the correct way (dashed green line) as right and left accumulated errors attenuate each other. However, the vertical error accumulates in a greater magnitude as shown by Figure 5.18bB. Figure 5.18 also displays the error from the correct stopping point considering every fault injection (C) and only the ones with complete (i.e., neither Hang or UT) application (D). Note how the number of completed simulations (in red) remains at the same levels of the reduced simulations (i.e., 100 frames), around 55% when compiled with an identical GCC flag (i.e., O3).

Most of the existing software projects use O2 as standard shipping optimization flag because a four times slowdown is not acceptable. In the impossibility of improving the software compiled with O0, the last experiment attempts to reduce the number of unexpected terminations when using the O3 option in the GCC. This fault campaign (Figure 5.19d) uses aggressive loop and branch optimizations, given more freedom to the compiler to allocate the instructions orders. Again, it demonstrates the GCC lack of support to algorithm customizations as most of the code transformations are locked to hardcoded arguments.

5.8 Closing Remarks

This chapter introduced the basic concepts of machine learning and its applicability to system development. Further, we reviewed the state-of-the-art ML approaches to improve the reliability assessment of large-scale systems. By observing the lack of methodologies targeting embedded multicore systems software stack, this Thesis proposes the use of ML algorithms to speed up dependability investigations during early design space explorations. We proposed a cross-layer investigation tool which performs multivariable and statistical analyses using the gem5 microarchitectural information combined with fault injection campaigns. This tool enables reducing the number of fault injection campaigns or improving the mitigation technique efficiency by observing microarchitectural symptoms. To explore the functionalities of the promoted development cycle, we investigated the reliability of a visual odometry algorithm common on self-driven cars.

6 CONCLUSION AND FUTURE WORKS

The continuous transistor shirking features lead to increasing system vulnerability to high-energy radiation particles causing malfunctions and abnormal behaviors. In this context, soft errors is an emerging concern in ground-level electronics devices such as smartphones and self-driven cars. Those systems deploy a complex software stack on top of commercial processor architectures, which does not possess an inherent fault tolerance technique. Investigating the system reliability since early design space explorations is fundamental to achieve a dependable system.

6.1 Contributions

This section briefly resumes this work contributions and achieved results.

6.1.1 Early soft error evaluation

This Thesis presented a flexible and fast simulation framework by including a pair of fault injectors with complementary characteristics. The instruction-accurate OVPsim provides a dynamic and adaptable environment for simulating a myriad of platforms. However, it lacks microarchitectural (e.g., pipeline) models limiting its utilization when considering hardware components explorations. To fulfill this gap, the gem5 covers microarchitectural modeling aspects and thus enabling an early architectural design space exploration. The precision leads to an order of magnitude in performance loss and a lack of scalability when comparing to the OVPsim.

HPC evaluations and emerging embedded applications require an early investigation of the soft error over multi/manycore systems. This work extends the framework from single to multi/manycore ARM Cortex processor architectures. For the sake of knowledge of the authors, this is the first work to address multi/manycore fault injection in virtual platforms. The simulation speed is fundamental to rapid investigate the reliability during early design stages. For this purpose, we include and enhance the simulation infrastructure to include new speed-up techniques such as multicore processors, checkpointing, and distributed simulation.

6.1.2 Fault Injection Controllability

This Thesis developed new fault injection techniques and analysis tools to trace the individual software component error source. For example, this extension enables fault injections onto the virtual memory ranges, functions, variables, among others with a completely automatic and transparent tool. Additionally, we enhance the traditional error analysis by including customizable inspections during the application execution. This enhanced tooling enables disentangling the cause and effect of soft errors from different software components.

6.1.3 Instruction-accurate fault injection consistency

To investigate the fault injection consistency between two instruction and cycle-accurate virtual platforms this work performed a comprehensive fault injection campaign. This evaluation comprised hundreds of scenarios targeting complex software stacks (i.e., Linux, parallelization libraries) and multiple hardware configurations (i.e., single, dual, quad, and octa-core ARM Cortex-A9 and A72 processor models).

6.1.4 Extensive investigation of the software stack impact on the system reliability

This work uses the proposed FI framework scalability to explore early design decisions impact on the system reliability, e.g., architecture, number of cores, ISA, OS, parallelization library, among other possible configurations. For this purpose, this work ports high-performance applications, including 35 OpenMP and MPI-based applications and ten serial from the Rodinia benchmarks and the NASA NAS Parallel Benchmark (NPB) suites. This work comprises 3,344,000 fault injections which require a total of 2 million simulation hours. During this extensive fault campaigns, we covered multi-core processors, different parallelization APIs, distinct simulation abstraction levels, and architectures.

6.1.5 Using machine learning techniques to explore the software stack reliability

This work proposed a cross-layer investigation tool to correlate microarchitectural parameters and fault injection campaigns. The investigation uses machine learning techniques such as linear regression to discover hidden relationships between two parameters. The developed tool helps to uncover critical parameters to the system dependability.

6.2 Future Works

This Thesis proposed collection of soft error evaluation tools enabling the rapid prototyping of complex systems. To take full advantage of this developed framework we propose two lines of research: Section 6.2.1 describes a self-aware soft error mitigation system. In Section 6.2.2, we outline the next steps to build a self-driven reliability assessment tool.

6.2.1 Soft Error Mitigation

Many/multicore system requires an adaptable and flexible methodology, to dynamically evaluate both application requirements and system state. The traditional techniques target a single-thread application with design-time built-in fault tolerance techniques. Emerging manycore systems require flexible and adaptable solutions, capable of dealing at runtime with the unpredictable occurrence of permanent and transient faults. In this regard, a checkpoint and restore scheme, called *SafetyNet* (SORIN et al., 2002), targeting multiprocessor architecture is proposed. SafetyNet includes 512 kb Checkpoint Log Buffers (CLBs) to store the node information. The memory consistency uses a logical time technique to atomize each transaction. Also, the fault detection occurs in parallel with the normal execution to avoid synchronization barriers. The SafetyNet performance is tightly coupled with its design parameters: checkpoint granularity and CLB size. Using a one-million cycles interval and 512 kb dedicated memory leads to a 0.3% overhead. In contrast, a 5,000 cycles interval achieves a 4% overhead, and a 256 kb CLB has a performance degradation up to 50%. (PRVULOVIC; ZHANG; TORRELLAS, 2002) proposes the *ReVive* a hardware trade-off between dedicated hardware and performance. The RAM stores the checkpoint logs and the cache do not require any modification. For instance, the fault-free overhead achieves up to 22%.

DepMan (KOKOLIS et al., 2016) is a python-based checkpoint and restore at application level. It stores periodical checkpoints for manycore applications without extra hardware. The DepMan framework continuously evaluates the error rate at the target application to estimate the optimal checkpoint interval. Additionally, using the error rate information the framework improves the application performance by controlling the DVFS module.

In a multiprocessor system, the task mapping plays a major role in the system performance and reliability. Task mapping algorithms distribute and assign working threads among the available resources (i.e., processing units - PE). This algorithm should take into account parameters such as task communication, length, and execution pattern as well the system environment conditions like the workload distribution, temperature, DVFS, among others. In recent years the academic begins to explore mapping heuristics aim system reliability improve. Chen *et al.* (CHEN et al., 2016) proposes reliability-aware mapping algorithm based on Hungarian Algorithm. The algorithm includes the core-to-core frequency on an iso-ISA manycore system, multi-version applications, dateline awareness, and redundant multithreading. In a similar approach, (DAS et al., 2014) promotes a multi-goal mapping algorithm considering the aging effects and voltage-frequency variation into the application soft error tolerance. Also, it deploys selective redundant multithreading to increase the application reliability. The main mapping algorithm uses a genetic sorting algorithm with the considered parameters.

Task migration mechanisms can be explored to mitigate the soft error occurrence. On (CHAKRAVORTY; MENDES; KALé, 2006), the solution employs the virtual processor concept to abstract an MPI applications. A manager maps the virtual processor to physical processor, which has the inherent capability to migrate virtual processor. Tasks are migrated whenever a fault warning arises. The MADNESS (MELONI et al., 2012) approach includes a self-test hardware to detect permanent faults and TMR replications. After an error detection, each processor has a built-in task migration hardware to move the task to another processor.

Bolchini *et al.* (BOLCHINI; MIELE; SCIUTO, 2012) proposes an online and adaptive fault tolerance approach for a P2012 16-core cluster system. This approach modifies the task dispatcher to create three task copies automatically. Also, to avoid bottlenecks in the cluster and excessive wearing, several *healthy tables* compute the overall utilization. In a posterior extension (BOLCHINI; CARMINATI; MIELE, 2013), the adaptation engine includes four three distinct tolerance techniques: Duplication with Comparison (DWC), Duplication with Comparison and Re-execution (DWCR), and Triplication (TMR). Moreover, the control system incorporates an Observe–Decide–Act (ODA) control loop (LAFRIEDA et al., 2007) to introduces the self-adaptive engine. This engine defines the protection level and granularity on-the-fly for the application considering the system status. However, the application should be modeled using a task-graph to provide the necessary information to the engine algorithm.

We propose a combined hardware and software solution to improve the soft errors mitigation of multi/manycore systems through the use of thread-redundancy. The target core is a general-purpose COTS processor architecture without any inherent hardware fault tolerance components built-in. The proposed self-adaptive soft error mitigation framework consists of series of protection techniques and error detection mechanisms working collectively to improve the application and system dependability. This solution relies on a mitigation library for the application and a manager module embedded into the main core operating systems. This mitigation technique decides on-the-fly the protection granularity depending on the system environment (e.g., temperature, workload) and user constraints regarding the application criticality, dateline, latency, among others. Advising the redundancy coarseness incurs several challenges regarding performance, energy efficiency, and intrusiveness. For instance, replicating the entire application induces in larger overheads. To alleviate this issue, the soft error mitigation into software level usually employs simple code replication by the use of explicit parallelization into the source code. In other words, the software designer should explicitly impose the parallelization granularity, APIs, and configurations. Among the main advantages of this proposed self-adaptive soft error mitigation framework are:

(i) **Flexible software stack**: It is easily portable to other architectures as thread redundant fault tolerance techniques that do not require any ISA previous knowledge or modification.

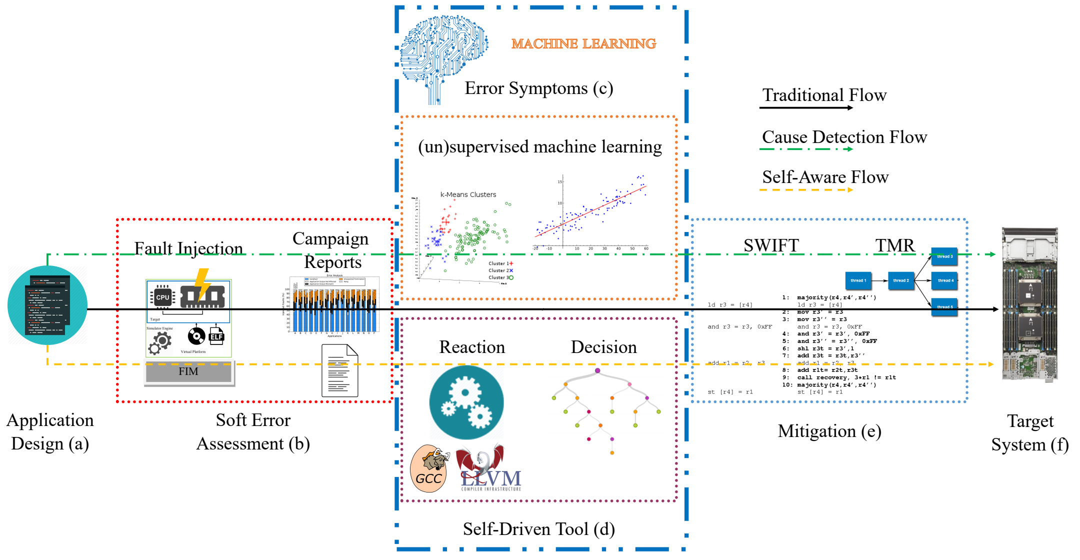
(ii) **Adaptable hardware stack**: The components are easily adaptable to other architectures such as ARMv7, RISCV or MIPS. No hardware alteration (e.g., hardware duplication or additional instructions) is required enabling the use of COTS components.

(iii) **Modularity**: Besides the thread replication technique, other mitigation algorithms can be easily included in the system design.

6.2.2 Self-Driven Reliability Assessment and Reaction

Machine learning provides a powerful framework to investigate large quantities of data, which has been presented on Chapter 5. Figure 6.1 display the traditional application development cycle without any modifications (a,b, e and f) and the proposed solution. This Thesis developed an investigation tool that deploys supervised and unsupervised machine learning techniques to correlate profiling information with fault injection campaigns as described on Section 5.3. Further, the adopted machine learning approach can be further improved by using an automatic grouping algorithm (e.g., k-means) to split complex features into small subsets. This tool objective is to find the most relevant parameters helping the software engineer to understand the source of errors and improve its system reliability. The developed algorithm shows the relationship strength between any two parameters, however, modifications are a user responsibility.

Figure 6.1: Proposed application design cycle using reinforced learning.



Source: The Authors

ML has the ability to create reactive and intelligent algorithms by learning from previous experiences. We propose a second ML development phase which provides a self-driven and pro-active tool using reinforced machine learning approaches. In other words, iteratively improving the target system reliability. This tool collects information from the previous phases (Figure 5.1b-c) to build a knowledge database from which it will manage the next exploration step. The tool algorithm has two main parts: *Decision making* and *Reaction*. A decision tree provides the 'intelligence' to steer the investigation taking in to account a cost function for each complete iterations. Successive executions increase should maximize the application reliability considering several parameters and goals defined by the software engineer. The acquired information will be supplied to a decision tree (or similar) algorithm deciding the best mitigation technique to be applied in the next iteration. After choosing an action, the tool changes the application by applying automatic GCC optimization flags (e.g., O3, funroll-all-loops) to improve the system reliability. For instance, after observing a high concentration of *Hangs* the tool decides to recompile the target application using an unrolling loop flag[1]. Following the previous section mitigation technique with variable thread/task replication is also applicable in such situation. Considering the adopted profiling tools and user information about the application behavior, it is possible to selective protect the target algorithm. Further, we plan to study and develop an LLVM reliability optimization module using the compiler intermediary representation (IR). IR enables a better control on the performance optimizations and facilitates the inclusion of other mitigation techniques at instruction-level. The IR optimization would be assigned by the system intelligence (i.e., decision tree) from a pool of LLVM optimizations.

[1]Loop unrolling, also known as loop unwinding, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size. The goal of loop unwinding is to increase a program's speed by reducing or eliminating instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration. To remove this computational overhead, loops can be re-written as a repeated sequence of similar independent statements. From https://en.wikipedia.org/wiki/Loop_unrolling.

REFERENCES

AKERKAR, R.; SAJJA, P. S. **Intelligent Techniques for Data Science**. Springer International Publishing, 2016. ISBN 978-3-319-29205-2. Available from Internet: <//www.springer.com/gp/book/9783319292052>.

ALAM, M. A. et al. A comprehensive model for PMOS NBTI degradation: Recent progress. **Microelectronics Reliability**, v. 47, n. 6, p. 853–862, jun. 2007. ISSN 0026-2714. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0026271406003751>.

ALIAN, M.; KIM, D.; KIM, N. S. pd-gem5: Simulation Infrastructure for Parallel/Distributed Computer Systems. **IEEE Computer Architecture Letters**, v. 15, n. 1, p. 41–44, jan. 2016. ISSN 1556-6056.

ARM. **Technologies | big.LITTLE**. 2017. Available from Internet: <https://developer.arm.com/technologies/big-little>.

ARTOLA, L.; HUBERT, G.; SCHRIMPF, R. Modeling of radiation-induced single event transients in SOI FinFETS. In: **Reliability Physics Symposium (IRPS), 2013 IEEE International**. [S.l.: s.n.], 2013. p. SE.1.1–SE.1.6.

ASHRAF, R. A. et al. Understanding the Propagation of Transient Errors in HPC Applications. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: ACM, 2015. (SC '15), p. 72:1–72:12. ISBN 978-1-4503-3723-6. Available from Internet: <http://doi.acm.org/10.1145/2807591.2807670>.

AUSTIN, T.; LARSON, E.; ERNST, D. SimpleScalar: an infrastructure for computer system modeling. **Computer**, v. 35, n. 2, p. 59–67, feb. 2002. ISSN 0018-9162.

BAILEY, D. H. et al. The NAS parallel benchmarks summary and preliminary results. In: **Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)**. [S.l.: s.n.], 1991. p. 158–165.

BARAZA, J. et al. A prototype of a VHDL-based fault injection tool. In: **IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2000. Proceedings**. [S.l.: s.n.], 2000. p. 396–404.

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

BAUTISTA-GOMEZ, L. et al. Unprotected Computing: A Large-Scale Study of DRAM Raw Error Rate on a Supercomputer. In: **SC16: International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2016. p. 645–655.

BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. **Proceedings of the Annual Conference on USENIX Annual Technical Conference**, p. 41–41, 2005. Available from Internet: <http://dl.acm.org/citation.cfm?id=1247360.1247401>.

BELTRAME, G. et al. A Framework for Reliability Assessment and Enhancement in Multi-Processor Systems-On-Chip. In: **22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)**. [S.l.: s.n.], 2007. p. 132–142.

BELTRAME, G.; BOLCHINI, C.; MIELE, A. Multi-level Fault Modeling for Transaction-level Specifications. In: **Proceedings of the 19th ACM Great Lakes Symposium on VLSI**. New York, NY, USA: ACM, 2009. (GLSVLSI '09), p. 87–92. ISBN 978-1-60558-522-2. Available from Internet: <http://doi.acm.org/10.1145/1531542.1531565>.

BELTRAME, G.; FOSSATI, L. ReSP: A Design and Validation Tool for Data Systems. In: **Data Systems In Aerospace, 2008. DASIA 2008**. [s.n.], 2008. v. 665, p. 22. ISBN 1609-042X. Available from Internet: <http://adsabs.harvard.edu/abs/2008ESASP.665E..22B>.

BIENIA, C. et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: **Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques**. New York, NY, USA: ACM, 2008. (PACT '08), p. 72–81. ISBN 978-1-60558-282-5. Available from Internet: <http://doi.acm.org/10.1145/1454115.1454128>.

BINDER, D.; SMITH, E. C.; HOLMAN, A. B. Satellite Anomalies from Galactic Cosmic Rays. **IEEE Transactions on Nuclear Science**, v. 22, n. 6, p. 2675–2680, dec. 1975. ISSN 0018-9499.

BINKERT, N. et al. The Gem5 Simulator. **SIGARCH Comput. Archit. News**, v. 39, n. 2, p. 1–7, aug. 2011. ISSN 0163-5964. Available from Internet: <http://doi.acm.org/10.1145/2024716.2024718>.

BISHOP, R. **Intelligent Vehicle Technology And Trends**. Boston: Artech House, 2005. ISBN 978-1-58053-911-1.

BOLCHINI, C.; CARMINATI, M.; MIELE, A. Self-Adaptive Fault Tolerance in Multi-/Many-Core Systems. **J. Electron. Test.**, v. 29, n. 2, p. 159–175, abr. 2013. ISSN 0923-8174. Available from Internet: <http://dx.doi.org/10.1007/s10836-013-5367-y>.

BOLCHINI, C.; MIELE, A.; SCIUTO, D. An adaptive approach for online fault management in many-core architectures. In: **2012 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2012. p. 1429–1432.

BORKAR, S.; CHIEN, A. A. The future of microprocessors. **Communications of the ACM**, v. 54, n. 5, p. 67, may 2011. ISSN 00010782. Available from Internet: <http://cacm.acm.org/magazines/2011/5/107702-the-future-of-microprocessors/fulltext>.

BORKAR, S. et al. Parameter variations and impact on circuits and microarchitecture. In: **Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)**. [S.l.: s.n.], 2003. p. 338–342.

BORODIN, D.; JUURLINK, B. H. Protective Redundancy Overhead Reduction Using Instruction Vulnerability Factor. In: **Proceedings of the 7th ACM International Conference on Computing Frontiers**. New York, NY, USA: ACM, 2010. (CF '10), p. 319–326. ISBN 978-1-4503-0044-5. Available from Internet: <http://doi.acm.org/10.1145/1787275.1787342>.

BUCHANAN, S. **NOAA kicks off 2018 with massive supercomputer upgrade | National Oceanic and Atmospheric Administration**. 2018. Available from Internet: <http://www.noaa.gov/media-release/noaa-kicks-off-2018-with-massive-supercomputer-upgrade>.

BUTKO, A. et al. Accuracy evaluation of GEM5 simulator system. In: **2012 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)**. [S.l.: s.n.], 2012. p. 1–7.

CHAKRAVORTY, S.; MENDES, C. L.; KALé, L. V. Proactive Fault Tolerance in MPI Applications Via Task Migration. In: **High Performance Computing - HiPC 2006**. Springer, Berlin, Heidelberg, 2006. (Lecture Notes in Computer Science), p. 485–496. ISBN 978-3-540-68039-0 978-3-540-68040-6. Available from Internet: <https://link.springer.com/chapter/10.1007/11945918_47>.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: **2009 IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.: s.n.], 2009. p. 44–54.

CHEN, K. H. et al. Task Mapping for Redundant Multithreading in Multi-Cores with Reliability and Performance Heterogeneity. **IEEE Transactions on Computers**, v. 65, n. 11, p. 3441–3455, nov. 2016. ISSN 0018-9340.

CHO, H. et al. Quantitative evaluation of soft error injection techniques for robust system design. In: **2013 50th ACM / EDAC / IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2013. p. 1–10.

DAS, A. et al. Combined DVFS and mapping exploration for lifetime and soft-error susceptibility improvement in MPSoCs. In: **2014 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2014. p. 1–6.

DIDEHBAN, M.; SHRIVASTAVA, A. nZDC: A Compiler Technique for Near Zero Silent Data Corruption. In: **Proceedings of the 53rd Annual Design Automation Conference**. New York, NY, USA: ACM, 2016. (DAC '16), p. 48:1–48:6. ISBN 978-1-4503-4236-0. Available from Internet: <http://doi.acm.org/10.1145/2897937.2898054>.

DINECHIN, B. de et al. A clustered manycore processor architecture for embedded and accelerated applications. In: **2013 IEEE High Performance Extreme Computing Conference (HPEC)**. [S.l.: s.n.], 2013. p. 1–6.

DITOMASO, D. et al. Dynamic error mitigation in NoCs using intelligent prediction techniques. In: **2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2016. p. 1–12.

DIXIT, A.; WOOD, A. The impact of new technology on soft error rates. In: **2011 International Reliability Physics Symposium**. [S.l.: s.n.], 2011. p. 5B.4.1–5B.4.7.

DODD, P. E.; SEXTON, F. W. Critical charge concepts for CMOS SRAMs. **IEEE Transactions on Nuclear Science**, v. 42, n. 6, p. 1764–1771, dec. 1995. ISSN 0018-9499.

DOYLE, B. et al. Tri-Gate fully-depleted CMOS transistors: fabrication, design and layout. In: **2003 Symposium on VLSI Technology, 2003. Digest of Technical Papers**. [S.l.: s.n.], 2003. p. 133–134.

EBNENASIR, A.; HAJISHEYKHI, R.; KULKARNI, S. S. Facilitating the Design of Fault Tolerance in Transaction Level SystemC Programs. In: BONONI, L. et al. (Ed.). **Distributed Computing and Networking**. Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, 7129). p. 91–105. ISBN 978-3-642-25958-6 978-3-642-25959-3. Available from Internet: <http://link.springer.com/chapter/10.1007/978-3-642-25959-3_7>.

EBRAHIMI, M. et al. Layout-Based Modeling and Mitigation of Multiple Event Transients. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 35, n. 3, p. 367–379, mar. 2016. ISSN 0278-0070.

ESMAEILZADEH, H. et al. Power Limitations and Dark Silicon Challenge the Future of Multicore. **ACM Trans. Comput. Syst.**, v. 30, n. 3, p. 11:1–11:27, aug. 2012. ISSN 0734-2071. Available from Internet: <http://doi.acm.org/10.1145/2324876.2324879>.

ESMAEILZADEH, H. et al. Dark silicon and the end of multicore scaling. In: **2011 38th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2011. p. 365–376.

GARIBOTTI, R. et al. Simultaneous multithreading support in embedded distributed memory MPSoCs. In: **2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2013. p. 1–7.

GEIGER, A.; LENZ, P.; URTASUN, R. Are we ready for autonomous driving? The KITTI vision benchmark suite. In: **2012 IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2012. p. 3354–3361.

GEIGER, A.; LENZ, P.; URTASUN, R. Are we ready for autonomous driving? the kitti vision benchmark suite. In: **Conference on Computer Vision and Pattern Recognition (CVPR)**. [S.l.: s.n.], 2012.

GEIGER, A.; ZIEGLER, J.; STILLER, C. Stereoscan: Dense 3d reconstruction in real-time. In: **Intelligent Vehicles Symposium (IV)**. [S.l.: s.n.], 2011.

GEISSLER, F. de A.; KASTENSMIDT, F. L.; SOUZA, J. P. Soft error injection methodology based on QEMU software platform. In: **Test Workshop - LATW, 2014 15th Latin American**. [S.l.: s.n.], 2014. p. 1–5.

GIURGIU, I. et al. Predicting DRAM Reliability in the Field with Machine Learning. In: **Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track**. New York, NY, USA: ACM, 2017. (Middleware '17), p. 15–21. ISBN 978-1-4503-5200-0. Available from Internet: <http://doi.acm.org/10.1145/3154448.3154451>.

GRANLUND, T.; GRANBOM, B.; OLSSON, N. Soft error rate increase for new generations of SRAMs. **IEEE Transactions on Nuclear Science**, v. 50, n. 6, p. 2065–2068, dec. 2003. ISSN 0018-9499.

GROPP, W.; THAKUR, R.; LUSK, E. **Using MPI-2: Advanced Features of the Message Passing Interface**. 2nd. ed. Cambridge, MA, USA: MIT Press, 1999. ISBN 978-0-262-57134-0.

GUAN, Q. et al. Design, Use and Evaluation of P-FSEFI: A Parallel Soft Error Fault Injection Framework for Emulating Soft Errors in Parallel Applications. In: **Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques**. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016. (SIMUTOOLS'16), p. 9–17. ISBN 978-1-63190-120-1. Available from Internet: <http://dl.acm.org/citation.cfm?id=3021426.3021429>.

GUAN, Q. et al. F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability. In: **2014 IEEE 28th International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2014. p. 1245–1254.

GUTHAUS, M. et al. MiBench: A free, commercially representative embedded benchmark suite. In: **2001 IEEE International Workshop on Workload Characterization, 2001. WWC-4**. [S.l.: s.n.], 2001. p. 3–14.

GUTIERREZ, A. et al. Sources of error in full-system simulation. In: **2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.: s.n.], 2014. p. 13–22.

HARI, S. K. S. et al. 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**. New York, NY, USA: ACM, 2012. (ASPLOS XVII), p. 123–134. ISBN 978-1-4503-0759-8. Available from Internet: <http://doi.acm.org/10.1145/2150976.2150990>.

HARI, S. K. S. et al. GangES: Gang Error Simulation for Hardware Resiliency Evaluation. In: **Proceeding of the 41st Annual International Symposium on Computer Architecuture**. Piscataway, NJ, USA: IEEE Press, 2014. (ISCA '14), p. 61–72. ISBN 978-1-4799-4394-4. Available from Internet: <http://dl.acm.org/citation.cfm?id=2665671.2665685>.

HASHIMOTO, M.; LIAO, W.; HIROKAWA, S. Soft error rate estimation with TCAD and machine learning. In: **2017 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)**. [S.l.: s.n.], 2017. p. 129–132.

HENKEL, J. et al. Reliable On-chip Systems in the Nano-era: Lessons Learnt and Future Trends. In: **Proceedings of the 50th Annual Design Automation Conference**. New York, NY, USA: ACM, 2013. (DAC '13), p. 99:1–99:10. ISBN 978-1-4503-2071-9. Available from Internet: <http://doi.acm.org/10.1145/2463209.2488857>.

HENNING, J. L. SPEC CPU2006 Benchmark Descriptions. **SIGARCH Comput. Archit. News**, v. 34, n. 4, p. 1–17, sep. 2006. ISSN 0163-5964. Available from Internet: <http://doi.acm.org/10.1145/1186736.1186737>.

HUBERT, G.; ARTOLA, L. Single-Event Transient Modeling in a 65-nm Bulk CMOS Technology Based on Multi-Physical Approach and Electrical Simulations. **IEEE Transactions on Nuclear Science**, v. 60, n. 6, p. 4421–4429, dec. 2013. ISSN 0018-9499.

HUBERT, G.; ARTOLA, L.; REGIS, D. Impact of scaling on the soft error sensitivity of bulk, FDSOI and FinFET technologies due to atmospheric radiation. **Integration, the VLSI Journal**, v. 50, p. 39–47, jun. 2015. ISSN 0167-9260.

IBE, E. et al. Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule. **IEEE Transactions on Electron Devices**, v. 57, n. 7, p. 1527–1538, jul. 2010. ISSN 0018-9383.

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

Inside HPC. **Supercomputers Unlocking Mysteries of the Subatomic World**. 2018. Available from Internet: <https://insidehpc.com/2018/05/supercomputers-unlocking-mysteries-subatomic-world/>.

JOHANSSON, K. et al. Neutron induced single-word multiple-bit upset in SRAM. **IEEE Transactions on Nuclear Science**, v. 46, n. 6, p. 1427–1433, dec. 1999. ISSN 0018-9499.

JONES, W. D. **Building Safer Cars**. 2002. Available from Internet: <http://spectrum.ieee.org/transportation/advanced-cars/building-safer-cars>.

KAHNG, A. B. The ITRS design technology and system drivers roadmap: Process and status. In: **2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2013. p. 1–6.

KALIORAKIS, M. et al. Differential Fault Injection on Microarchitectural Simulators. In: **2015 IEEE International Symposium on Workload Characterization**. [S.l.: s.n.], 2015. p. 172–182.

KANTER, D. **ARM Goes 64-bit**. 2012. Available from Internet: <http://www.realworldtech.com/arm64/>.

KARNIK, T.; HAZUCHA, P. Characterization of soft errors caused by single event upsets in CMOS processes. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 2, p. 128–143, abr. 2004. ISSN 1545-5971.

KOKOLIS, A. et al. Runtime interval optimization and dependable performance for application-level checkpointing. In: **2016 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2016. p. 594–599.

KOOLI, M.; NATALE, G. D. A survey on simulation-based fault injection tools for complex systems. In: **Design Technology of Integrated Systems In Nanoscale Era (DTIS), 2014 9th IEEE International Conference On**. [S.l.: s.n.], 2014. p. 1–6.

KOUROU, K. et al. Machine learning applications in cancer prognosis and prediction. **Computational and Structural Biotechnology Journal**, v. 13, p. 8–17, jan. 2015. ISSN 2001-0370. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S2001037014000464>.

LAFRIEDA, C. et al. Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor. In: **37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)**. [S.l.: s.n.], 2007. p. 317–326.

LI, T. et al. Processor Design for Soft Errors: Challenges and State of the Art. **ACM Comput. Surv.**, v. 49, n. 3, p. 57:1–57:44, nov. 2016. ISSN 0360-0300. Available from Internet: <http://doi.acm.org/10.1145/2996357>.

MAGNUSSON, P. S. et al. Simics: A Full System Simulation Platform. **Computer**, v. 35, n. 2, p. 50–58, feb. 2002. ISSN 0018-9162. Available from Internet: <http://dx.doi.org/10.1109/2.982916>.

MARTIN, M. M. K. et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. **SIGARCH Comput. Archit. News**, v. 33, n. 4, p. 92–99, nov. 2005. ISSN 0163-5964. Available from Internet: <http://doi.acm.org/10.1145/1105734.1105747>.

MAY, T. C.; WOODS, M. H. A New Physical Mechanism for Soft Errors in Dynamic Memories. In: **16th International Reliability Physics Symposium**. [S.l.: s.n.], 1978. p. 33–40.

MCCLUSKEY, B. Connected cars - the security challenge [Connected Cars Cyber Security]. **Engineering Technology**, v. 12, n. 2, p. 54–57, mar. 2017. ISSN 1750-9637.

MCKINNEY, W. pandas: a foundational python library for data analysis and statistics. **Python for High Performance and Scientific Computing**, 2011.

MELONI, P. et al. System Adaptivity and Fault-Tolerance in NoC-based MPSoCs: The MADNESS Project Approach. In: **2012 15th Euromicro Conference on Digital System Design**. [S.l.: s.n.], 2012. p. 517–524.

MISERA, S. et al. A Mixed Language Fault Simulation of VHDL and SystemC. In: **9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006**. [S.l.: s.n.], 2006. p. 275–279.

MOORE, G. E. Cramming More Components onto Integrated Circuits. **Electronics**, IEEE, v. 38, n. 8, p. 114–117, abr. 1965. ISSN 0018-9219. Available from Internet: <http://dx.doi.org/10.1109/jproc.1998.658762>.

MUKHERJEE, S. **Architecture Design for Soft Errors**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 978-0-08-055832-5 978-0-12-369529-1.

MUKHERJEE, S. S. et al. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In: **Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 2003. (MICRO 36), p. 29–. ISBN 978-0-7695-2043-8. Available from Internet: <http://dl.acm.org/citation.cfm?id=956417.956570>.

NIE, B. et al. Characterizing Temperature, Power, and Soft-Error Behaviors in Data Center Systems: Insights, Challenges, and Opportunities. In: **2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)**. [S.l.: s.n.], 2017. p. 22–31.

NORDRUM, A. **Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated**. 2016. Available from Internet: <http://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated>.

NORMAND, E. et al. First Record of Single-Event Upset on Ground, Cray-1 Computer at Los Alamos in 1976. **IEEE Transactions on Nuclear Science**, v. 57, n. 6, p. 3114–3120, dec. 2010. ISSN 0018-9499.

PALAU, J.-M. et al. Device simulation study of the SEU sensitivity of SRAMs to internal ion tracks generated by nuclear reactions. **IEEE Transactions on Nuclear Science**, v. 48, n. 2, p. 225–231, abr. 2001. ISSN 0018-9499.

PANDINI, D. Variability in Advanced Nanometer Technologies: Challenges and Solutions. In: **Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation**. Springer, Berlin, Heidelberg, 2009. p. 2–2. Available from Internet: <https://link.springer.com/chapter/10.1007/978-3-642-11802-9_2>.

PATEL, A. et al. MARSS: A Full System Simulator for Multicore x86 CPUs. In: **Proceedings of the 48th Design Automation Conference**. New York, NY, USA: ACM, 2011. (DAC '11), p. 1050–1055. ISBN 978-1-4503-0636-2. Available from Internet: <http://doi.acm.org/10.1145/2024724.2024954>.

PEDREGOSA, F. et al. Scikit-learn: Machine Learning in Python. **J. Mach. Learn. Res.**, v. 12, p. 2825–2830, nov. 2011. ISSN 1532-4435. Available from Internet: <http://dl.acm.org/citation.cfm?id=1953048.2078195>.

PERERA, C. et al. Context Aware Computing for The Internet of Things: A Survey. **IEEE Communications Surveys Tutorials**, v. 16, n. 1, p. 414–454, 2014. ISSN 1553-877X.

PRVULOVIC, M.; ZHANG, Z.; TORRELLAS, J. ReVive: Cost-effective Architectural Support for Rollback Recovery in Shared-memory Multiprocessors. In: **Proceedings of the 29th Annual International Symposium on Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 2002. (ISCA '02), p. 111–122. ISBN 978-0-7695-1605-9. Available from Internet: <http://dl.acm.org/citation.cfm?id=545215.545228>.

RAMACHANDRAN, P. et al. Statistical Fault Injection. In: **IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008**. [S.l.: s.n.], 2008. p. 122–127.

REHMAN, S. et al. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In: **2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)**. [S.l.: s.n.], 2011. p. 237–246.

RIGO, S. et al. ArchC: a systemC-based architecture description language. In: **16th Symposium on Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004**. [S.l.: s.n.], 2004. p. 66–73.

ROSA, F. R. et al. Impact of dynamic voltage scaling and thermal factors on FinFET-based SRAM reliability. In: **2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)**. [S.l.: s.n.], 2015. p. 137–140.

ROSA, F. R. et al. Impact of dynamic voltage scaling and thermal factors on SRAM reliability. **Microelectronics Reliability**, v. 55, n. 9–10, p. 1486–1490, aug. 2015. ISSN 0026-2714. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0026271415301001>.

RUANO, O. et al. A Simulation Platform for the Study of Soft Errors on Signal Processing Circuits through Software Fault Injection. In: **IEEE International Symposium on Industrial Electronics, 2007. ISIE 2007**. [S.l.: s.n.], 2007. p. 3316–3321.

RUSS, C. ESD issues in advanced CMOS bulk and FinFET technologies: Processing, protection devices and circuit strategies. **Microelectronics Reliability**, v. 48, n. 8–9, p. 1403–1411, aug. 2008. ISSN 0026-2714.

SAMUEL, A. L. Some Studies in Machine Learning Using the Game of Checkers. **IBM Journal of Research and Development**, v. 3, n. 3, p. 210–229, jul. 1959. ISSN 0018-8646.

SEIFERT, N. Radiation-induced Soft Errors: A Chip-level Modeling Perspective. **Found. Trends Electron. Des. Autom.**, v. 4, n. 2-3, p. 99–221, feb. 2010. ISSN 1551-3939. Available from Internet: <http://dx.doi.org/10.1561/1000000018>.

SEIFERT, N. et al. Soft Error Rate Improvements in 14-nm Technology Featuring Second-Generation 3d Tri-Gate Transistors. **IEEE Transactions on Nuclear Science**, v. 62, n. 6, p. 2570–2577, dec. 2015. ISSN 0018-9499.

SHAFIK, R.; ROSINGER, P.; AL-HASHIMI, B. SystemC-Based Minimum Intrusive Fault Injection Technique with Improved Fault Representation. In: **On-Line Testing Symposium, 2008. IOLTS '08. 14th IEEE International**. [S.l.: s.n.], 2008. p. 99–104.

SHAO, Y. S. et al. Co-designing accelerators and SoC interfaces using gem5-Aladdin. In: **2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2016. p. 1–12.

SHIVAKUMAR, P. et al. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In: **Proceedings of the 2002 International Conference on Dependable Systems and Networks**. Washington, DC, USA: IEEE Computer Society, 2002. (DSN '02), p. 389–398. ISBN 978-0-7695-1597-7. Available from Internet: <http://dl.acm.org/citation.cfm?id=647883.738394>.

SLAYMAN, C. Soft errors Past history and recent discoveries. In: **2010 IEEE International Integrated Reliability Workshop Final Report**. [S.l.: s.n.], 2010. p. 25–30.

SORIN, D. J. et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In: **Proceedings of the 29th Annual International Symposium on Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 2002. (ISCA '02), p. 123–134. ISBN 978-0-7695-1605-9. Available from Internet: <http://dl.acm.org/citation.cfm?id=545215.545229>.

SUBASI, O. et al. MACORD: Online Adaptive Machine Learning Framework for Silent Error Detection. In: **2017 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.: s.n.], 2017. p. 717–724.

TANIKELLA, K. et al. gemV: A validated toolset for the early exploration of system reliability. In: **2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)**. [S.l.: s.n.], 2016. p. 159–163.

UNPINGCO, J. **Python for Probability, Statistics, and Machine Learning**. Springer International Publishing, 2016. ISBN 978-3-319-30715-2. Available from Internet: <//www.springer.com/gp/book/9783319307152>.

VALDERAS, M. et al. Advanced Simulation and Emulation Techniques for Fault Injection. In: **IEEE International Symposium on Industrial Electronics, 2007. ISIE 2007**. [S.l.: s.n.], 2007. p. 3339–3344.

VISHNU, A. et al. Fault Modeling of Extreme Scale Applications Using Machine Learning. In: **2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2016. p. 222–231.

VLACIC, L.; PARENT, M.; HARASHIMA, F. **Intelligent Vehicle Technologies: Theory and Applications**. [S.l.]: Butterworth-Heinemann, 2001. Google-Books-ID: oL6M0U3QEacC. ISBN 978-0-7506-5093-9.

WALLMARK, J. T.; MARCUS, S. M. Maximum packing density and minimum size of semiconductor devices. In: **1961 International Electron Devices Meeting**. [S.l.: s.n.], 1961. v. 7, p. 34–34.

WATT, T. **ARM Discloses Technical Details Of The Next Version Of The... - ARM**. 2011. Available from Internet: <https://www.arm.com/about/newsroom/arm-discloses-technical-details-of-the-next-version-of-the-arm-architecture.php>.

WOO, S. et al. The SPLASH-2 programs: characterization and methodological considerations. In: **, 22nd Annual International Symposium on Computer Architecture, 1995. Proceedings**. [S.l.: s.n.], 1995. p. 24–36.

YAN, J.; ZHANG, W. Compiler-guided Register Reliability Improvement Against Soft Errors. In: **Proceedings of the 5th ACM International Conference on Embedded Software**. New York, NY, USA: ACM, 2005. (EMSOFT ’05), p. 203–209. ISBN 978-1-59593-091-0. Available from Internet: <http://doi.acm.org/10.1145/1086228.1086266>.

YOSHIDA, J. **Toyota Case: Single Bit Flip That Killed | EE Times**. 2015. Available from Internet: <http://www.eetimes.com/document.asp?doc_id=1319903>.

YOURST, M. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In: **IEEE International Symposium on Performance Analysis of Systems Software, 2007. ISPASS 2007**. [S.l.: s.n.], 2007. p. 23–34.

ZANELLA, A. et al. Internet of Things for Smart Cities. **IEEE Internet of Things Journal**, v. 1, n. 1, p. 22–32, feb. 2014. ISSN 2327-4662.

ZHANG, Y. et al. Lighting the dark silicon by exploiting heterogeneity on future processors. In: **2013 50th ACM / EDAC / IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2013. p. 1–7.

ZIEGLER, J. F.; LANFORD, W. A. Effect of Cosmic Rays on Computer Memories. **Science**, v. 206, n. 4420, p. 776–788, nov. 1979. ISSN 0036-8075, 1095-9203. Available from Internet: <http://science.sciencemag.org/content/206/4420/776>.

Appendices

AppendixA LIST OF WORKS PUBLISHED BY THE AUTHOR

This section presents a list of publications related to the author's Ph.D.:

International Journals:

(i) Rosa, F., Brum, R., Wirth, G., Kastensmidt, F., Ost, L., Reis, R. Impact of dynamic voltage scaling and thermal factors on SRAM reliability. Microelectronics Reliability, vol. 55(9-10), pp. 1486–1490 2015.

(ii) E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, J. Tonfat, E. Macchione, F. Aguirre, N. Added, N. Medina, V. Aguiar, M. A. G. Silveira, L. Ost, R. Reis, S. Cuenca-Asensi, and F. L. Kastensmidt. Reliability on ARM Processors Against Soft Errors Through SIHFT Techniques. IEEE Transactions on Nuclear Science, vol. PP, no. 99, pp. 1–9, 2016.

(iii) E. Chielle, F. Rosa, G. Rodrigues, F. Kastensmidt, R. Reis, and S. Cuenca-Asensi. Reliability on ARM Processors against Soft Errors by a Purely Software Approach. 25th International Conference on Radiation Effects on Components and Systems (RADECS'15), 2015.

(iv) G. Rodrigues, F. ROSA, A. de Oliveira, F. L. Kastensmidt, L. Ost, and R. Reis, "Analyzing the Impact of Fault Tolerance Methods in ARM Processors under Soft Errors running Linux and Parallelization APIs," IEEE Transactions on Nuclear Science, vol. PP, no. 99, pp. 1–1, 2017.

International Conferences:

(i) Rosa, F., Bandeira, V., L., Ost, L., Reis, R. Extensive Evaluation of Programming Models and ISAs Impact on Multicore Soft Error Reliability. Accepted for publication in the proceedings of the 55th Design Automation Conference. San Francisco, 2018.

(ii) Rosa, F., Rodrigues, G. S., Kastensmidt, F. L., Ost, L., Reis, R. SOFIA: A Fault Injection Tool for Detailed and Efficient Multicore Soft Error Vulnerability Analysis. Accepted as a work-in-progress poster at the 55th Design Automation Conference. San Francisco, 2018.

(iii) F. R. Rosa, L. Ost, R. Reis, S. Davidmann, and L. Lapides. Evaluation of Multicore Systems Soft Error Reliability Using Virtual Platforms. 15th IEEE International

New Circuits and Systems Conference (NEWCAS) Puerto Vallarta, 2018. Strasburg , 2017.

(iv) Rosa, F., Ost, L., Reis, R. gem5-FIM: A flexible and scalable multicore soft error assessment framework to early reliability design space explorations. 2018 IEEE 9th Latin American Symposium on Circuits & Systems (LASCAS). Puerto Vallarta, 2018.

(v) Rosa, F., Ost L., Raupp, T., Moraes, F., Reis, R. Fast Energy Evaluation of Embedded Applications for Many-core Systems. 24th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'14). Palma de Mallorca, 2014.

(vi) Rosa, F., Ost L., Reis, R., Sassatelli G. Instruction-driven Timing CPU Model for Efficient Embedded Software Development using OVP. 20th IEEE International Conference on Electronics, Circuits, and Systems (ICECS'13), 2013. Abu-dhabi, 2013.

(vii) Mandelli, M., Rosa, F., Ost L., Abdoulaye, G., Sassatelli G, Moraes, F. Multi-level MPSoC Modeling for Reducing Software Development Cycle. 20th IEEE International Conference on Electronics, Circuits, and Systems (ICECS'13). Abu-dhabi, 2013.

(viii) Rosa, F., Kastensmidt, F., Reis, R. and Ost, L. A Fast and Scalable Fault Injection Framework to Evaluate Multi/Many-core Soft Error Reliability. 2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS'15). Boston, 2015.

(ix) Rosa, F.R., Brum, R.M., Wirth, G., Kastensmidt, F., Ost, L., Reis, R. Impact of Dynamic Voltage Scaling and Thermal Factors on FinFET-Based SRAM Reliability. 22th IEEE International Conference on Electronics, Circuits, and Systems (ICECS'15). Egypt, 2015.

(x) Rodrigues, G., Rosa, F.R., F, Ost, L., Reis, R., Kastensmidt, F. Analyzing the Impact of Using Pthreads versus OpenMP under Fault Injection in ARM Cortex-A9 Dual-Core, European Conference on Radiation and Its Effects on Components and Systems (RADECS'16). Bremen, 2016.

(xi) Abich, G., Mandelli, M., Rosa, F.R., Moraes, F, Ost, L., Reis, R. Extending FreeRTOS to Support Dynamic and Distributed Mapping in Multiprocessor Systems. 23th IEEE International Conference on Electronics, Circuits, and Systems (ICECS'16). Monaco, 2016

(xii) F. R. Rosa, L. Ost, R. Reis, S. Davidmann, and L. Lapides. Fast Fault Injection to Evaluate Multicore Systems Soft Error Reliability. 15th Embedded World Conference. Nuremberg, 2017.

(xiii) Rodrigues, G., Rosa, F.R., F, Ost, L., Reis, R., Kastensmidt, F. Investigating parallel TMR approaches and thread disposability in Linux. 24th IEEE International Conference on Electronics, Circuits, and Systems (ICECS'17). Batumi, 2017.

(xiv) Rosa, F., F., Ost, L., Reis, R. Early Evaluation of Multicore Systems Soft Error Reliability Using Virtual Platforms. 2018 Conference on Ph.D. Research in Microelectronics and Electronics (PRIME-LA). Puerto Vallarta, 2018.

(xv) Moore, L., Graham, D., Davidmann, S., Rosa, F. . Cycle Approximate Timing Simulation of RISC-V Processors. 16th Embedded World Conference. Nuremberg, 2018.

National Conferences:

(i) Rosa, F.R., Ost, L., Reis, R. Instruction-driven Timing CPU Model Thread Extension. 30th South Symposium on Microelectronics (SIM'2015). Santa Maria, 2015.

(ii) Rosa, F., Ost, L., Reis, R. Fast Instruction-driven Timing Processor Model for Many-core Embedded Systems. 5th Workshop on Circuits and Systems Design (WCAS'2015). Salvador, 2015