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

JONAS FOGLIARINI GAVA

An Automated Framework for Early Soft Error Assessment, Identification, and Mitigation

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

Advisor: Prof. Dr. Ricardo Reis
Coadvisor: Prof. Dr. Luciano Ost

Porto Alegre
March 2021

CIP — CATALOGING-IN-PUBLICATION

Gava, Jonas Fogliarini

An Automated Framework for Early Soft Error Assessment, Identification, and Mitigation / Jonas Fogliarini Gava. – Porto Alegre: PGMICRO da UFRGS, 2021.

103 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2021. Advisor: Ricardo Reis; Coadvisor: Luciano Ost.

1. Soft errors. 2. Reliability. 3. Fault injection. 4. Fault tolerance. 5. Virtual Platforms. 6. Microelectronics. I. Reis, Ricardo. II. Ost, Luciano. III. Título.

ABSTRACT

Multicore electronic computing systems are incorporating more functionalities and new technologies into their software stacks (i.e., kernels, drivers, and heavy applications). The software stacks running on such architectures differ in terms of security, reliability, performance, and power requirement. While supercomputer software development considers performance as primary criteria, software stacks embedded in cars must comply with strict safety and reliability requirements, which are defined by specific standards such as the ISO 26262 Road vehicles Functional Safety. Such systems are expected to integrate artificial intelligence (AI) and machine learning (ML) techniques that will be just as complex as those found in today's data centers. Soft error mitigation techniques implemented in software do not impact the manufacturing cost. Nonetheless, there are impacts regarding the execution time, code size, and development effort to port to new architectures and multiple programming languages. This can be time-consuming and not provide a good trade-off on large projects. One solution to reduce the energy and performance overhead is to apply selective hardening covering only the application's critical parts. This work focuses on enhancing the SOFIA framework capability by including a soft error mitigation module, which supports automatic code protection by applying different software-based soft error mitigation techniques, also called software-implemented hardware fault tolerance (SIHFT). The proposed approach broadens SOFIA's capabilities by making it the first fully automated framework that supports fast and early soft error assessment, diagnosis, and susceptibility reduction evaluation. The developed mitigation module includes partial and full TMR protection as well as a novel mitigation technique called RAT, which allocates the critical kernel/application function to a specific pool of general-purpose processor registers. Finally, an extensive framework validation is done with over a million fault injections considering distinct Arm processors' configurations. Experiments show that bare metal applications without external dependencies present promising soft error reliability results, as we have access to most of the executed code. On the other hand, for the majority of Linux applications, the code protection is not as effective. For the three evaluated ML algorithms, results show that partial TMR protection's improvement is similar to TMR and has up to 50% less performance penalty for all scenarios. The CNN application results show that replication techniques might not be suitable for resource-constraints platforms and that new and lightweight techniques must be investigated.

Keywords: Soft errors. Reliability. Fault injection. Fault tolerance. Virtual Platforms. Microelectronics.

Um Framework Automatizado para Avaliação, Identificação e Mitigação de Erros Transientes

RESUMO

Os sistemas de computação multicore estão incorporando mais funcionalidades e novas tecnologias em suas pilhas de software (ou seja, kernels, drivers e aplicações pesadas). As pilhas de software em execução nessas arquiteturas diferem em termos de segurança, confiabilidade, desempenho e requisitos de energia. Enquanto o desenvolvimento de software de supercomputador considera o desempenho como critério principal, as pilhas de software embutidas em carros devem cumprir requisitos estritos de segurança e confiabilidade, que são definidos por padrões específicos como a ISO 26262. Espera-se que esses sistemas integrem inteligência artificial (IA) e técnicas de aprendizado de máquina (ML), que serão tão complexas quanto as encontradas nos data centers atuais. As técnicas de mitigação de erros transientes implementadas em software não afetam o custo de fabricação. No entanto, existem impactos em relação ao tempo de execução, tamanho do código e esforço de desenvolvimento para portar para novas arquiteturas e várias linguagens de programação. Isso pode consumir muito tempo e não oferecer uma boa compensação em grandes projetos. Uma solução para reduzir o *overhead* de energia e desempenho é aplicar proteção seletiva cobrindo apenas as partes mais críticas da aplicação. Este trabalho foca no aprimoramento da capacidade do framework SOFIA, através da inclusão de um módulo de mitigação de erros, que oferece suporte à proteção automática de código aplicando diferentes técnicas de mitigação de erros transientes baseadas em software. A abordagem proposta amplia os recursos do SOFIA, tornando-o a primeira ferramenta totalmente automatizada que oferece suporte à avaliação rápida e precoce de *soft errors*, diagnóstico e avaliação de redução de suscetibilidade. O módulo de mitigação desenvolvido inclui proteção TMR parcial e total, bem como uma nova técnica de mitigação chamada RAT, que aloca a função crítica do kernel/aplicação para um pool específico de registradores. Finalmente, uma extensa validação do framework é feita com mais de um milhão de injeções de falha considerando configurações de processadores Arm distintos. Experimentos mostram que aplicações *bare metal* sem dependências externas apresentam resultados promissores de confiabilidade de erros transientes, já que temos acesso à maior parte do código executado. Por outro lado, para a maioria das aplicações Linux, a proteção do código não é tão eficaz. Para os três algoritmos de ML, os resultados mostram que a melhoria usando proteção parcial do TMR é semelhante ao TMR e tem até 50% menos penalidade de desempenho para todos os cenários. Os resultados da avaliação da aplicação CNN mostram que as técnicas de replicação podem não ser adequadas para plataformas de restrição de recursos e que técnicas de mitigação novas e leves devem ser investigadas.

Palavras-chave: Erros transientes, Confiabilidade, Injeção de falhas, Tolerância a falhas, Plataformas virtuais, Microeletrônica.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
CMOS	Complementary Metal–Oxide–Semiconductor
CNN	Convolutional Neural Network
CPU	Computing Processing Unit
EDAC	Error Detection and Correction
FIM	Fault Injection Module
FPGA	Field Programmable Gate-Array
HPC	High-Performance Computing
IR	Intermediate Representation
ISA	Instruction Set Architecture
MTTF	Mean Time To Failure
MWTF	Mean Work To Failure
OMM	Output Memory Mismatch
ONA	Output not Affected
OS	Operating System
OVP	Open Virtual Platforms
OVPsim	Open Virtual Platforms Simulator
SDC	Silent Data Corruption
SEB	Single Event Burnout
SER	Soft Error Rate
SEE	Single Event Effect
SEGR	Single Event Gate Rupture
SEL	Single Event Latchup
SET	Single Event Transient

SEU	Single Event Upset
SIMD	Single Instruction Multiple Data
SoC	System-on-a-Chip
TMR	Triple Modular Redundancy
UT	Unexpected Termination
Van	Vanished

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Multicore electronic computing systems are incorporating more functionalities and new technologies to their software stacks (i.e., kernels, drivers and heavy applications). The software stacks running on such architectures differ in terms of security, reliability, performance and power requirement. While supercomputer software development considers performance as main criteria, software stacks embedded in safety-critical applications (e.g., autonomous vehicles, avionics, medical) must comply with strict safety and reliability requirements, which are defined by specific standards such as the ISO 26262 Road vehicles Functional Safety (ISO, 2011). Such standards will undoubtedly impose more restrictions, mainly due to the advance of autonomous vehicles, which are going to be making decisions that can put human lives at risk. Such systems are expected to integrate artificial intelligence (AI) and machine learning (ML) techniques that will be just as complex as those found in today's data centers.

To deal with the excessively high complexity of such algorithms, researchers and industry leaders are investigating more efficient instruction set architectures (CHEN et al., 2019), ML inference processors (ARM..., 2020), and accelerators (REUTHER et al., 2019). The majority of ML algorithms demand high computational power to perform massive and complicated calculations, which restricts their adoption in resource-constrained devices. In this regard, software libraries and Application Programming Interfaces (APIs) are being proposed (LAI; SUDA; CHANDRA, 2018; CAPOTONDI et al., 2020) to reduce the critical resource usage. Such software libraries/APIs provide a set of functions and kernels devoted to optimize the performance and minimize the memory footprint of deep learning algorithms, thus allowing their efficient execution on low-resource devices (AMOH; ODAME, 2019). With the constant growth of software stack code size and complexity, designing fast, flexible and cost-effective tools that enable in-depth soft error susceptibility analysis of complex software stacks become of utmost importance.

With this in mind, new alternatives, such as the use of virtual platform fault injection (FI) have been proposed (HARI et al., 2012; KALIORAKIS et al., 2015; TANIKELLA et al., 2016). Such FI frameworks allow enormous productivity gains over the hardware-based approaches, mostly at the cost of ignoring the impact of new technologies on soft error rates. The majority of such FI frameworks are reasonable environments, which provide flexible and detailed soft error assessment and identification. However, to ensure the failsafe functionality of emerging electronic computing systems, designers should be able to not only assess and identify, but also to promote efficient alternatives to mitigate the occurrence of soft errors. Authors in (BANDEIRA et al., 2019) proposed SOFIA, a framework developed on the basis of the OVP (Open Virtual Platforms) simulator (IMPERAS, 2019) which supports several non-intrusive fault injection techniques for early soft error assessment and vulnerability analysis.

This work focuses on enhancing SOFIA capability by including a soft error mitigation module, which supports automatic code protection through the application of different mitigation techniques. The proposed approach broadens SOFIA's capabilities by making it the first fully automated framework that supports fast and early soft error assessment, diagnosis and susceptibility reduction evaluation. Promoted extension moves SOFIA beyond the classical toolsets, thereby furthering potential advantages that fill the gap between the available tools and the industry requirements. The developed mitigation module includes partial and full TMR (Triple Module Redundancy) (LYONS; VANDERKULK, 1962) protection as well as a novel mitigation technique called Register Allocation Technique (RAT), which allocates the critical kernel/application function to a specific pool of general-purpose processor registers.

1.1 Dissertation Goal

In order to address solutions to the problems mentioned above, the strategic goal of this Dissertation is to combine well adopted and a novel fault mitigation technique into a module and finally integrate them into the SOFIA framework aiming to enable early soft error mitigation evaluation.

To accomplish the Dissertation goal, the following objectives should be fulfilled:

- Implement well-known state-of-the-art software-based fault mitigation techniques using the LLVM compiler framework;
- Port of the LLVM passes to a most recent compiler version to make use of the last optimizations introduced;
- Proposal and development of a novel soft error mitigation technique aiming to reduce the tradeoff between reliability and performance;
- Combine all these techniques in a mitigation module that can be used standalone or integrated into the automated SOFIA flow.

1.2 Original Contributions

Figure 1.1 illustrates the main contribution of this work, which is the development and integration of a soft error mitigation module into the SOFIA framework (described in Chapter 4 and Chapter 6).

The main contributions of this Dissertation are described as follows:

1.2.1 Fully Automated Soft Error Analysis Flow

Completely automated soft error analysis flow, which leverages the high simulation performance of M*DEV (IMPERAS, 2019) to efficiently acquire representative error/failure-related data considering state-of-the-art multicore processor architectures, such as ARMv7, ARMv8. ML tools can also be used to correlate data and find some particular application's behavior when distinct parameters are set.

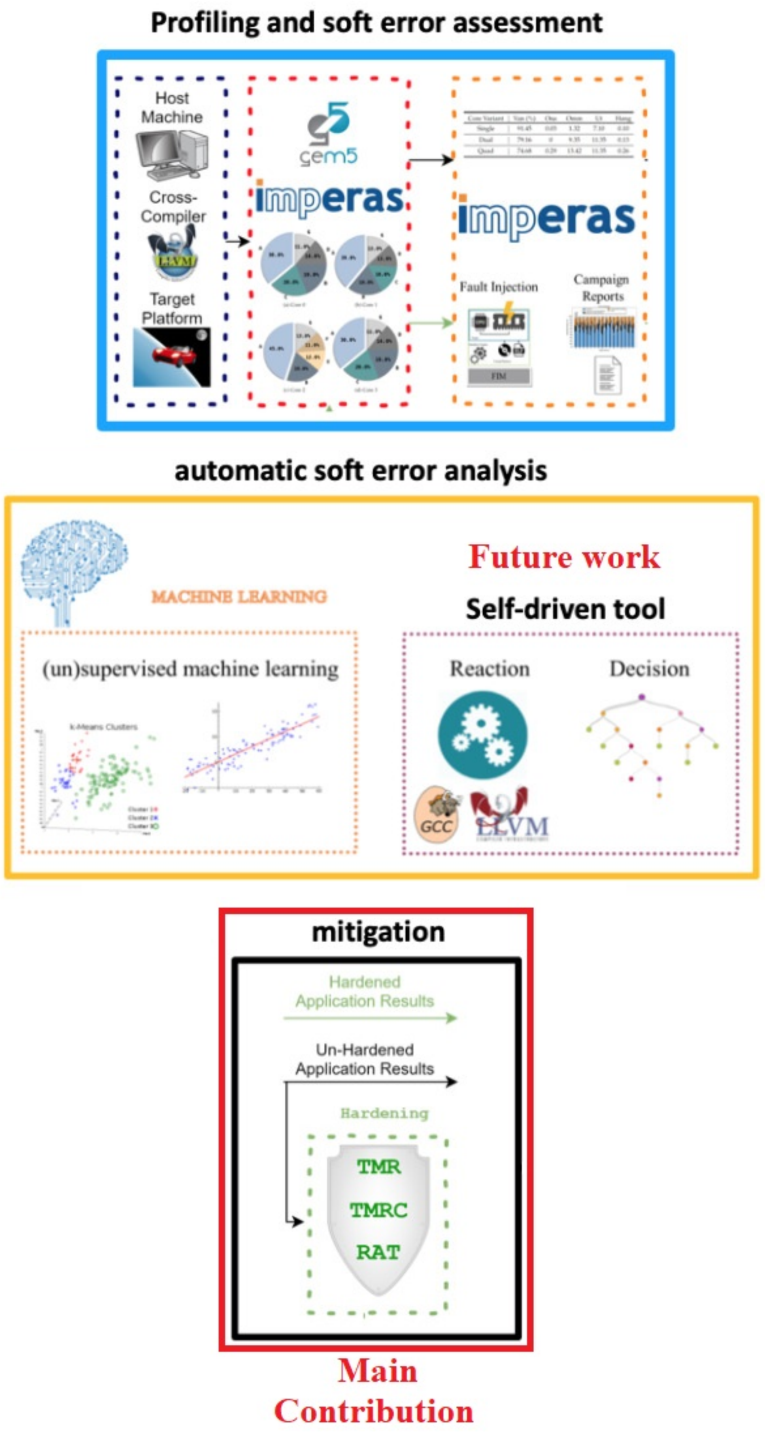
1.2.2 Automated Well-known Mitigation Techniques

Integration of soft error software-based mitigation techniques (e.g., TMR) from literature, enabling susceptibility analyses of real soft stacks considering different configurations. The mitigation techniques are developed using the LLVM tooling, which relies on modifying an architecture-independent intermediate code and later generating the desired target machine code. The techniques were developed to provide partial protection (i.e., P-TMR) at the function level in addition to the full application hardening. That is, it is possible to protect only the more sensitive (or critical) functions in order to obtain a result that is more energy-efficient.

1.2.3 Proposal of a Novel Mitigation Technique

Fault mitigation techniques involving redundancy are the most common and offer good reliability improvement in general. However, in addition to the high cost in performance and energy-efficiency, there are some limitations when a resource-constrained architecture is chosen as a target. In this Dissertation, a novel mitigation technique called Register Allocation Technique (RAT) is proposed. It consists of guiding the compiler registers' allocation phase by restricting the registers available to selected critical functions of the target application (GAVA; REIS; OST, 2020).

Figure 1.1: Simplified framework view showing the main contribution of this work.



Source: Authors

1.2.4 Extensive Validation

Extensive framework validation through more than 1300k fault injections considering distinct architecture, number of cores, OS, parallelization libraries. Different from other works, the promoted framework uses a realistic software stack comprising multiple unmodified operating systems (e.g., Linux 4.3, Linux 3.13) alongside parallelization libraries (e.g., OpenMP, MPI, Pthreads) and ML libraries (e.g., CMSIS NN). To reinforce the experiments, some performance measurements are made using real microcontroller boards.

1.3 Dissertation Outline

The rest of this work is organized as follows.

Chapter 2 - Background: presents basic concepts related to atmospheric radiation environment, the classification of single-event effects, the fault tolerance taxonomy, some well-known reliability metrics, and soft error mitigation techniques.

Chapter 3 - Related Works: presents related works in soft error early assessment using virtual platform fault injection simulators, as well as fault mitigation capable tools found in the literature.

Chapter 4 - SOFIA Framework: is devoted to describing how the adopted framework is structured showing each module: Cross-Compilation module, Profiling module, Fault Injection module, ML module, Analysis and Visualization module, and finally the main contribution *Hardening module*.

Chapter 5 - SOFIA Workflow Validation: validates the framework and the entire workflow using examples.

In **Chapter 6 - Proposed Soft Error Mitigation Module:** an overview of the implemented software-based soft error mitigation techniques are made, as well as the integration of such techniques into our framework.

In **Chapter 7 - Soft Error Assessment:** the mitigation techniques' efficiency is evaluated considering distinct software stacks and processor architectures. Two more case studies are also explored regarding ML applications and the use of custom parameters for the RAT mitigation technique.

Finally, **Chapter 8 - Conclusions:** summarizes this work contribution until this point. Also, it describes the future works related to this Dissertation.

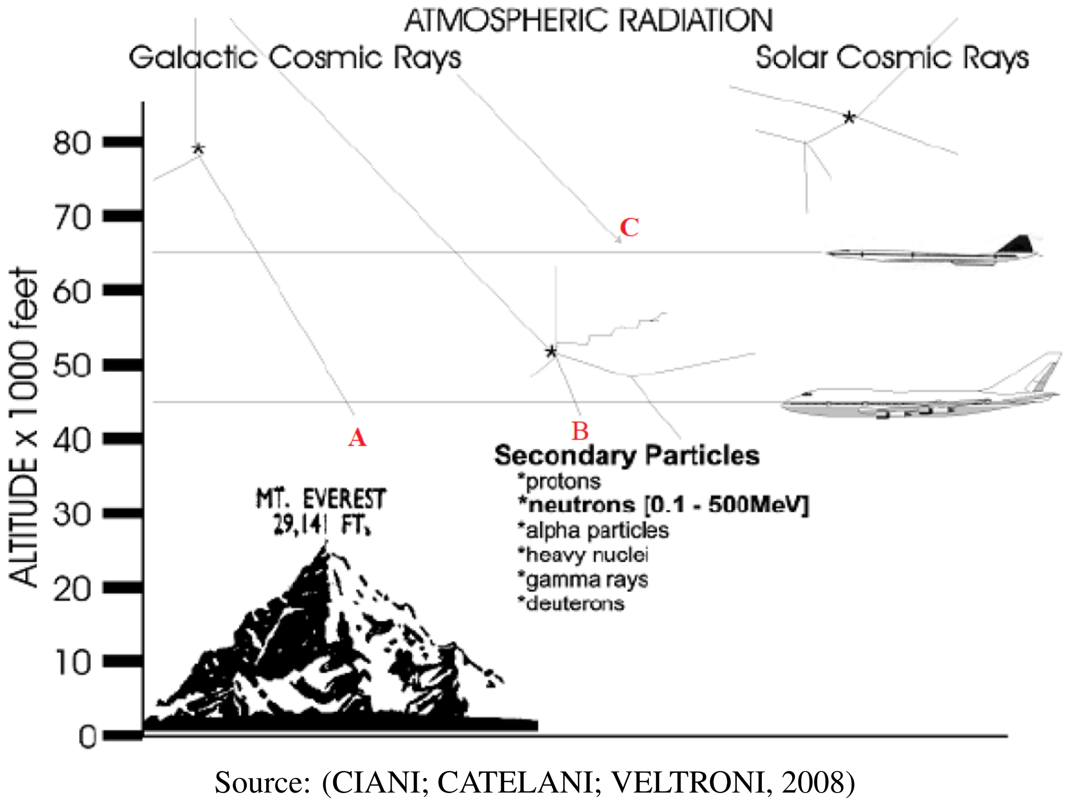
2 BACKGROUND IN RADIATION EFFECTS AND FAULT MITIGATION TECHNIQUES

This chapter aims to introduce the basic concepts regarding radiation-induced errors and their impact on electronic computing system devices. Also, soft error assessment and mitigation literature are wide, requiring a taxonomy to classify the different fault outcomes. Lastly, different forms of hardware and software fault tolerance techniques are presented, as well as the metrics adopted to evaluate such approaches.

2.1 Radiation Environment

Although most people think the atmosphere is benign, the reality is different, even concerning the ionizing radiation environment (NORMAND, 1996; VELAZCO; FOUILLAT; REIS, 2007). In this section, the atmosphere will be described from the point of view of its ionizing radiation components. The three main components of ionizing radiation, neutrons, protons, and heavy ions, are described below for review purposes, as shown in Figure 2.1.

Figure 2.1: Atmospheric Radiation Environment.



Source: (CIANI; CATELANI; VELTRONI, 2008)

2.1.1 Neutrons

Atmospheric neutrons are the main cause of single-event effects at high altitudes (NORMAND, 1996) (Figure 2.1-A). Neutrons in the atmosphere are created by the interaction of cosmic rays with the oxygen and nitrogen atoms in the air. Neutrons in the atmosphere vary with altitude and latitude. The high energy neutron fluxes of interest vary between 10 particles/cm^2/h at sea level and about 10^3 particles/cm^2/h at an altitude of 30,000 feet with variation due to solar activity (LERAY, 2007; HESS; CANFIELD; LINGENFELTER, 1961).

2.1.2 Protons

Neutrons are charged particles produced by the reaction of primary cosmic rays' interaction with particles in the air (Figure 2.1-B). In general, the distribution of protons is similar to that of neutrons, especially concerning energy and altitude. These two particles can also cause SEEs in a very similar way (NORMAND, 1996).

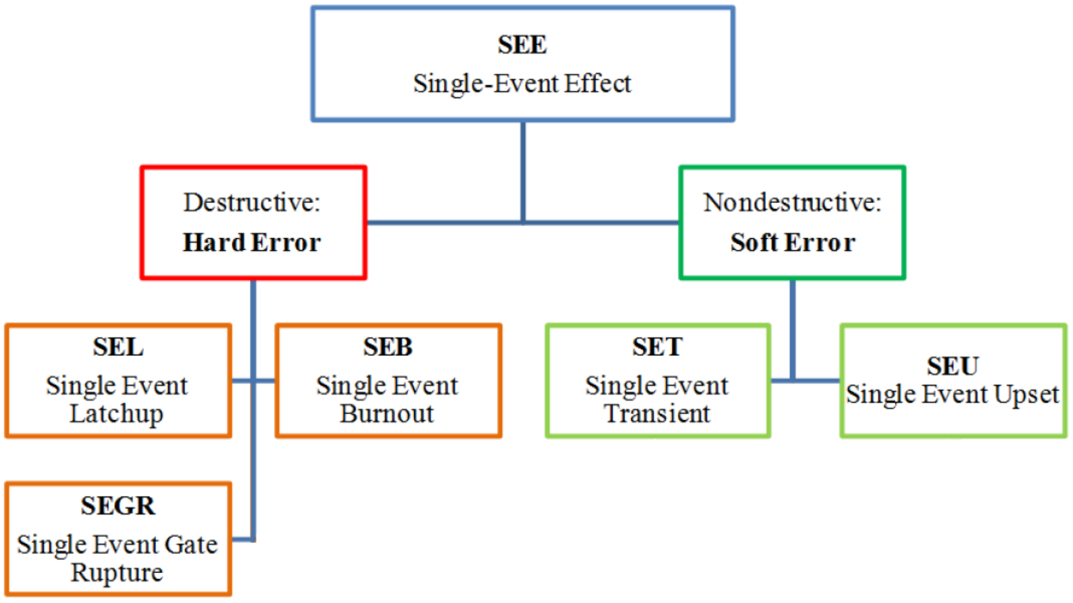
2.1.3 Heavy Ions

Heavy-ion refers to a particle with one or more electric charge units and a mass exceeding that of the alpha particle. The flow of heavy ions within the primary cosmic rays is quickly attenuated with increasing atmospheric depth due to fragmentation (NORMAND, 1996) (Figure 2.1-C).

2.2 Classification of Single Event Effects (SEE)

This discussion aims to give an idea of the types of radiation effects on CMOS transistors. Soft errors are only a subset of the single event effects (SEE). They are any measurable or observable change in the state or performance of a device, component, subsystem, or microelectronic system (digital or analog) resulting from a single energy particle strike. First, destructive SEEs, hard errors will be presented, followed by non-destructive, soft errors (see Figure 2.2) (JEDEC, 2006).

Figure 2.2: Classification of Single Event Effects (SEE).



Source: Adapted from (LADBURY, 2007).

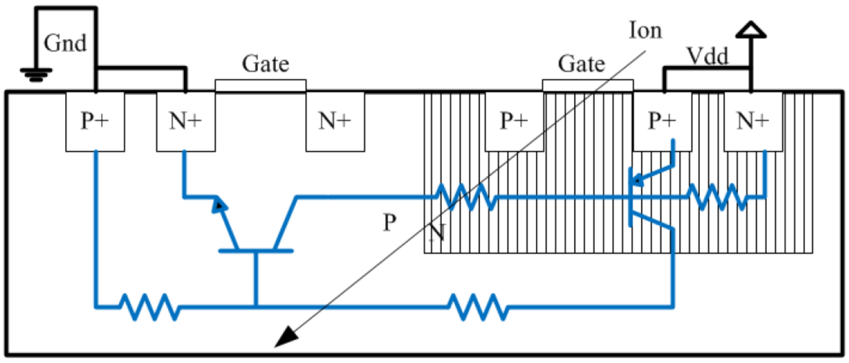
2.2.1 Destructive SEE - Hard Errors

This type of SEE is an irreversible change in a circuit or device's functioning. This effect translates into permanent damage to one or more elements of the circuit or device (rupture of the door oxide, destructive locking events). This type of error is called hard error because there is a permanent loss of data, and the component no longer works properly, even after a restart. The following are the different hard errors.

2.2.1.1 Single Event Latchup (SEL)

An SEL is a high current state in a device caused by the passage of an energetic particle close to two neighboring PMOS and NMOS transistors. It can activate the parasitic PNPN thyristor structure (Figure 2.3) formed by the NMOS-PMOS pair by shortening the power to ground. This mechanism can amplify currents to the point where the device fails due to thermal stress. If the effect is not permanent, turning off the power supply disables the thyristor. The length of the channel and the epitaxial layer's thickness play a dominant role in SEL susceptibility. With the gradual decrease in the transistor's size, new technologies are more vulnerable to this effect. Also, there is an increase in SELs at high temperatures (BRUGUIER; PALAU, 1996; JOHNSTON et al., 1991). In some cases, where SEL does not provoke permanent damage, it could also be classified as a soft error.

Figure 2.3: Parasitic PNPN structure formed by an energetic particle during an SEL.

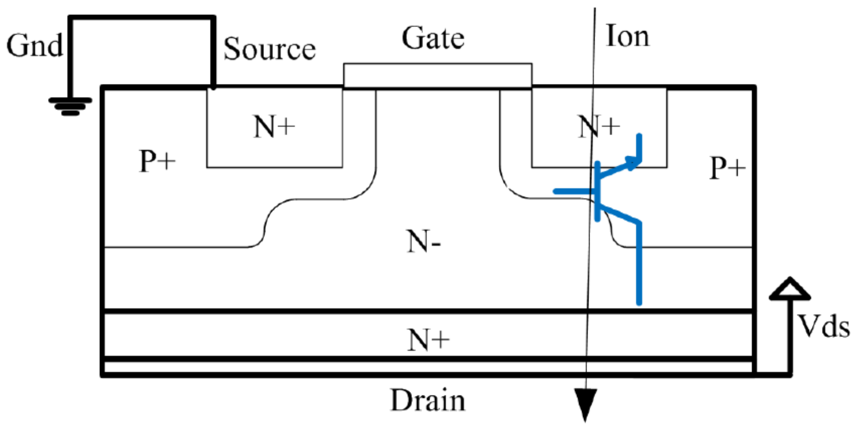


Source: TORO (2014).

2.2.1.2 Single Event Burnout (SEB)

Mainly affecting high-powered MOS transistors, such as Vertical Diffused MOS (VDMOS), a SEB occurs when an ion attack activates the parasitic BJT structure (Figure 2.4), usually in n-channel MOSFETs. The resulting breakage causes a high current state and can cause the device's thermal failure. These transistors are vulnerable to SEB only when they are in the OFF state and only when the applied voltages (VDS and VGS) are outside the region of regular and safe operation. When a particle hits, it can create a high-density current. If the voltage decreases at the parasitic bipolar transistor's base-emitter junction, the transistor is turned on. This happens because of the avalanche current of the BJT scavenger. This can create excessive heating at the junction, which can cause the transistor to burn (WROBEL et al., 1985).

Figure 2.4: Parasitic BJT transistor structure formed by an energetic particle impact in a power VDMOS transistor during an SEB.

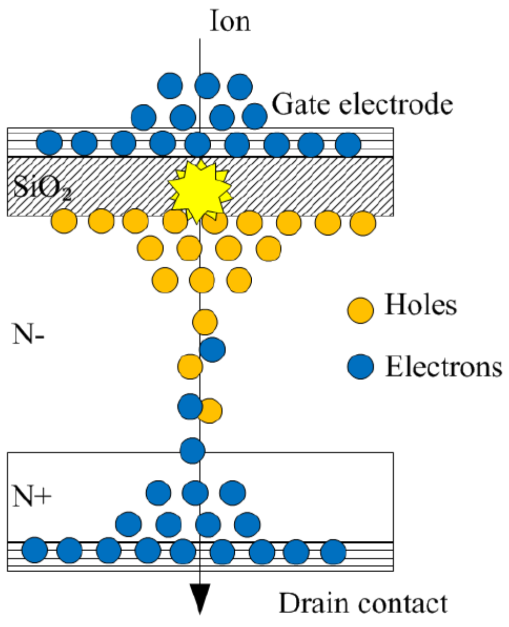


Source: TORO (2014).

2.2.1.3 Single Event Gate Rupture (SEGR)

Like SEB, SEGR mainly affects power MOSFETs. When an ion reaches close to the gate's Si / SiO2 interface, the ion attack holes accumulate under the gate (Figure 2.5). The electric field through the oxide of the MOSFET gate is increased until its dielectric break. The resulting leakage current can also cause a door oxide's thermal failure (TITUS et al., 1998).

Figure 2.5: Accumulation of charges during an SEGR under the gate of a VDMOS transistor.



Source: TORO (2014).

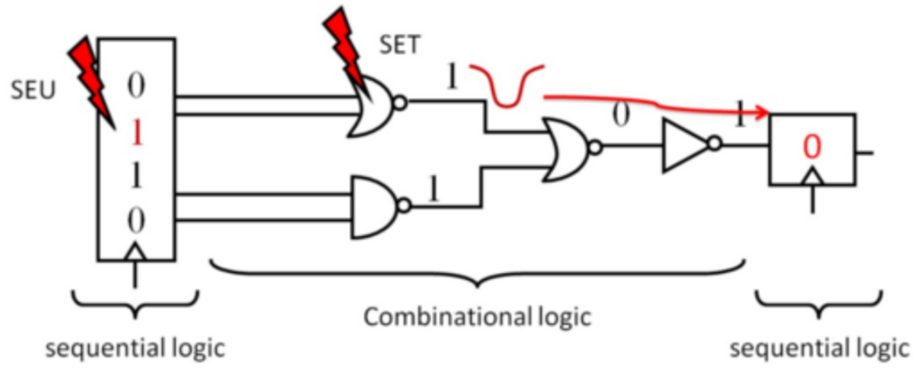
2.2.2 Non-destructive SEE - Soft Errors

Soft errors are non-destructive functional errors induced by attacks of energetic ions. The transistor scale and the lower supply voltage led to lower noise margins and a smaller amount of load, representing a little bit of information. This is expressed in terms of Critical Load (DC), which is the amount of load required to change the logic state when the voltage becomes equal to half the supply voltage. This scale trend increased the susceptibility to soft errors. The stakes for protection against soft errors are high for current and future technologies (DODD et al., 2010). Now, the different soft errors in CMOS technologies will be presented.

2.2.2.1 Single Event Transient (SET)

A Single Event Transient is a voltage spike in the drain of a MOS transistor caused by the charge collection mechanism after a high energy particle creates an ionization track. This affects the combinatorial logic circuits. As illustrated in Figure 2.6, the peak voltage will propagate through logic. If the peak is important enough to avoid electrical masking, it can reach a memory element (latch, flip-flop). However, nowadays, the effects of radiation do not occur solely and exclusively in this way. Other effects appear with technology scaling, such as charge sharing, which causes signal degradation by transferring charges from one electronic domain to another (FERLET-CAVROIS; MASSENGILL; GOUKER, 2013). As the technology's scalability has increased, CMOS circuits' sensitivity to SETs has become a significant problem (MAY; WOODS, 1978; DODD et al., 2010). Despite that, FinFET nanotechnologies show improvements in terms of fault resilience.

Figure 2.6: Single Event Upset and Single Event Transient effects on a circuit.



Source: KASTENSMIDT (2007).

2.2.2.2 Single Event Upset (SEU)

The single event upset differs from SET in its affected target and its duration over time. An SEU occurs in a memory element (e.g., latch, flip-flop, RAM cell, asynchronous memory logic) as a result of latching a SET or an ionizing strike hitting the memory element, as shown in Figure 2.6. As an SEU is stored in a memory element, the error is no longer transient. It can remain several clock cycles for synchronous logic or until the next transition of an input signal in asynchronous logic (TABER; NORMAND, 1993). As in SEL, electrical masking can occur when the signal interference is not enough to change the logic state; in SEU, logical masking can occur through an operation that overwrites the value in the memory element.

2.3 Fault Tolerance Taxonomy

This work considers the definitions from (AVIŽIENIS; LAPRIE; RANDELL, 2004) for fault, error, and failure. A fault is an event that may cause the system's internal state to change, e.g., a radiation particle strike. When a fault affects the system's internal state, it becomes an error. If the error causes a deviation of the correct system behavior, it is considered a failure.

Another important step to understanding fault tolerance is interpreting the data from fault injection campaigns. The two most used and accepted classifications are from (MUKHERJEE; EMER; REINHARDT, 2005) and (CHO et al., 2013). The former classification for soft error assessment considers three classes: *Silent Data Corruption* (SDC) occurs when the system does not detect a fault and the outcome of the application is affected; In *Detected Unrecoverable Error* (DUE) on the other hand, the fault is detected, and it is not possible to continue the execution (e.g., segmentation fault); and *Masked* when the application outcome and the system state are the same as a faultless execution. The latter classification has five fault classes, which increase the level of details in the dataset for future soft error assessment w.r.t. other classifications from the literature, as described in details in Table 2.1.

Table 2.1: Fault effects classification by CHO et al. 2013.

Class	**Description**
Vanished	No fault traces are left
Output Not Affected (*ONA*)	The resulting memory is not modified; nevertheless, one or more remaining bits of the architectural state is incorrect
Output Memory 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 removal after a threshold execution time

Source: (CHO et al., 2013)

2.4 Reliability Metrics

The literature also presents many metrics that help better understand how reliable an application is, and compare two versions of the same app. In this manner, the choice of adequate reliability metrics is crucial to guide the experiments' analysis.

In contrast to Mean Time to Failure (MTTF), the *Mean Work To Failure* (MWTF) (REIS et al., 2005b) can indicate the reliability of hardware and software-based fault mitigation techniques considering the trade-off with performance. The MWTF is defined in Equation (2.1), as the workload that a system can complete before failing. To calculate the MWTF, we measured the applications' runtimes using the gem5 simulator. In addition, our framework uses the fault injection results to measure the Architectural Vulnerability Factor (AVF). In general, the most critical vulnerability is presented by the occurrence of OMM. For example, in safety-critical applications, such as autonomous cars, an OMM error can alter the detection of an obstacle in front of the vehicle, which can lead to an accident. For this reason, this work used the OMM-based AVF (AVF_{OMM}).

$$MWTF = \frac{1}{AVF_{OMM} * execution\ time} \tag{2.1}$$

The Extrapolated Absolute Failure Counts (EAFC)(SCHIRMEIER; BORCHERT; SPINCZYK, 2015) is a metric that considers the fault-space dimension (w) of each application (Equation (2.2)). The fault-space is given by $w = \Delta t * \Delta m$, where the Δt represents the application CPU cycles, and the amount of registers memory in bits Δm it uses. $F_{sampled}$ indicates the number of failures found in the simulations and is defined as the sum of OMM, UT, and Hang. The $N_{sampled}$ is the total number of simulations. The MWTF and EAFC complement each other. The former focuses on failures that alter the application's output without alarming the system which can result in a critical fail. The second metric group all types of failures, including crashes and hangs, and gives an overall vision on the reliability impact.

$$EAFC = w * \frac{F_{sampled}}{N_{sampled}} \tag{2.2}$$

Also, another essential metric to present is fault coverage. This metric describes the percentage of faults that are either detected or masked. It is represented as the ratio of detected and masked faults (or vanished faults) to the total number of faults that occurred, as shown in Equation (2.3). In this scenario, faults detected are defined as Unexpected Termination results. Faults undetected are expressed as the sum of ONA, Output Mismatch and Hang results. We assume that hang is undetected because the real system generally does not have a timeout service to detect.

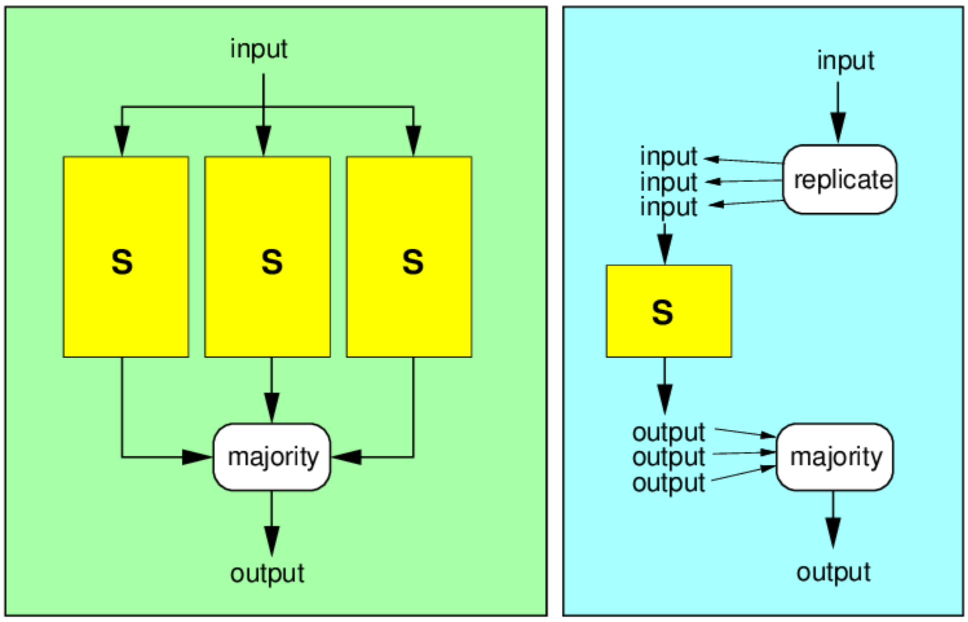
$$F_{coverage} = \frac{F_{detected} + F_{masked}}{F_{total}} = 1 - \frac{F_{undetected}}{F_{total}} \tag{2.3}$$

Typically, soft errors are not a concern for single-user commercial applications while for safety-critical applications error correction and redundancy techniques are mandatory (BAUMANN, 2005). Different methods of soft error protection can be developed at different levels of abstraction. The following are some existing solutions.

2.5 Mitigation Techniques

The search for methods to reduce the effects of radiation in electronic systems has dramatically increased in recent years (KASTENSMIDT; CARRO; REIS, 2006). One of the most utilized techniques to achieve high resiliency to radiation-induced errors is the temporal redundancy. With temporal sampling, the same combinatorial logic is effectively used at many separate times. In the other hand, with spatial redundancy, the computing element is replicated many times (MAVIS; EATON, 2002), as shown in Figure 2.7.

Figure 2.7: Spatial (left) and temporal (right) triple modular redundancy.



Source: GOLDSTEIN; BUDIU (2003).

The best-known mitigation techniques are those that involve modular redundancy. While the Double Modular Redundancy (DMR) creates one copy of a module and can detect that an error has occurred if the outputs are not identical, the Triple Modular Redundancy (TMR) creates two copies, thus managing to recover the system in addition to just detecting the error. Figure 2.7 shows two ways to apply the TMR, running the modules in parallel or sequentially, respectively.

The occurrence of soft errors problems can be tackled both in hardware and software. While hardware approaches lead to the area and power overhead, software techniques are generally implemented on a per-application basis that usually incurs performance penalties. Table 2.2 shows some pros and cons comparing these two approaches. Following are some basics about hardware and software-implemented protection techniques found in the literature.

Table 2.2: Hardware vs Software approaches to mitigate soft errors.

	HARDWARE	**SOFTWARE**
Pros	*performance* and *high reliability*	*cheap* and *easy to deploy*
Cons	*area* and *power* penalties	*performance* and *power* penalties
Examples	Bit Parity, ECC, TMR	EDDI, SWIFT, SWIFT-R, CFSS, SETA, S-SETA

Source: Authors.

2.5.1 Hardware-only

Designers frequently introduce redundant hardware to detect or recover from transient faults. Storage structures, such as caches and memory, typically include extra information in the form of parity or Error-Correcting Codes (ECC), which let these hardware structures detect and recover from such faults. However, protecting all transistors is difficult without significant area, power, and performance penalties (REIS; CHANG; AUGUST, 2007). Hardware-based techniques can also be classified as intrusive (e.g., register bank redundancy) and non-intrusive (e.g., redundant multithreading).

A well-known hardware-based mitigation technique is the lockstepping (HORST; HARRIS; JARDINE, 1990). It performs the computation on two different cores and checks the outputs to detect errors (i.e., DMR). Although the fault coverage is very high, the area and power-efficiency suffer considerably. Another famous technique is the Redundant MultThreading (RMT) (REINHARDT; MUKHERJEE, 2000), which takes advantage of the multiple hardware contexts of SMT, thus making better use of available system resources while reducing the validation overhead through the elimination of cache misses.

In (ABDULHAY et al., 2018), the authors proposed three versions of majority voters for quintuple modular redundancy (QMR). Their focus was in a fault-tolerant medical imaging system. Its implementation in FPGA presents results of logic elements, delay and energy dissipation comparable to the state-of-the-art. Authors in (WANG et al., 2021) explore the impact of single-event upsets (SEUs) on convolutional neural networks using a Xilinx Zynq FPGA. They evaluate the soft error reliability when quantization and a TMRed version of ZynqNet were used. In addition to fault injections, neutron exposure was also utilized in the experiments. Their results show that the quantized version presented a better reliability improvement, with a circuit area reduction of about 44.76%, in contrast with the 111.92% area increase of the TMRed version.

Work in (HUSSAIN; SHAFIQUE; HENKEL, 2019) presents a fine-grained and power-efficient reliability management system that takes into account vulnerabilities and power consumption during application execution. They achieved this power-efficient reliability through a Dual Modular redundancy with Re-Execution (DMR-RE) and Triple Modular Redundancy (TMR) implemented for different architectural components that can be controlled. Compared to the two state-of-the-art techniques, the proposed method achieves significant reliability improvements up to 33% while reducing power up to 13%. These results suggest that considerations of both vulnerability and power variations during different phases of the application provide opportunities for fine-grained and power-efficient reliability management.

2.5.2 Software-only

Software-only approaches to fault detection and recovery can significantly improve reliability without requiring hardware modifications and are therefore cheaper and easier to deploy. However, software-based techniques are generally implemented on a per-application basis that usually incurs performance penalties. Deployment of redundancy techniques in the field is essential, because designers might incorrectly estimate the soft-error rate or the machine's usage condition might change. Changes to the hardware's operating environment can noticeably affect reliability and require the deployment of software redundancy techniques.

The software mitigation techniques can be classified into *data-flow* and *control-flow* protection methods. All forms of protection aim to increase the reliability and fault tolerance of a program. However, they sacrifice performance and power-efficiency. Whenever a program uses more memory or runs longer, the chance of a fault occurrence increases. *Data-flow* techniques aim to detect or correct faults affecting the data, i.e., the registers' values, and the memory. In order to do so, such techniques duplicate (i.e., DMR) or triplicate (i.e., TMR) the instructions, make copies of registers used by the application, and later compare with its replicas. *Control-flow* protection ensures that the program executes all instructions in the correct flow. This type of technique is necessary because the program counter, function return addresses, and branch instructions introduce single points of failure that can not be mitigated using the data-flow approach. The control-flow mitigation ensures that the transitions between code regions are legal and halt the system if an unexpected or illegal change in the execution order occurs. These code regions are referred to as basic blocks, which are defined to be sections of code with a single entry and exit points.

To be able to limit the performance and energy penalties while maintaining good fault coverage, some mitigation techniques are developed on a per-application basis. That is the case of (LI et al., 2021), which presents two algorithmic-based fault tolerance (ABFT) methods aiming to detect errors on deep learning recommendation systems. For the general matrix multiplication (GEMM), results show more than 95% error detection accuracy with a performance overhead of about 20%. And for the EmbeddingBag algorithm, they achieved 99% detection accuracy with less than 26% performance overhead.

2.5.3 Hybrid

Hybrid methods aim to combine changes in the application code with some sort of external hardware support. In general, it consists of a watchdog device that monitors the hardware signals and informs the application running whenever an error is detected. Following this approach, a mix of hardware and software techniques is used to achieve a better tradeoff between reliability and involved costs (performance, area, and power-efficiency penalties). These type of methods could be more flexible, by having the option to chose either higher reliability or a lower overhead than those using software-only or hardware-only hardening techniques (GOLOUBEVA et al., 2006).

For control-flow checking, the source code is divided into basic blocks at compile-time, and a signature instruction is added into the block. The signature instruction has a field that contains an identifying opcode and a field that contains the reference signature.

The opcode could be a coprocessor opcode included in the processor's instruction set, or it could be a specific addition to the instruction set. During the runtime execution, the watchdog observes the executed instructions and generates each basic blocks' runtime signature using dedicated hardware. For data hardening, the watchdog processor can be used to execute assertions concurrently. The assertions are inserted at different program points, stating what intends to be true for the variables. It can be written on the basis of some algorithm property or specifications (GOLOUBEVA et al., 2006). For example, in the inverse problem, if an integer variable is supposed to be positive, a negative check can be used as an assertion.

Authors in (REIS et al., 2005a) proposed the Compiler-Assisted Fault Tolerance (CRAFT), a hybrid hardware/software fault detection systems that capture existing systems' features while reducing the overheads. CRAFT combines SWIFT (SoftWare Implemented Fault Tolerance), a software-only technique, with lightweight hardware structures obtained from hardware-only techniques such as RMT. They implement and evaluate three variations of such technique to demonstrate the potential of hybrid systems. The first version, Checking Store Buffer (CSB), uses the hybrid method to take care of errors affecting store instructions while SWIFT take care of errors affecting load instructions. The second version, Load Value Queue (LVQ), does the opposite by using SWIFT for store instructions and the hybrid approach for load instructions. Finally, the third version, CSB + LVQ, uses only the hybrid method. Their results indicate that the evaluated techniques (hardware, software, or hybrid) present distinct behaviors depending on specific application characteristics. Also, they discovered that adding or enhancing even a single hardware structure can significantly improve system reliability, showing that partial protection can be used to get a better tradeoff.

Some hybrid mechanisms integrate software and hardware mitigation aiming to take advantage of both approaches. However, this work will focus only on software-based techniques as it is cheaper and extendable. Further, in Chapter 3, a more in-depth analysis is done regarding well-known mitigation techniques comparing with this work's approach.

3 RELATED WORKS

Given trends for ever-increasing application/kernel code size and complexity, cost-effective tools to assess the soft error resilience of multicore-based systems become of utmost importance to identify the most unreliable system functionalities early in the design phase.

Simulation-based fault injection approaches are proposed to enable complex soft error resilience analysis regarding different system configurations at an acceptable time. Simulation-based fault injection frameworks allow early evaluation of the system reliability when only system components models are available. Although simulations performed either at the register-level or gate-level provide more accurate results than instruction-level, there are two main issues that reduce their relevance for performing fault injection campaigns in complex multicore systems. Commercial processors are rarely available to users in register-level or gate-level description, and required simulation time is extremely high, even for relatively small processors, making the investigation of systems composed of more complex processors impractical (Rosa et al., 2015).

More recently, researchers proposed extensions to virtual platform simulators aiming to enable fault injection analysis at early design phases. Virtual platforms simplify the development of fault injection modules and the subsequent analysis due to their design flexibility (e.g., several processor models available) and debugging capabilities (e.g., GDB support) (ROSA et al., 2019a; BANDEIRA et al., 2019).

3.1 Early Soft Error Assessment, Identification and Mitigation

Most of the frameworks available in the literature only provide support to soft error assessment and identification. Table 3.1 shows a comparison of related works that will be discussed next.

In (GEISSLER; KASTENSMIDT; SOUZA, 2014), the authors perform 8k fault injections considering an x86 architecture and a Real-Time Operating System (RTEMS), but use only four in-house applications. P-FSEFI (GUAN et al., 2016) framework focus on performing fault injections on parallel applications. They use the NAS Benchmark suite as a case study. However, their instrumentation can add 30$\times$ on top of the tenfold overhead added by QEMU (GUAN et al., 2016). Further, QEMU has a small number of supported architectures. While the approaches presented by (HARI et al., 2012; HARI et

Table 3.1: Related works in soft error assessment, identification and mitigation frameworks developed on the basis of virtual platforms (VPs).

Works	Virtual Platforms	OS	Features			
			Multicore	**Profiling**	**ML**	**Mitigation**
(HARI et al., 2012; HARI et al., 2014)	Simics + GEMS	OpenSolaris				
(GEISSLER; KASTENSMIDT; SOUZA, 2014)	QEMU	RTEMS				
(KALIORAKIS et al., 2015)	MARSS + gem5	None				
(TANIKELLA et al., 2016)	gem5	None				
(GUAN et al., 2016)	gem5	Syscall Mode				
(DIDEHBAN; SHRIVASTAVA, 2016)	gem5	None				
(KHOSROWJERDI; MEINKE; RASMUSSON, 2018)	QEMU	None			✓	
(ROSA et al., 2019a)	M*DEV + gem5	Unmodified Linux	✓	✓	✓	
(BANDEIRA et al., 2019)	M*DEV	Unmodified Linux	✓	✓	✓	
This work	**M*DEV + gem5**	**Unmodified Linux**	✓	✓	✓	✓

al., 2014) propose a hybrid simulation framework for SPARC core using Simics (MAGNUSSON et al., Feb./2002) and GEMS (MARTIN et al., 2005) simulators, remaining works rely on a single virtual platform simulator.

In (KALIORAKIS et al., 2015), authors present two new fault injection tools (MaFIN and GeFIN), one based on MARSS (x86) and the other on gem5 (x86/ARM). For both frameworks, faults are random in time and can target the register bank, cache control among other internal components. Nonetheless, the validation includes ten bare metal applications from the MiBench suite (GUTHAUS et al., 2001). Similarly, authors in (DIDEHBAN; SHRIVASTAVA, 2016) propose a framework based on gem5 capable of targeting the register file, pipeline registers, functional units, and load-store queue. Also, they offer a compilation framework to introduce fault detection mechanisms into the applications. However, the implementation is restricted to Arm assembly, and their analysis employs only ten bare-metal applications. Likewise, (TANIKELLA et al., 2016) considers fault injection in instruction, load-store, and pipeline queue; reorder buffer; renaming unit; and register file. They use two benchmark suites, MiBench and SPEC-Int 2006, for a total of 33k fault injections. Both (KALIORAKIS et al., 2015; TANIKELLA et al., 2016) fail to consider complex software stacks and operating systems. In (Rosa et al., 2015), authors add a fault-injection module to the OVPsim simulator that can manage multiple simulations in parallel. A case study with over 10B instructions running on a Real-Time Operating System (FreeRTOS) and an Arm processor model was conducted. However, the analysis is fixed with a single classification and fault injection configuration. The toolset showed in (BANDEIRA et al., 2019) adds support for bespoke soft error classification and analysis as well as new fault injection techniques. They validate it with the use of a single application.

In (KHOSROWJERDI; MEINKE; RASMUSSON, 2018), authors introduced ML techniques aiming to reduce the number of fault injections required without increasing the error margin. Two automotive applications were tested using the QEMU simulator to validate their implementation. While this approach focuses on reducing the time needed for the fault injection campaign, our approach aims to correlate large subsets of application profiles and architecture characteristics with fault injection results to extract the most relevant parameters on the target system. Further, authors report a slowdown of 3x to simulate the application without FI and 35x, or more depending on the metrics extracted, with FI. On the other hand, our approach an worst case of 4x slowdown with FI.

3.2 Software-based Soft Error Mitigation

The fault mitigation problem can be tackled both in hardware and software. Hardware techniques led to area overhead, thus increasing the system cost. This cost problem can be alleviated by selectively replicating system components or using approximate methods such as approximate-TMR (ATMR) (ALBANDES et al., 2018). On the other hand, software-based techniques usually add execution time and memory overhead. Software techniques can range from low-level (e.g., modifying or adding assembly code) to high-level approaches (e.g., C/C++ libraries, function wrappers). The former may lead to less overhead, but it is architecture and application dependent; while the latter is application dependent. As mentioned in Section 2.5.2, the techniques can be classified in data-flow or/and control-flow hardening. Table 3.2 show some details about well-known software-based fault tolerance techniques.

3.2.1 Control-flow Hardening

Control-flow protection ensures that the program executes all instructions in the correct flow. This type of technique is necessary because the program counter, function return addresses, and branch instructions introduce single points of failure that can not be mitigated using the data-flow approach. The control-flow mitigation ensures that the transitions between code regions are legal and halt the system if an unexpected or illegal change in the execution order occurs. These code regions are referred to as basic blocks, which are defined to be sections of code with a single entry and exit points.

Control flow checking via software signatures (CFCSS) (OH; SHIRVANI; MC-

Table 3.2: Related works in software-based soft error mitigation techniques.

Work	**Technique/Tool Name**	**Type**	**Classification**	**Hardening Level**
(BENSO et al., 2000)	RECCO	Error Recovery	Data-flow	C/C++
(OH; SHIRVANI; MCCLUSKEY, 2002a)	CFCSS	Error Detection	Control-flow	Assembly
(OH; SHIRVANI; MCCLUSKEY, 2002b)	EDDI	Error Detection	Data-flow	Assembly
(NICOLESCU; VELAZCO, 2003)	C2C Translator	Error Detection	Data-flow & Control-flow	C/C++
(GOLOUBEVA et al., 2003)	YACCA	Error Detection	Control-flow	Assembly
(REIS et al., 2005b)	SWIFT	Error Detection	Data-flow & Control-flow	Assembly
(REIS; CHANG; AUGUST, 2007)	SWIFT-R	Error Recovery	Data-flow & Control-flow	Assembly
(FETZER; SCHIFFEL; SÜSSKRAUT, 2009)	EC-AN	Error Detection	Data-flow	LLVM
(FENG et al., 2010)	Shoestring	Error Detection	Data-flow & Control-flow	LLVM
(FENG et al., 2011)	Encore	Error Recovery	Data-flow & Control-flow	LLVM
(Azambuja et al., 2011)	CFT	Error Detection & Recovery	Data-flow & Control-flow	Assembly
(VEMU; ABRAHAM, 2011)	CEDA	Error Detection	Control-flow	Assembly
(ABDI et al., 2012)	CBD	Error Detection	Data-flow	Assembly
(DIDEHBAN; SHRIVASTAVA, 2016)	nZDC	Error Detection	Data-flow & Control-flow	LLVM
(KUVAISKII et al., 2016)	ELZAR	Error Recovery	Data-flow & Control-flow	LLVM
(Chielle et al., 2016)	SETA	Error Detection	Control-flow	Assembly
(DIDEHBAN; SHRIVASTAVA; LOKAM, 2017)	NEMESIS	Error Recovery	Data-flow & Control-flow	LLVM
(DIDEHBAN; LOKAM; SHRIVASTAVA, 2017)	InCheck	Error Recovery	Data-flow & Control-flow	LLVM
(THATI et al., 2019)	SDSC	Error Detection	Data-flow	Assembly
(BOHMAN et al., 2018)	COAST	Error Detection & Recovery	Data-flow & Control-flow	LLVM
(GAVA; REIS; OST, 2020)	RAT	Error Mitigation	-	LLVM

Source: Authors

CLUSKEY, 2002a) requires each basic block to have a unique signature value, assigned in the compilation process. When the program is running, a register keeps track of the current signature. Whenever a new basic block is entered, the signature tracker is checked to match the basic block signature. If the signatures do not match, the program execution is then handed off to an user-defined error handler. But this method does not solve the illegal short jumps into the same basic block, for example. In (GOLOUBEVA et al., 2003), the authors presented Yet Another Control-Flow Checking using Assertions (YACCA). The technique exploits the program Graph's information to check if the basic block is reached from a legal block during the program execution; if not, a control flow error is detected. Its difference to CFCSS consists of the signature generation and control check using the adopted technique. The experiments demonstrated higher fault coverage than the one achieved considering the CFCSS and ECCA (ALKHALIFA et al., 1999) approaches while maintaining similar memory and performance overheads. In (VEMU; ABRAHAM, 2011), Control-Flow Error Detection Using Assertions (CEDA) was proposed integrated into GCC compiler. Like the last technique, it statically analyzes a program and inserts instructions to update and check a runtime signature. Also, they used a set of rules for restricting the possible signature values, which reduces the performance overhead associated with the method while still having a high error coverage. Table 3.3 shows a comparison between CFCSS, YACCA, and CEDA techniques while running five SPEC2000 benchmark programs.

Table 3.3: Comparison of Percent Undetected Errors (UF) and Percent Performance Overhead (PO) for control-flow Hardening Techniques

	CFCSS		YACCA		CEDA	
Benchmark	%UF	%PO	%UF	%PO	%UF	%PO
parser	4.6	14.36	1.0	33.9	1.1	13.79
gzip	3.4	57.7	0.7	84.32	0.6	57.8
ammp	4.7	4.45	0.3	78.97	0.2	3.15
twolf	2.8	7.5	0.6	39.8	0.6	9.8
equake	2.8	18.81	0.5	33.9	0.5	17.9

Source: Adopted from (VEMU; ABRAHAM, 2011).

SETA (Software-only Error-detection Technique using Assertions) (CHIELLE et al., 2016) is another signature-based approach. It improves fault tolerance by giving unique identifiers to not only each block individually but networks of basic blocks. SETA, when combined with the VAR3+ data-flow technique, increases the mean work to failure by a significant amount. The same authors also developed a selective hardening approach called S-SETA (CHIELLE et al., 2015). This technique ignores some small basic blocks with a lower probability of being affected by a fault to reduce the performance and code overhead. Results show a slight reduction in fault coverage but a better tradeoff between reliability and performance.

3.2.2 Data-flow Hardening

Considering the high-level software-based approach, (BENSO et al., 2000) and (NICOLESCU; VELAZCO, 2003) tools that apply fault mitigation techniques in C/C++ applications are proposed. Supported transformations are architecture-independent, but the language is fixed, and the compiler may remove redundant code during the optimization phases. The focus of (NICOLESCU; VELAZCO, 2003) is on low-cost safety-critical applications, where the high memory and speed overheads (about 3–4 times) are not essential metrics. Another similar tool is the REliable Code COmpiler (RECCO) (BENSO et al., 2000), which relies on code reordering and selective variable duplication. (SHIRVANI et al., 2000) propose a software implementation of EDAC, i.e., an independent task executed periodically. Results show that their approach provides protection for code segments and can enhance the system reliability with a lower check-bit overhead w.r.t. other techniques (e.g., Hamming, Parity).

In (SERRANO-CASES et al., 2019), authors use genetic algorithms to find a combination of optimization parameters (i.e., compilation flags) that increase the final binary's reliability and present a reasonable trade-off in terms of performance, and memory size. The proposed technique was evaluated considering an FPGA implementation that was exposed to a proton irradiation test. In (RODRIGUES et al., 2016), the authors implemented in C code two mitigation techniques: the Triple Modular Redundancy (TMR) and the Conditional Modular Redundancy (CMR). Their results have shown that both techniques do not provide reasonable protection to a complex system executing Linux kernel. According to the authors, the OS itself is an enormous source of errors and need to be protected if employed on safety-critical systems. (JAMES et al., 2019) extended the COAST (COmpiler Assisted Software fault Tolerance) tool, which automatically inserts redundancies (DMR and TMR) in the program code through an LLVM plugin. This latest version added support for new processor platforms such as RISC-V and Xilinx SoC-based products. Their mitigation tool is similar to ours, but the soft error evaluation is minimal;

with only two simple applications for the tests, it is difficult to predict what occurs when a real application is deployed. In addition, some experiments were carried out to assess the system's reliability by changing cache memory configuration, which showed that different architectures have distinct behaviors. In (FETZER; SCHIFFEL; SÜSSKRAUT, 2009), AN-encoding is presented as a form of data protection using arithmetic codes. First, each value is multiplied by a constant A. The data must be divisible by A, and the variables must be periodically checked by testing if the datum is equal to zero modulus A. This model relies heavily on correct arithmetic operations, which, if done improperly, causes the application to fail. AN encoding shows a high fault coverage using 128-bit instruction extensions and Streaming SIMD Extensions (SEE). However, AN-encoding would be challenging to do on a low-cost system because of the overhead associated with many modulus operations.

The downside of the approaches mentioned above is that parts of the protected code (e.g., redundant functions) may be wrongly removed during the compiler optimization. One solution to overcome such restriction relies on modifying the assembly code after the compilation or building your own compiler. An option is EDDI (Error Detection by Duplicated Instructions) (OH; SHIRVANI; MCCLUSKEY, 2002b), which is a well-known data flow protection approach. EDDI applies fine-grained mitigation, duplicating each computation instruction. The duplicates are periodically compared with the original data to detect a mismatch. A detected error triggers some form of user-specified error handler. The purpose is to detect SDC and then stop the processor. This approach is also known as duplicate with compare (DWC).

The SWIFT (SoftWare Implemented Fault Tolerance) (REIS et al., 2005b) objective was to reduce the overhead associated with EDDI. They remove duplicate store instructions, saving both memory and execution time. This approach was possible because they assumed that some error correction mechanism protected the system memory. The SWIFT method showed a 14% speedup over EDDI when tested using an Intel Itanium 2 without reducing the fault coverage. Authors in (DIDEHBAN; SHRIVASTAVA, 2016) improved SWIFT technique by checking the load instructions right after a store instruction and creating redundant load instructions in critical sections to achieve near-zero SDC. A popular instruction-level mitigation technique introduced by (REIS; CHANG; AUGUST, 2007) is the SWIFT-R, which implements TMR to recover from soft errors in the register file. Instead of duplicating instructions, it triplicates, and change the checking points to a voter mechanism.

In (FENG et al., 2010), authors presented Shoestring. This technique exploits a low-cost symptom-based error detection mechanism and focuses on applying instruction duplication to protect only those code segments that are likely to result in user-visible faults and do not exhibit symptomatic behavior. Results show that it can recover from an additional 33.9% of soft errors that are undetected by a symptom-only approach. (FENG et al., 2011) presents Encore, a software-based error recovery mechanism (paired with other error detection technique) that combines program analysis, profile data, and simple code transformations to create code portions that can recover from faults at a minimal cost. The experimental results show that Encore can recover from 97% of transient faults on average with 14% additional runtime overhead. In (ABDI et al., 2012), the Critical Block Duplication (CBD) is presented. The technique replicates instructions and check registers only in the critical basic block selected by the higher number of fan-outs aiming to reduce faults' propagation. The results show that CBD reduces the fault coverage by about 28% with a 61% performance overhead compared to full duplication. Another TMR'd technique developed by (KUVAISKII et al., 2016) is ELZAR. It triplicates arithmetic and logical operations, and the voting mechanisms are inserted between register operands of memory and control flow operations for recovery. To reduce the performance overhead, they utilize Intel AVX extensions for vectorization. The experiments show that the performance overhead is reasonable for CPU-intense applications with many floating-point operations. However, for some case studies, the instruction-level parallelism was inefficient, resulting in a performance penalty that surpassed the SWIFT-R technique.

The NEMESIS technique introduced by (DIDEHBAN; SHRIVASTAVA; LOKAM, 2017) is a duplication with recovery technique. It replicates instructions and checks the results of memory write operations and branches' direction. If an error is detected, it then recovers to a valid state if possible; otherwise, a power restart is needed. The results show that at least 97% of the detected errors were recoverable considering the ten selected applications. Another error recovery technique is InCheck (DIDEHBAN; LOKAM; SHRIVASTAVA, 2017), which is an extension of their older technique nZDC. The mitigation technique has error detection, diagnosis, and recovery schemes. Unlike SWIFT-R, InCheck protects the execution of error handling routines in addition to the main program instructions. The authors claim that their technique offers complete error coverage for the tested applications. In (THATI et al., 2019), the authors compare CBD and nZDC against their error detection technique called Selective Duplication and Selective Comparison (SDSC). They follow the Vulnerable Path Duplication (VPD) strategy, which protects the longest path in a Control-flow Graph (CFG) because of the higher probability of error occurrence in such a path. The results show that SDSC has a higher error detection ratio than both techniques but has a higher execution time overhead.

Authors in (MARTINEZ-ALVAREZ et al., 2011) present a compiler-directed methodology that guides the early design of fault-tolerant systems. They developed a generic intermediate code needed for the hardening process and a compilation infrastructure (frontend and backend) to transform the high-level code language to generic instructions (GI) and later transform it to the target architecture code. Furthermore, a selective soft error mitigation technique based on SWIFT-R was evaluated using an 8-bit Picoblaze microcontroller. Although the idea is interesting, a considerable effort is necessary to add support for a new processor architecture, limiting its usability. In this direction, authors in (CHIELLE et al., 2012) proposed the CFT-tool, which is a framework that modifies assembly code by applying different data-flow and control-flow protection techniques. Although this approach does not suffer with compiler optimization, it is architecture-dependent. The CFT-tool uses a configuration file to minimize this limitation, however, this file needs to be hand-made for each new Instruction Set Architecture (ISA). The lower level language tends to have higher efficiency in the reliability result, but the difficulty to work at this level is not feasible when applied to complex soft stacks.

(HOFFMANN et al., 2014) developed the Combined Redundancy (CoRed) mitigation technique for selective protection of safety-critical applications. This technique combines TMR with Arithmetic error coding schemes (AN codes) and engages between application and operating system. The work focus on describing the challenges and pitfalls using the AN codes and present an in-depth analysis of their protected majority voter aiming to eliminate the single-points of failure. The voter reliability was validated by running an extensive fault-injection campaign covering 100 per cent of the fault space for 1-bit and 2-bit errors. They show good results, but the code overhead of 38 to 92 machine instructions for the enhanced voter can be very restrictive.

In general, software or hardware methods are based on redundancy (e.g., DMR, TMR) (REIS et al., 2005b; REIS; CHANG; AUGUST, 2007; FENG et al., 2010; DIDEHBAN; SHRIVASTAVA; LOKAM, 2017). While hardware-based solutions incur in extra silicon area (e.g. use of sensors, monitoring infrastructures, replication of wires), software-based solutions may be power and performance inefficient (e.g., time redundancy). An alternative solution is the use of lightweight mitigation techniques that does not involve redundancy such as RAT (GAVA; REIS; OST, 2020), a technique that allocates the critical kernel/application function to a specific pool of general-purpose processor registers aiming to reduce the register file vulnerabilities to soft errors.

4 SOFIA FRAMEWORK

Rather than implementing a toolset from scratch, we have adopted SOFIA (BANDEIRA et al., 2019) a flexible virtual platform (VP) that provides us with the necessary means (i.e., simulator with processor and component models, full software behavior observability) to implement the proposed technique. The modular framework has a wide range of features that can work independently or as part of a complex workflow (Figure 5.1, later explained in Chapter 5). The framework can generate an extensive and detailed soft error analysis for a given application when all modules are enabled. Currently available modules are described next.

4.1 Cross-Compilation Module

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. Our framework supports both GCC and Clang/LLVM[1] for C/C++ based applications. While GCC is the default compiler available in most GNU/Linux distributions, the LLVM has many useful resources to increase our tooling. LLVM aims to make program analysis and transformation available for arbitrary software, in a manner that is transparent to programmers (LATTNER; ADVE, 2004). The framework support for these compilers is delivered by a set of scripts and CMake files.

The proposed framework simulation module is based on two well-known simulators: gem5 (BINKERT et al., 2011a) and the Multicore Developer (M*DEV) virtual platform (IMPERAS, 2019). Each simulator brings its advantages and drawbacks. The gem5 simulator provides a cycle-accurate simulation and many insightful metrics from the underlying architecture at the cost of simulation turn-around. On the other hand, M*DEV sacrifices cycle-accuracy in favor of speed by simulating instructions behaviors. To achieve the best results for our workflow, we use them where appropriate. We leverage the accuracy of gem5 to perform application and architecture profiling, while M*DEV is better suited for massive campaigns of fault injection and soft error assessment. Note that we achieve an apples-to-apples comparison by using the same application binary and Linux kernel between the simulators.

[1]Note that the fault mitigation techniques integrated into our framework only works with code compiled with LLVM

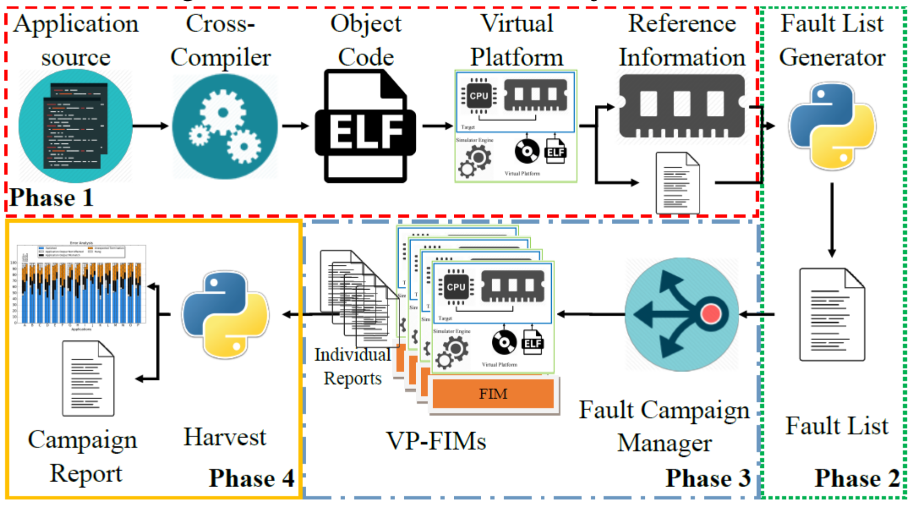
4.2 Profiling Module

To gather more useful information about the soft error reliability of multicore systems, our framework has support for tracing and profiling an application's behavior under the presence of faults. The underlying toolset provides software engineers with crucial information. For example, a detailed profile of executed instructions — the number of each executed instruction by their opcode (`add, bneq`) or their class (e.g., arithmetic, branch) — as well as the utilization percentage of processor registers, enabling the evaluation of different instructions reliability and register criticality. To execute the target platform binary in the host machine, the simulator (M*DEV) needs to translate the `opcode` from the target machine into a host machine code. It uses a callback to perform this translation. This callback is responsible for fetching an instruction, decoding it and then calling the routines that describe its behavior. During the callback, a user-defined routine changes the instruction's behavior aiming to generate a bespoke profile.

4.3 Fault Injection Module

As shown in Figure4.1, fault injection can target register bank, main memory, program variables, compiled source code saved in memory. Further, the framework is also able to isolate a function and perform the above techniques, e.g., fault injection on the register bank only when the isolated function is on the processor context. The fault injection comprises four steps. *First phase*, Golden Execution, the target architecture is simulated in the absence of faults to extract the system reference behavior (i.e., the context of the registers and final memory state), while the *second phase* creates a list of target faults (i.e., register, injection time, and bit position). In the *third phase*, the scenario under test suffers the fault injection and a report is created with available information, e.g., final processor context, number of executed instructions, memory values — this phase can be executed multiple times, i.e., perform a fault injection campaign. The module has support for distributed (multiple hosts within the same network) and parallel (multiple processors and cores in the same host) fault injection simulations. Finally, *fourth phase* assembles all individual reports to create a single fault injection results database.

Figure 4.1: OVPSim-FIM Fault Injection Flow.



Source: ROSA et al. (2019b).

4.4 Machine-Learning Module

Converting fault injection explorations into actual system reliability improvements is not straightforward. By extending the study in ROSA et al. (ROSA et al., 2019a), our work deploys 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 using M*DEV). Underling 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. The framework also provides users with a flexible investigation, where several information sources can be easily included, selected, and conformed to different machine learning investigation techniques.

4.5 Analysis and Visualization Module

Decoupled from the fault injection module, we provide a module to perform analysis and the visualization of the data from the fault injection campaign. The framework is capable of performing soft error analysis using the classifications from the literature

(Section 2.3) or user-customized classifications. This module can generate all figures showcasing the results of fault injection analysis and profiling present in this work. The availability of this visualization feature can increase the productivity of application engineers, making raw data easy to understand and find patterns.

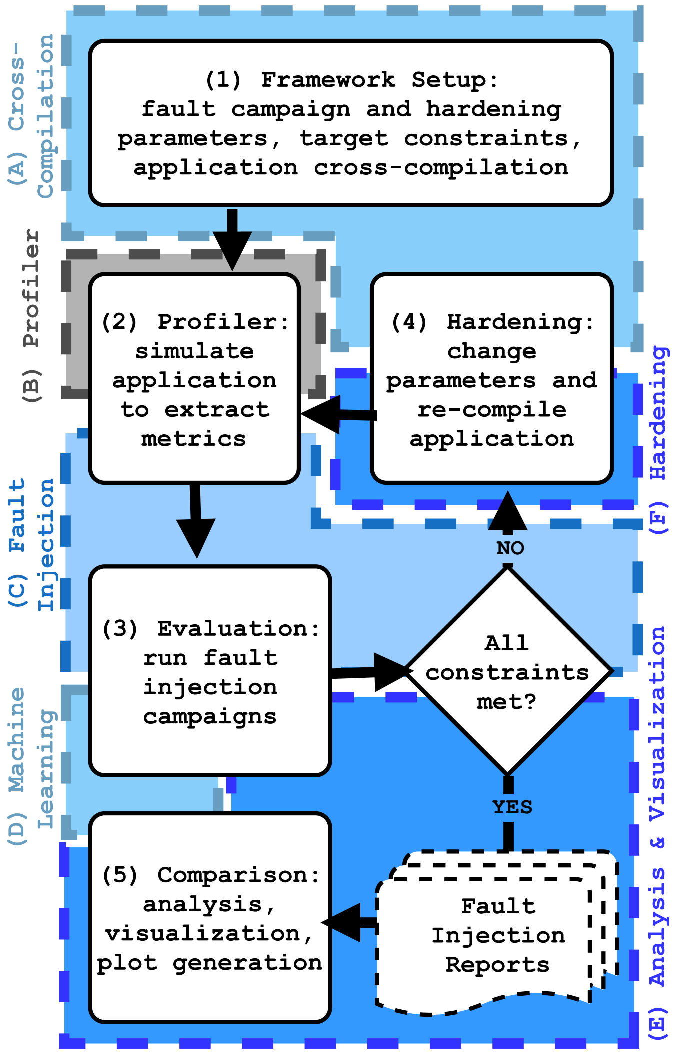
4.6 Application Hardening Module

This framework module focuses on architecture-independent software-based protection by leveraging LLVM features to implement mitigation techniques on the basis of COAST tool (BOHMAN et al., 2018). Thus, without modifying the original application source code. The LLVM operation flow comprises the following steps: **High-level language to Intermediate code Representation (IR):** a compiler driver (i.e., front end) receives the high-level language source code (e.g., C/C++) and outputs LLVM IR instructions — which describes a program using an abstract RISC-like instruction set but with essential higher-level information for effective analysis. **Transformations:** uses the IR to apply optimizations (e.g., `-O3`) and other code modifications. **Code Generator:** the final step is to transform the architecture-independent internal representation into assembly code for a given target (e.g., x86, ARM, RISC-V). The mitigation techniques provided with the framework take place (i.e., code transformations) at early code generator steps, thus guaranteeing architecture independence. This mitigation module is described in more details on Chapter 6.

5 SOFIA WORKFLOW VALIDATION

In this chapter, a workflow using all available modules and features is proposed to validate our framework. Further, we showcase the framework and the workflow with a step-by-step example for an application. The proposed workflow comprises six phases (Figure 5.1).

Figure 5.1: Framework modules (A–F) and proposed workflow (1-5): from source code to hardened application.


(A) Cross-Compilation
(1) Framework Setup: fault campaign and hardening parameters, target constraints, application cross-compilation
(B) Profiler
(2) Profiler: simulate application to extract metrics
(4) Hardening: change parameters and re-compile application
(F) Hardening
(C) Fault Injection
NO
(3) Evaluation: run fault injection campaigns
All constraints met?
(D) Machine Learning
YES
(5) Comparison: analysis, visualization, plot generation
Fault Injection Reports
(E) Analysis & Visualization

The first step is to set the framework configuration: choose the target platform, fault injection campaign parameters (e.g., number of experiments, fault target), and cross-compile the application source code (unhardened version) to the target platform. The second step extracts performance metrics of the application using the available profiling tools. In the third step, the unhardened binary goes through an initial soft error evaluation. The data from the profiling and soft error evaluation is fed to the machine learning module, which is able to find correlations that can aid to choose the correct hardening technique. If the constraints are not all met (e.g., the MWTF is below a threshold), the framework tried to add or change the current hardening parameters in the fourth step and repeats steps 2 and 3 with the new hardened binary. When all the constraints are met, the workflow advances to the last step to combine all the fault injection reports from the multiple evaluation loops and profiling data from the different binaries to generate graphs and a final report.

To show in details each step of the workflow, we chose the OpenMP version of application Integer Sort (IS) from NAS Parallel Benchmarks (NPB) (NASA, 2019).

5.1 Framework Setup and Configuration

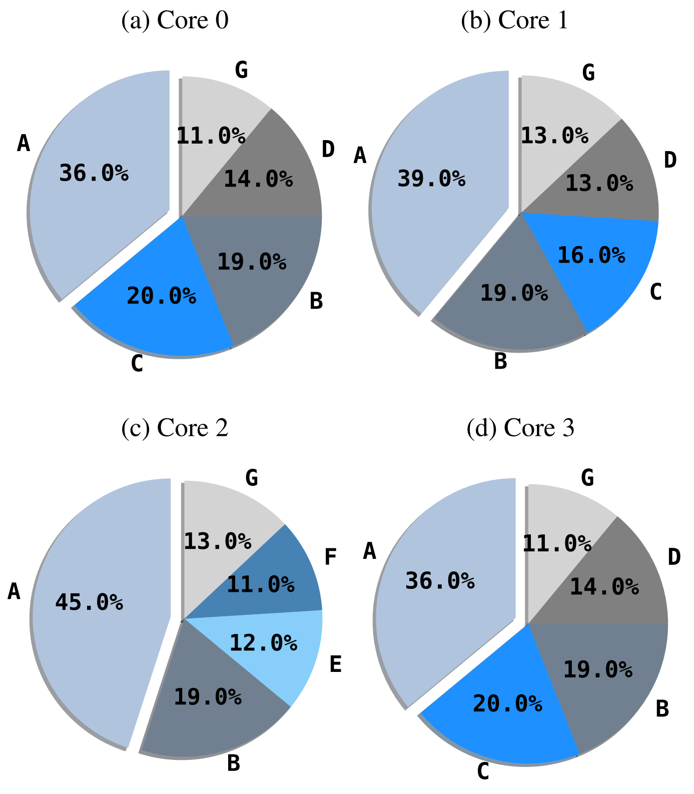
The framework uses CMake configuration scripts to set compiler related ISA flags. Then we introduce two function calls (i.e., INIT and EXIT) at the beginning and end of the main function. These functions tell the fault injector module the interval time where the application executes. After this initial setup the framework compiles the application using the template CMake script with some optional specific compilation flags (e.g., -O2, -mcpu, -mfloat-abi) passed through the command line.

5.2 Profiling

After creating the binary for the target platform, the framework simulates the application without faults and does the profiling (Figure 5.2) using both gem5 and M*Dev. Data from these initial simulations are used later for the ML analysis and the mitigation techniques.

Taking a closer look at the function profile for the unhardened program version running on a quad-core Arm Cortex-A72 system, Figure 5.2, we can see that the most performed function is `.omp_outlined.6`, created by the OpenMP library. There are dozens of functions in the "others" category, of which over 90% belong to the Linux routines. It is also worth to note that thread management routines (e.g., `kmp_fork_barrier`, `kmp_join_barrier`, `kmp_hyper_barrier`) represent a large portion of the total execution time. These thread synchronization functions are more sensitive to register bit-flips. When increasing the execution time of other program functions by inserting

Figure 5.2: Sub-figures (a-d) show the function profile for each core on a quad-core system for the unhardened program version. *Function* **A:** .omp_outlined.6, **B:** randlc, **C:** kmp_hyper_barrier, **D:** kmp_fork_barrier, **E:** kmp_join_barrier, **F:** kmp_barrier, **G:** others.



redundancies, the probability of errors in the operating system's critical tasks is reduced. Finally, the reason that makes Core 2's profile different from other Cores is that it contains the main thread, so it runs function A longer and has other specific tasks for synchronizing threads (E, F).

5.3 Soft Error Evaluation

For an initial soft error evaluation, we perform three fault injection campaigns varying only the number of processor cores (i.e., 1, 2, and 4). Each FI campaign injects $3.1k$ random bit-flips in the Cortex A72 processor general-purpose registers — 99% confidence level and a 2.3% error margin according to (LEVEUGLE et al., 2009a). As the number of cores increases, so does the operating system thread management routines, thus Table 5.1 shows the increase in OMM and decreases in Vanished. MWTF is also severely affected with a reduction of $19.40\times$ from single-core to dual-core and a further $2.29\times$ reduction from dual-core to quad-core. The same is true for EAFC (lower values are better), where a 2.81× increase from single-core to dual-core is observed and a further 1.56× addition from dual-core to quad-core.

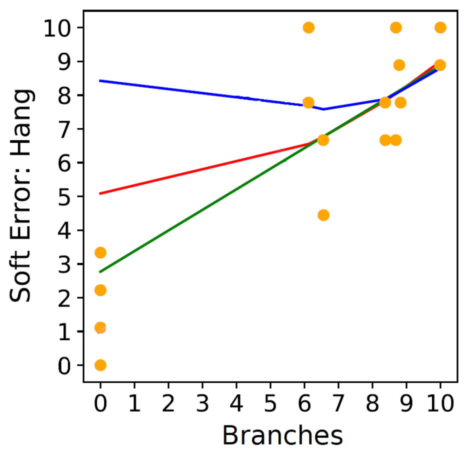
Table 5.1: NAS Parallel Fault Injection Campaign Results.

Core Variant	**Van.**	**ONA**	**OMM**	**UT**	**Hang**	**MWTF**	**EAFC** (10^6)
Single	91.45	0.03	1.32	7.10	0.10	88.45	5.34
Dual	79.16	0	9.35	11.35	0.13	4.56	15.00
Quad	74.68	0.29	13.42	11.35	0.26	1.99	23.43

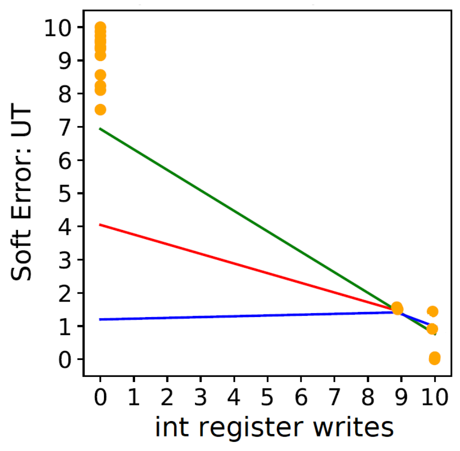
Soft error values are in %.

Figure 5.3: Sub-figures (a-d) show ML analysis examples generated using a set of FI campaigns. The *circles* are FI campaigns, and the lines are distinct ML regression techniques: *red*, poly; *green*, linear; and, *blue*, rbf regression).

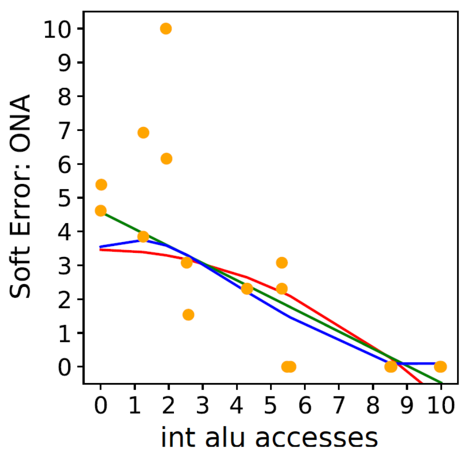
(a) Hang vs Branches



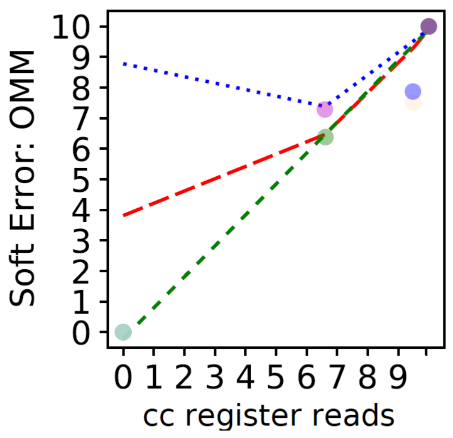
(b) UT vs Register



(c) ONA vs ALU

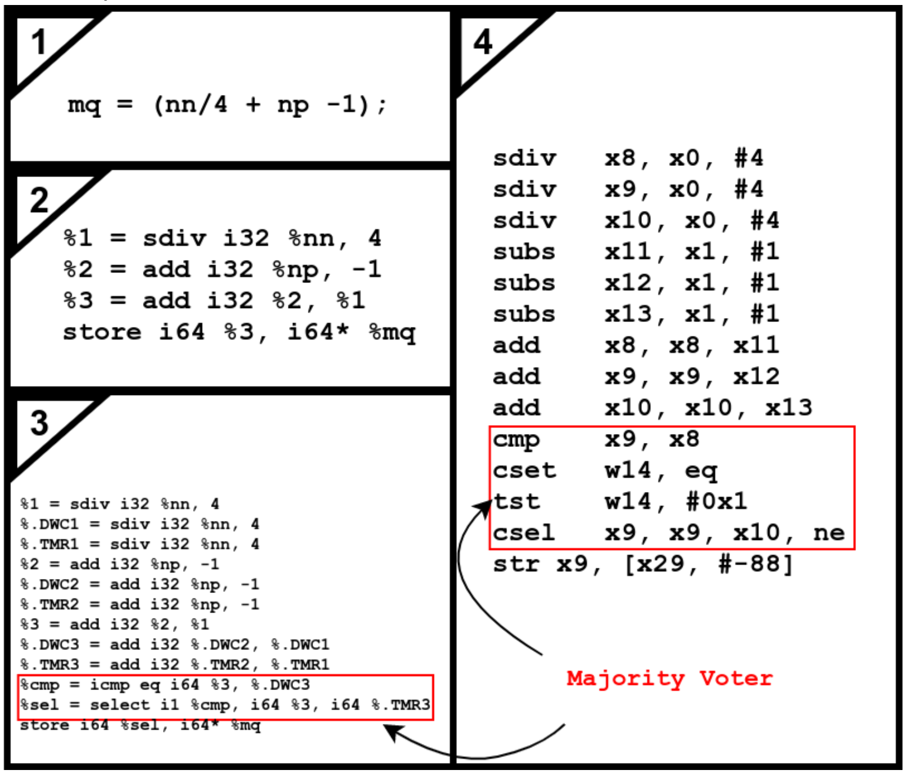


(d) OMM vs Register



The ML module combines data from the profile and soft error to speed up the analysis of the application's behavior for different configurations and parameters. Figure 5.3 show four ML correlations generated for a set of FI campaigns for the same application, varying some configurations (e.g., number of cores, compiler flags). (a) shows that the increase in the number of branch instructions is directly related to the rise in the system Hangs. In (b) we see that the increase in the number of entries in the integer register is related to the decrease in UT. In (c), we see that the rise in the number of accesses to the ALU is associated with the reduction of ONA. Finally, in (d), we have a direct relationship between the number of readings from control registers and the increase in OMM.

Figure 5.4: Example of the TMR applied in an operation inside IS application. From (1) C code, to (2) LLVM IR code, then (3) LLVM IR with TMR, and finally the (4) protected Aarch64 assembly.



```
mq = (nn/4 + np -1);
```

```
%1 = sdiv i32 %nn, 4
%2 = add i32 %np, -1
%3 = add i32 %2, %1
store i64 %3, i64* %mq
```

```
%1 = sdiv i32 %nn, 4
%.DWC1 = sdiv i32 %nn, 4
%.TMR1 = sdiv i32 %nn, 4
%2 = add i32 %np, -1
%.DWC2 = add i32 %np, -1
%.TMR2 = add i32 %np, -1
%3 = add i32 %2, %1
%.DWC3 = add i32 %.DWC2, %.DWC1
%.TMR3 = add i32 %.TMR2, %.TMR1
%cmp = icmp eq i64 %3, %.DWC3
%sel = select i1 %cmp, i64 %3, i64 %.TMR3
store i64 %sel, i64* %mq
```

```
sdiv   x8, x0, #4
sdiv   x9, x0, #4
sdiv   x10, x0, #4
subs   x11, x1, #1
subs   x12, x1, #1
subs   x13, x1, #1
add    x8, x8, x11
add    x9, x9, x12
add    x10, x10, x13
cmp    x9, x8
cset   w14, eq
tst    w14, #0x1
csel   x9, x9, x10, ne
str x9, [x29, #-88]
```

5.4 Application Hardening

In this step, the mitigation technique is selected. The initial analysis considers TMR and P-TMR mitigation techniques. Figure 5.4 shows the transformation of a block of C code to LLVM IR, then to protected LLVM IR, and finally to the aarch64 assembly code. Note that the TMR majority voter has two instructions in the LLVM IR code and four instructions in the ARMv8-A. Later on, we show that an application with a large number of majority voters can increase susceptibility to soft errors due to the increase in the single points of failure, indicating that the use of a selective protection technique may be more appropriate depending on the application.

5.5 Results Comparison

With the data from the unhardened binary and the two hardened versions (`TMR` and `P-TMR`), Software engineers can use the visualization module to compare the reliability vs performance trade-off and then define which protection strategy is best suited for the chosen application.

Figure 5.5: Comparison between mitigation techniques, values are normalized by unhardened version.

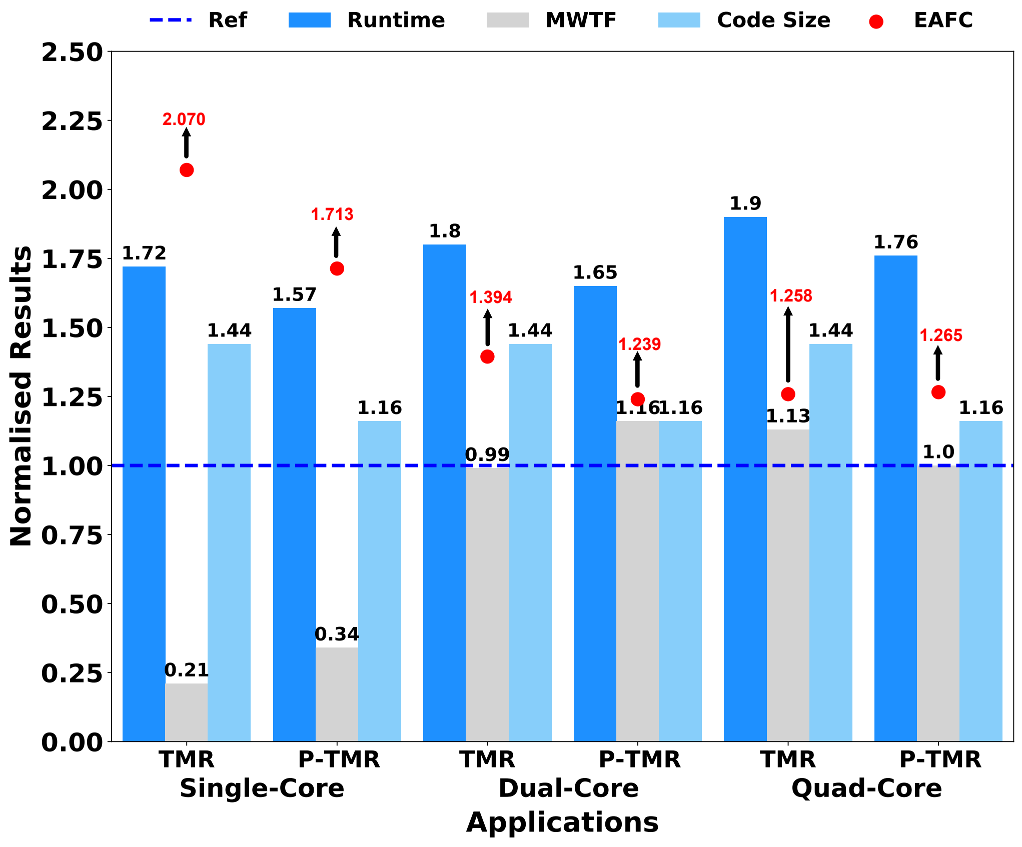


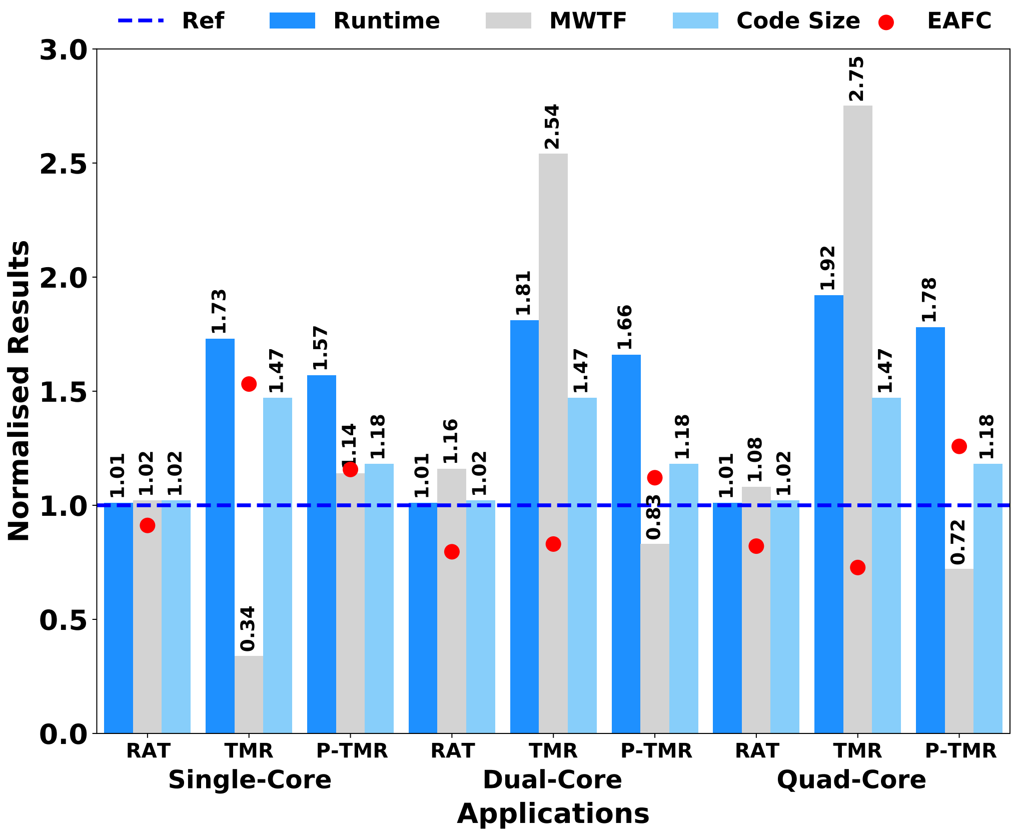
Figure 5.5 shows the results regarding system reliability and the overhead for runtime and code size. This data set considers two different mitigation techniques and three core variants, with all values normalized by the unhardened version. The program code size has a 44% increase when applying `TMR` against 16% in `P-TMR`, showing an interesting finding for memory-constrained projects. Considering the reliability metrics, there is a significant impact on the normalized MWTF and EAFC for the single-core platform. The MWTF for `TMR` and `P-TMR` has a reduction of 79% and 66% (respectively), while EAFC increases by $2.1\times$ and $1.7\times$ — thus, showing that the memory errors mitigation was inefficient. In contrast, the reliability regarding overall errors (OMM + UT + Hang), represented by the EAFC, was positively impacted. The reason for this behavior is that

the application already has a natural high fault-masking rate (90%+), and our tool cannot reach external functions and OS routines that use the most susceptible registers. However, the normalized reliability improves with the number of system cores. This effect occurs because we have a significant increase in the execution of OpenMP functions and other thread management routines, making the protection worth in this case. The MWTF gain is ∼16% for the dual-core `P-TMR`, and ∼13% for the quad-core `TMR`.

Overall, the majority of applications uses a small fraction of the 31 available general-purpose registers in the ARMv8 ISA, while the operating system handles them more evenly. Thus, a fault that occurs inside an application's scope can remain in the register after the program's execution and affect external functions, libraries, and the kernel subroutines.

Figure 5.6 presents results with the feature of RAT enable. Data points consider three processor model variants (single-core, dual-core, quad-core), the unhardened version with custom register allocation (`RAT`) and two different mitigation techniques (`TMR(+RAT)`, `P-TMR(+RAT)`). All the scenarios presented are normalized by the unhardened application (`Ref`).

Figure 5.6: Results applying `RAT` for unhardened, `TMR`, and `P-TMR` versions. All data points are normalized with unhardened version without `RAT`.



The first notable case we can see is the unhardened version with RAT. With minimal addition of runtime or code size overhead, there is an increase in MWTF — 2%, 16%, and 8% for single-core, dual-core, and quad-core respectively — and a decrease of EAFC — 8.83%, 20.37%, and 17.91%. When we apply `TMR` technique, a massive increase in MWTF gain can be observed — about 2.54× and 2.75× for dual-core and quad-core, respectively. This effect occurs due to the more significant number of registers used by the program since all their functions have redundant code. In this way, it is possible to distribute the use of registers more evenly. However, we do not see a significant effect on `P-TMR(+RAT)`, meaning that choosing the least used registers strategy does not work in all cases.

6 PROPOSED SOFT ERROR MITIGATION MODULE

A processor-based system can be affected by two main types of soft errors: control-flow and data-flow. A control-flow error occurs when the error causes deviation from the correct program flow (e.g., incorrect branch). The data-flow error refers to the soft error caused by a bit-flip in a storage device, such as a register or memory element. They can affect the output of the program, but not its execution.

This work focus on software mitigation techniques based on data-flow protection. This approach was chosen because, in general, the majority of soft errors affect data-flow rather than control-flow, and the protection in software added by the control-flow methods alone are sometimes questionable as discussed in (RHISHEEKESAN; JEYAPAUL; SHRIVASTAVA, 2019).

6.1 TMR (Triple Module Redundancy)

The first fault mitigation technique implemented was TMR, a method widely used at the hardware level (LYONS; VANDERKULK, 1962). It creates two replicas of a component and then checks the outputs through a majority voting mechanism to mask the error. However, when analyzed at the software-level, the performance penalty is often quite significant, so it is necessary to evaluate it on a case-by-case basis. The following subsections describe all the rules that define how the implemented technique works and how they are inserted in the application's compilation flow to allow automated protection.

6.1.1 Data-Flow Rules

Authors in (CHIELLE; KASTENSMIDT; CUENCA-ASENSI, 2016) describe a set of rules for data-flow techniques that aim to detect faults affecting values stored in the register bank and memory devices (Table 6.1). To provide soft error recovery instead of just detection, this work uses triplication instead of duplication.

Table 6.1: Data-Flow Rules.

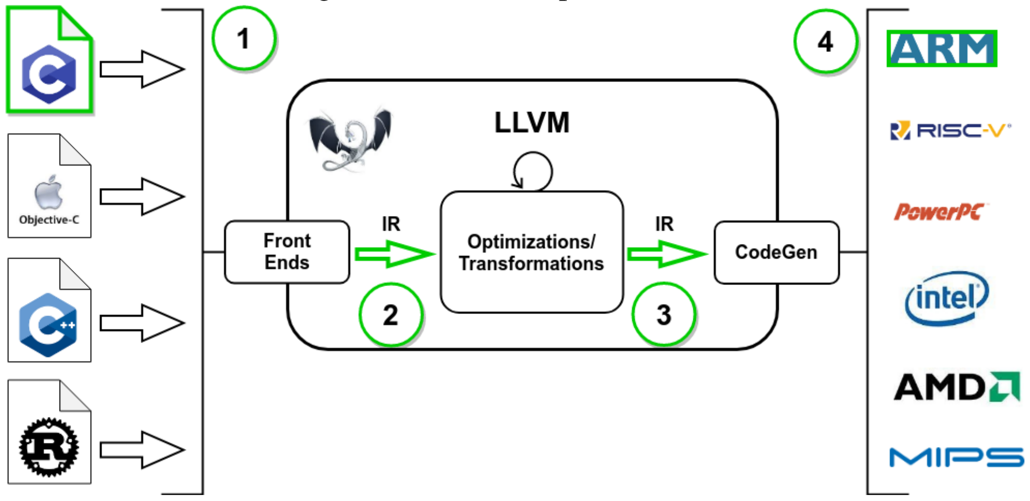
	Global Rules (valid for all techniques)
G1	Each used register has a replica
	Duplication Rules (same operation on the register's replica)
D1	All instructions except branches
D2	All instructions, except branches and stores
	Checking Rules (compare the value of a register with its replica)
C1	Before each read on the register (except load, store, branch)
C2	After each write on the register
C3	Before loads, on the register that contains the address
C4	Before stores, on the register that contains the data
C5	Before stores, on the register that contains the address
C6	Before branches

Source: CHIELLE; KASTENSMIDT; CUENCA-ASENSI (2016).

6.1.2 LLVM Compiler Infrastructure

The LLVM aims to make program analysis and transformation available for arbitrary software, in a manner that is transparent to programmers (LATTNER; ADVE, 2004).

Figure 6.1: LLVM Operation Flow.



Source: Authors.

The LLVM operation flow, shown in Figure 6.1, comprises the following modules: **(1)** ***Front-end.*** is the compiler driver that outputs LLVM IR code, the most mature of these is Clang (CLANG, 2019). Clang is used in production to build performance-critical software like Chrome or Firefox, for example. Figure 6.1a shows a simple example of a C function that takes two integer values as input and returns the sum of these elements. **(2)** ***Typed SSA IR.*** The LLVM intermediate code representation describes a program using an abstract RISC-like instruction set but with essential higher-level information for effective analysis. Figure 6.1b shows the transformation of the C example in LLVM IR code. **(3)** ***Optimizations/Transformations*** uses an intermediate code representation to describe the program. Through the so-called passes, this representation is traversed to collect information or perform transformations to the code. This is the core of the LLVM project. Figure 6.1c shows the IR code that results from the TMR technique applied to the first IR example code. **(4)** ***Code Generator*** transforms the architecture-independent internal representation into assembly code for a given target (e.g., x86, ARM, RISC-V). The register allocation, which is one of the explored features of this work, is done at this point. Figure 6.1d shows the protected Aarch64 assembly code generated.

Figure 6.2: Sub-figures (a,b,c,d) show an example for the LLVM steps.

(a) C code

```
int Sum( int a, int b){
    return a + b;
}
```

(b) LLVM IR

```
define i32 @Sum( i32 %a, i32 %b){
    entry:
    %add = add nsw i32 %b, %a
    ret i32 %add
}
```

(c) LLVM IR with TMR

```
define i32 @Sum( i32 %a, i32 %b){
    entry:
    %add = add nsw i32 %b, %a
    %add.DWC = add nsw i32 %b, %a
    %add.TMR = add nsw i32 %b, %a
    %cmp = icmp eq i32 %add, %add.DWC
    %sel = select i1 %cmp, i32 %add, i32 %add.TMR
    ret i32 %sel
}
```

(d) Aarch64 assembly

```
Sum:
//%bb.0:
    add w8, w1, w0
    add w9, w1, w0
    add w0, w1, w0
    cmp w8, w9
    cset w9, eq
    tst w9, #0x1
    csel w0, w8, w0, ne
    ret
```

Source: Authors.

6.1.3 Automated Software-Based Fault Mitigation

Our framework presents two selective TMR variants based on the VAR3+ technique from (Azambuja et al., 2011), as shown in Table 6.2. These techniques were chosen for their capability of increasing reliability while maintaining a low overhead. The **first**

version has the original VAR3+ definition, all instructions, except for branches and stores, are replicated (rules G1, D2). The replicas are checked before every load, store, or branch instruction (rules C3, C4, C5, C6). In the **second version**, VAR3+ technique can be applied selectively on one or more specific functions (e.g., critical tasks) instead of the entire application code. This approach can considerably reduce code size and execution time while maintaining similar reliability. For simplicity, we call these versions just `TMR` and `P-TMR`, respectively. These techniques can be activated with custom flags added to the Clang frontend, thus hardening applications with minimal effort.

Table 6.2: Techniques and Rules.

Technique	**Duplication Rule**	**Checking Rules**
VAR3+	D2	C3, C4, C5, C6

Source: CHIELLE; KASTENSMIDT; CUENCA-ASENSI (2016).

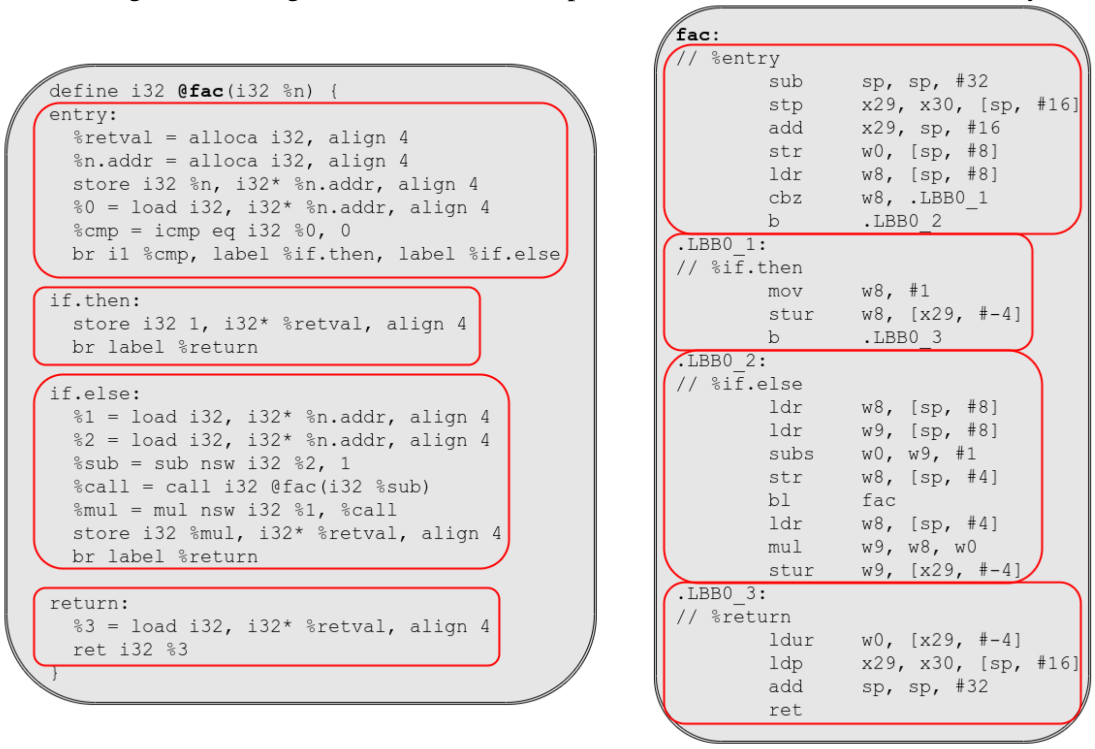
6.2 RAT - Register Allocation Technique

Register Allocation is executed during the Code Generation phase. It consists of mapping a program with an unlimited number of virtual registers (like in the LLVM IR) to a program that contains a limited number of physical registers of some particular architecture. Each target architecture has a different number of physical registers. If the number of physical registers is not enough to accommodate all the virtual registers, some of them will have to be mapped into memory. These virtual registers are called spilled virtuals.

The LLVM has several different register allocators, and by default, the *Fast* version is used most of the time. This allocator is called a local register allocator, which has the most straightforward strategy when compared with the others. It scans the program linearly and assigns virtual registers to physical registers at the basic blocks level, as shown in Figure 6.3.

In this work, the *Fast* register allocator was used as a basis to introduce the desired functionalities. When mapping virtual to physical registers, a function is called to search for a free register among all registers compatible with a particular type of value (for example, int32). At this point, a new condition is imposed. If the basic block being mapped belongs to the specific function passed as a parameter, the mapping can only be done using the register pool provided to the compiler. With this modification, the allocator will

Figure 6.3: Register allocation example – LLVM IR to Aarch64 assembly



```
define i32 @fac(i32 %n) {
entry:
  %retval = alloca i32, align 4
  %n.addr = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  %0 = load i32, i32* %n.addr, align 4
  %cmp = icmp eq i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:
  store i32 1, i32* %retval, align 4
  br label %return

if.else:
  %1 = load i32, i32* %n.addr, align 4
  %2 = load i32, i32* %n.addr, align 4
  %sub = sub nsw i32 %2, 1
  %call = call i32 @fac(i32 %sub)
  %mul = mul nsw i32 %1, %call
  store i32 %mul, i32* %retval, align 4
  br label %return

return:
  %3 = load i32, i32* %retval, align 4
  ret i32 %3
}
```

```
fac:
// %entry
        sub     sp, sp, #32
        stp     x29, x30, [sp, #16]
        add     x29, sp, #16
        str     w0, [sp, #8]
        ldr     w8, [sp, #8]
        cbz     w8, .LBB0_1
        b       .LBB0_2
.LBB0_1:
// %if.then
        mov     w8, #1
        stur    w8, [x29, #-4]
        b       .LBB0_3
.LBB0_2:
// %if.else
        ldr     w8, [sp, #8]
        ldr     w9, [sp, #8]
        subs    w0, w9, #1
        str     w8, [sp, #4]
        bl      fac
        ldr     w8, [sp, #4]
        mul     w9, w8, w0
        stur    w9, [x29, #-4]
.LBB0_3:
// %return
        ldur    w0, [x29, #-4]
        ldp     x29, x30, [sp, #16]
        add     sp, sp, #32
        ret
```

Source: Authors

be forced to use only the registers available in the defined pool. However, it is essential to note that if an inadequate number of registers or invalid registers names are set, the compilation process can be aborted. For this reason, an automated process was built to handle the analysis of dynamic and static registers' usage and finally apply the allocation technique.

In the **first step**, the disassembler code is analyzed to extract the number of general-purpose registers utilized by the defined critical function. With this information, it is possible to set the size of the desired customized register pool. In the **second step**, dynamic register profiling is done aiming to capture the application's most and least used registers.

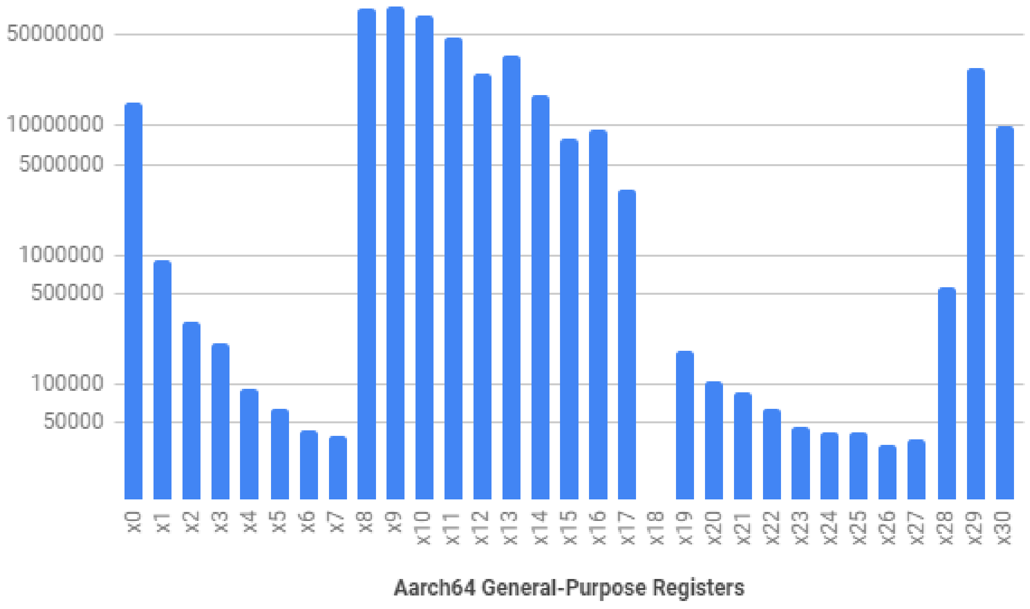
Different strategies can guide the selection of the pool. Figure 6.5 shows an example of general-purpose aarch64 registers' profile of an application after a normal compilation process, and the Figure 6.6 shows the registers' profile of the same application using a custom register allocation targeting the critical function with the register pool set to X18-X28, as expressed in the command line in Figure 6.4. The x-axis of the plots shows the 31 general-purpose 64bit registers from Armv8 ISA. The y-axis shows how often each register was referenced as a source or destiny in the entire application execution. When the technique is applied in the critical function (i.e., most executed by default), there are a usage reduction in the intermediary registers (X8-X17) and an increase in the last registers' usage (X18-X28).

Figure 6.4: Clang/LLVM command-line example to apply the custom register allocation.

```
clang --target=<ARCH> app.c -o app.elf -mllvm -regalloc=fastX
-mllvm -funcList='crit' -mllvm -regPool='X18,X19,...'
```

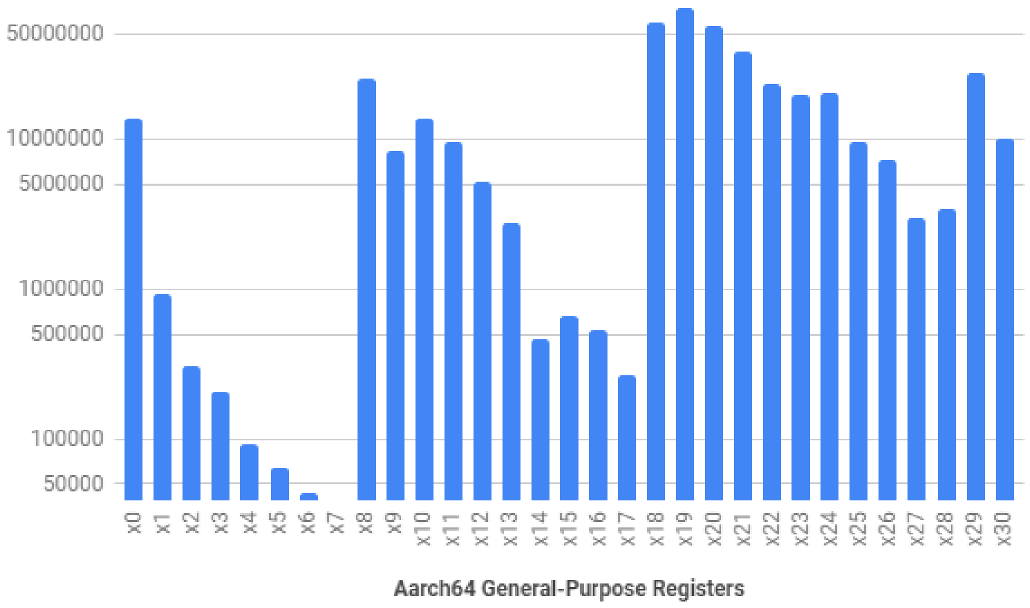
Source: Authors

Figure 6.5: Register's references for default register allocation in log scale.



Source: Authors.

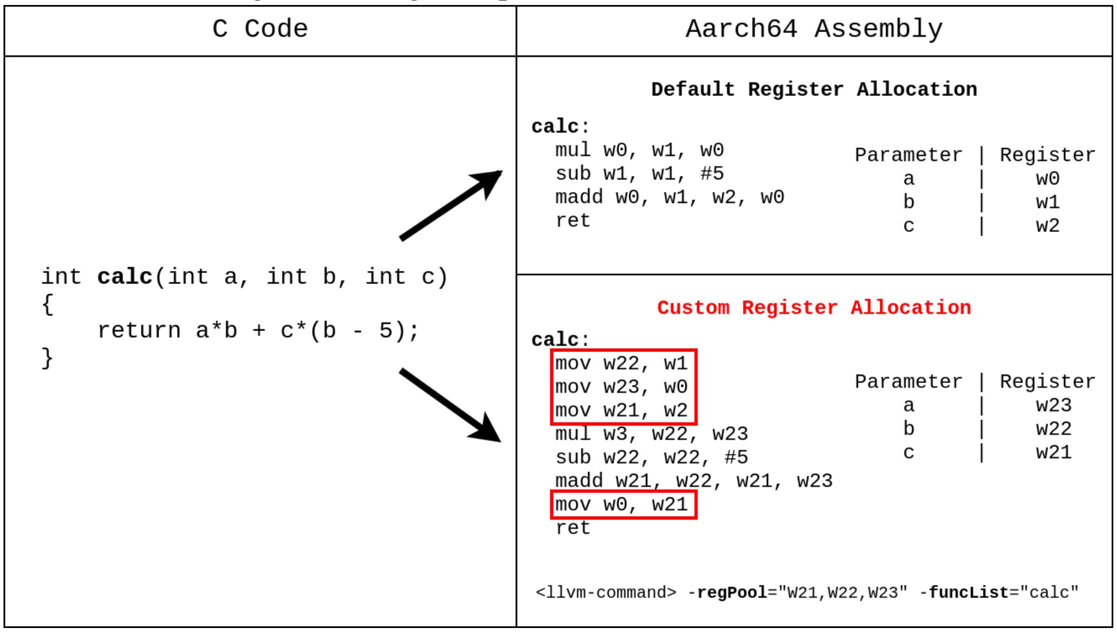
Figure 6.6: Register's references after applying RAT in log scale.



Source: Authors.

The left-hand of Figure 6.7 shows an example of a C language function that takes three integer parameters as input, performs arithmetic operations, and returns an integer value. The resulting 64-bit ARM (Aarch64) assembly code is shown in the right-hand of fig. 6.7, where at the top the default register allocation is shown. In turn, at the bottom right-hand of Figure 6.7 the RAT technique is applied, limiting the function register pool to “W21, W22, W23”. As mentioned before, `RAT` is a compiler-based mitigating technique, thus it can be associate with other techniques as well. Such capacity is explored in the Chapter 7.

Figure 6.7: Example of C code (left) conversion to Aarch64 assembly without (top right) and with (bottom right) `RAT` flags compilation.



```
int calc(int a, int b, int c)
{
    return a*b + c*(b - 5);
}
```

```
calc:
  mul w0, w1, w0
  sub w1, w1, #5
  madd w0, w1, w2, w0
  ret
```

```
Parameter | Register
    a     |    w0
    b     |    w1
    c     |    w2
```

```
calc:
  mov w22, w1
  mov w23, w0
  mov w21, w2
  mul w3, w22, w23
  sub w22, w22, #5
  madd w21, w22, w21, w23
  mov w0, w21
  ret
```

```
Parameter | Register
    a     |    w23
    b     |    w22
    c     |    w21
```

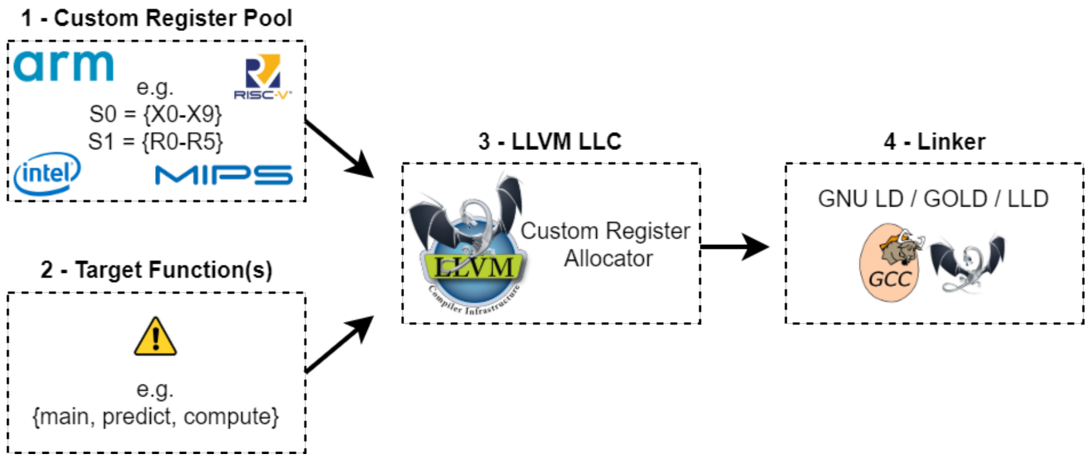
```
<llvm-command> -regPool="W21,W22,W23" -funcList="calc"
```

Source: Authors

6.2.1 RAT Custom Parameters

To be able to provide more flexibility to engineers, RAT enables some fine-grained customization to be applied in the target application through command-line parameters. As shown in Figure 6.8, the first parameter that can be modified manually is the registers' sets (pools) that restrict the application register allocation. These register pools need to be composed of registers available in the target architecture, and the minimum size of each pool needs to follow the application restriction. For example, some Arm standard instructions such as `ADD R0, R1, R2` need three registers to work. If a pool is set to `{R0, R1}`, the compiler will throw an error. Moreover, note that even when the compiler compiles the code correctly, the more you restrict the pool's size, the more the allocator needs to spill registers values to memory, which could significantly reduce the application's performance.

Figure 6.8: Custom parameters configuration flow for RAT.



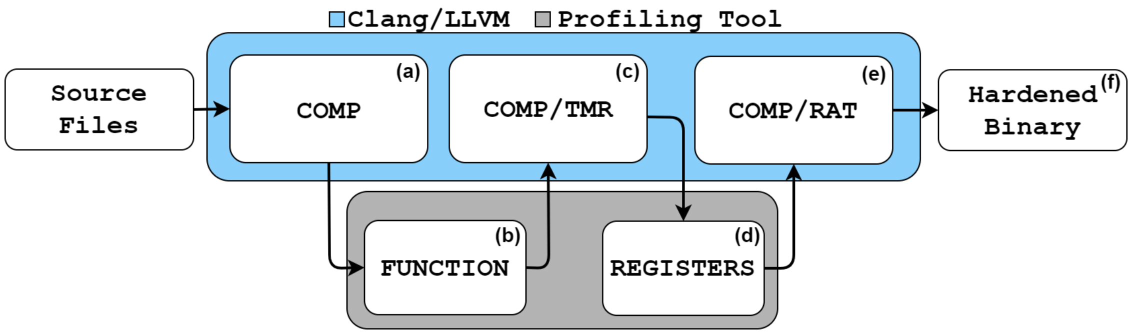
Source: Authors

The second parameter that can be set is the list of application functions (e.g., the critical ones) that the engineer wants to harden. This parameter is also passed by command-line to the LLVM static compiler (LLC). The functions are limited to those available in the program source code; that is, functions from external libraries do not suffer any effect as they are already in the machine code format. A functionality that was not introduced yet is the assignment of particular register pools to specific functions. After setting the parameters, the LLC produces the assembly code, and finally, the binary is generated by any linker compatible (e.g., LLD, GOLD, GNU LD).

6.3 Integration of Mitigation Techniques in the Proposed Framework

To apply the `P-TMR` technique of selective protection automatically, a series of steps must be followed (Figure 6.9): **(a) Default Compilation:** configure software stack and compile the application using any frontend compiler compatible with LLVM (e.g., Clang). **(b) Function Profile:** run the profiling tool with the trace flag enabled, this will capture the execution time of each of the application's function and determine which is the critical one. Note that the software engineer can either determine the most critical application function or use our toolset's default option, which selects the most executed one. **(c) Compilation TMR:** with the information provided by the profile, the framework automatically re-compiles the application with the required flags — i.e., applying the fault mitigation technique to the target function(s). One limitation of this approach is that its source code must be available and under the application's own scope to protect the function. This limitation can drastically affect the efficiency of the mitigation techniques and will be discussed in more details on Chapter 7. **(d) Register Profile:** since the last step changed the binary, it is needed to profile the application again to extract register's usage. **(e) Compilation with Register Allocation Technique (RAT):** In this step, a new compilation is performed taking into account the critical function and a register pool (i.e., least used registers). The underlying compilation uses a modified version of the LLVM Fast Register Allocator, which considers arguments (i.e., restrictions) that are passed to LLC (LLVM Static Compiler) through a command line, as shown in Figure 6.7. **(f) Hardened Binary:** Finally, the resulting hardened binary is generated by the LLD linker.

Figure 6.9: Framework cross compilation flow for application hardening. Blue box shows steps executed by Clang or LLVM, while grey box by profiling tools.



Source: Authors

7 SOFT ERROR MITIGATION MODULE VALIDATION

This Chapter evaluates the effectiveness of the developed soft error mitigation module and techniques considering different case studies: (i) Bare metal vs. OpenMP linux applications (Section 7.1) . (ii) ARMv7-A vs. ARMv8-A (Section 7.2). (iii) ML in resource-constrained devices (Section 7.3). (iv) RAT customization analysis (Section 7.4).

7.1 Case Study I - Considering Distinct Software Stacks

This Case study considers two different sets of benchmarks (WCET (GUSTAFSSON et al., 2010) and Rodinia (Che et al., 2009)) using two reliability metrics (MWTF and EAFC). To showcase the proposed framework's range, we select benchmarks with different characteristics. WCET benchmark do not consider an operating system, single-core processor, and the applications are relatively small w.r.t. the other benchmark; however simple each application is, they comprise the basis for more complex applications — for example, matrix multiplication and linear equation solver are the basis for many machine learning routines. On the other hand, Rodinia benchmark leverage the use of external libraries to manage multicore processing in parallel, thus adding a new layer of complexity as well as a Linux operating system.

Table 7.1 shows the experimental setup; with 3.1k fault injections per campaign, our data has a 99% confidence level and a 2.3% error margin according to (LEVEUGLE et al., 2009a). This work assumes that an error-correcting code protects the main memory, thus the targets for fault injection are the general-purpose registers X0-X30.

7.1.1 WCET

The applications selected for this section comes from the real-time benchmark suite WCET. In Table 7.2 we have a brief description of each one. They are concise applications with distinct behaviours (e.g., loop, recursion, arrays), and no external library dependencies (e.g., `string.h, math.h`). Due to the lack of operating system and external calls, our framework has access to 100% of the code that is executed, which ensures a fine grain control over the flow and fault mitigation techniques for soft error

Table 7.1: Experimental Setup

Framework Setup	
Processor	Arm Cortex A72
Compiler	Clang/LLVM 6.0
Fault Model	Random Single Bit-Flip
Fault Injection per Scenario	3100
WCET Benchmark	
Core Variants	Single-Core
OS	None
Mitigation Techniques	`TMR`, `P-TMR`
Number of Applications	25
Rodinia Benchmark	
Core Variants	Single, Dual and Quad-Core
OS	Linux (kernel 4.3)
Mitigation Techniques	`RAT`, `TMR(+RAT)`, `P-TMR(+RAT)`
Number of Applications	13

Source: Authors

protection.

Table 7.2: Reference Baremetal Benchmarks

#	Benchmark	Description	Exec. Instr. (x10^3)
1	adpcm	Adaptive pulse code modulation algorithm.	292.37
2	binary_search	Binary search for an array of integer elements.	174.03
3	bit_manipulation	Complex embedded code.	365.82
4	blowfish	Encryption algorithm.	1008.40
5	bubble	Bubblesort program.	261.23
6	compress	Data compression program.	320.18
7	counts	Counts non-negative numbers in a matrix.	243.04
8	crc	Cyclic redundancy check computation.	253.36
9	edn	Finite Impulse Response (FIR) filter calculations.	171.28
10	expint	Series expansion for computing an integral function.	346.00
11	factorial	Calculates the factorial function.	347.88
12	fdct	Fast Discrete Cosine Transform.	474.53
13	fibonacci	Simple iterative Fibonacci calculation.	324.52
14	hanoi	Tower of Hanoi puzzle game.	704.68
15	harm	Harmonic calculations with recursive calls.	321.89
16	insert_sort	Insertion sort on a reversed array.	271.16
17	jfdct_int	Discrete-cosine transformation.	224.04
18	matrix_mult	Matrix Multiplication 20x20.	345.05
19	mdc	Calculates the greatest common divisor.	176.35
20	peakspeed	Memory bounded program.	20.59
21	petri_net	Simulate an extended Petri Net.	233.15
22	prime	Calculates whether numbers are prime.	712.79
23	switch_cases	Control bounded program.	5339.43
24	ud	Linear Equations by LU Decomposition.	509.87
25	usqrt	Sqrt calculation without multiplications or divisions.	257.98

Source: Authors.

Table A.1 shows all results for unhardened binary (`R`) and with the mitigation techniques (`P-TMR` and `TMR`). Although `TMR` generally offers a better MWTF and EAFC, the difference to the `P-TMR` results is minimal in the majority of cases. The average normalized runtime, code size, MWTF, and EAFC is about 2.76×, 1.24×, 5.97×, and 1.32×, respectively for `TMR`; and 2.51×, 1.16×, 4.76×, and 1.33× for `P-TMR`. However, on a few applications (highlighted in blue), protecting only the critical function (`P-TMR`) has better reliability vs performance trade-off than `TMR`. Also, applications 6, 11, (highlighted in grey) and 14 (in blue) had worse results in terms of MWTF and EAFC on both mitigation techniques. These results indicate that adding protection can also create new points of failure. The TMR majority voters implementation for the ARMv8-A ISA uses four instructions, making the application more vulnerable if there are a significant number of voters in the code.

Analyzing the average of all normalized results gives an overview of the effectiveness of the implemented mitigation techniques. Comparing the P-TMR technique with TMR, we observed a 9% increase in MWTF, 7.5% in code size, and 10.5% in runtime. Although the average difference in code size seems small, note that some applications are so simple that they have only a main function, or the main just calling another function. This characteristic of some benchmarks ends up diluting this gain, which is quite significant in some cases. For example, the `adpcm` benchmark which has code size overhead equals to 2% and 21.2%, for P-TMR and TMR respectively.

Table 7.3: Summary of aggregated fault injection effect results, for the bare metal applications.

	Reference					**TMR**					**P-TMR**				
#	Van	ONA	OMM	UT	Hang	Van	ONA	OMM	UT	Hang	Van	ONA	OMM	UT	Hang
Max	99.25	8.67	27.00	22.00	3.38	99.62	1.43	16.93	10.38	5.38	99.62	1.78	18.38	13.54	4.38
Min	50.83	0.00	0.00	0.00	0.00	79.42	0.00	0.00	0.00	0.00	72.79	0.00	0.00	0.00	0.00
Avg	85.58	1.98	6.90	5.20	0.34	93.83	0.42	3.10	2.38	0.28	93.22	0.47	3.39	2.62	0.30
Std	13.20	2.62	9.20	6.54	0.84	5.81	0.41	4.52	2.79	1.08	6.80	0.57	5.07	3.29	0.93

Source: Authors.

Looking at Table 7.3, we have an aggregate of the fault injection effect results for all unprotected and protected bare metal benchmarks. The most important points of this table are the values of Van and OMM (which include the SDC). While in the reference version R, we have an application with a minimum fault-masking rate of 50.83%, in versions TMR and P-TMR, we have a minimum of 79.49% and 72.79% respectively. Comparing the OMM values, we observe that the average and maximum values are 6.90% and 27.00%, respectively, for the Reference version. And for TMR and P-TMR, there is an abrupt decrease in the number of errors that propagate until the end of the application—the average and maximum values are around 3% and 17%, respectively, for both protection techniques.

In (CHIELLE; KASTENSMIDT; CUENCA-ASENSI, 2016), different software mitigation techniques were applied to benchmarks running on an Arm Cortex-A9 CPU system. Fault injections were made by simulation and heavy-ion radiation testing. Their results showed that there is a significant impact on system reliability due to cache memory that is not modelled on simulators like OVP, for example. Yet they conclude that the use of simulation is acceptable for comparing mitigation techniques.

7.1.2 Rodinia

There are many benchmark suites for parallel computing on general-purpose architectures. Rodinia targets heterogeneous computing infrastructures with support to multicore architectures, thus adding complexity when compared to WCET applications. The applications run on top of a Linux operating system and use the OpenMP parallelization library. Table 7.4 shows the domain and the average number of instruction for each application/kernel compiled with O0 optimization flag on single-core platform.

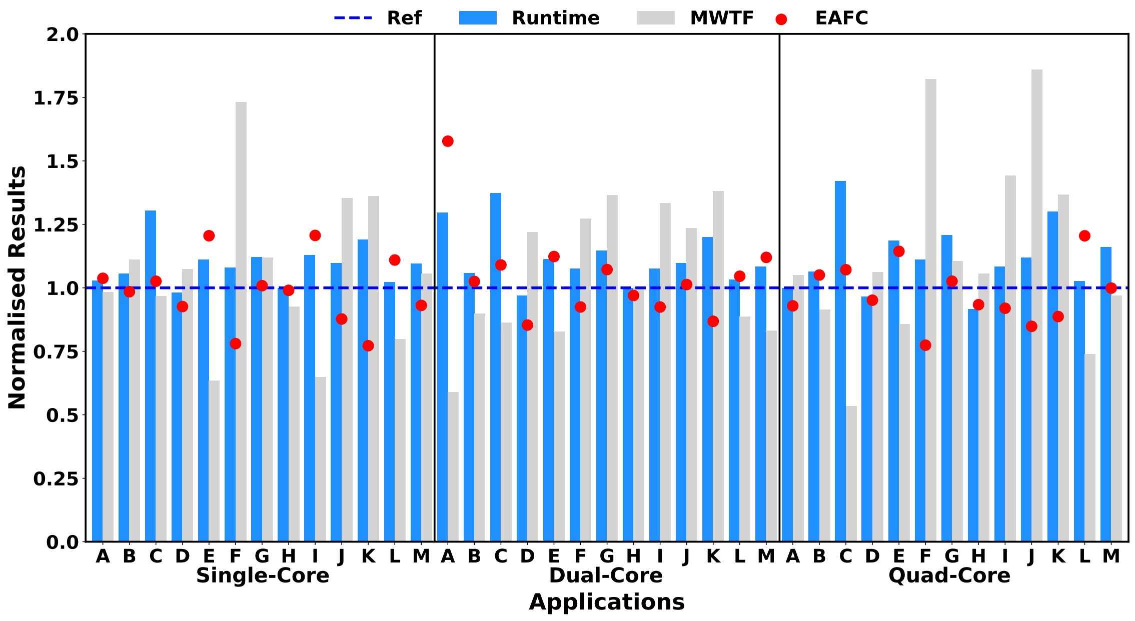
Table 7.4: Reference Rodinia Benchmarks

#	Application	Domain	Average Executed Instructions (10^6)
`A`	Backprop	Pattern Recognition	32.24
`B`	BFS	Graph Algorithms	66.02
`C`	HeartWall	Medical Imaging	81.34
`D`	HotSpot	Physics Simulation	69.15
`E`	HotSpot3D	Physics Simulation	164.37
`F`	Kmeans	Data Mining	40.28
`G`	LUD	Linear Algebra	43.28
`H`	Myocyte	Biological Simulation	31.77
`I`	NN	Data Mining	96.03
`J`	ParticleFilter	Medical Imaging	60.73
`K`	PathFinder	Grid Traversal	47.35
`L`	SradV1	Image Processing	200.16
`M`	SradV2	Image Processing	47.88

Source: Authors.

Figures 7.1 and 7.2 show the normalised runtime, MWTF, and EAFC for the two mitigation techniques considering single-core, dual-core, and quad-core platforms. Comparing the normalised average runtime results ($1.11\times$ vs $1.04\times$), MWTF ($1.08\times$ vs $1.07\times$) and EAFC ($1.01\times$ vs $0.97\times$) between the `TMR` and the `P-TMR`, we see a small difference, but when we observe isolated cases, some discrepancies appear. For example, in the application F running on quad-core, we have a normalised MWTF and EAFC of $1.82\times$ and $0.77\times$ for `TMR`, and of $0.90\times$ and $1.09\times$ for `P-TMR`. This behaviour shows that protecting only the most performed function is insufficient in some applications. However, observing the application K in single-core, we have a normalised MWTF and EAFC of $1.36\times$ and $0.77\times$ for `TMR`, and of $1.67\times$ and $0.72\times$ for `P-TMR`. Thus, different mitigation techniques need to be tested to ensure which one best fits to each case.

Figure 7.1: `TMR` reliability comparison with reference.

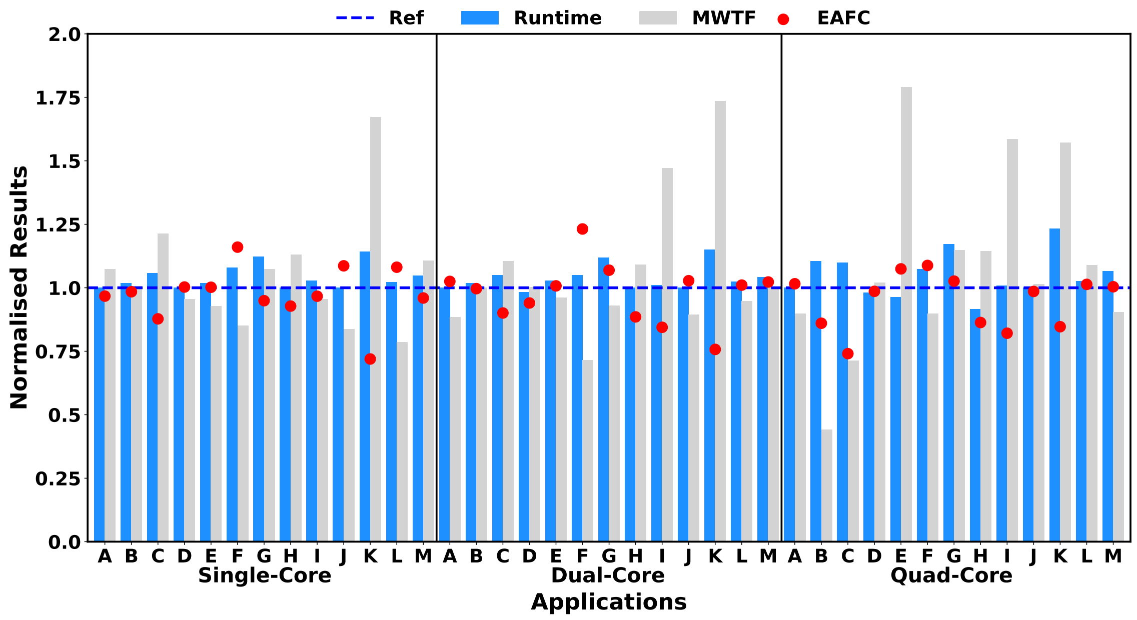


Source: Authors

Another point to note here is that in some cases (D, H), the protected program's execution time is less than the execution time of the reference version. This behaviour occurs as each FI campaign uses a different binary, thus the scheduling may also be different. In these specific cases, almost 100% of the execution time was dedicated to function calls from libraries responsible for I/O communication and related to physics (math functions). That is, the `TMR` was applied to something that runs less than 1% of the time, making the protection ineffective.

Figure 7.3 shows the code size increase comparison that highlights one of the main differences between these two techniques evaluated. In the Rodinia benchmark, the trade-off between reliability and code size is evident. For instance, application K has the highest reliability gain overall. Its code size increases by only 14.15% when applying the `P-TMR` technique in a single-core processor while `TMR` has a similar reliability gain, but a code size increase of 35.23%. Similar behaviour occurs for all benchmark applications, thus

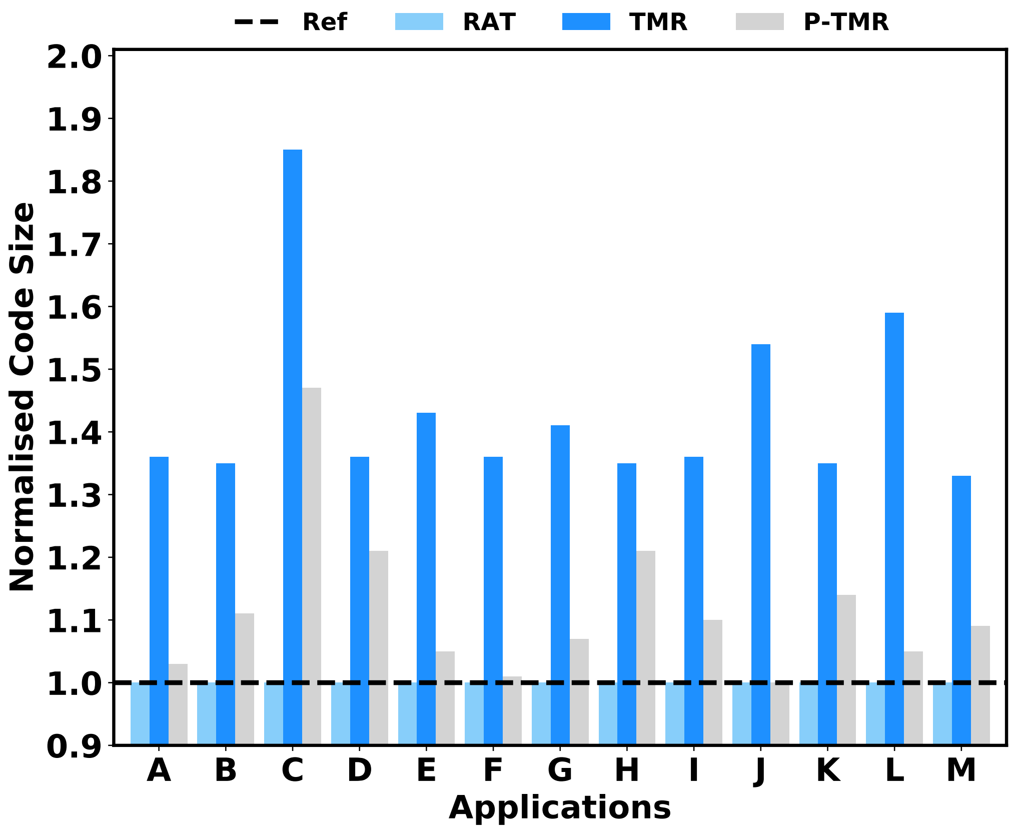
Figure 7.2: `P-TMR` reliability comparison with reference.



Source: Authors

`P-TMR` technique has a better trade-off between reliability and code size.

Figure 7.3: Comparison between Rodinia's applications code size increase for each mitigation technique.



Source: Authors

When comparing the WCET and Rodinia reliability results, a massive reduction can be seen in the average normalised MWTF gain from $5.97\times$ (WCET) to $1.08\times$ (Rodinia). This effect occurs due to a large number of calls to external functions in more complex applications (e.g., math/physics libraries). The presented mitigation techniques cover only the functions inside the application's scope. There are two possible solutions to this problem. Our tool allows function calls replication; however, aside from the considerable performance overhead, there are many possible collateral damages to this approach (e.g., modifying the same data structure multiple times). The other solution would be to compile the library using our mitigation techniques; however, library hardening falls out of this work scope.

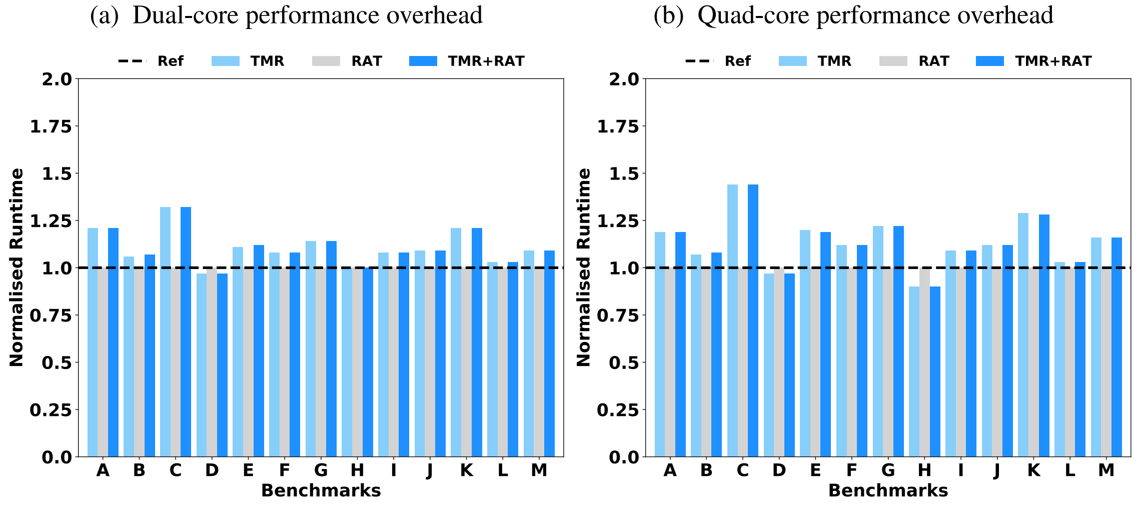
7.1.2.1 RAT Performance Overhead

To provide relevant overhead measures, the **performance** figures were obtained from the gem5 full system simulator (BINKERT et al., 2011b). Results in Figure 7.4 show that the use of the `TMR` can lead to up to 38.5% and 50% of performance penalty (benchmark C) when running on dual and quad-core ARM Cortex A72 processors. The reason why there is an increase in the execution time in the quad-core when compared to the dual-core is due to the increasing execution of OS thread synchronization routines that is not linear with the number of cores. Note that the additional execution time of `TMR` is small for a technique that triples instructions and inserts voters into the code. This is justified by the fact that only instructions inside the application's scope are replicated, and the majority of Rodinia applications rely on external library calls. One possible solution to this problem implies replicating function calls; however, there are possible collateral damages inherent to this approach (e.g., modifying the same data structure multiple times).

7.1.2.2 RAT Soft Error Reliability Evaluation

Figures 7.5 and 7.6 show the reliability comparison between the three mitigation techniques. In terms of MWTF on Figure 7.5, the `TMR` implementation provides higher reliability in 5 out of 13 cases (C, D, F, I, K), while the `RAT` in 4 cases (A, E, J, L), and the `TMR+RAT` in the other 4 cases (B, G, H, M). Results show that RAT can also provide reliability improvements of up to 40% in some cases w.r.t. `TMR`. Results also show that, depending on the application nature, `TMR+RAT` is an appropriated combination to improve system reliability. For instance, taking the benchmarks B and K as examples, it is possible to identify a considerable difference in the MWTF gain when comparing

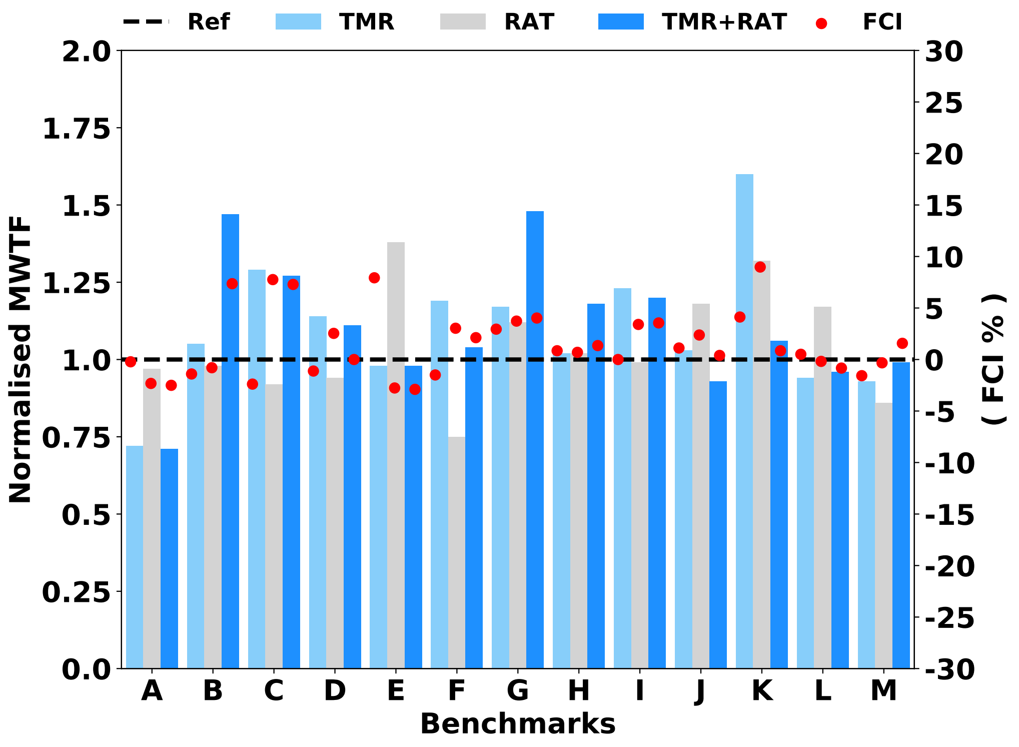
Figure 7.4: Performance overhead for dual (b) and quad-core (c) ARM Cortex-A72 processor when comparing the impact of the mitigation techniques w.r.t. the original reference benchmark (**–Ref**).



Source: Authors

the two TMR implementations. While benchmark B showed a reliability improvement of 40% for `TMR+RAT`, the use of `TMR` provides an improvement of 51% for K.

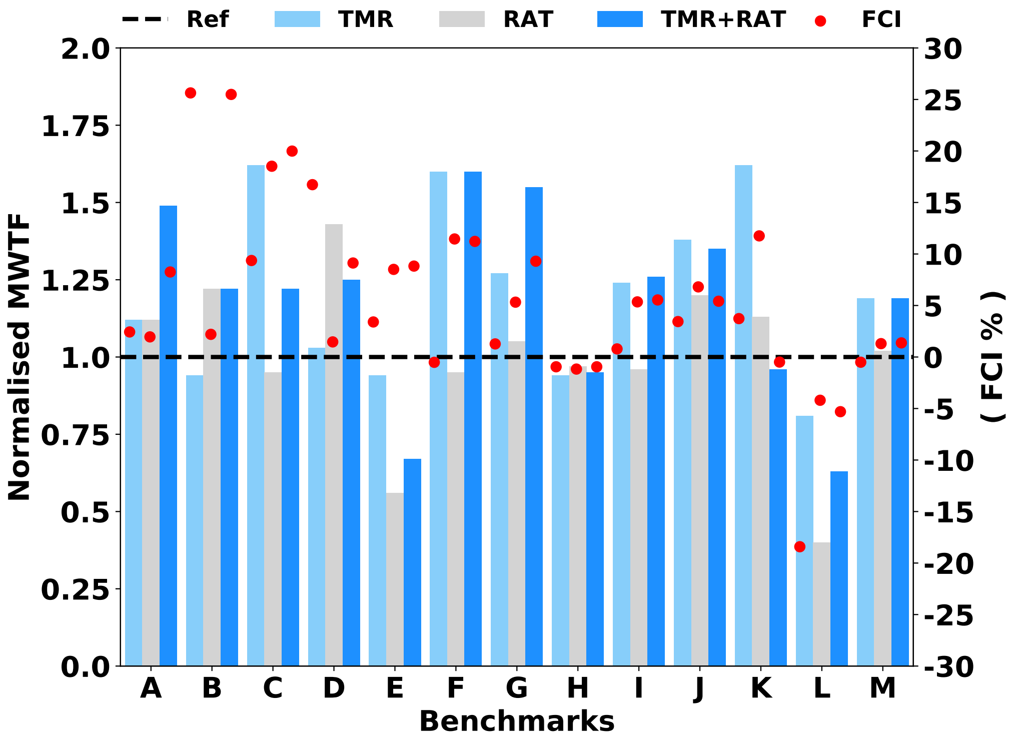
Figure 7.5: Normalized reliability comparison between each technique considering the original benchmark code as reference (**–Ref**) for a dual-core ARM Cortex-A72.



Source: Authors

Figure 7.6 shows a significant increase in the FCI average compared to the results in the dual-core processor, 5.47% versus 1.48%. Note that all reliability metrics have

Figure 7.6: Normalized reliability comparison between each technique considering the original benchmark code as reference (**–Ref**) for a quad-core ARM Cortex-A72.



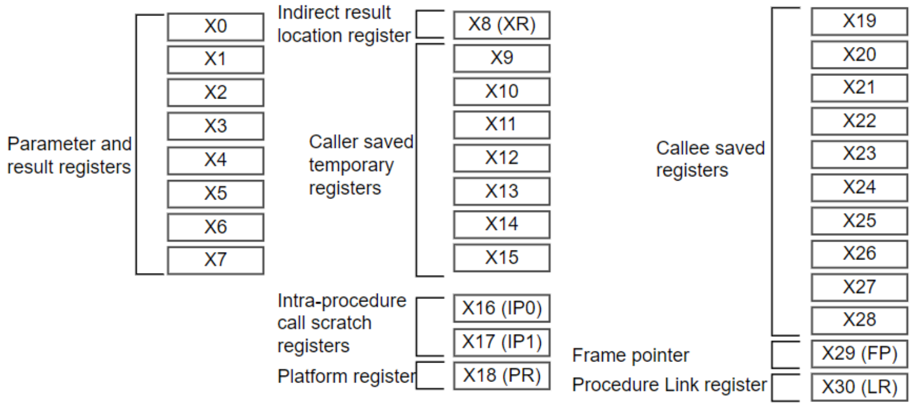
Source: Authors

been reduced from dual-core to quad-core, the increase was only about the reference benchmark. This behavior occurs precisely due to the rise in the execution of thread management tasks, which have a higher susceptibility to soft errors, as mentioned earlier. The TMR technique obtained better reliability results in 6 of the 13 benchmarks (C, E, F, J, K, L), RAT was better in 2 cases (D, H), and TMR+RAT was better in 5 cases (A, B, G, I, M). Note that the applications' reliability varies from one mitigation technique to another. For that reason, we claim that engineers can use our toolset to analyze the impact of different mitigation techniques at the system-level, so they might be able to identify the most suitable one considering their application/system's constraints. Further, a more in-depth analysis is carried out, verifying the results of the fault injections in each register for a specific case study.

7.1.2.3 Register Criticality Analysis

Figure 7.7 shows how the 64-bit ARM (AArch64) calling convention works. The X0-X7 registers are used for input parameters and return functions; the X8 is used to hold an indirect return location address; the X9-X15 are used to hold local variables (caller saved); the X16 and the X17 are the Intra-Procedure-call scratch registers; the X18 can be used for some OS-specific purpose; the X19-X28 are callee-saved registers; the X29 is the frame pointer; while the X30 is the link register, used to return from subroutines. To better explain the RAT benefits, we chose the particlefilter benchmark (**J**) as as case study.

Figure 7.7: Allocation of the general-purpose registers following the AArch64 calling convention. (ARM, 2020)

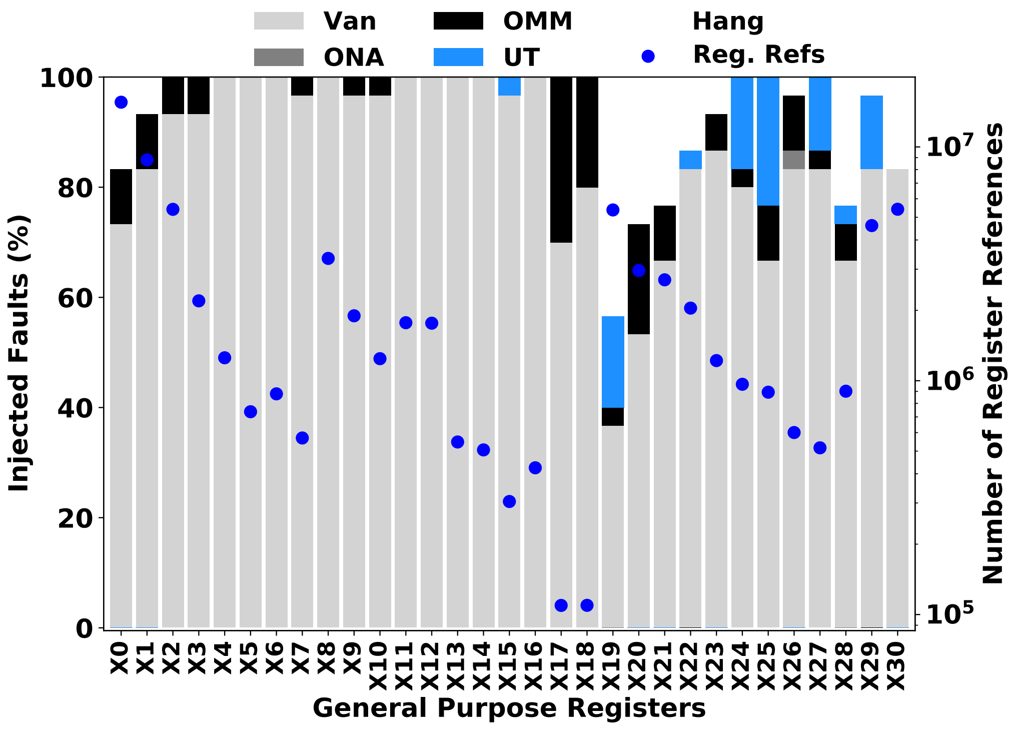


Source: Authors

The results show that half of the registers (X0-X16) do not suffer significantly from soft errors (Figure 7.8), when the particlefilter benchmark (**J**) is executed on a **dual-core** ARM Cortex-A72 processor. In contrast, the rest of the registers strongly suffers from the injected faults. Especially the callee-saved category that are used to hold long-lived values that must be preserved across calls and are used by the Linux kernel. That is, theoretically, they are registers that take a longer time to get written, but they are continuously read. However, as shown in Figure 7.9, the fault masking increases when we apply the RAT technique and limit the number of registers that will be used to execute the most performed function. In general, this effect occurs because when entering the critical function, the callee-saved registers are saved in memory and return to their original values at the end of the execution. In practice, this behavior ends up reducing the lifetime of these registers, making them more resilient to soft errors. The best examples are from the X17 and X19 registers. For the X17, we have a fault-masking rate of 70% in the reference application, and 98% when using the RAT mitigation technique. For the X19 register, we have a fault-masking rate of 36.67% in the reference application, and 58% when using the RAT technique.

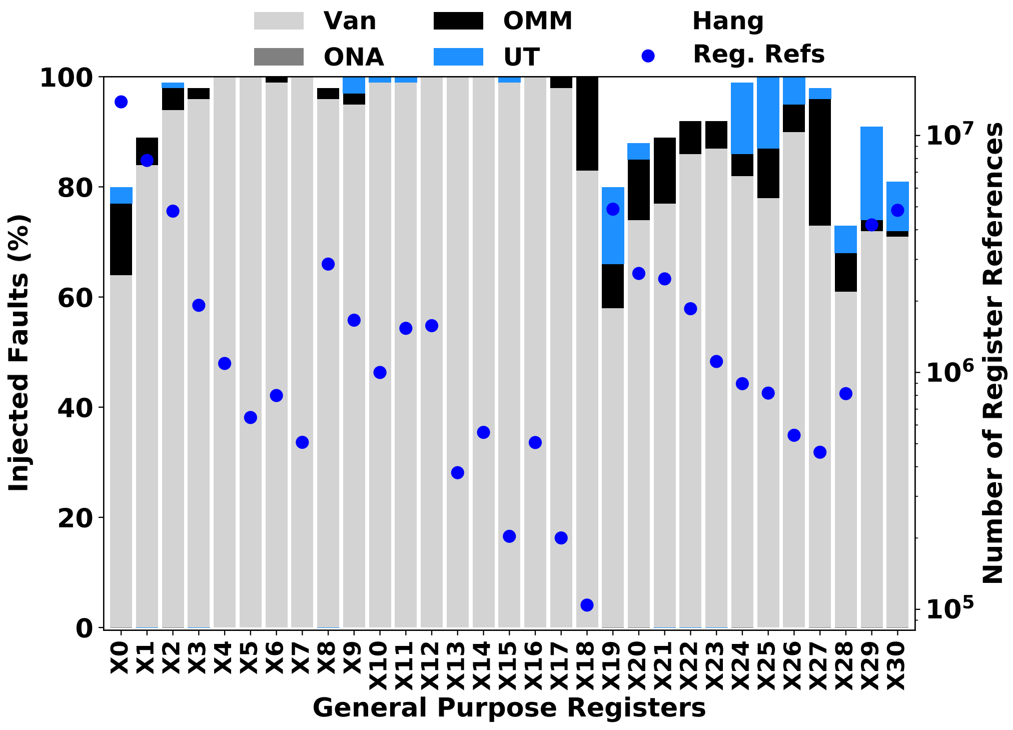
Results demonstrated that RAT reduces the code size and performance overheads while providing reliability improvement when considering a state-of-the-art 64-bits processor, which has a large register pool (i.e., 32 general-purpose registers). Researchers and industrial leaders are also developing optimized machine-learning algorithms (ABICH et al., 2020), aiming to enable their execution in resource-constrained devices. The resulting

Figure 7.8: Registers criticality for the **Reference** particlefilter benchmark running on a dual-core ARM Cortex-A72.



Source: Authors

Figure 7.9: Registers criticality for the **RAT** version of particlefilter benchmark running on a dual-core ARM Cortex-A72.



Source: Authors

scenario calls for lightweight soft error mitigation techniques such as the one proposed here. The next Section investigates the RAT efficiency when applied to a more resource-constrained architecture.

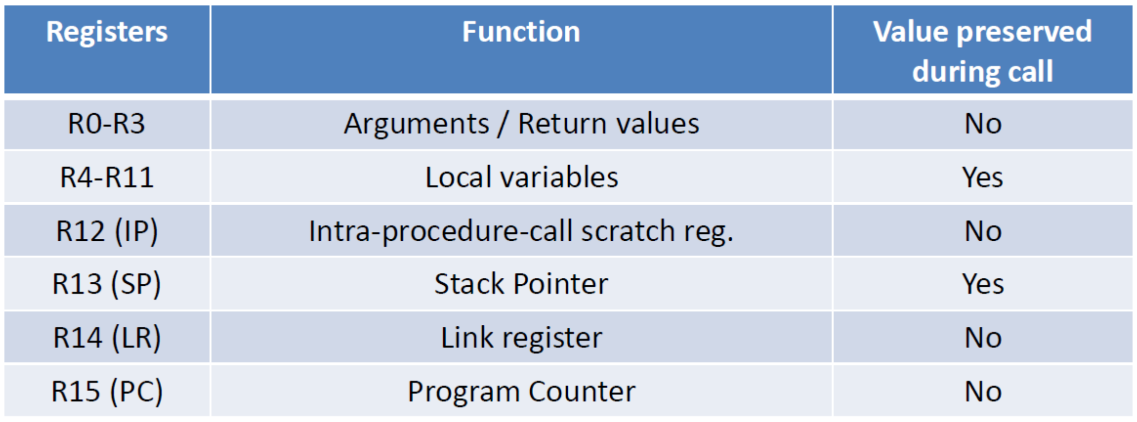
7.2 Case Study II - Considering Distinct Processor Architectures

To assess the impact of the processor architecture on RAT efficiency, this Section considers the ARMv7-A 32-bit and the ARMv8-A 64-bit instruction set architectures.

7.2.1 ARMv7-A General-purpose Registers

The ARMv7-A has 16 registers (R0-R15) with 32 data bits each. Removing the special use registers (IP, SP, LR, PC), there are only 12 extra registers that RAT can use to allocate the application critical function. As explained in the Section 7.1.2.3, there is also a particular ARMv7-A calling convention. As shown in Figure 7.10, the initial registers (R0-R3) are used to pass input and function return parameters, the R4-R11 are used for local variables, and the R12-R15 are special registers responsible for managing stack, function return address, and jumps during the application execution.

Figure 7.10: Register usage for ARMv7 architecture.



Registers	Function	Value preserved during call
R0-R3	Arguments / Return values	No
R4-R11	Local variables	Yes
R12 (IP)	Intra-procedure-call scratch reg.	No
R13 (SP)	Stack Pointer	Yes
R14 (LR)	Link register	No
R15 (PC)	Program Counter	No

Source: Authors

For example, if a routine has more than four arguments, besides using R0-R3, the stack will need to be used to store the extra parameters. Moreover, if R4-R11 are not sufficient, R0-R3 and R12 can be used, and even LR when there are no other subroutine calls.

7.2.2 Soft Error Reliability Assessment for the ARMv7-A considering different Mitigation Techniques

In order to understand how limiting the number of available registers affects the soft error reliability results, the experiments consider a subset of seven applications of the Rodinia Benchmark Suite executing in dual-core and quad-core Arm Cortex-A9 processors. For each scenario, 1600 SEU fault injections were performed targeting the 16 general-purpose registers. Based on the equation defined in (LEVEUGLE et al., 2009a), our results have a margin of error of 2.45% with a 95% confidence level.

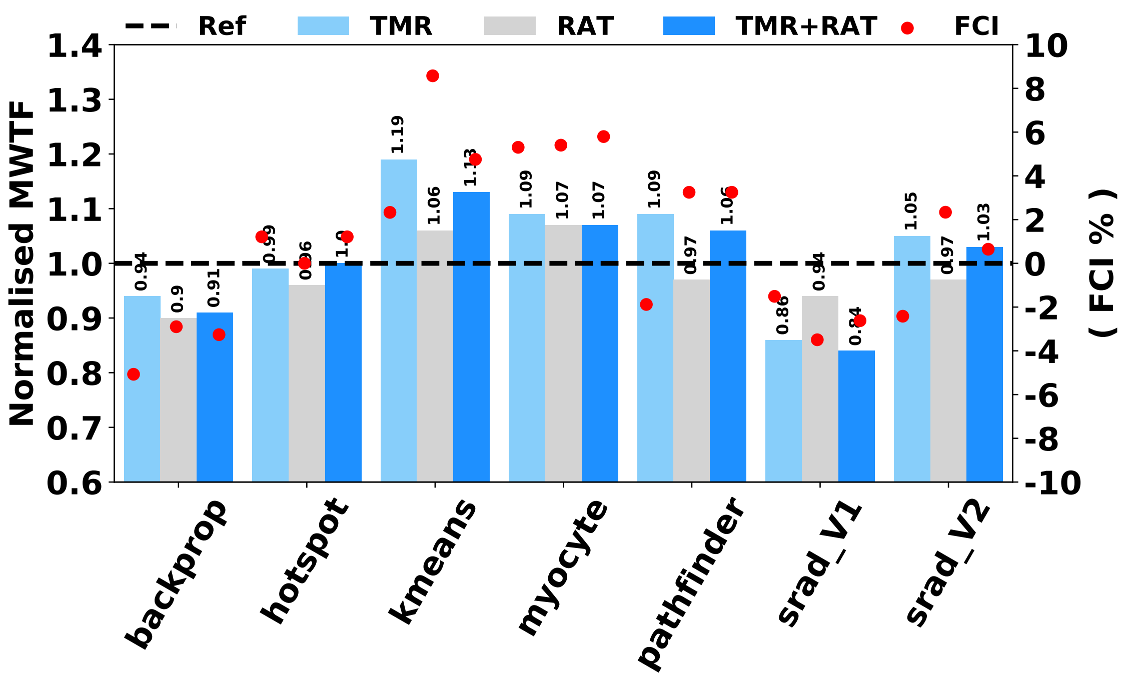
Figure 7.11 shows the MWTF results normalized by the reference application version indicated on the left y-axis. Each bar in the graph indicates a mitigation technique, and each group of three bars refers to a different application. The right y-axis shows the increase/decrease in the fault coverage for each case, which are indicated by the red dots. Following the same format adopted in the previous Section, results consider dual-core and quad-core processor architectures and the three soft error mitigating techniques (i.e., TMR, TMR-RAT and RAT). For the dual-core results (Figure 7.10a), it is possible to observe a low soft error reliability improvement when applying the mitigation techniques (see MWTF and FC values). While TMR presents the higher MWTF factor for the *kmeans* application (19%), RAT shows the best FCI factor for the same application (9%).

The application of `RAT` leads to a low reliability improvement (MWTF factor equal to 7% - best case) at a low extra code overhead. The low reliability improvement is expected; since the number of available registers is low, the registers' allocation can be precisely the same as the reference version if the function defined as critical already uses all possible registers.

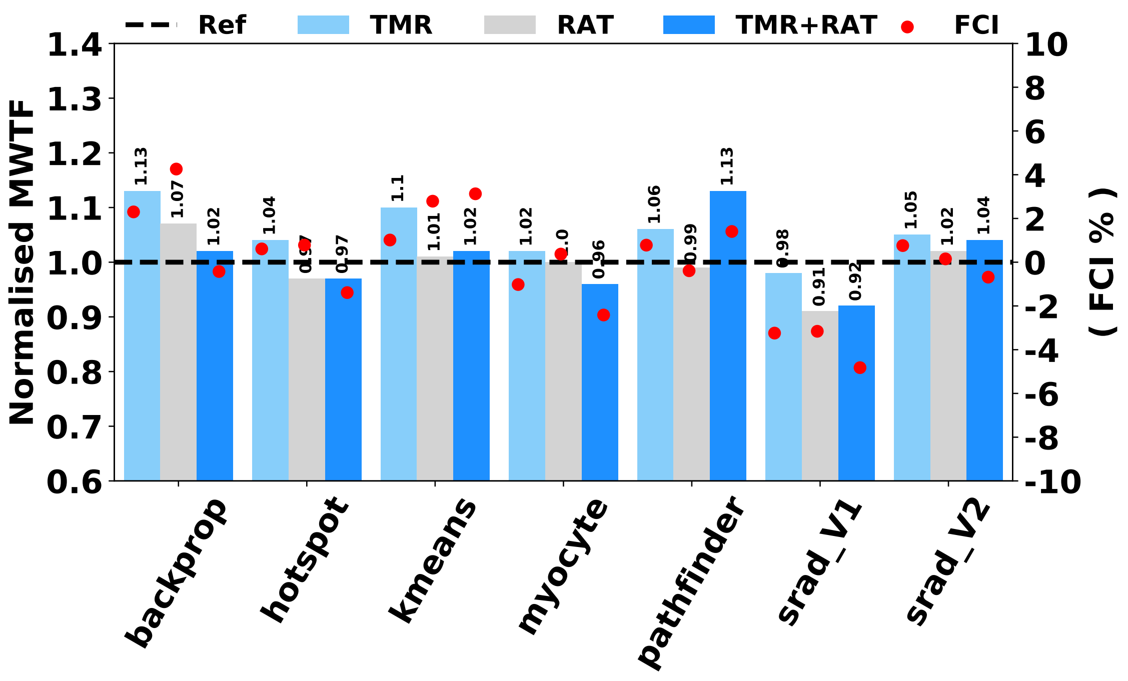
Quad-core soft error reliability results (Figure 7.10b) provide a lower MWTF and FCI average w.r.t. the dual-core configuration. The more cores the higher is the probability of a fault happen during the operating system execution. In this case, the operating system puts more pressure on the registers, leading to more spilling to store temporary values in memory, thus requiring an increase in the proportional time slice of the application's total execution. This reduces the chance of a fault being masked within one of the hardened functions. For instance, the best achieved FCI factor is only 4% when RAT is applied to the backprop application. In turn, the higher MWTF factor of 13% is achieved when TMR is applied for the same application.

Figure 7.11: Reliability improvement for dual (a) and quad-core (b) Arm Cortex-A9 processor when comparing the impact of the mitigation techniques w.r.t. the original reference benchmark (**–Ref**).

(a) Dual-core reliability results



(b) Quad-core reliability results



Source: Authors

7.2.3 RAT Efficiency Comparison: ARMv7-A *vs* ARMv8-A

The purpose of this section is to make a more detailed comparison of the reliability results when applying TMR, TMR-RAT and RAT techniques to seven Rodinia applications running on different processor architectures.

The Figure 7.12 shows the normalized MWTF of each application (i.e., unprotected and protected versions) obtained from the fault injection campaigns considering the Arm Cortex-A72 and the Arm Cortex-A9. Each bar in the 4-bar structure of the graph indicates a different version of each application.

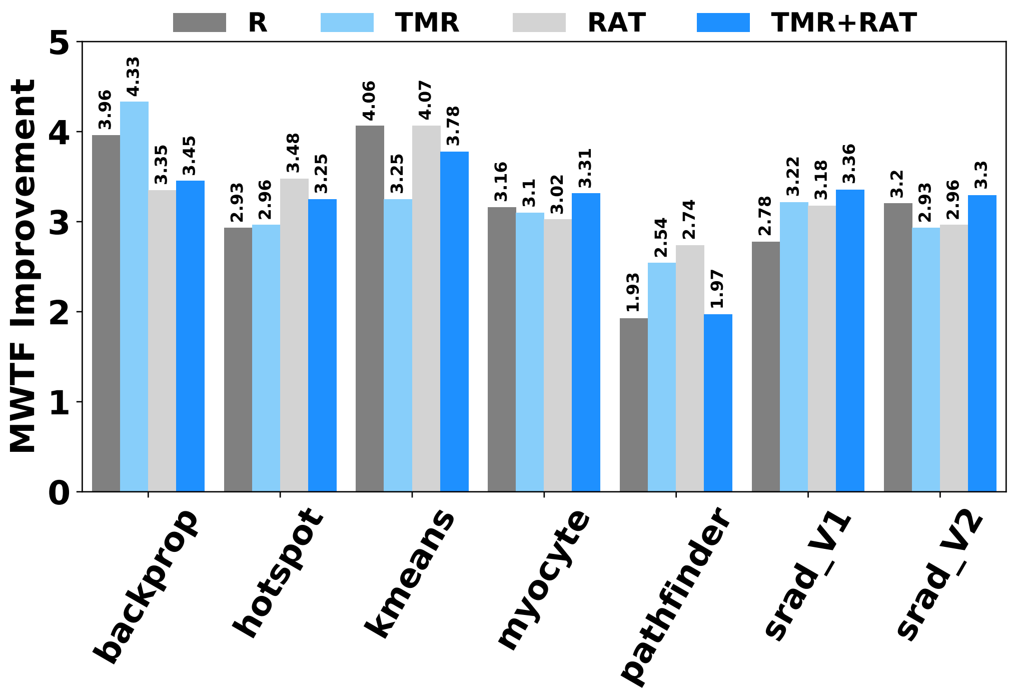
Analyzing Figure 7.11a, we see that the ARM Cortex-A72 dual-core provides a significant MWTF improvement in all applications. The minimum increase is $1.93\times$ (pathfinder - R), and the maximum $4.33\times$ (backprop - TMR). Results show RAT can benefit from processors with a larger number of registers. Results obtained from the quad-core processor scenarios (Figure 7.11b), show a reasonable reduction in the MWTF improvement. The minimum reliability improvement of $0.92\times$ is achieved when applying RAT to the hotspot application. In turn, the use of TMR+RAT incurs in an improvement of $3.11\times$ for the kmeans application. Therefore, the increase of system resource utilization leads to a decrease of more than 70% in the normalized MWTF in some cases (i.e., hotspot and myocyte).

7.3 Case Study III - Considering Machine Learning Applications

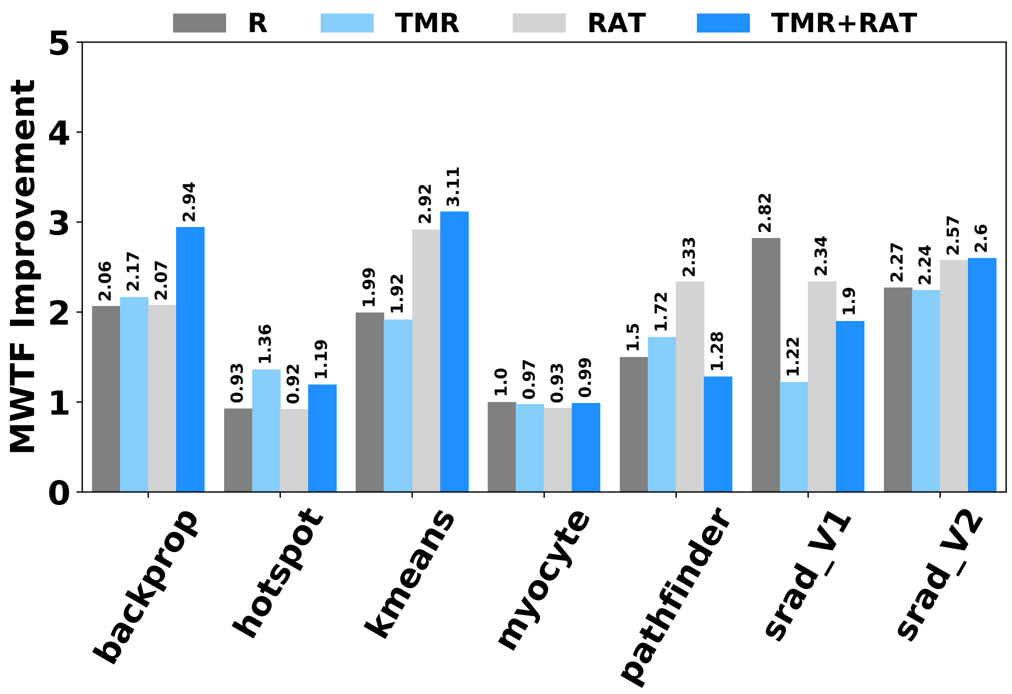
The number of products integrating Machine Learning (ML) algorithms is continuously increasing. With this in mind, researchers have started to investigate the impact of radiation-induced soft errors on the reliability of ML algorithms. For example, (LI; HARI et al., 2017) examined soft error effects on an accelerator for a CNN, while (SANTOS; PIMENTA et al., 2018) assessed the reliability of a CNN on a Graphics Processing Unit (GPU). (Da Rosa et al., 2019) investigated the impact of soft errors in an automotive vehicle application based on a CNN. Results showed that the occurrence of soft errors affects a vehicle's travel and must be mitigated. In turn, (REAGEN et al., 2018) proposed a framework used to demonstrate that the soft error reliability varies across Deep Neural Networks (DNNs). The findings are built upon several fault injections performed at specific DNN design points, including weights, activation functions, and hidden states. (TRINDADE; COELHO et al., 2019) assessed the soft error reliabil-

Figure 7.12: Reliability mismatch for dual (a) and quad-core (b) Arm Cortex-A72 processor when comparing with Arm Cortex-A9.

(a) Dual-core reliability mismatch results



(b) Quad-core reliability mismatch results



Source: Authors

ity of an FPGA-implemented Support Vector Machine (SVM) under neutron irradiation, and (KHOSHAVI; BROYLES; BI, 2020) examined the effects of soft errors on the layers of a Binarized Neural Network (BNN) running on an FPGA. (Libano et al., 2018) investigated the reliability of Feed-forward Neural Networks (FNN) by performing fault injection campaigns by simulation and heavy-ions irradiation. To the best of our knowledge, the only work that explores the use of mitigation techniques is (Libano et al., 2019), where selective hardening approaches are applied to improve the soft error reliability of FNNs and CNNs.

7.3.1 Simple ML Algorithms

The literature on supervised ML classification algorithms is abundant and features countless algorithm variations, making it challenging to define the most distinguished and suitable ones to run on resource-constrained processors. To cover as many ML algorithms as possible, we chose three families, i.e., algorithms with different behaviors, where each representative of the family is based on its popularity and applicability. Aiming to cover the three mains groups, we have implemented the following three ML algorithms: SVM from the group based on statistical methods; Random Forest (RF) from the group based on decision trees; and Artificial Neural Network (ANN) from the group based on neural networks. Each algorithm is described as follows:

Support Vector Machines were first introduced as binary classifiers (CORTES; VAPNIK, 1995), i.e., a classifier for problems that have two possible classes as output. During the training phase, it generates an optimal linear classifier in the following format:

$$score(\vec{x}) = b + \sum_{n=1}^{N}[y_n\alpha_n(\vec{x_n}\dot{\vec{x}})] \tag{7.1}$$

where $\vec{x}$ is the unknown input sample. Parameters $\vec{x_n}, y_n, \alpha_n, N$ and b are found during training time. To overcome the limitation of accepting only binary datasets, we have used the One-vs-One technique (HSU; LIN, 2002), in which an SVM is trained for each pair of classes. In the end, a majority voting algorithm decides the final class.

Artificial Neural Networks, unlike SVM, inherently support multiclass problems. The Neural Network classifier is a layered structure, where each layer is composed of a set of "neurons". Each of these neurons comprises a set of weights and an activation function. A neuron receives neurons' results in the previous layer as input, weighs them, evaluates the result in its activation function, and sends them to neurons in the next layer. Each neuron in the final layer represents one class of the dataset. Therefore, the final class is inferred from the class represented by the neurons in the last layer with the highest value.

Random Forest was introduced as a technique to counter the tendency of Binary Decision Trees (BDTs) to overfit (BREIMAN, 2001). It is composed of multiple BDT nodes, where each one divides the dataset in two, based on a given logical condition. For example, let us consider a Flower dataset where the class separation condition is the sepal length < 1.5. The goal is to separate samples from different classes, using the training dataset to validate the chosen conditions. In this process, the left child node continues separating samples for those with sepal length < 1.5, while the right child node does for those with sepal length $>= 1.5$. Eventually, if the tree grows deep enough, the leaves will contain only the same class samples. To classify a new sample, the tree must be traversed, evaluating the conditions in the incoming sample until a leaf is reached, which reveals the final class of the sample. Depending on the parameters, a BDT can become very deep, overfitting the data. To overcome this problem, the Random Forest algorithm trains multiple BDTs, each with a subset of the training dataset. As each BDT within the forest will produce a class, potentially different from each other, a majority voting algorithm is used to infer the final class.

7.3.1.1 Experimental Setup

Table 7.5 shows our experimental setup. Fault analyses are obtained by injecting random bit flips in the general-purpose and floating-point register files of an ARM Cortex-M4F. Conducted experiments include more than 150K fault injections in a bare-metal system, considering three ML algorithms running with and without protection (reference results). To ensure the results' statistical significance, this work injects 17k faults per campaign, which generates a margin of error of 1% with a 99% confidence level, according to (LEVEUGLE et al., 2009b).

The cross-compiler chosen to generate the binaries was Clang/LLVM 6.0.1 using the O2 optimization flag, which has already shown reliability results superior to GCC in several applications (GAVA et al., 2019a).

7.3.1.2 Soft Error Analysis

Figure 7.13 shows the fault effects for three ML algorithms (Section 7.3.1) with no protection (none), full protection (TMR), and partial protection (P-TMR). Concerning the reliability improvements brought by the TMR technique, it is possible to see an increase in Vanishes in all ML algorithms. In addition, the number of ONA results remain

Table 7.5: Experimental Setup.

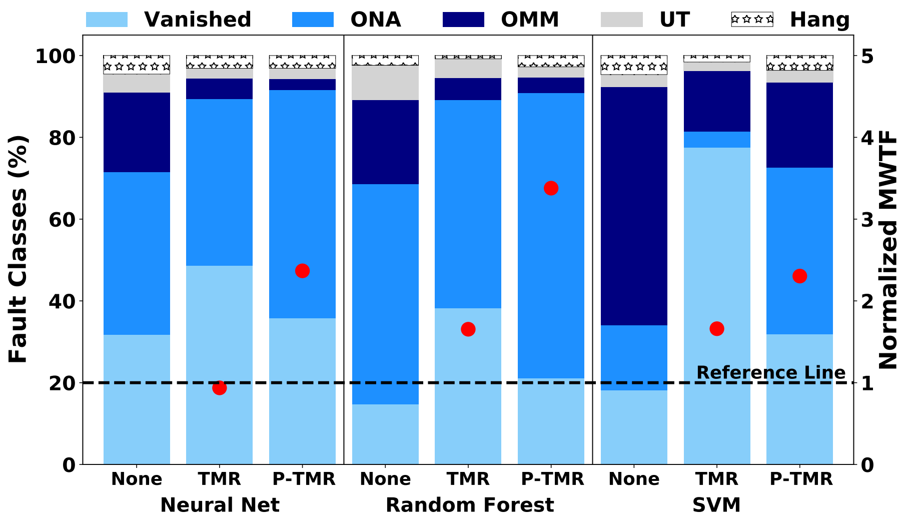
Processor	Arm Cortex-M4F
Software Stack	Bare metal
Compiler	Clang 6.0.1
Optimization Flag	O2
Number of ML algorithms	3
Mitigation Technique	TMR, P-TMR
Fault Injections per Scenario	17000
Number of Scenarios	9
Total Fault Injections	153,000

Source: Authors

quite significant for Neural Net and Random Forest. In general, this effect occurs due to faults that hit registers not used by the program. The Cortex-M4F has sixteen 32-bit general-purpose registers and another thirty-two 32-bit floating point registers. The SVM has more floating-point operations w.r.t. the other two algorithms, which leads to extra utilization of registers resulting in less ONA occurrence.

Moreover, as ONA does not affect the memory, this work assumes that the correct output is the sum of Vanishes + ONAs. In this sense, the TMR technique's increase in reliability regarding the correct outputs is 17.82%, 20.48%, and 47.33% for Neural Net, Random Forest, and SVM, respectively. However, such reliability improvements can be misleading when compared to other hardening techniques. Although Cho's classification

Figure 7.13: Fault effects and normalized MWTF (red dots) for 3 ML algorithms with no protection (none), TMR, and P-TMR.



Source: Authors

offers a useful metric for classifying soft error effects, it does not express the trade-off correctly between reliability increase and performance degradation. In other words, when hardening a program code by inserting additional instructions, it also increases its exposure to faults, thus decreasing its reliability. Therefore, it must consider the decrease in critical faults and the additional execution time inherent to the hardening technique's extra instructions. For this reason, this work uses the Mean Workload to Failure (MWTF) factor (REIS; CHANG et al., 2005) in conjunction with Cho's classification. The MWTF is defined in Eq. 2.1, as the workload that a system can complete before failing. To calculate the MWTF, we measured the applications' runtimes on an STM32F407VGT microcontroller, which uses an Arm Cortex-M4F processor running at 168Mhz. In addition, the SOFIA framework uses the fault injection results to measure the Architectural Vulnerability Factor (AVF). For ML algorithms, the most critical vulnerability is presented by the occurrence of OMM. For example, in safety-critical applications, such as autonomous cars, an OMM error can alter the detection of an obstacle in front of the vehicle, which can lead to an accident. For this reason, this work used the OMM-based AVF (AVF_{OMM}).

The resulting MWTF is used to compare both hardening techniques when applied to the three ML algorithms. For the Neural Net algorithm, the defined critical function was `exp`, a kernel that calculates the exponential of a value used in the activation layer. While the `predict` function was defined as critical for the Random Forest application, and the `classify` function was set as critical for the SVM application.

Figure 7.13 shows that the correct output improvement for the P-TMR was 20.10%, 22.22%, and 38.58% for Neural Net, Random Forest, and SVM, respectively. Regarding Cho's classification, the aforementioned results represent a greater soft error susceptibility compared to the full TMR. However, when considering the normalized MWTF, the P-TMR presents the best ratio between the increased reliability and performance degradation. This behavior occurs due to the significant performance overhead resulting from TMR's redundant code execution (up to 4.5 times - Table 7.6). On average, it has an increase of 2.60$\times$ in the number of executed instructions for P-TMR, which represents 35.78% less overhead than TMR.

It is essential to highlight that, in some cases (i.e., Neural Net and Random Forest), the P-TMR presents better reliability, with significantly less overhead with respect to the full TMR. This is because P-TMR is applied to a specific function/layer, thus reducing the replication cost and the register pressure. As these ML algorithms already use a few registers, the probability of a fault reaching an unused register increases when compared

Table 7.6: Executed instructions and execution time (Cortex-M4F).

Application	Mitigation technique	Executed instructions ($\times 10^3$)	Execution time (ms)
Neural Net	none	357	5.68
	TMR	1,619 ($4.53\times$)	23.11 ($4.06\times$)
	P-TMR	1,260 ($3.53\times$)	17.39 ($3.06\times$)
Random Forest	none	40	1.06
	TMR	153 ($3.83\times$)	2.41 ($2.27\times$)
	P-TMR	102 ($2.55\times$)	1.68 ($1.58\times$)
SVM	none	28	0.76
	TMR	106 ($3.78\times$)	1.79 ($2.75\times$)
	P-TMR	48 ($1.71\times$)	0.92 ($1.21\times$)

Source: Authors

to TMR, which usually has more ONA occurrences. Results show a significant reduction of OMM, which is a critical fault for most ML applications when either TMR technique is applied. When applying the full TMR, it is possible to observe an OMM decrease of 73.74% for the Neural Net, 73.23% for the Random Forest, and 74.72% for the SVM. For P-TMR, the OMM reduction achieves 86.21% for the Neural Net, 81.26% for the Random Forest, and 64.22% for the SVM. Therefore, for critical systems where the performance loss is not restrictive, TMR can still be an acceptable option. However, results indicate that bespoke and partial protection obtained through the use of P-TMR is the best option for resource-constrained systems.

7.3.2 CNN Application

An additional ML application is also used to validate the proposed mitigation flow. This case study includes a CNN application (CMSIS-NN kernel (LAI; SUDA; CHANDRA, 2018)) trained with the CIFAR-10 dataset (KRIZHEVSKY; HINTON et al., 2009), which consists of 60k 32x32 color images divided into ten output classes. The adopted CNN topology consists of convolution layers interspersed with non-linear activation layers, pooling layers, and a fully-connected layer. Unlike classic CNNs, the CMSIS-NN kernel uses low-precision fixed-point representation, making it suitable for execution on resource-constrained processors that support SIMD instructions, such as the adopted Arm Cortex-M4 processor.

7.3.2.1 Experimental Setup

Table 7.7 shows the adopted experimental setup. Each fault injection considers a single input image for one CNN execution, thus not considering the fault propagation to the subsequent executions. Also, this CNN implementation does not use floating-point; that is, fault injections are only performed in general-purpose registers (i.e., R0-R15, including SP, LR, PC). Lastly, we maintain the same number of fault injection campaigns to ensure the results' statistical significance.

Table 7.7: Experimental Setup.

Processor	Arm Cortex-M4
Mitigation Technique (executed instructions)	none (32 millions), TMR (118 millions), P-TMR (84 millions)
Software Stack	CMSIS-NN Kernel
Dataset	CIFAR-10
CNN Topology	3 Convolution, 3 ReLU Activations, 3 Max Pooling, 1 Fully-Connected, 1 Softmax
Injections per Scenario	17,000
Number of Scenarios	36
Total Fault Injections	612,000

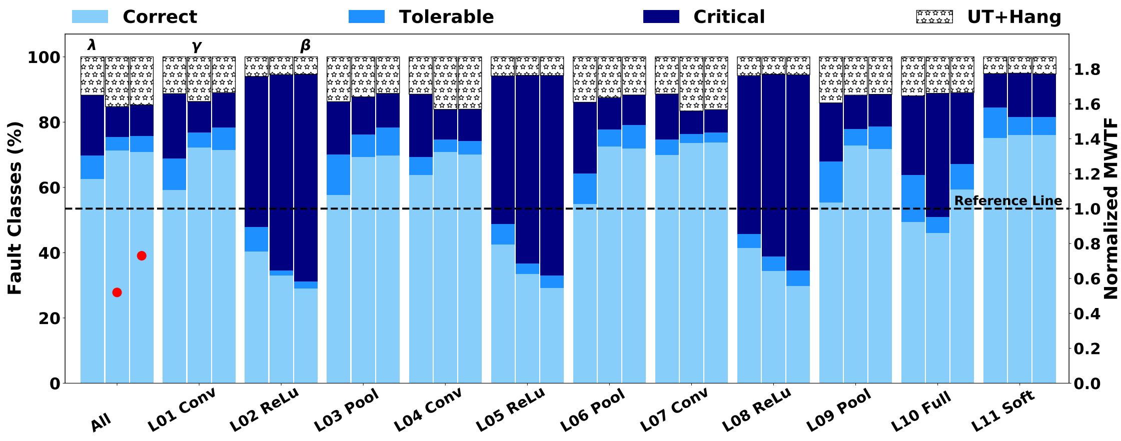
Source: Authors

Complementary to Cho's classification, this work adopts a second classification to evaluate the impact of soft errors on the target CNN's output probabilities. This classification identifies the output results as correct output, effective faults, critical faults, and tolerable faults. *Correct outputs* are the scenarios where the output probabilities are the same as faultless execution. *Effective faults* consider all non-masked faults (i.e., OMM, UT, and Hang). The *Critical faults* consider only the OMM faults that affect the output with incorrect probabilities and no predictions. The remaining OMMs are the *tolerable faults*, which have the same top-ranked classification of the fault-free execution.

7.3.2.2 Reliability Analysis

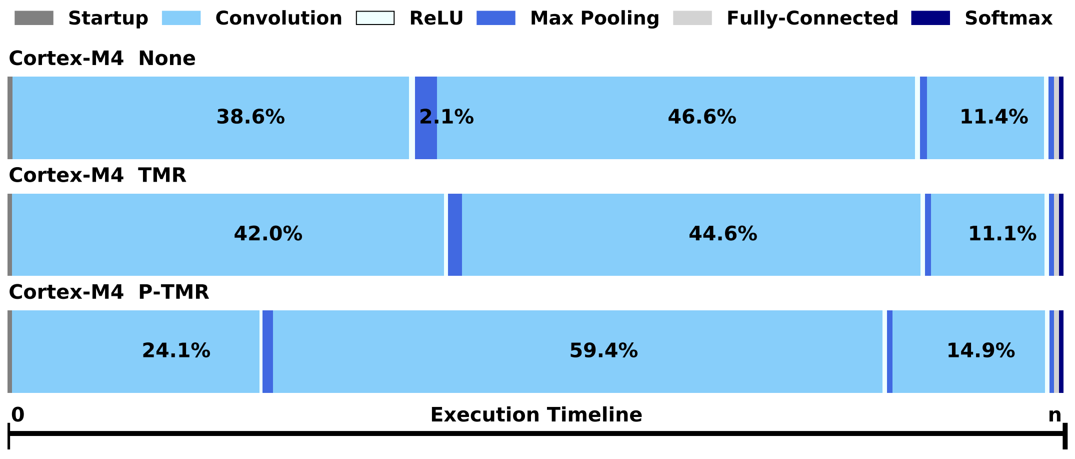
Figure 7.14 shows the correct output classification and the normalized MWTF for the customized fault injection campaigns applied to the CNN application. The x-axis has three bars for each CNN layer, representing the application with no protection (λ), full TMR (γ), and P-TMR (β), respectively. The y-axis shows the percentage of soft errors gathered from the fault injection campaigns. The results in the first three bars (`All` - Figure 7.14) refer to campaigns that consider the injection of a single flipped bit in a single register during the application execution time. Results show a greater amount of critical errors in the activation layers (ReLu), reaching $5\times$ more than the average for all layers. This is due to the fact that the ReLu layers are implemented with a single loop that iterates over all the elements, which access the memory all the time. This behavior increases the possibility of fault to propagate to the memory and generate a critical error. On the other hand, ReLu performs faster than the other layers. When combined, the three ReLu layers correspond to $\sim$0.5% of the overall execution time, as shown in Figure 7.15. Thus, the probability of a fault affecting these three layers is lower w.r.t. other layers.

Figure 7.14: Fault classification and normalized MWTF results comparing no protection (λ), with TMR (γ), and P-TMR (β) mitigation techniques.



Source: Authors

Figure 7.15: CNN execution timeline for None, TMR, and P-TMR.



Source: Authors

Figure 7.14 shows the reliability improvement of the CNN application when both mitigation techniques are used. Both mitigation techniques show similar soft error reliability improvement in terms of correct outputs (i.e., 17% - `All`) with respect to the unprotected application. However, the full TMR presents an overhead of $3.7\times$ in terms of executed instructions, as shown in Table 7.7. This is due to the high pressure on the register bank. The full TMR technique is applied at an intermediate code level, where we have an unlimited number of registers. However, when the register assigner (which has only 13 available registers) is performed, the compiler is forced to spill the variables to memory, which incurs in extra execution time. In addition, the spilling creates new load/store instructions that are unprotected, thus increasing the CNN application's vulnerability. In turn, the overhead presented by the P-TMR is $2.6\times$, which leads to an improvement in the MWTF factor, as shown in Figure 7.14.

However, the achieved reliability improvement is at the cost of an excessive increase in the execution time, thus leading to a worse MWTF with respect to the unhardened version (i.e., 0.52 for TMR and 0.73 for P-TMR, as shown in Figure 7.14). The reason for that is mainly due to the high pressure on the register bank, which increases execution time and introduces points of failure due to memory spilling. One possible way to solve the underlying problem is the use of an assembly-level selective mitigation technique, which can be applied after the register allocation, targeting the code with the lower granularity.

7.3.2.3 Registers Criticality Analysis

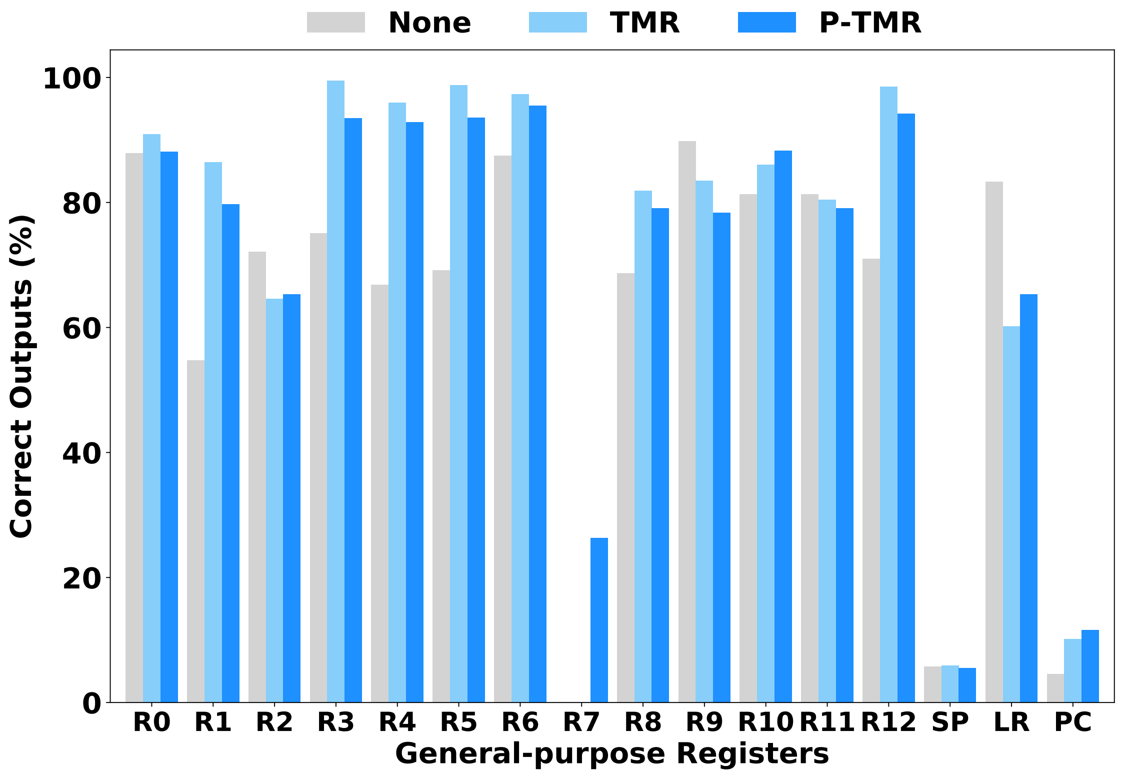
To understand the register bank's influence, we analyze the results of fault injection in each register. Figure 7.16 shows the percentage of correct outputs for each register (i.e., R0-R12, SP, LR, PC). The x-axis presents three bars that inform the result for None (gray), TMR (light blue), and P-TMR (dodger blue), respectively.

Figure 7.16 shows that the full TMR presents a higher number of correct outputs in most cases (i.e., R0, R1, R3-R6, R8, R12, SP). On the other hand, the P-TMR wins in three cases (i.e., R7, R10, PC), and the version without mitigation technique shows better results in four cases (i.e., R2, R9, R11, LR).

Registers without protection and showing reliability improvements have a low utilization and short exposure time. As the adopted mitigation techniques introduce overhead in the application execution time, their reliability does not compensate for the exposure time suffered by these less-used registers, which have little effect on application reliability due to their low usage.

The R7 presents a very particular behavior when comparing the mitigation tech-

Figure 7.16: Comparison of the correct output classifications considering the CNN execution with None, TMR, and P-TMR mitigation technique for each general-purpose register.



Source: Authors

niques. Such register is responsible for storing the base address that is used to access the variables stored in memory due to the spilling. This register is written at the beginning of each function call and only read at the end of the function's execution. Therefore, the interval at which this register is vulnerable is exceptionally high.

The code protected with P-TMR presents less spilling when compared to full TMR. It also creates a time window in which R7 is not vulnerable to faults. This behavior shows that full code replication may not always give the best result in terms of reliability. In this regard, software engineers must have a flexible flow to assess case by case and decide the mitigation technique that would cover the majority of the reliability requirements of each ML application.

7.4 Case Study IV - Considering RAT Custom Parameters

A series of tests was carried out to investigate these parameter customization' impact. The functions list was defined in the same way as the previous experiments, just selecting the application's function that runs most of the time. This parameter has not been explored as much as it would increase the number of simulations exponentially, and the main focus is, in fact, on register sets. As shown in Table 7.8, six sets of registers have been defined. The first two (S1, S2) are limited to the registers in the lower and upper half of the Arm Cortex-A72 processor's general-purpose register bank. Next, the registers were divided into four parts and defined for each of the other sets (S3-S6). The reason for dividing the registers into these sets was to explore how applications behave when the compiler forces a configuration other than the default calling convention. The question is how do these parameters impact the application's reliability?

Table 7.8: Defined register pools for RAT.

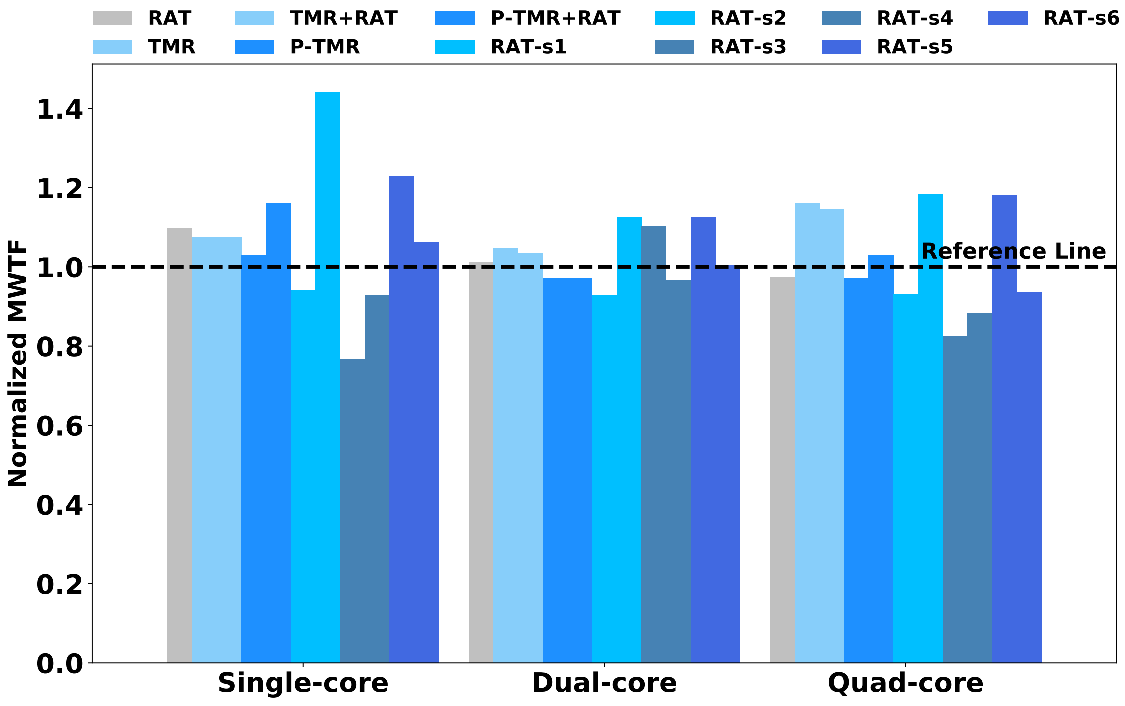
Name	**Register Pool**
`S1`	X0-X14
`S2`	X15-X29
`S3`	X0-X6
`S4`	X7-X13
`S5`	X14-X20
`S6`	X21-X27

Source: Authors

Figure 7.17 shows the average normalized MWTF results for each mitigation technique grouped by CPU core configuration. Each bar is an average of the 13 applications of the Rodinia benchmark. The RAT-s2 mitigation technique is the one that stands out the most on the graph, showing results superior to all other techniques. This behavior reinforces what was seen in Section 7.1.2.2, indicating that the top register (x15-x29) is more susceptible to soft errors in general. It is also possible to notice that the RAT-s3 generates results significantly inferior to the other techniques, indicating that allocating the X0-X6 registers to the most performed function is a bad idea in general. The overall TMR reliability results were lower, but it was expected since all applications have many unhardened external libraries. However, some particular cases differ widely.

Figure 7.18 show *Kmeans* and *Pathfinder* applications' MWTF results in more depth. For the first application, it is possible to observe that single-core and quad-core results are similar, while for dual-core, there are notable differences. This behavior comes from the operating system managing the threads in the two processing cores differently from the other configurations. Another point to be investigated is that the MWTF values for the TMR and TMR+RAT techniques differ significantly from the general average of the applications. For single-core, an improvement can be observed in the normalized MWTF of $1.87\times$ and $1.44\times$ for TMR+RAT and RAT-s2, respectively — and of $1.86\times$ and $1.79\times$ for quad-core. However, the results are quite different from the previous one,

Figure 7.17: Average normalized MWTF results per CPU core configuration for the Rodinia benchmark.

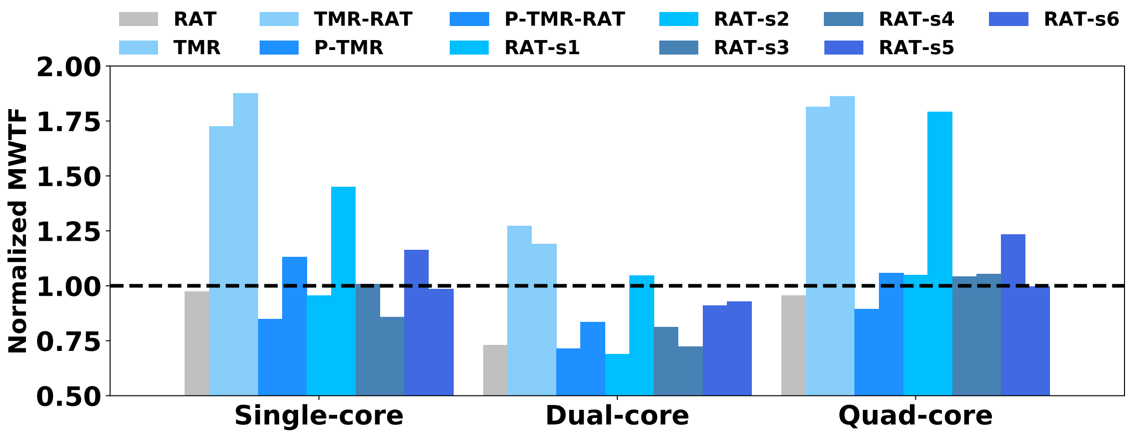


Source: Authors

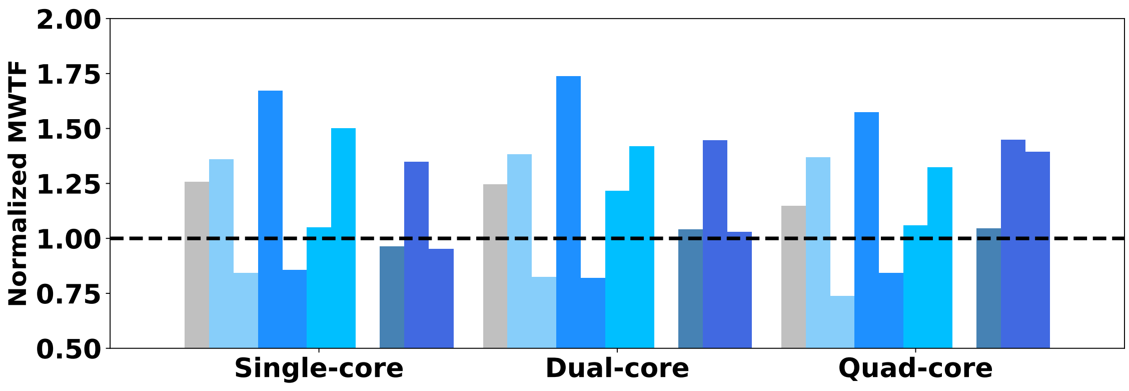
for the Pathfinder application. For this case, the configuration of CPU cores does not significantly affect the system's reliability. The partial mitigation (P-TMR) of the application is more effective than the others techniques, showing an increase of $1.67\times$, $1.73\times$, $1.57\times$ in MWTF normalized to single-core, dual-core, and quad-core, respectively. Another curiosity in this example is that it was not possible to compile the application limited to the initial registers (RAT-s3). This problem may have occurred due to architectural restrictions, which specify that particular instructions can only use specific sets of registers. Therefore, as in the previous analyses, each application needs to be evaluated individually by testing different mitigation options to select the technique that best fits the project's requirements.

Figure 7.18: Sub-figures a and b show the MWTF normalized with the unhardened version for 11 mitigation techniques for two specific Rodinia applications (Kmeans and Pathfinder) grouped by CPU core configuration.

(a) Kmeans



(b) Pathfinder



Source: Authors

8 CONCLUSIONS

Soft error mitigation techniques implemented in software do not impact the manufacturing cost. Nonetheless, there are impacts regarding the execution time, code size, and development effort to port to new architectures and multiple programming languages. This can be time-consuming and not provide a good trade-off on large projects. One solution is to apply selective hardening in the most critical routines of the program. This work proposes a complete framework where software engineers can evaluate and improve the soft error reliability of applications during the early design stages.

Experiments show that bare metal applications without external dependencies present promising soft error reliability results, as we have access to most of the code executed. On the other hand, for the majority of Rodinia's applications, the code protection is not as effective. These applications rely on the OpenMP library, thus increases the number of OS function calls and reduces the scope that our techniques can be used. In other words, the most effective way to protect this kind of application is to improve or use a more reliable library. This trend is reversed as we increase the number of cores, and the proportion of time consumed by functions with redundant code is improved.

This work promotes an early and automated soft error mitigation flow to boost the design space exploration and reduce the development effort, enabling susceptibility reduction evaluation through the full or partial application of TMR. Some analyses are made for a new mitigation technique called RAT throughout this work, considering the Rodinia Suite and NAS Parallel benchmarks. This technique guides the registers allocation based on static and dynamic profile data. The results showed that its use could improve reliability with minimal code size and runtime overhead. Further, a more detailed analysis is made showing the use of each ARMv8-A general-purpose register for a specific application. A soft error evaluation was also conducted to show RAT custom parameters efficiency. The results revealed that limiting the critical function register allocation to X15-X29 registers generates, in general, a hardened application with higher soft error resiliency.

For the three ML algorithms evaluated, experiments show that P-TMR protection's improvement is very similar to TMR and has up to 50% less performance penalty for all scenarios. The CNN application results show that replication techniques might not be suitable for resource-constraints platforms and that new and lightweight techniques must be investigated. Conducted investigations and results also demonstrate the importance of providing engineers with appropriate means to identify not only the occurrence but also the characteristics of ML-based applications that contribute more to the event of soft errors. Authors also believe that the high statistical significance presented gives this work the potential to be a reference for other studies with concerns about the soft error resilience of ML-based application executing on resource-constraint Arm processors.

8.1 Future Works

Several ideas have emerged as possible future works, including some that are currently underway.

- Port the current LLVM mitigation tool from LLVM 6.0 to newer versions (9.0+), including updates to newer processor architectures support (e.g., RISC-V).
- Introduce new well-known data-flow and control-flow mitigation techniques into the application hardening module.
- Create a novel mitigation technique based on the Machine Learning module output information to guide the mitigation techniques and improve the tradeoff between reliability and performance.
- Assess the soft error reliability of other ML-based applications but considering architectures with more resources available.
- Explore the soft error reliability of new architectures and more complex applications.
- Radiation experiments to validate the simulation results.
- Implement a novel software-based mitigation technique in a lower code-level to surpass some limitations of LLVM IR.

REFERENCES

ABDI, A. et al. An optimum instruction level method for soft error detection. *International Review on Computers and Software*, Praise Worthy Prize, FEDERICO II University 21 Claudio Naples I 80125 Italy, v. 7, n. 2, p. 637–641, 2012.

ABDULHAY, E. et al. Fault-tolerant medical imaging system with quintuple modular redundancy (QMR) configurations. *Journal of Ambient Intelligence and Humanized Computing*, Springer, p. 1–13, 2018.

ABICH, G. et al. Soft error reliability assessment of neural networks on resource-constrained iot devices. In: IEEE. *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. [S.l.], 2020. p. 1–4.

ALBANDES, I. et al. Design of approximate-tmr using approximate library and heuristic approaches. *Microelectronics Reliability*, Elsevier, v. 88, p. 898–902, 2018.

ALKHALIFA, Z. et al. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 10, n. 6, p. 627–641, 1999.

AMOH, J.; ODAME, K. M. An optimized recurrent unit for ultra-low-power keyword spotting. *ACM IMWUT*, ACM New York, NY, USA, 2019.

ARM. *ARMv8-A parameters in general-purpose registers*. 2020. Available from Internet: <https://developer.arm.com/docs/den0024/latest/the-abi-for-arm-64-bit-architecture/register-use-in-the-aarch64-procedure-call-standard/parameters-in-general-purpose-registers>.

ARM Ethos-N77 processor - Datasheet. 2020. Available from Internet: <https://developer.arm.com/ip-products/processors/machine-learning/arm-ethos-n/ethos-n77>.

AVIŽIENIS, A.; LAPRIE, J.-C.; RANDELL, B. Dependability and its threats: a taxonomy. In: *Building the Information Society*. [S.l.]: Springer, 2004. p. 91–120.

Azambuja, J. R. et al. Evaluating the efficiency of data-flow software-based techniques to detect sees in microprocessors. In: *2011 12th Latin American Test Workshop (LATW)*. [S.l.: s.n.], 2011. p. 1–6. ISSN 2373-0862.

BANDEIRA, V. et al. Non-intrusive Fault Injection Techniques for Efficient Soft Error Vulnerability Analysis. In: *VLSI-SoC*. [S.l.: s.n.], 2019.

BAUMANN, R. C. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, IEEE, v. 5, n. 3, p. 305–316, 2005.

BENSO, A. et al. AC/C++ source-to-source compiler for dependable applications. In: IEEE. *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. [S.l.], 2000. p. 71–78.

BINKERT, N. et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, ACM New York, NY, USA, v. 39, n. 2, p. 1–7, 2011.

BINKERT, N. et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, ACM New York, NY, USA, v. 39, n. 2, p. 1–7, 2011.

BOHMAN, M. et al. Microcontroller compiler-assisted software fault tolerance. *IEEE Transactions on Nuclear Science*, IEEE, v. 66, n. 1, p. 223–232, 2018.

BREIMAN, L. Random forests. *Machine Learning*, 2001.

BRUGUIER, G.; PALAU, J.-M. Single particle-induced latchup. *IEEE transactions on nuclear science*, IEEE, v. 43, n. 2, p. 522–532, 1996.

CAPOTONDI, A. et al. Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.

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. ISSN null.

CHEN, Y. et al. An instruction set architecture for machine learning. *ACM Transactions on Computer Systems*, v. 36, n. 3, p. 1–35, 8 2019.

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

CHIELLE, E.; KASTENSMIDT, F. L.; CUENCA-ASENSI, S. Overhead reduction in data-flow software-based fault tolerance techniques. In: ____. *FPGAs and Parallel Architectures for Aerospace Applications*. [S.l.]: Springer, 2016. p. 279–291.

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

Chielle, E. et al. Reliability on arm processors against soft errors through sihft techniques. *IEEE Transactions on Nuclear Science*, v. 63, n. 4, p. 2208–2216, 2016.

CHIELLE, E. et al. Reliability on ARM processors against soft errors through SIHFT techniques. *IEEE Transactions on Nuclear Science*, IEEE, v. 63, n. 4, p. 2208–2216, 2016.

CHO, H. et al. Quantitative evaluation of soft error injection techniques for robust system design. In: ACM. *Proceedings of the 50th Annual Design Automation Conference*. [S.l.], 2013. p. 101.

CIANI, L.; CATELANI, M.; VELTRONI, L. Fault tolerant techniques to diagnose and mitigate single event upset (seu) effects on electronic programmable devices. In: CITESEER. *Proc. of 16th ImEKo TC4 symposium*. [S.l.], 2008.

CLANG. *Clang: a C language family frontend for LLVM*. 2019. Available from Internet: <http://clang.llvm.org>.

CORTES, C.; VAPNIK, V. Support-vector networks. *Machine Learning*, 1995.

Da Rosa, F. R. et al. Using machine learning techniques to evaluate multicore soft error reliability. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2019.

DIDEHBAN, M.; LOKAM, S. R. D.; SHRIVASTAVA, A. Incheck: An in-application recovery scheme for soft errors. In: IEEE. *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. [S.l.], 2017. p. 1–6.

DIDEHBAN, M.; SHRIVASTAVA, A. nzdc: A compiler technique for near zero silent data corruption. In: IEEE. *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. [S.l.], 2016. p. 1–6.

DIDEHBAN, M.; SHRIVASTAVA, A.; LOKAM, S. R. D. Nemesis: A software approach for computing in presence of soft errors. In: IEEE. *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. [S.l.], 2017. p. 297–304.

DODD, P. et al. Current and future challenges in radiation effects on cmos electronics. *IEEE Transactions on Nuclear Science*, IEEE, v. 57, n. 4, p. 1747–1763, 2010.

FENG, S. et al. Shoestring: probabilistic soft error reliability on the cheap. *ACM SIGARCH Computer Architecture News*, ACM New York, NY, USA, v. 38, n. 1, p. 385–396, 2010.

FENG, S. et al. Encore: low-cost, fine-grained transient fault recovery. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. [S.l.: s.n.], 2011. p. 398–409.

FERLET-CAVROIS, V.; MASSENGILL, L. W.; GOUKER, P. Single event transients in digital cmos—a review. *IEEE Transactions on Nuclear Science*, IEEE, v. 60, n. 3, p. 1767–1790, 2013.

FETZER, C.; SCHIFFEL, U.; SÜSSKRAUT, M. AN-encoding compiler: Building safety-critical systems with commodity hardware. In: SPRINGER. *International Conference on Computer Safety, Reliability, and Security*. [S.l.], 2009. p. 283–296.

GAVA, J. et al. Evaluation of compilers effects on openmp soft error resiliency. In: IEEE. *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. [S.l.], 2019. p. 259–264.

GAVA, J. et al. Evaluation of Compilers Effects on OpenMP Soft Error Resiliency. In: IEEE. *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. [S.l.], 2019. p. 259–264.

GAVA, J.; REIS, R.; OST, L. Rat: A lightweight system-level soft error mitigation technique. In: IEEE. *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*. [S.l.], 2020. p. 165–170.

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

GOLDSTEIN, S. C.; BUDIU, M. Molecules, gates, circuits, computers. *Molecular Nanoelectronics (Mark A. Reed and Takhee Lee, eds.), Amer-106 NANO, QUANTUM AND MOLECULAR COMPUTING ican Scientific Publishers*, Citeseer, v. 25650, p. 91381–1439, 2003.

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

GOLOUBEVA, O. et al. *Software-implemented hardware fault tolerance*. [S.l.]: Springer Science & Business Media, 2006.

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. 2016.

GUSTAFSSON, J. et al. The Mälardalen WCET benchmarks: Past, present and future. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. [S.l.], 2010.

GUTHAUS, M. R. et al. Mibench: A free, commercially representative embedded benchmark suite. In: IEEE. *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*. [S.l.], 2001. p. 3–14.

HARI, S. K. S. et al. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. v. 47, n. 4, p. 123–134, 2012.

HARI, S. K. S. et al. GangES: Gang error simulation for hardware resiliency evaluation. In: *ISCA*. [S.l.: s.n.], 2014.

HESS, W.; CANFIELD, E.; LINGENFELTER, R. Cosmic-ray neutron demography. *Journal of Geophysical Research*, Wiley Online Library, v. 66, n. 3, p. 665–677, 1961.

HOFFMANN, M. et al. A practitioner's guide to software-based soft-error mitigation using an-codes. In: IEEE. *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. [S.l.], 2014. p. 33–40.

HORST, R. W.; HARRIS, R. L.; JARDINE, R. L. Multiple instruction issue in the nonstop cyclone processor. *ACM SIGARCH Computer Architecture News*, ACM New York, NY, USA, v. 18, n. 2SI, p. 216–226, 1990.

HSU, C.-W.; LIN, C.-J. A comparison of methods for multiclass support vector machines. *IEEE Transactions on Neural Networks*, 2002.

HUSSAIN, S.; SHAFIQUE, M.; HENKEL, J. A fine-grained soft error resilient architecture under power considerations. In: IEEE. *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. [S.l.], 2019. p. 972–975.

IMPERAS. *Open Virtual Platforms (OVP)*. 2019. Available from Internet: <http://www.ovpworld.org/>.

ISO. Norm, *Road vehicles – Functional safety*. [S.l.]: ISO, Geneva, Switzerland, 2011.

JAMES, B. et al. Applying compiler-automated software fault tolerance to multiple processor platforms. *IEEE Transactions on Nuclear Science*, IEEE, v. 67, n. 1, p. 321–327, 2019.

JEDEC, J. Measurement and reporting of alpha particles and terrestrial cosmic ray-induced soft errors in semiconductor devices: Jesd89a. *JEDEC STANDARD, JEDEC Sold State Technology Association, No. 89*, p. 1–85, 2006.

JOHNSTON, A. et al. The effect of temperature on single-particle latchup. *IEEE Transactions on nuclear science*, IEEE, v. 38, n. 6, p. 1435–1441, 1991.

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

KASTENSMIDT, F. L. SEE mitigation strategies for digital circuit design applicable to ASIC and FPGAs. In: *IEEE NSREC Short Course*. [S.l.: s.n.], 2007.

KASTENSMIDT, F. L.; CARRO, L.; REIS, R. A. da L. *Fault-tolerance techniques for SRAM-based FPGAs*. [S.l.]: Springer, 2006.

KHOSHAVI, N.; BROYLES, C.; BI, Y. A survey on impact of transient faults on bnn inference accelerators. *arXiv preprint arXiv:2004.05915*, 2020.

KHOSROWJERDI, H.; MEINKE, K.; RASMUSSON, A. Virtualized-fault injection testing: A machine learning approach. In: IEEE. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.], 2018. p. 297–308.

KRIZHEVSKY, A.; HINTON, G. et al. Learning multiple layers of features from tiny images. Citeseer, 2009.

KUVAISKII, D. et al. Elzar: Triple modular redundancy using intel avx (practical experience report). In: IEEE. *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. [S.l.], 2016. p. 646–653.

LADBURY, R. Radiation hardening at the system level. In: *IEEE NSREC Short Course*. [S.l.: s.n.], 2007. p. 1–94.

LAI, L.; SUDA, N.; CHANDRA, V. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: IEEE COMPUTER SOCIETY. *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. [S.l.], 2004. p. 75.

LERAY, J. Effects of atmospheric neutrons on devices, at sea level and in avionics embedded systems. *Microelectronics Reliability*, Elsevier, v. 47, n. 9-11, p. 1827–1835, 2007.

LEVEUGLE, R. et al. Statistical fault injection: Quantified error and confidence. In: EUROPEAN DESIGN AND AUTOMATION ASSOCIATION. *Proceedings of the Conference on Design, Automation and Test in Europe*. [S.l.], 2009. p. 502–506.

LEVEUGLE, R. et al. Statistical fault injection: Quantified error and confidence. In: *DATE*. [S.l.: s.n.], 2009.

LI, G.; HARI, S. K. S. et al. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In: *ACM/IEEE Supercomputing Conference*. [S.l.: s.n.], 2017.

LI, S. et al. Efficient soft-error detection for low-precision deep learning recommendation models. *arXiv preprint arXiv:2103.00130*, 2021.

Libano, F. et al. On the reliability of linear regression and pattern recognition feedforward artificial neural networks in fpgas. *IEEE Transactions on Nuclear Science*, v. 65, n. 1, p. 288–295, 2018.

Libano, F. et al. Selective hardening for neural networks in fpgas. *IEEE Transactions on Nuclear Science*, v. 66, n. 1, p. 216–222, 2019.

LYONS, R. E.; VANDERKULK, W. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development*, IBM, v. 6, n. 2, p. 200–209, 1962.

MAGNUSSON, P. et al. Simics: A full system simulation platform. *Computer*, Feb./2002.

MARTIN, M. M. K. et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 2005.

MARTINEZ-ALVAREZ, A. et al. Compiler-directed soft error mitigation for embedded systems. *IEEE Transactions on Dependable and Secure Computing*, IEEE, v. 9, n. 2, p. 159–172, 2011.

MAVIS, D. G.; EATON, P. H. Soft error rate mitigation techniques for modern microcircuits. In: IEEE. *2002 IEEE International Reliability Physics Symposium. Proceedings. 40th Annual (Cat. No. 02CH37320)*. [S.l.], 2002. p. 216–225.

MAY, T. C.; WOODS, M. H. A new physical mechanism for soft errors in dynamic memories. In: IEEE. *16th International Reliability Physics Symposium*. [S.l.], 1978. p. 33–40.

MUKHERJEE, S. S.; EMER, J.; REINHARDT, S. K. The soft error problem: An architectural perspective. In: IEEE. *11th International Symposium on High-Performance Computer Architecture*. [S.l.], 2005. p. 243–247.

NASA. *Nas Parallel Benchmarks*. 2019. Available from Internet: <https://www.nas.nasa.gov/publications/npb.html>.

NICOLESCU, B.; VELAZCO, R. Detecting soft errors by a purely software approach: method, tools and experimental results. In: ____. *Embedded Software for SoC*. [S.l.]: Springer, 2003. p. 39–51.

NORMAND, E. Single-event effects in avionics. *IEEE Transactions on nuclear science*, IEEE, v. 43, n. 2, p. 461–474, 1996.

OH, N.; SHIRVANI, P. P.; MCCLUSKEY, E. J. Control-flow checking by software signatures. *IEEE transactions on Reliability*, IEEE, v. 51, n. 1, p. 111–122, 2002.

OH, N.; SHIRVANI, P. P.; MCCLUSKEY, E. J. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, IEEE, v. 51, n. 1, p. 63–75, 2002.

REAGEN, B. et al. Ares: A framework for quantifying the resilience of deep neural networks. In: *DAC*. [S.l.: s.n.], 2018.

REINHARDT, S. K.; MUKHERJEE, S. S. Transient fault detection via simultaneous multithreading. In: IEEE. *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201)*. [S.l.], 2000. p. 25–36.

REIS, G. A.; CHANG, J.; AUGUST, D. I. Automatic instruction-level software-only recovery. *IEEE micro*, IEEE, v. 27, n. 1, p. 36–47, 2007.

REIS, G. A.; CHANG, J. et al. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2005.

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

REIS, G. A. et al. Swift: Software implemented fault tolerance. In: IEEE COMPUTER SOCIETY. *Proceedings of the international symposium on Code generation and optimization*. [S.l.], 2005. p. 243–254.

REUTHER, A. et al. Survey and benchmarking of machine learning accelerators. In: *IEEE High Performance Extreme Computing Conference (HPEC)*. [S.l.: s.n.], 2019. p. 1–9.

RHISHEEKESAN, A.; JEYAPAUL, R.; SHRIVASTAVA, A. Control Flow Checking or Not?(for Soft Errors). *ACM Transactions on Embedded Computing Systems (TECS)*, ACM, v. 18, n. 1, p. 11, 2019.

RODRIGUES, G. S. et al. Analyzing the impact of using pthreads versus OpenMP under fault injection in ARM Cortex-A9 dual-core. In: *RADECS*. [S.l.: s.n.], 2016.

Rosa, F. et al. A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability. In: *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. [S.l.: s.n.], 2015. p. 211–214. ISSN 2377-7966.

ROSA, F. R. da et al. Using machine learning techniques to evaluate multicore soft error reliability. *IEEE Transactions on Circuits and Systems I: Regular Papers*, IEEE, v. 66, n. 6, p. 2151–2164, 2019.

ROSA, F. R. da et al. Using machine learning techniques to evaluate multicore soft error reliability. *IEEE Transactions on Circuits and Systems I: Regular Papers*, IEEE, v. 66, n. 6, p. 2151–2164, 2019.

SANTOS, F. F. dos; PIMENTA, P. F. et al. Analyzing and increasing the reliability of convolutional neural networks on gpus. *IEEE Transactions on Reliability*, 2018.

SCHIRMEIER, H.; BORCHERT, C.; SPINCZYK, O. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In: IEEE. *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. [S.l.], 2015. p. 319–330.

SERRANO-CASES, A. et al. Non-intrusive automatic compiler-guided reliability improvement of embedded applications under proton irradiation. *IEEE Transactions on Nuclear Science*, IEEE, 2019.

SHIRVANI, P. P. et al. Software-Implemented EDAC Protection Against SEUs. *IEEE Transactions on Reliability*, v. 49, p. 273–284, 2000.

TABER, A.; NORMAND, E. Single event upset in avionics. *IEEE Transactions on Nuclear Science*, IEEE, v. 40, n. 2, p. 120–126, 1993.

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

THATI, V. B. et al. Selective duplication and selective comparison for data flow error detection. In: IEEE. *2019 4th International Conference on System Reliability and Safety (ICSRS)*. [S.l.], 2019. p. 10–15.

TITUS, J. et al. Effect of ion energy upon dielectric breakdown of the capacitor response in vertical power mosfets. *IEEE Transactions on Nuclear Science*, IEEE, v. 45, n. 6, p. 2492–2499, 1998.

TORO, D. G. *Temporal Filtering with Soft Error Detection and Correction Technique for Radiation Hardening Based on a C-element and BICS*. Thesis (PhD) — Télécom Bretagne; Université de Bretagne Occidentale, 2014.

TRINDADE, M. G.; COELHO, A. et al. Assessment of a hardware-implemented machine learning technique under neutron irradiation. *IEEE Transactions on Nuclear Science*, 2019.

VELAZCO, R.; FOUILLAT, P.; REIS, R. *Radiation effects on embedded systems*. [S.l.]: Springer Science & Business Media, 2007.

VEMU, R.; ABRAHAM, J. Ceda: Control-flow error detection using assertions. *IEEE Transactions on Computers*, IEEE, v. 60, n. 9, p. 1233–1245, 2011.

WANG, H.-B. et al. Impact of single event upsets on convolutional neural networks in xilinx zynq fpga. *IEEE Transactions on Nuclear Science*, IEEE, 2021.

WROBEL, T. et al. Current induced avalanche in epitaxial structures. *IEEE Transactions on Nuclear Science*, IEEE, v. 32, n. 6, p. 3991–3995, 1985.

ANNEX A — WCET RESULTS TABLE

Table A.1: WCET benchmark results for runtime, code-size, MWTF, and EAFC normalised with the Reference application.

#	Runtime		CodeSize		MWTF		EAFC	
	P-TMR	TMR	P-TMR	TMR	P-TMR	TMR	P-TMR	TMR
1	2.33	2.93	1.02	1.21	5.08	3.63	0.69	0.94
2	2.67	2.78	1.10	1.12	4.66	6.90	1.02	0.89
3	2.28	2.33	1.12	1.21	0.56	0.74	1.90	1.78
4	2.02	2.70	1.03	1.16	1.47	1.95	1.25	1.26
5	2.31	2.31	1.09	1.12	12.97	14.43	0.43	0.43
6	2.50	2.88	1.03	1.30	0.33	0.33	2.86	2.92
7	1.83	2.50	1.08	1.13	5.04	1.97	0.60	1.02
8	3.00	3.85	1.09	1.24	0.60	1.02	1.88	1.87
9	1.67	2.67	1.02	1.40	2.50	2.36	0.92	1.15
10	3.41	3.41	1.29	1.32	7.15	8.55	0.68	0.69
11	2.55	2.65	1.03	1.09	0.18	0.18	4.67	3.42
12	2.75	2.75	1.45	1.46	0.60	0.67	2.13	2.00
13	3.69	3.69	1.08	1.08	5.82	5.82	0.76	0.76
14	1.62	1.62	1.01	1.07	0.31	0.21	2.61	3.12
15	2.22	2.28	1.03	1.10	2.38	1.78	0.92	0.98
16	2.64	2.64	1.09	1.09	5.29	5.12	0.76	0.82
17	3.00	3.00	1.43	1.47	0.57	0.66	2.30	2.13
18	2.24	2.35	1.06	1.11	2.76	3.10	0.90	0.90
19	2.00	2.20	1.03	1.12	3.26	2.64	0.83	1.06
20	4.00	4.00	1.19	1.19	1.95	2.11	1.39	1.30
21	3.42	3.42	1.99	1.99	26.66	29.13	0.63	0.58
22	2.28	2.94	1.09	1.17	19.26	33.71	0.35	0.30
23	1.08	1.13	1.27	1.46	1.33	0.89	0.90	1.13
24	2.16	2.68	1.28	1.37	2.48	2.00	1.08	1.08
25	3.15	3.31	1.09	1.13	5.79	16.43	0.73	0.46
Avg	2.51	2.76	1.16	1.24	4.76	5.97	1.33	1.32

ANNEX B — LIST OF PUBLICATIONS

Appendix B shows the main works published during the journey taken in the scientific environment until reaching this point. The first task involved studying the susceptibility to soft errors of OpenMP applications compiled using different cross-compilers and different code optimization levels. This work was named "Evaluation of Compilers Effects on OpenMP Soft Error Resiliency" and generated a publication at the *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (GAVA et al., 2019b). After completing this project, I started to work with software-based fault mitigation techniques (e.g., full and partial TMR) for multi-core and multi-processed systems. This second project's main focus was on developing and evaluating a new lightweight fault tolerance technique called RAT comparing it to the well-known TMR technique. This work was called "RAT: A Lightweight System-level Soft Error Mitigation Technique" and generated a publication in the *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)* (GAVA; REIS; OST, 2020). Meanwhile, other projects were also carried out in partnership with colleagues. In "Soft Error Reliability Assessment of Neural Networks on Resource-constrained IoT Devices", I assisted in analyzing the results and writing the text. In "Evaluation of the soft error assessment consistency of a JIT - based virtual platform simulator", I was responsible for part of the experiments compiling applications using different compilers and optimization flags, as well as doing most of the fault injection simulations.

Publications:

- **J. Gava**, R. Reis and L. Ost, "RAT: A Lightweight System-level Soft Error Mitigation Technique," 2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC), Salt Lake City, UT, USA, 2020, pp. 165-170, <https://doi.org/10.1109/VLSI-SOC46417.2020.9344080>

- G. Abich, **J. Gava**, R. Reis and L. Ost, "Soft Error Reliability Assessment of Neural Networks on Resource-constrained IoT Devices," 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Glasgow, UK, 2020, pp. 1-4, <https://doi.org/10.1109/ICECS49266.2020.9294951>

- **J. Gava**, V. Bandeira, R. Reis and L. Ost, "Evaluation of Compilers Effects on OpenMP Soft Error Resiliency," 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Miami, FL, USA, 2019, pp. 259-264, <https://doi.org/10.1109/ISVLSI.2019.00055>

- Abich, G., Garibotti, R., Bandeira, V., da Rosa, F., **Gava, J.**, Bortolon, F., Medeiros, G., Moraes, F.G., Reis, R. and Ost, L. (2021), Evaluation of the soft error assessment consistency of a JIT-based virtual platform simulator. IET Comput. Digit. Tech. <https://doi.org/10.1049/cdt2.12017>

- **Gava, J.**, Reis, R., Ost, L. (2021) RAT: A Lightweight Architecture Independent System-level Soft Error Mitigation Technique. In: AICT 621: VLSI-SoC. Springer Nature [Prelo]