POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

MASTER's Degree Thesis

AI-Based Soft Error Detection for Embedded Application

Supervisors

Prof. STEFANO DI CARLO

Prof. ALESSANDRO SAVINO

Doctor. ALESSIO CARPEGNA

Candidate

ENRICO MAGLIANO

OCTOBER 2023

Summary

Soft errors pose significant challenges to the reliability of Safety-Critical Real-Time Embedded Systems (SACRES).
Traditionally, Double Modular Redundancy (DMR) techniques have been employed to mitigate such errors. The increasing complexity of these system, however, with the adoption of multi-core processors, hierarchical memory systems, with multiple levels of cache, and specialized co-processors has made this solution increasingly unaffordable and expensive.
This thesis aims to investigate the utilization of Artificial Intelligence (AI)-based detection of soft errors.

Soft errors can bring different possible outcomes: crash, silent data corruption, benign, hangs or reboots. The hardest to detect is the Silent Data Corruption (SDC), which let the program reaches the end of the computation, but cause a wrong outcome. This behaviour makes the SDC very dangerous, because the system does not crash and continues the operations with the wrong data/information, that can bring to potentially dangerous decisions.

The objective of this thesis is to understand if using machine learning techniques it is possible to detect malfunctions in the architecture of a processor, capable of working with various target software applications. A potential approach is to explore micro-architectural event attributes as fault detection features. Hardware Performance Counters (HPC) are used to monitor micro-architectural events, for example the memory access, cache hit/miss or branch events that occur during the execution of the target application. Many processors already track these attributes using the Performance Monitoring Unit (PMU). However, it remains an open question to what extent these attributes are sufficient to detect faulty executions.

Acknowledgements

This thesis work is also thanks to the numerous people who have been close to me and who directly or indirectly contributed. In particular, I want to thank my supervisors and co-supervisors, Alessio Carpegna, Stefano Di Carlo, and Alessandro Savino, who have always been available to guide me in my choices and help me with the difficulties of my work; without them all this would not have been possible.
I also thank my parents and relatives, who always believed in me and never let me lack their support.
Finally, I want to thank my friends, including my roommates and all the guys at lab6 who welcomed me as one of their own from day one. These few lines are not enough, but I hope they can give you an idea of my gratitude.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

SACRES
Safety-Critical Real-Time Embedded Systems

DMR
Double Modular Redundancy

SDC
Silent Data Corruption

ONA
Output Not Affected

PMU
Performance Monitoring Unit

HPC
Hardware Performance Counter

ISA
Instruction Set Architecture

PC
Program Counter

OS
Operating System

ECC

Error Correction Code

SBU

Single-Bit Upset

FC-FFNN

Fully Connected Feed Forward Neural Network

HTDR

Hard To Detect Region

HWIFI

Hardware-Implemented Fault Injection

SWIFI

Software-Implemented Fault Injection

FIT

Failure In Time

LOC

Line Of Code

SOC

System on Chip

HAL

Hardware Abstraction Level

IDE

Integrated Development Environment

ELF

Executable and Linkable Format

XSCT

Xilinx Software Command-Line Tool

PCA

Principal Component Analysis

SVM

Support Vector Machine

K-NN

K-Nearest Neighbors

Chapter 1

Introduction

1.1 Problem Presentation

In the Safety-Critical Real-Time Embedded Systems (SACRES) computation can occur errors during the software execution due to many different phenomena. For example instantaneous voltage spikes, electromagnetic interference, neutron strikes and out-of-range temperatures.
Those phenomena can cause a switch state in the transistor, from open to closed or vice versa. Consequently, this transistor switch can result in a bit-flipping causing a soft-errors, a transient corruption of data stored in memory (RAM, ROM, cache or CPU register) of the embedded system.
Soft errors pose a significant challenge to reliability, trustworthiness and security, this lacks brings potentially the system into a dangerous state, that in some areas can have unsafe consequences in the real world and harm people, automotive and aerospace are two examples of these sensitive areas.

Soft-errors can lead to the occurrence of faults in application execution, in particular the fault effect has different possible outcomes, that can be classified into these categories:

1. *Crash:* the program completely stops working and exits.
2. *Silent Data Corruption (SDC):* the program reaches the end of the computation, but its outcome is wrong.
3. *Benign:* even if there was a fault, the program's outcome is correct.
4. *Hangs:* the program is stuck within a loop.
5. *Reboots:* the operating system reboots.

Crash, Hangs and the Reboots are trivial to detect, basically because in the hangs the computation doesn't reach the end, while for the crash and the reboots the computation doesn't produce a valid outcome.
The real cumbersome to detect is the Silent Data Corruption because the computation ends with a valid outcome, but the system has no idea that it is wrong, and this wrong output can be used by the system to take potentially harmful deductions.

To mitigate the in-memory bit-flipping caused faults, the ECC memory is largely adopted, where Error Correction Code is used to detect where data corruption occurs caused by a bit-flipping in this kind of memory.
While for CPU based bit-flipping caused faults, traditionally, Double Modular Redundancy (DMR) techniques have been employed to mitigate such faults. This solution consists of Double the errors-sensitive hardware, for example, processor cores, co-processors or caches.
Thanks to the redundancy introduced by doubling the computation in two independent modules, in the end, by comparing the two outputs, we can detect if the outcome is wrong or not:

- if the two module's outputs are the same, we can presume that the two output are both correct, which means that also the outcomes are correct.
- if the two module's outputs are different, one of the two modules occurs in a faults, without knowing which is the correct one, we need to reschedule the task.

This solution works under the promise that the two modules don't occur in the same error. However, the most significant con is regarding the cost. The increasing the complexity of these systems, however, with the adoption of multi-core processors, hierarchical memory systems, with multiple levels of cache, and specialized co-processors have made this solution increasingly unaffordable and expensive.

1.2 Basic Machine Learning based Solution

The basic idea is to use Artificial Intelligence (AI) to detect soft errors, in practice implement a machine learning model able to classify the outcome of the computation.
To reach the final goal of detecting malfunctions in the architecture of a processor, over different target software applications. This aims to detect malfunctions without using expensive redundancy solutions.

On the other hand machine learning model prediction is affected by uncertainty, according to the model accuracy, which will never reach 100%.

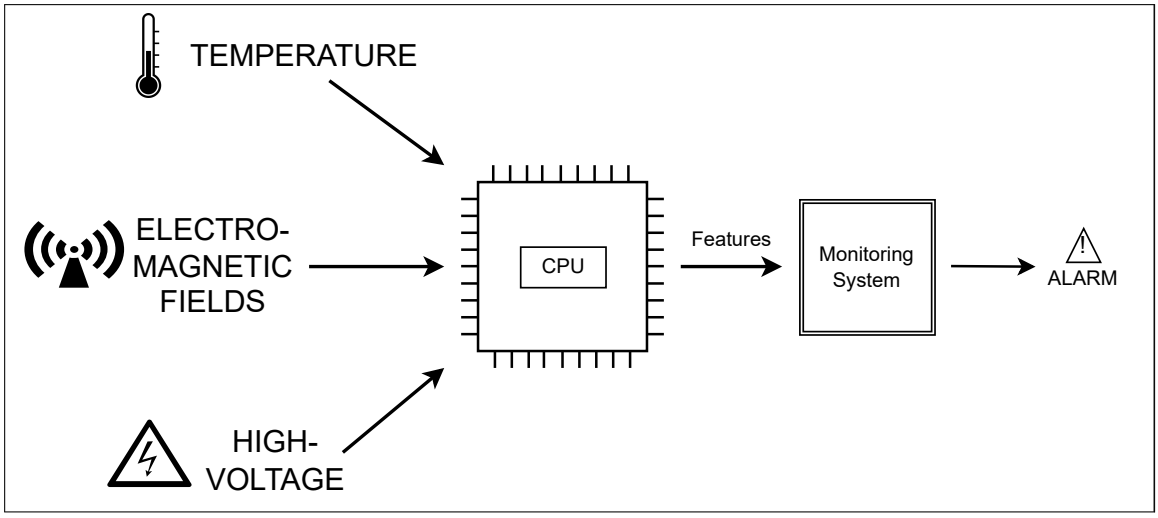


Figure 1.1: This figure represented the basic idea of a Monitoring System (can be ML-based) to detect faults

For training the model we need a dataset, collected with a lot of golden or faulty labelled executions, the golden ones are the ones that reaches the end of the computation without any soft errors.

Two open questions remain; how induce or simulate the faults and how to convert task execution in a features representation suitable for the machine learning model.

1.3 Our Solution

The solution discussed in this Thesis is composed of three part:

1. Translation of the task execution in a form suitable for the Machine learning model.
2. Faults injection.
3. Data Collection in a Dataset used to train a ML model.

First for representing the executions in a form suitable for the Machine Learning model, we decide to use the Architectural events, already tracked in most of the CPU in a specific unit called Performance Monitoring Unit (PMU), which use specific counters called Hardware Performance Counter (HPC). These micro-architectural events, for example, the memory access, cache hit/miss or branch

events that occur during the execution of the target application.
However, it remains an open question to what extent these attributes are sufficient to detect faulty executions.

Second the solution discussed in this thesis used a fault injector simulator, able to inject transient faults in a random part of the processor. This choice is due to the inability to create real faults within the target architecture.

Our fault injector simulator is a basic script running in a host machine that using serial interface to submit command to the target's debugger, setting a breakpoint in a random position, running the task and when the execution reaches the breakpoint, select a random bit in a random register and change his value (bit flipping) and resuming the computation.

Finally to create the dataset the simulator (executed in the host) execute more time the task (executed on the target platform), performing the fault injection during the computation and at the end of every computation access to the Performance Monitoring Unit to collect all events counted in the Hardware Performance Counter. In this scenario, a series of numbers (event counts) are used to represent the execution, that can be labelled based on the outcomes, if is golden (without a fault), or if instead a fault has been injected into it, depending on the type of the fault.

Algorithm 1 Pseudo Code of the Fault injector Simulator.

```
1: procedure FAULT_INJECTION(number_of_execution)
2:     ▷ number_of_execution is the number of time the target task is executed
3:
4:     i = 0
5:     hpcSet = newSet()
6:     while i < number_of_execution do
7:         i = i + 1
8:
9:         setRandomBreakPoint()
10:        runTargetTask()
11:        waitUntilTargetReachBreakPoint()
12:        performingBitFlipping()
13:        resumeTargetTask()
14:        waitUntilTargetReachEnd()
15:        hpc =collectHPCfromTarget()
16:        hpcSet.add(hpc)
17:
18:    end while
19:    return hpc_set
20: end procedure
```

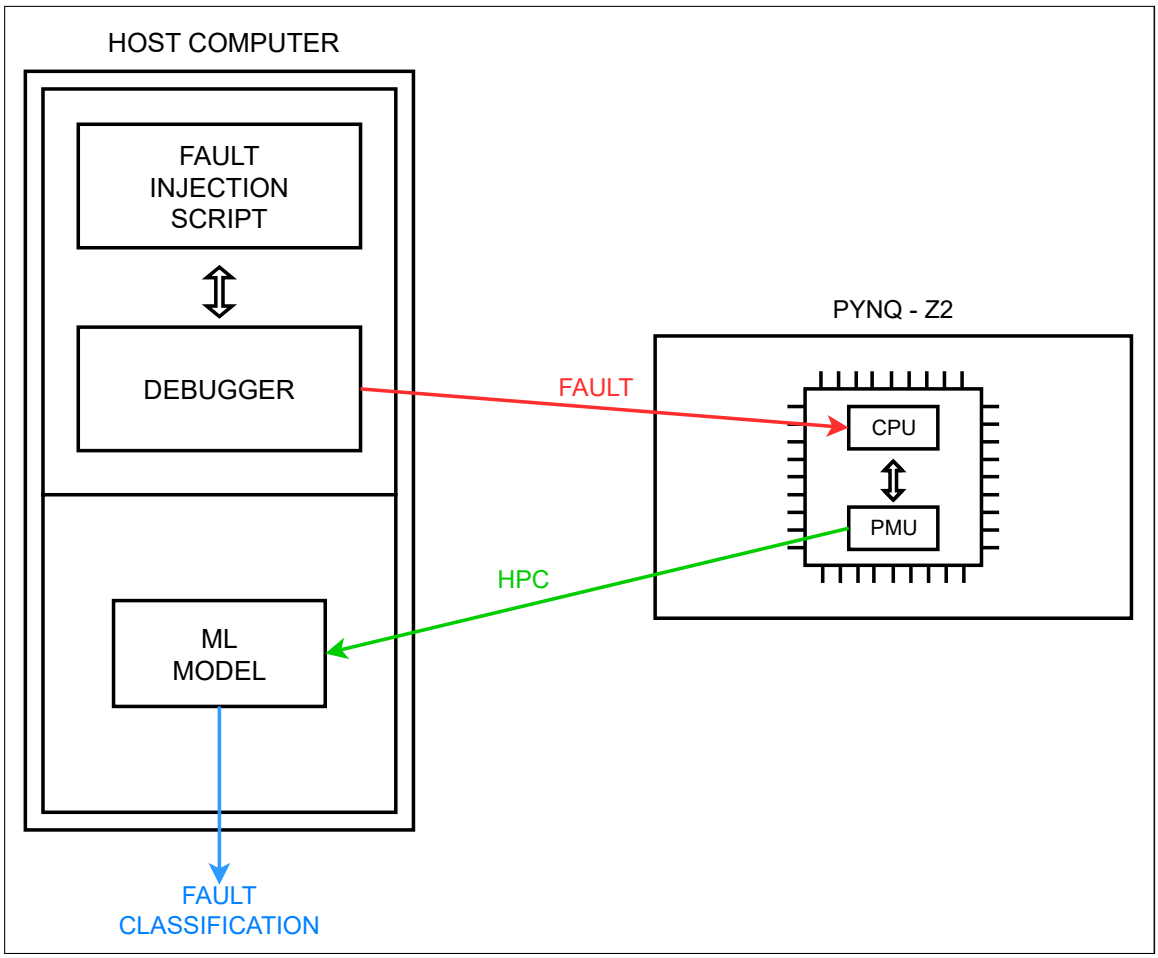


Figure 1.2: This figure represented our solution where the faults are generated by a simulator that uses the debugger in the host computer, and the HPC are used as the features for a machine learning model that performs the fault classification.

Chapter 2

State Of the Art

In the last decades, many studies have been conducted on this topic, and many mitigation to reduce the radiation effects. Within embedded systems, soft errors potentially cause harmful malfunctions.
The design of the digital systems requires the system to expose a very high degree of reliability and to provide fault detection mechanisms. Some works have tried to overcome the Double Module Redundancy by applying machine learning techniques, both in real or simulated environments.

This chapter is split into two subsections; the first aims to explore the state of the art of the Fault Injection, while the second aims to explore one of the Machine Learning techniques to detect soft errors.

2.1 Fault Injection

Literature distinguishes two main approaches to perform fault injection:

- **HWIFI:** fault is injected by means of specific hardware and physical processes like a laser beams or electromagnetic fields.
- **SWIFI:** Software that emulates faults by means of various debugging or virtual environments.

2.1.1 HWIFI

A widely adopted method in this field of study is to irradiate hardware to produce a bit flipping. Karlsson J Liden P. Dahlgren P. Johansson R. and Gunneflo U. in this research [1] inject faults into embedded systems by using heavy-ion radiation. This approach performs hardware bit flipping, as aspected, but on the other hand

This method generates uncontrollable results, consumes a lot of time, another negative aspect is that it may corrupt the entire system.

Due to the fact that beam experiments generate no observable faults, except when they manifest at the application output or when they compromise the system responsiveness. Without no information about the spatial and temporal location of the fault and no information about its propagation pattern.
For all these reasons in this work [2], the researcher decided to combine beam experiments and microarchitectural fault injection over 13 benchmarks executed on the top of Linux on an ARM Cortex-A9 system. Then, comparing the error rate predicted with beam experiments and fault injection, we evaluate at which level fault injection can be used to emulate the beam experiment.

The crucial points of this paper are:

1. Experimental evaluation of the reliability of ARM devices based on beam experiments.
2. A fault injections analysis of the vulnerability of codes.
3. Comparison of the error rate predicted through beam experiment and fault injection.

To avoid the bias of a single application, they use different benchmarks with different computational characteristics. The same input values are used for the benchmark for both bean experiments and fault injection.

The neutron beam experiments setup is created for irradiating the chip uniformly without affecting the onboard DDR. Then the output is compared with the one from the goldel execution, if the two are not equal the output is marked as an SDC. Furthermore the target board send periodically same alive message, if the host PC does not receive for a long period these messages, it tries to reconnect to the boaed, if this is not posssible the execution is marked as a System Crash, instead the execution is marked as an Application Crash.
Due to the fact that the neutron flux is low, it is highly unlikely to see more than a single corruption during program execution.

2.1.2 SWIFI

Some works aim to induce Radiation soft errors using a Software-Implemented Fault Injection (SWIFI).
Due to the fact that some errors can hardly be detected and monitored, the general solution is to generate faults manually.

Unfortunately, generating real fault is usually expensive and unworkable. For this reason, many works used simulated hardware faults through Software-Implemented Fault Injection (SWIFI), which provides machine code-level fault injection into the architectural state.

Due to all these downsides, recent studies to modify the hardware state of the system use software-based fault injection tools, which allow the simulation of amounts of hardware faults.
Soft Error Fault Injection frameworks are used to analyze the target software sensibility to soft errors by injecting, in a Virtual Machine, soft error alterations in the logic operations using the virtual machine and processor emulator (QEMU) [3]

One more sophisticated example is the Level Soft Error Fault Injection Tool (B-SEFI) [4] realized by Ying Wang, Jian Dong, Sen Zhang, and Decheng Zuo to face the challenge of making hardware faults simulation more precise and distributed. This work uses binary-level fault injection technology that has proved to be accurate and efficient in performing injection close to the source code, this approach is going to modify the binary code.
Practically, they designed and developed a Pin-based fault injector [5] capable of injecting soft errors in memory or registers and analyzing the consequence.

The Fault Model performs a single bit flipping for each application run, adopting a random, uniform distribution for selecting the instructions and another for choosing the bit to flip.
Pin is an analysis framework designed by Intel that performs runtime binary instrumentation and is used for implementing fault injection in multiple architectures. Abstracting the details of the architecture allows a high portability.

B-SEFI is a fault injection tool based on binary instrumentation that injects faults at the machine code level, using PIN to target the executable binary program without using the source code, which is biased by the programming language.
B-SEFI is made of three modules: controller, scanner, and fault injection module.

In conclusion, this solution proposes a fault injection tool to simulate soft errors at machine code level. This solution is evaluated using five classical machine learning applications, thousands of fault injections are performed to measure the impact of the soft errors. Experimental results show that different applications are sensitive to soft errors on performance and accuracy but with different impacts. Compared to the classical SWIFI, like QEMU [3], this technique has the pro of presenting a high granularity in the code fault injection.

Some other works, for exemple [6], use a fault injection simulation using FIMSIM [7], a framework based on gem5 [8]
The setup, figure (2.1), includes gem5 simulator used to emulate the x86 Instruction Set Architecture (ISA) hardware due to the fact that virtualization simplifies the faults injection, while the software stack, including a Linux kernel and the target tasks, six Mibench applications (qsort, dijkstra, susan, sha, bitcount and basicmath).
Thanks to the simulation in this study, they performed a Bit flipping in the Integer Register File (intRF) at any instant in time. This solution allows a high control-

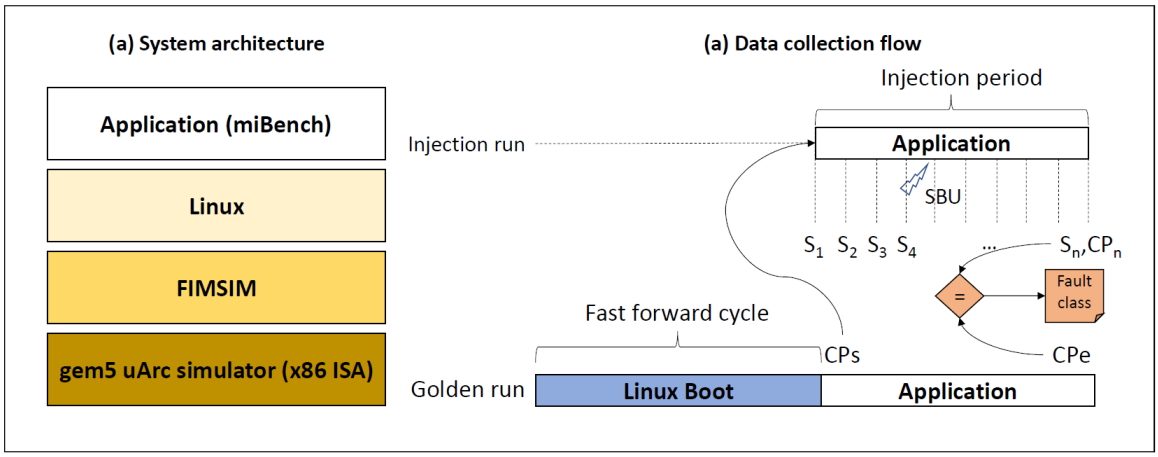


Figure 2.1: Experimental setup based gem5 and FIMSIM.

lability of the fault injection process, without any change in the source code and allows the automation possibility through scripting and a high observability of the fault through the performance counter in PMU.

In the literature, there are only a few reports about debugger-based fault injection, especially applied to embedded systems.
One of these is the work done by Michał MOSDORF and Janusz SOSNOWSKI [9]. They use an ARM7 microcontroller, on it the faults are injected by the JTAG interface available in the ARM platform, using the GNU Debugger (GDB).

The Fault Injector architecture, in figure 2.2, is composed of a JTAG interface controlled by the GDB remote server that provides a GDB-compliant serial interface used by the GNU Debugger. The remote debugging happens through a TCP connection.

GDB is highly adopted across a large variety of architectures, provides easy access to processor memory and registers by the standard interface.
Thanks to an application called Control Software, implemented in .Net, we can control the process of triggering a fault injection and performing if using a set of the GDB commands and then checking the fault injection effects. GDB needs

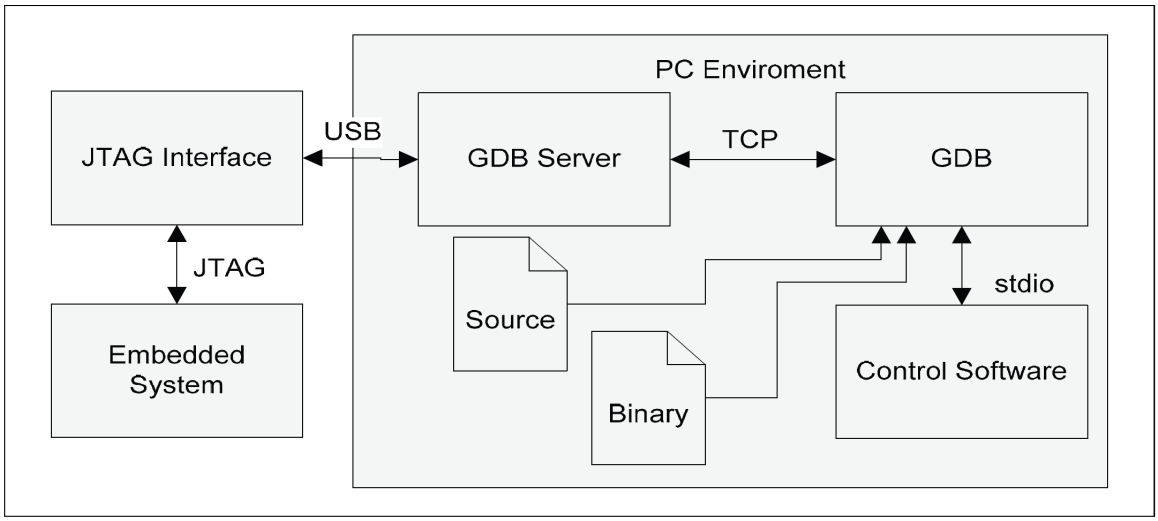


Figure 2.2: Architecture of GDB-Based fault injectorl.

access to a binary file and source codes of the target program to retrieve the debug symbol information. To trigger the Fault injection, they propose different schemes that can be used:

- Time trigger: a specific moment is selected by the control software, and break instruction is executed by the GDB.
- Breakpoint: A specific source code location is chosen by the control software to perform bit-flipping.
- Whatchpoint: injection moment is defined by programmed watch point for a chosen processor memory address.

The presented fault injector does not need software instrumentation and an additional result observation channel.
Thanks to the debugger, this approach allows a full controllability and repeatability of the faults. The only one cons is that it requires the suspension of the target program to inject faults or to observe their effects, and this introduces a temporal overhead, the average overhead, measured by the control software, is 42 ms or, in the case of handling the observational breakpoint and 52 ms in case of fault injection that additionally requires writing a variable.

In the performed experiments, it figured out that the most sensitive memory areas, having injected faults with equal distribution within the whole RAM address space are memory cells within addresses 0-10k (static global and dynamically allocated variables) and 63k-64k (the program stack area holding function calls data and local variables).

Simulating faults directly in processor registers, in particular it gives the possibility to disturbing the program control flow by injecting faults on the program counter register (PC), the result obtained disturbing the PC with a random distribution in time:

- Data abort – 45%
- Prefetch abort - 20%
- Undefined instruction - 15%
- Incorrect result - 10%
- Correct result - 10%

In conclusion, this paper proves that GDB, thanks to the JTAG interface, provides the possibility of creating a fault injector, given the possibility of performing experiments in a real system with easy access to processor registers and system memory. In general, this paper confirms the high controllability and observability of the injection processes at the the expense of some overhead (typically 40-50 ms per simulated fault).

The growing susceptibility of multicore systems to soft errors pose the challenge of asses the soft error resilience view in this paper [10].

In this work, they employ two fault injection frameworks over two virtual platforms: OVPsim-FM [11] and gem5-FIM [12]. This second is event-driven and performs better, up to 2-3 MIPS instead of 1 MIPS. Both frameworks give the possibility to inject faults in different ISAs (multicore, ARM ...).

The workflow is composed of four phases:

1. Golden execution on target architecture and extract the system behavior
2. Create a fault list
3. compare the faulty execution behavior with the golden one

4. create a dataset with all individual report

The fault injection framework can perform a single bit flip in a single register or memory address. In this study only the storage elements are injected with the faults following a random uniform distribution during the target application lifespan (OS startup is not a faults target).
Even if the fault injection process does not change the target application source, it introduces an overhead in terms of simulation time and ML analysis time.

Returning now on this work [2], that compare a HWIFI approach with a SWIFI approach. The SWIFI Fault injection microarchitectural model is based on Gem5 simulator [8], while genFin fault injection framework [13] was used on top of Gem5, GemFin was configured to inject single fault during each simulation on the CPU memory components (L2 Cache, L1 Data and Instruction Caches, Physical Register file, Data and Instruction Translation Lookaside Buffers (TLB)).
Microarchitecture-level fault injection offers a significant amount of observability, Also, in this case, the soft errors have been classified in those categories: SDC, System Crash, Application Crash.

To avoid any bias in the results, exactly the same source code, compiler (and compiler options), and input data for both fault injection and beam experiments. They used seven counters: CPU cycles, branch misses, L1 data cache accesses, L1 data cache misses, L1 data TLB misses, L1 instruction cache misses, and L1 Instruction TLB misses to show if the same differences are in the two setups. This reports acceptable deviations between the two setups, with one main difference in the implementation of TLB of Gem5 and ARM Cortex microarchitectures.

In Conclusion, this paper presents a first detailed analysis that reports a comparison of these two reliability assessment methods.
The failure in time (FIT) rate differences between the two experiments are extremely small (not exceed one order of magnitude in all types of errors).

2.2 Machine Learning Model

The artificial resilience concept implies training system to detect and possibly recover the faults, in particular for Safety-Critical Real-Time Embedded System (SACRES).
Using microarchitectural events as features (executed instructions, cache misses, or incorrectly anticipated branching) traced through a Performance Monitoring Unit (PMU).

In this paper [6], the Authors propose a preliminary study to understand if microarchitectural features can be exploited to train an AI-based hardware soft error detector.
Particular emphasis is on understanding whether event timing could bring additional information to the model.

This preliminary study focused on Single-Bit Upsets in the Integer Register File (intRF), faults are injected at random locations and time intervals during the execution of the target task.
To speed up the experiments, they made every fault injection run start from a checkpoint collected at the end of the fast-forward cycle corresponding to the end of the Linux boot process (CPs) The fault effect was classified by comparing the last checkpoint of the fault injection run (CPn) with the golden execution (CPe).

The HPC collected (from gem5 stats) during fault injection amounts to about 600 features used to create the dataset. Data are preprocessed and normalized. SDC and Benign classes are not balanced in a fault injection experiment. The dataset was balanced using downsampling on the Benign (major) class to enable a fair analysis in the training phase, avoiding bias. During the operative phase this unbalancing must be considerated.

All experiments used a 19-32-2 network architecture composed of three layers. These numbers are found using trial and error. that consists in determine the best model by recognizing and removing errors or failures through various model setup. The first model considered is a Fully Connected Feed feed-forward neural Network (FC-FFNN). The accuracy does not improve it, using a small subset of features is 80%, and adding more features. This means that the discriminant information is brought mainly by a few correlated features.

A second temporal model, based on Long-Short Term Memory (LSTM), was trained on the time-expanded dataset. The performance of the LSTM model confirms no gain on the different metrics.
Since fault detection latency is crucial in SACRES, FC-FFNN model is trained on a cumulative dataset (cumulative data available at different checkpoints) without waiting for the end of the program execution. This could minimize the error detection latency.

In conclusion, even if the results show, in terms of accuracy (the percentage of fault correctly classified). that micro-architectural events are able to detect faulty executions. The presence of Hard To Detect Regions (HTDR) suggests that pure microarchitectural attributes are insufficient, especially for simple tasks where corruption is limited to the data domain.

Returning to this [10] paper that poses the challenge of asses the soft error resilience using Machine Learning techniques. The soft error assessment tool execution flow is organized in three phases:

1. Feature acquisition and data homogenization, basically the tool extracts the information in two datasets, one for the fault injection information and one for the microarchitectural information.

2. Feature transformation and selection, for filtering the features, rearranges the collected data to improve accuracy. Normalize all parameters by the application length (number of instructions) to compare different applications. The data is resized to range between 1 and 10 to enable the comparison among distinct features.

3. Multidimensional Feature Transformation and Selection, soft error classification is a five-dimensional problem (5 error class), prunes the features and measures the feature impact (score) from 0 to 1 to remove the magnitude. The tool ranks the 50 most relevant features using the soft error score, revealing multidimensional correlations.

This paper considers the NAS Parallel Benchmark suite [14] with 29 applications. Some are CPU intensive, some others are memory-bounded, instead for the architectural configurations, it considers single, dual, quad, and octa-core ARM Cortex-A9 and AMD Cortex-A72.

First, they exploited the access to the raw information to perform the profiling of both soft error results and the microarchitectural parameters. This profiling reveals patterns that can be used to group applications according to their behavior. From the initial exploration, two main possible pattern hypotheses:

1. hypothesis: the increase of the feature branches leads to more occurrence of Hang and Vanish.

2. hypothesis: Memory-bound applications present more ONA (Output not affected).

While the first hypothesis is confirmed by the individual feature analysis, the second one is contradicted by the result that indicates that the occurrence of faults is due to the application itself. The applications of Group A, the one with high OMM incidence and low number of Vanishes, are deeply studied because they are appropriate targets to improve system reliability.
The assessment flow based on ML techniques shows that the cache memory activity

impacts the Group A reliability. A higher hit frequency on the cache memory results in a greater number of Hang and Vanish while it decreases the SDC incidence.

After analyzing the impact of one single feature, this work also investigates how the combination of more features affects the system's dependability. To do this, they automatically, using tools, add and multiply data-frame columns in all possible combinations for searching for more complex correlations along both microarchitectural parameters and fault injection information.
For example, they observe that invalid references in cache memory cause CPU stalls and consequently increase the number of context switches. If the bit-flip is injected into a register that will be used later, the memory is considered dirty. Even if the soft error is masked, the memory will remain dirty, causing an increase in SDC. The increase in valid references of each CPU's cache memory positively influences the reliability of Group A.

In conclusion, this paper describes a soft error flow that enables software to identify the occurrence of soft errors in complex software stacks and also determine the correlation between multicore architectural features and detected soft errors using supervised and unsupervised machine learning techniques.
The effectiveness of this soft error assessment flow was evaluated through an extensive data set gathered from more than 1.2 million fault injections.

Chapter 3

Fault Injector

The Fault Injector is the module in charge of performing a bit-flipping. The bit-flipping, performed in the memories on the board (Cache, RAM, CPU Register), simulates the beam radiations that cause the soft error.
Before presenting the technical details and problems, some questions need an answer.

First of all, when injecting a fault, in this case, the answer is trivial, during the execution of the benchmark at any instant. If the benchmark runs over an operating System, the OS must be kept out of the Fault injection. Injection fault on the OS will cause a high percentage of crash, hangs and reboots that we want keep out from the classification.

The second question is where, in this other case, the answer is more cumbersome. In an embedded system, there are many memories on which it is possible to perform a bit-flipping, the RAM, the ROM, the Cache, and the CPU Register.
In a safety-critical system, Error Correction Code memory (ECC memory) can be used to mitigate the system's trustworthiness. For this reason, studying the effects of soft errors in this kind of memory is less attractive. In this thesis, we decide to inject faults only in the CPU's registers, where ECC mechanisms are not used and soft error effects are potentially more dangerous.

How many, or is it better to say with which frequency do we have to inject a fault in our system, at least one, exactly one, or at most one for embedded application runs?
Before answering this question, an important consideration has to be made. Not all the Fault Injected brings to a Soft Error, as mentioned in the introduction of this thesis Chapter 1, the possible outcomes are more (crash, Silent data corruption benign hangs and reboots), and not all cause a soft error, like benign.
Increasing the number of faults per run also increases the percentage of soft errors

in the dataset.
We decide for exactly one, which means one fault for every run. This allows us to build a dataset that contains a good percentage of soft errors without injecting too many faults that would otherwise simulate an unrealistic behavior because, in the real system, it is improbable that more than two faults will occur in a short period of time.

Another important aspect is the choice of the benchmarks. To cover various scenarios, we decided to use more than one, with different characteristics, using applications suitable for an embedded system, both memory intensive and CPU intensive.

Considering now the main challenge, in practice, how to create a fault injector able to work in a real embedded system? As mentioned in chapter 2, there are many alternatives to create a Fault Injector, physical or software-based:

- Using a hardware-implemented fault injector.
- Using a simulator like Gem5 [8].
- Using compile-time injection.
- Using a time-based triggers a timer that generates an interrupt that performs bit-flipping.
- Using a debugger that stops the execution and accessing the register performs the bit-flipping.

In literature, all approaches are documented and reported as valid methodologies to perform a fault injection.
Using a hardware-implemented method is not worth for our goal due to the infeasibility and the uncontrollability of the injection. In practice, it is not easy to inject a single bit flipping in a certain instant.

Using a simulator allows us to inject a single fault and fully control the consequences, but on the other hand, the simulated architecture could present some different respect to the real ones due to the fact that we want to implement our model in a real embedded system, for maintaining the conformity with the final target we choose to collect the dataset over a real system, avoiding using a simulator. Furthermore, this thesis work aims to extend the preliminary study about micro-architectural features as soft-error markers in embedded safety-critical systems [6], that is made over the gem5 simulator [8], proving its applicability in real embedded safety-critical systems.

A compile-time injection is an injection technique that allows the injection of simulated faults into a system, modifying the source code.
Given the fact that we need to collect a whole dataset of executions with different faults in position and time, this means we have to compile the target program every time we want to change the fault position into the register and the fault timing inside the execution. This makes this approach infeasible and time-consuming.

The last two possibilities are using time-based interrupt and using the debugger. The first possibility needs to use a timer, which could be hardware or software. Suppose we decide to use a hardware timer. In that case, the problem is that the configuration of a timer and its behavior could bring the alteration of some architectural events that potentially could train the model over biased data, different from the data that will be used during the operational phase of the model when the fault injector is no more used.
Using a software timer presents another problem, but first, we have to take a step forward. To execute the benchmarks, we decided to use a real-time Operating System, FreeRTOS. Our OS implements the software timer. Unfortunately, the ones implemented by FreeRTOS present a low granularity of one millisecond.
Given the fact that the total target application execution duration is of the order of a few milliseconds, a potential injection triggered by the FreeRTOS timer would also present a low granularity in terms of a temporal level.

In the end, we decided to use the debugger. Modern processors have a module dedicated to debugging, able to control the software execution over the CPU, and able to access CPU registers both for reading and writing.
This approach introduces a little overhead in terms of time due to the fact that the normal execution is stopped by the external debug module. This module has to read and write the CPU registers and, at the end, resume the standard computation.
This behavior is repeated for each execution done to create the final dataset.
On the other hand, this approach does not modify the executable and so does not need to recompile the code and re-flash the board.
Moreover, this approach presents a good temporal granularity, allowing computation to be blocked at the assembly instruction level simply by inserting breakpoints, without any difference from a pure temporal approach using the timers, with the advantage of not using precise control, debugger provides precise control over the execution of the program, allowing you to set breakpoints, inspect memory, and step through code to pinpoint issues. Instead, using a timer lacks repeatability, as this approach relies on timing and interrupts, which might not always produce the exact conditions you want to test. For example, if we configure the same injection

point at the same time for different executions, the outcome could be different due to the fact that the OS schedule is not deterministic.
The timer is configured to trigger the interrupt after a specific period of time. In different executions, it could potentially stop the program in different positions in different lines of code (LOC).

Now, exploring more in depth the debugger approach. The modern CPUs usually integrate a debug module, which is mainly used for two modes:

- Self-hosted debug.
- External debug.

The most interesting mode for us is the second, which, as its name suggests, allows us to control the debug phase over our target architecture from an external host. For us, a key activity of an external debugger is to read and write the registers inside the processor core. To do this, the debugger (external host) needs to physically connect to the chip and then generate. Typically, an external debugger uses the JTAG protocols (IEEE 1149.1 [15]).
Another common debug activity is to halt a processor core by stopping the execution of the program and allowing an external debugger to check the current status of the processor core registers.

All these functionalities are exploited to create the fault injector. Given the number of executions, the basic idea is to set a breakpoint each time in a different random position. When the execution reaches that breakpoint, select a random register and perform a single bit-flipping, resume the execution, wait for its completion, collect the performance counter, and restart with a new execution. This procedure is shown more in depth in the image 3.1.

That is all for the more theoretical aspects, in the next section, we will instead go into more detail regarding the more practical aspects.

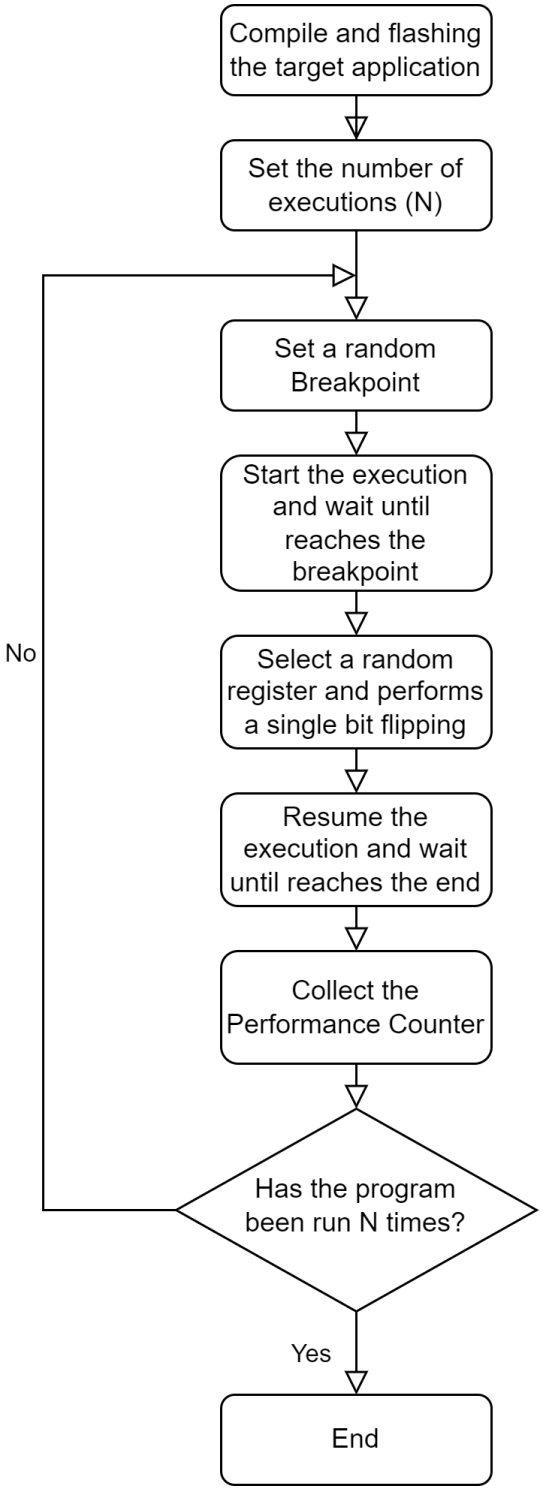


Figure 3.1: Flow Chart of the fault injector.

3.1 Setup

Now, let's move on to the more practical aspects for what concern the hardware and software setup used in this thesis work.

Starting with the hardware setup. The chosen target architecture is a Dual ARM Cortex A9 on board the Xilinx PYNQ Z2, image 3.2.
The PYNQ Z2 is an open-source project from Xilinx. It is a board based on Xilinx Zynq SoC, which is designed for the Xilinx University Program to support PYNQ (Python Productivity for Zynq). It has many features and interfaces that are useful for trying out the capabilities of the PYNQ framework.

Some of the main characteristics are:

- Processor: ARM Cortex A9.
- Programmable logic FPGA.
- Memory: 512MB DDR3.
- Storage: MicroSD.
- Video ports: HDMI In and Out.
- Network: 10/100/1000 Ethernet
- Expansion USB 2.0 Port
- GPIO and Other I/O like LEDs and buttons.

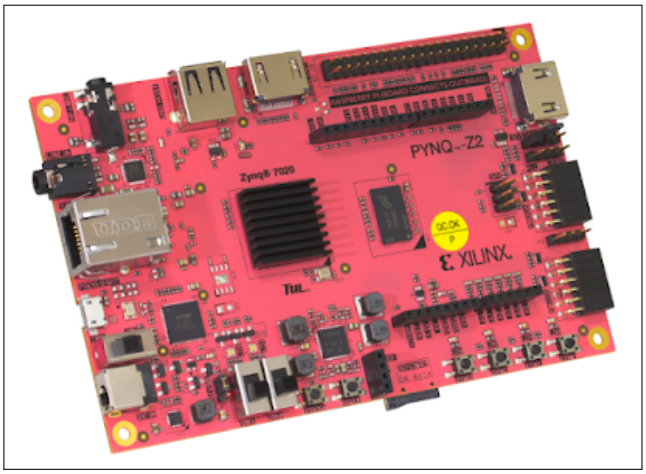


Figure 3.2: Pynq Z2 board.

We decided on this board given the Arm Cortex A9 processor, which is a highly adopted platform that well represents those used in the real environment in terms of cost and performance. Furthermore, a USB port is present, able to configure the board through JTAG.

The Arm Cortex A9 is a 32-bit dual-core processor. It has a frequency of 650 MHz and is based on ARMv7, but it also supports the Thumb and Thumb-2 instruction set and presents a coherent management of the cache, it presents a configurable Performance Monitoring Unit that is easy to configure and able to track a wide set of architectural events.
it is one of Arm's most widely adopted processors and is used in many systems on a Chip (SOC), like smartphones and videogames.
The Cortex-A9 processor features a dual-issue, partially out-of-order pipeline and flexible system architecture with configurable caches and system coherency, achieving a better than 50% performance over the Cortex-A8 processor in a single-core configuration.

The fact that this processor is dual-core is essential. Thanks to that, in a future work, we can run the target application, the benchmark, on a core, and thanks to the other core, we can run a specific program able to collect statistics about the execution of the target application. As already mentioned, the architectural events are tracked by the performance counter in the PMU. Once we collect these statistics, this specific support program can preprocess these data and pass them to the pre-trained ML Model that can perform the classification. In the end, it can also manage the outcome of the classification and potentially reschedule the target application task that occurs in a soft error. Potentially we can also run directly the pre-trained model on the other core of the Arm CPU.

One of the most relevant processors for our study is the Performance Monitoring Unit (PMU). According to the Technical Reference Manual, the Cortex-A9 PMU provides six counters to gather statistics on the operation of the processor and memory system, in other words, the architectural events. Each counter can count any of the 168 events available in the Cortex-A9 processor.
In the PMU are present the six counters registers and their associated event type registers, that specify the tracked event for each of them, plus other registers for the configuration. Accessible through the internal CP15 interface.

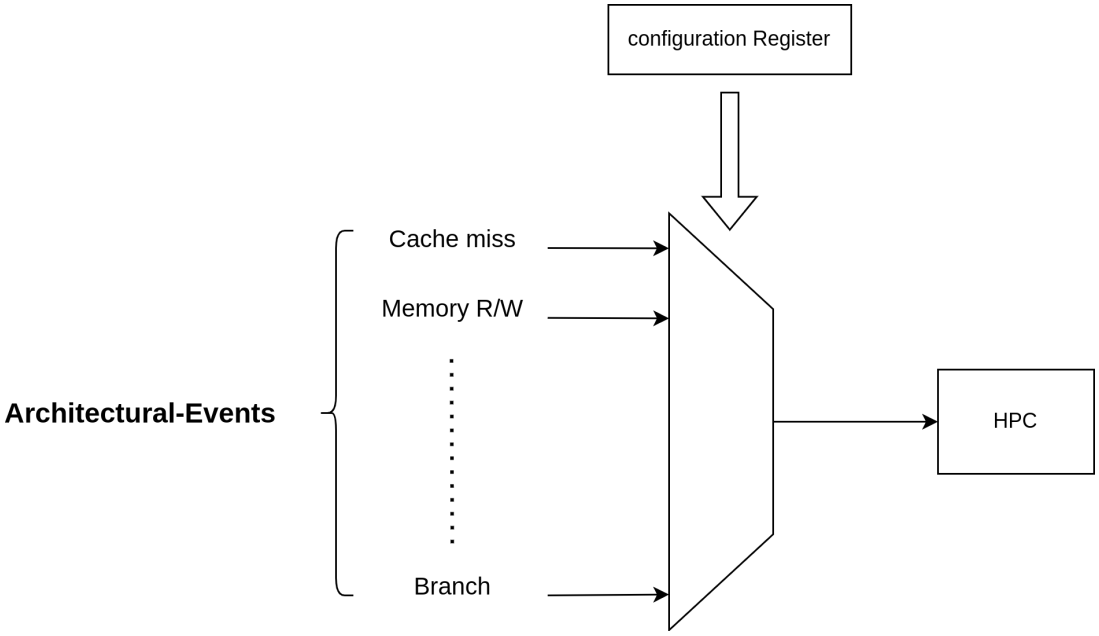


Figure 3.3: Event selection to configure a single hardware performance counter (HPC).

The performance monitoring events are defined in the ARM Architecture Reference Manual (https://developer.arm.com/documentation/ddi0388/latest/). Some examples are instruction cache miss, data cache access, data cache miss, data read, data write, exception taken, exception return, change of the PC, immediate branch, branch mispredicted or not predicted, and so on. For more information about these events, see the ARM Architecture Reference Manual.

As already mentioned, the PMU can track the architectural events in six registers that act like performance counters, but these registers need to be configured. To do that, other configuration registers are present in the PMU. According to the 32-bit based architecture, all registers are 32-bit width.
Let's check all the registers present in the PMU and their descriptions, table 3.1.

Now that we have seen the embedded system, particularly the processor on board, we can move to see the Software setup.
The software stack comprehends the Hardware Abstraction Level (HAL), an Operating System (OS), and, on the top, the application level.

The hardware abstraction level is a set of routines in software that provide access to hardware resources through programming interfaces that allow the writing of device-independent code by providing standard operating system (OS) calls. For

Table 3.1: PMU Register Descriptions. There are six PMXEVCNTR and six corresponding PMXEVTYPER identified by the last character that is a number from zero to five.

Name	**Type**	**Description**
PMXEVCNTR0-5	RW	Event Counter Register
PMCCNTR	RW	Cycle Count Regiter
PMXEVTYPER0-5	RW	Event Type Selection Register
PMCNTENSET	RW	Count Enable Set Register
PMCNTENCLR	RW	Count Enable Clear Register
PMINTENSET	RW	Interrupt Enable Set Register
PMINTENCLR	RW	Interrupt Enable Clear Register
PMOVSR	RW	Overflow Flag Status Register
PMSWINC	WO	Software Increment Register
PMCR	RW	Performance Monitor Control Register
PMUSERENR	RW	User Enable Register
PMSELR	RW	Event Counter Select Register

this work, this part is not so relevant; we have to know that it exists and is needed for the project, but it is not modified.

The application, in our case, acts like a target for the classification, like a benchmark. We decided to use MiBench [16], a free representative embedded benchmark suite that contains benchmarks and input files for each of the different programming

groups. We use a subset of them:

- **QSort** from the Automotive groups.
- **SHA** from the Security groups.
- **Dijkstra** from the Network groups.

They present different characteristics in terms of time and space complexity, as we can see in the table below 3.2.

Table 3.2: Benchmarks characteristics

Name	Time Complexity	Space Complexity
Dijkstra	$O(V^2)$ when using a simple implementation, but it can be optimized to $O(E + Vlog(V))$, (E = Edges, V = Vertices)	Depends on the data structures used for implementation. It can range from $O(V^2)$ to $O(V + E)$
QSort	$O(nlog(n))$, but it can degrade to $O(n^2)$ in the worst case.	$O(log(n))$ for the recursive call stack.
SHA	SHA-256 has a time complexity of $O(n)$.	Usually constant, as it operates on fixed-size blocks of data.

In summary, among the three benchmarks, SHA and Dijkstra's algorithms are typically more compute-intensive due to their computational complexity, while QuickSort is relatively less memory-intensive and more compute-intensive. However, it's important to note that the resource usage of these benchmarks can vary depending on the specific implementation and input data.

Another fundamental part of the software stack is the operating system. It is in charge of managing the application executions over the hardware, using the routines defined on the Hardware Abstraction Level.
We decided to use FreeRTOS, which is a real-time operating system for embedded systems. It is distributed under the MIT open-source License, and it is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors.

A real-time operating system is an OS for real-time computing applications, otherwise from a time-sharing operating system (Unix), which manages the sharing of system resources with a scheduler. All processing must occur within the defined constraints that are event-driven and preemptive, making the OS able to change the task priority. Event-driven systems switch between tasks based on their priorities, while time-sharing systems switch tasks based on clock interrupts.
This kind of OS, a specialized software system designed for applications where timing and responsiveness are critical, guarantees predictable and low latency, unlike general-purpose systems.
They are extensively used in industries such as aerospace, automotive, and industrial automation, ensuring tasks are executed within precise time constraints, making them ideal for Safety-Critical Real-Time Embedded Systems applications. That is exactly what we are looking for in our work.

In particular, FreeRTOS offers features like task scheduling and inter-task communication. It is mostly written in the C programming language to make it easy to port and maintain.
It provides methods for multiple threads, in this case, called task synchronization primitives, like mutex and semaphores, and also provides methods for static and dynamic memory allocation. Unlike Unix-based Os, it pones the emphasis on compactness, low overhead, and speed execution. FreeRTOS can be thought of as a thread library rather than an operating system.
Other key features are:

- POSIX-like input/output (I/O) abstraction.
- Scheduler can be configured for preemptive (priority) or cooperative multitasking (round-robin).
- Tick-less option for low-power applications.
- Coroutine support (lightweight tasks with limited use of the call stack).
- system timer based on tick (1 ms).

Now we have seen the complete software stack, image 3.4. The complete setup also includes a host machine with Ubuntu Linux and the Xilinx Vitis IDE installation. The host machine is in charge of compiling the freeRTOS, plus the benchmarks, and flashing it on the Pynq Z2 board, then controlling the execution, using the debugger commands, and, at the end, collecting the results.

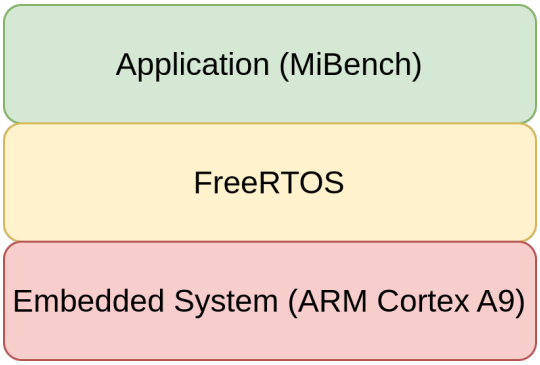


Figure 3.4: Software Stack.

3.2 Realization

First of all, starting with the right version of FreeRTOS for ARM Cortex A9 embedded processors, let us go to preparing the benchmark from Mibench.
To do this, we need to configure a new task for running the benchmark with the already implemented freeRTOS function **xTaskCreate()**. This function creates a new task and adds it to the list of task that is ready to run, take as parameters:

- The pointer to the task entry function (name of the function that implements the task).
- A text name for the task.
- The stack allocated to the task.
- The task parameter is not used set to null.
- The task priority (IDLE_PRIORITY is the lowest).
- The pointer for returning the handle to the created task.

Then we need to run this task starting the scheduler with the freeRTOS function **vTaskStartScheduler()**.

Listing 3.1: RTOS Task Creation Code

```
1  int main ( void ){
2      xTaskCreate(     faultInjectorTask ,
3                       ( const char * ) "fault Injector",
4                       configMINIMAL_STACK_SIZE,
5                       NULL,
6                       tskIDLE_PRIORITY + 1,
7                       &xTxTask );
8      vTaskStartScheduler();
9      for( ;; );
10 }
11 static void faultInjectorTask( void *pvParameters ){
12     confPMU();
13     beanchmark();
14     readPMU();
15     vTaskDelete( NULL );
16 }
```

In freeRTOS, tasks are normally implemented as an infinite loop; the function that implements the task must never attempt to return or exit. If all is well, the scheduler will now be running, and the infinite loop line will never be reached. If that line does execute, then there was insufficient FreeRTOS heap memory available

for the idle and/or timer tasks to be created.
At the end of the task with the vTaskDelete(NULL), delete the task.

Now we have a version of freeRTOS with installed our benchmark. The compilation results in an Executable and Linkable Format. The ELF format is flexible, extensible, and cross-platform, supporting different endiannesses and address sizes to be compatible with different CPU and instruction set architectures and many operating systems.

The main problem is the realization of a script able to control the target application execution using debug commands. Xilinx Software Command-Line Tool (XSCT) is an interactive and scriptable command-line interface to Xilinx SDK that also includes Vitis. XSCT is based on Tools Command Language (Tcl) and supports many actions, like creating and configuring hardware, board support packages () and application projects, and flash boot images.
XSCT also includes specific commands to control the debugging of the running application, investigating the hardware and the software.
To script those commands, we use a Python module called Pexpect. This module makes Python able to spawn child applications, control them, and respond to expected patterns in their output. This module is useful for automating applications, for example, ssh or ftp.

Now we have all the ingredients to create our fault injector:

- **XSCT** commands to control the program execution and debugging.
- **Pexpect** module to script those commands in a Python program.

Before starting to see in-depth the realization of the script, some problems must be addressed. First, the architectural events are a lot, more precise 168 for this specific architecture, but we have only six performance counters (PC) in the PMU. To solve this problem, we ran the target application with the same fault but tracking any time different events.
For us, a fault is a tuple of where and when a bit flipping is performed, the bit in the register, and the line of code.
Basically, we have:

$$\frac{number\_of\_events}{number\_of\_PC} = \frac{168}{6} = 28 = number\_of\_executions\_for\_every\_fault \tag{3.1}$$

This means that the number of total executions needed to create the final dataset is significantly increased by a factor of 28, considering collecting all possible events.

The second problem is originated by freeRTOS. In the fault injector, we need to know when the execution reaches the end to collect all the PC and restart with a new execution. The problem is that freeRTOS keeps executing the IDLE task if no other tasks are available for running, this means that the execution over the board never reaches the end. To address this problem, we can set a final breakpoint at the end of the task with the benchmark, wait for the program to reach that final breakpoint and then collect the PCs and start a new simulation.

According with the flow chart 3.1, the script needs to:

1. Set the global variable (number of faults, number of executions for faults, etc.) and create an empty dataset.
2. Spawn the XSCT console and.
3. Run the initialization script (init.tcl), which is a list of XSCT commands for connecting to the board and flashing the elf file on it.
4. set the final breakpoint.
5. Start the simulations.

Listing 3.2: example of pexpect code to setup the script

```
def main():
    f = open("dataset", "w")     #Output file

    xsct = pexpect.spawn("xsct")
    xsct.expect("xsct%")
    print(xsct.before.decode())

    xsct.sendline("source /home/enrico/Desktop/marvin/scriptTCL/init.
    tcl")
    xsct.expect(".*Successfully downloaded.*")

    xsct.sendline("bpadd -file freertos_hello_world.c -line " +
    final_bp)
    xsct.expect(".*Breakpoint 0.*")

    ...
```

In pexpect the main functions are spawn(), to spawn the child process, **expect()** to catch data in the standard output (stdout) of the child and wait for a certain pattern, **sendline()** to send a line in the standard input of the child (useful to submit commands).

Thanks to these commands, we can control the execution and realize the fault injector. But as already said, the pexpect Python module alone is not enough. We also need to use the XSCT commands to access the embedded system hardware. We need commands to:

- Connect and flash the target system.
- Control the debug.
- Access the CPU registers.
- Read and write the memory.

In particular, we need to set and delete breakpoints, start, stop, and resume executions, read and write CPU registers and memory, see table 3.3 or more in details on Xilinx site (https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/XSCT-Commands).
We can now put together all these commands in a Python script to create our fault injector. As mentioned before, it requires two cycles nested, the outer one cycles over the faults, and for each fault generated randomly:

- **Where**: selecting CPU register and bit position.
- **When**: Select a memory address in the code space for setting a breakpoint.

The inner cycle, instead, is in charge of executing the benchmark task more time for each fault. Due to the fact that we have only six performance counter to collect all the architectural events, as see in equation 3.1. Actually, as we will see in the next chapter, we collect only a subset of all architectural events, so the number of cycles in the inner cycle is reduced.

Another important particular is that the Python script runs on the host machine, and freeRTOS, which runs on the target system, can exchange information using the central memory of the target system. Basically, freeRTOS can read and write normally his memory space; instead, the host can use the XSCT commands to write and read in the target memory.

Given that, the inner cycle has to be written in the memory, in a specific address known by the target, and events have to be tracked. Then, set the breakpoint, chosen by the outer for, and run the execution, wait until the first breakpoint is reached, inject the faults, remove the fault breakpoints, resume execution until the last breakpoint is reached, or wait for a timeout exception to occur. In the end, in the outer for the host read in the target memory, in a specific address, the result written by freeRTOS and written in the dataset, benign or silent data corruption, or if a timeout occurs, the faults are marked as crash/hangs.

Listing 3.3: Fault injector script

```
for i in range(int(num_of_fault)): #Outer cycle
    rand_bp_pos = random.randint(int(init_task, base=16), int(
    fin_task, base=16))
    reg_flipping = random.randint(0, 10)
    pos_flipping = random.randint(0, 31)

    dataset.write(f"{i}: reg: {reg_flipping} pos: {pos_flipping} at
    LOC: {rand_bp_pos}\n")
    xsct.sendline("mwr 0x10200 " + str(i) ) #set events to track
    crash = False

    for y in range(num_of_run): #Inner cycle
        xsct.sendline("mwr 0x10000 0x0")
        xsct.sendline("mwr 0x10000 " + str(y) )
        rand_bp_cmd = "bpadd " + str(rand_bp_pos)
        xsct.sendline(rand_bp_cmd)
        xsct.sendline("con -addr 0x00100000")

        xsct.expect(".*Breakpoint.*")
        fault_injection(xsct, reg_flipping, pos_flipping, crash)
        xsct.sendline("bpremove " + str(num_bp_remove))
        num_bp_remove = num_bp_remove + 1
        xsct.sendline("con")

        try:
            xsct.expect(".*Breakpoint.*")
        except:
            #A timeout exception occurs
            crash = True
            break

    if crash == False:
        xsct.sendline("mrd 0x10100")
        xsct.expect(".*10100: *")
        value = xsct.readline().decode()
        if value[len(value)-5] == '1':
            dataset.write("benign\n")
        else:
            dataset.write("SDC\n")
    else:
        dataset.write("crash/hangs\n")
```

Now that we have seen the Python script, we can come back to the target task in freeRTOS. Beyond the benchmark, that task also has to configure the PMU to track the correct architectural events and read the performance counter.
Following this order, see code 3.1, the target task has to:

1. Configure the PMU.

2. Execute the benchmark.

3. Read the performance counter in the PMU.

For the benchmarks, we do not have much else to add. We tried some benchmarks from MiBench, a little modified to fit in freeRTOS, changing the input part and adding at the end the check of the output to control if the execution is benign or silent data corruption and write it on his memory in a specific address, read by the python script.

To configure the PMU, we need to use the register present on it. See table 3.1. The configuration procedure needs to configure the PMU, read in the memory from a specific address the events that, in this run, the PMU has to track, and then start the real configuration procedure. For writing in these PMU's registers, we have to use the ARM assembly command Move to Coprocessor from ARM Register (MCR <register> <value>) where the coprocessor is the PMU. To run this instruction from c code, we used the *asm* keyword that allows to embed assembler instruction within c code, associated with the *volatile* keyword that forces the compiler's optimizer to execute the code as is 3.4. First, we need to enable user mode access in the PMU, then in the Performance Monitor Control Register (PMCR), enabling all the events counter and reset them, and enable all performance counters in the Performance Monitor Count Enable Set register (PMCNTENSET). Loop to configure each counter. For each counter, we have to select it in the Performance Monitors Event Counter Selection Register (PMSELR) and select an event to track in the Performance Monitors Event Type Select Register (PMXEVTYPER).
See the complete code 3.4.

The procedure is similar for reading the PMU at the end of the benchmark, but instead of using the MCR instruction, the Move to ARM Register from Coprocessor is used. For each PC, we have to select the counter in the PMSELR, read the event type in PMXEVTYPER, and the counter in the Performance Monitors Event Counter Register (PMXEVNCNTR). At the end, print on the console the event type plus his value.

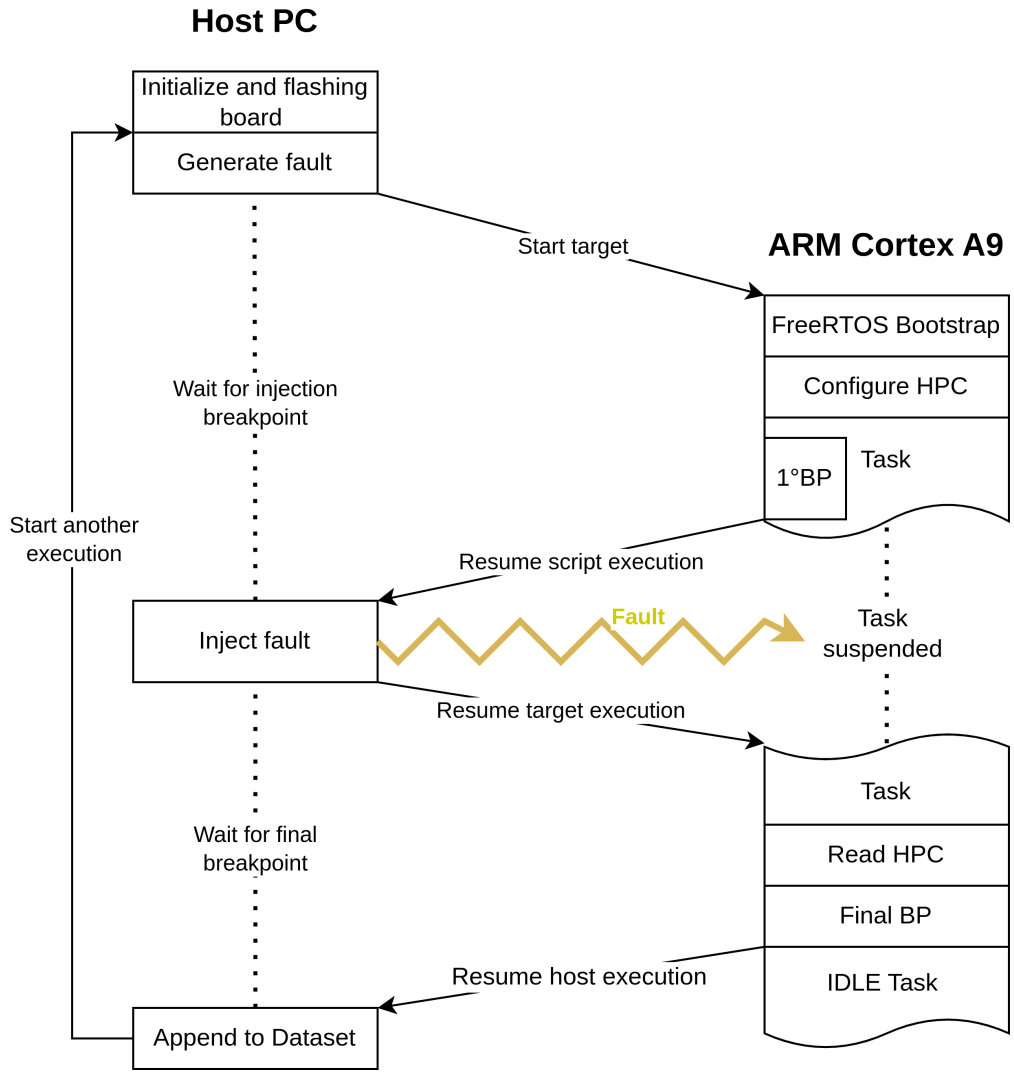


Figure 3.5: Flow software execution over the host and the target.

Now we have all the parts of our fault injector. The last thing that we need is a simple Python script, a listener that reads what is written on the serial port, in our case, the *'/dev/ttyUSB1'*, the one connected with the target system.
This script is needed to read and save the performance counter in the dataset.

Listing 3.4: PMU Configuration

```
static void confPMU(){
    char* pointer = (char*)0x10000;
    int c = (int)pointer[0];

    int events[42] = {1,3,4, ..., 144,145,146};

    // Enable user-mode access to performance counters
    asm volatile ("MCR p15, 0, %0, C9, C14, 0\n\t" :: "r"(1));
    //Enable bit in pmcr for enabeling the events counter and reset
    them
    asm volatile ("MCR p15, 0, %0, C9, C12, 0\n\t" :: "r"(0x4109300B)
    );
    // Enable all counters in pmcntenset
    asm volatile ("MCR p15, 0, %0, c9, c12, 1\t\n" :: "r"(0x8000003f)
    );
    for(int i = 0; i < 6; i++){
        // select PC in pmselr
        asm volatile ("MCR p15, 0, %0, c9, c12, 5\t\n" :: "r"(i));
        // select event to track in pmxevtyper
        if(c < 7)
            asm volatile ("MCR p15, 0, %0, C9, C13, 1" :: "r"(events[
    i+c*6]));
    }
}
```

Listing 3.5: PMU Reading

```
static void readPMU(){
    unsigned int counter_value;
    unsigned int evn_type;
    for(int i = 0; i < 6; i++){
        //selecting counter in pmselr
        asm volatile ("MCR p15, 0, %0, C9, C12, 5" :: "r"(i));
        //reading event type in pmxevtyper
        asm volatile ("MRC p15, 0, %0, C9, C13, 1" :"=r"(evn_type));
        //reading counter in pmxevncntr
        asm volatile ("MRC p15, 0, %0, C9, C13, 2" :"=r"(
    counter_value));
        xil_printf("%d: %d\n", evn_type, counter_value);
    }
}
```

Table 3.3: XSCT Commands description

Command	**Description**
connect [options]	Allows user to connect to a TCF server
disconnect	Disconnect from TCF server
targets <target id>	Return the list of all available targets, or if a target id is specified, set that target as the current target
rrd [options] <reg>	Read registe fot active target
rwr <reg> <value>	Write the <value> to active register specified by <reg>
state	Return the current execution state of target
stop	Suspend execution of active target
con <address>	Resume execution of active target, with <address we can optionally specified from where reasume the execution
mrd [options] <address> <num>	read <num> data values from the active target's memory <address>
mwr [options] <address> <values> <num>	Write <num> data values from list of <values> to active target memory address specified by <address>
dow [options] <file>	Download ELF file <file> to active target
fpga <bitstream-file>	Configure FPGA with given bitstream.
rst	Target reset
bpadd [option]	Set a breakpoint, option for specifing the location address or line
bpremove <id>	Remove the breakpoints/watchpoints specified by <id>, -all remove all breakpoints
bplist	List all the breakpoints

Chapter 4

Machine Learning Model

Artificial Intelligence (AI) is a vast world. Many are the models, like many are the applications from the easiest classifier to complex generative model.

Fortunately, in our case, we need a simple classifier able to distinguish the outcomes of a fault injection occurring during the execution of benchmarks.
Despite the fact that our goal is relatively simple, many of the possibilities are split into two main categories:

- Machine Learning models.
- Deep Learning models.

In the first category, we found all the classical classifiers like Gaussian, logistic regression, SVM, etc. Instead, in the second, we found classifiers based on neural networks and models derived from that.
The choice of the model and the choice of his hyper-parameters are crucial, but before that, we need to do a little analysis of our datasets.

4.1 Data Analysis

The datasets used in this work are three, each created over a different benchmark. That are SHA, QuickSort (QSort), and Dijkstra.
All the benchmarks include as features the values collected by the Performance Counters (PC). To reduce the collecting time, not all the available architectural events are tracked. Only the one that from preliminary studies figures it out stays always constant.
In total, we have 42 architectural events that are variable across different executions. According to the equation 3.1, we need $\frac{Events\_Number}{PC\_Number} = \frac{42}{6} = 7$ that is the number

of execution per fault needed to collect the 42 events. Comparing this result with 28 executions needed to collect all the available events, a reduction of a quarter allows us to significantly reduce the time required to create a dataset.

Recalling our goal, we need to perform a binary classification to separate the executions into two categories: Silent Data Corruption (SDC) and Benign. The other possible outcomes, crash, hang, and reboot, are trivial to detect in runtime and, for this reason, are not considered during the classification.
Due to this fact, the fault injection is only performed in the first eleven registers from R0 to R10, keeping out the special registers like the Stack Pointer Register, Link Register, and the Program Counter Register, which are the registers most likely to generate crashes, reboots and hangs.

On the other hand, injection faults only in standard registers (R0-R10) do not mean that crashes, hangs, and reboots are completely avoided. In a lower percentage, they are still present.
Those can occur for different reasons. One example could be a corruption of a pointer that leads the program to access a portion of memory that does not exist or in which it does not have permission to read/write, causing a segmentation fault and, in consequence, a crash.

In the table 4.1 we can see the breakdown of the various categories separated by benchmark.

Table 4.1: Breakdown of the various categories separated by benchmark

Benchmark	**Benign**	**SDC**	**Crash/hang**	**Total**
SHA	13657	1733	54	15444
QSort	4919	133	63	5115
Dijkstra	4920	228	32	5180

The category distributions reflect the benchmark characteristic. Looking at the high number of crashes in QSort and Dijkstra, we can presuppose that they are generated by the extensive use of structs and vectors in memory indexed by pointers.
Instead, the high number of Silent Data Corruption generated during the SHA dataset creation could be caused by the high number of CPU operations needed to compute the result over the same input data.

Anyways, all three datasets are filtered to remove crashes, hangs, and reboots, and keeping only the two classification's target categories benign and SDC.

Now, we have the datasets filtered with only the potentially useful information for the classification.
Each sample is described by 42 architectural events: 42 continuous variables and is preprocessed employing **Z-normalization**, which basically is centering and scaling to unit variance, equation 4.1.

$$z = \frac{x - \mu}{\sigma},\ \text{x = sample, } \mu = \text{mean, } \sigma = \text{standard deviation} \tag{4.1}$$

The Z-normalization is helpful for many reasons. Like the comparison of variables that use different units or scales, it can improve the numerical stability of certain calculations, and also, for machine learning algorithms, standardization helps gradient-based optimization algorithms converge more quickly.

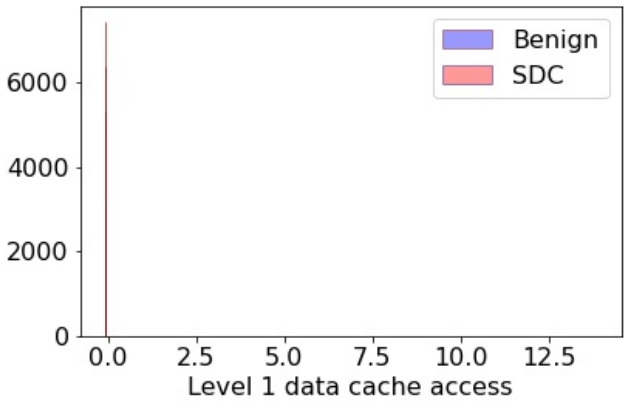
From figure 4.1, we can observe some examples of raw feature distributions, plotted by separating the SDC from the benign. We can notice that those raw distributions are very irregular and sometimes seem completely overlapped. This can negatively affect the classification.

For this reason, we have also tried to apply a **Gaussianization** as a prepossess technique. Following the equation 4.2.

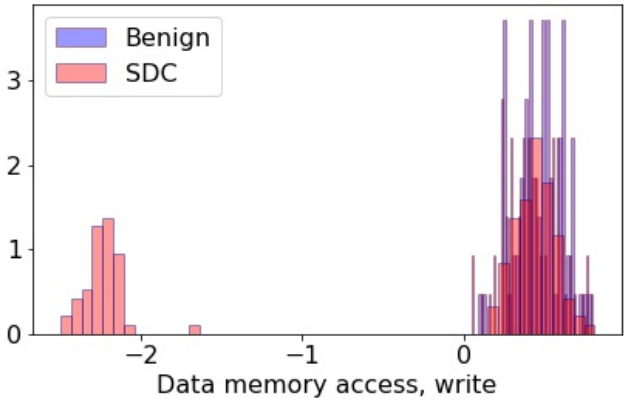
$$y = \Phi^{-1}(F(x)),\ \text{x = sample, } \Phi^{-1} = \text{inverse, } F(x) = \text{cumulative distribution func} \tag{4.2}$$

Gaussianization is a data transformation technique that makes data follow a Gaussian distribution.
Gaussianization can be beneficial for parametric modeling, where it assumes a specific functional form for the data distribution. When data is Gaussianized, it becomes easier to apply models like linear regression or the Gaussian classifier model, and in particular, the Gaussian is less sensitive to outliers, reducing the impact of outliers on statistical analyses.

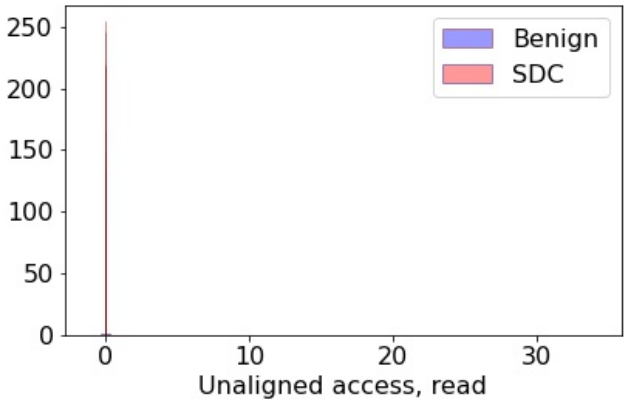
We can see from figures 4.2 that now the features are scaled to follow a Gaussian distribution and are more spread on the plot and, in same cases, also less overlapped.
This could suggest that a model based on a Gaussian Classifier could fit well our problem.



(a) QSort

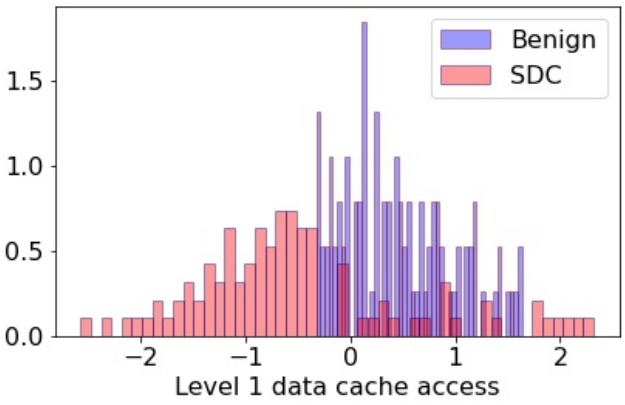


(b) Dijkstra

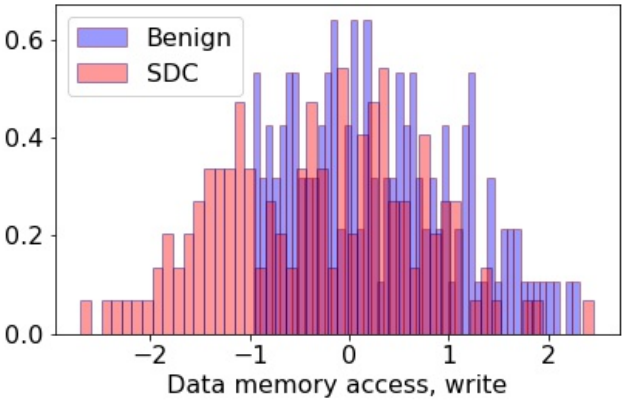


(c) SHA

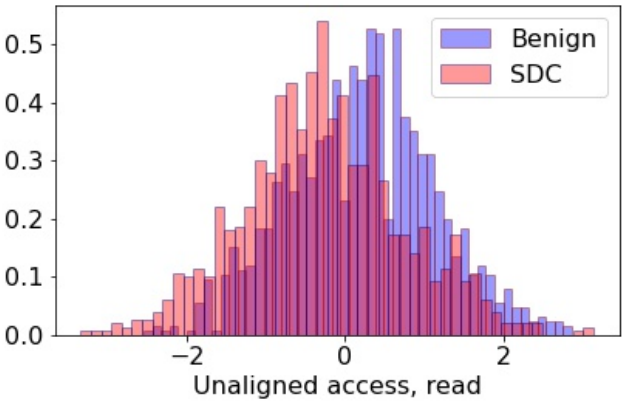
Figure 4.1: Histogram plot of raw features, only Z-normalized



(a) QSort



(b) Dijkstra



(c) SHA

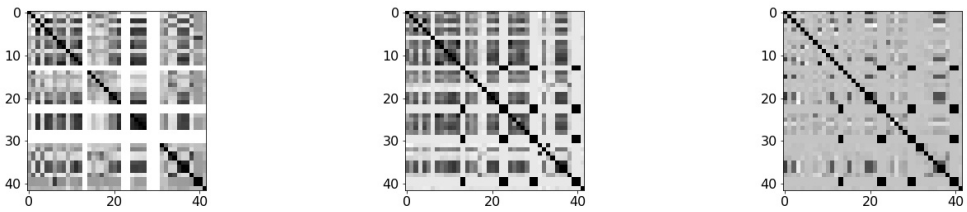
Figure 4.2: Histogram plot of Gaussianized features.

A correlation analysis of the features (heatmaps 4.3) shows some correlation computed as the absolute value of the Pearson correlation coefficient through the features themselves:

$$\left| \frac{Cov(X,Y)}{\sqrt{Var(X)}\sqrt{Var(Y)}} \right| \tag{4.3}$$

The Pearson correlation coefficient measures the linear relationship between two features. It varies between -1 and +1, with 0 implying no correlation. Correlations of -1 or +1 imply respectively a negative or positive exact linear relationship.
Instead, the p-value roughly indicates the probability of an uncorrelated system producing features that have a Pearson correlation at least as extreme as the one computed from these features. If this probability is lower than 5% ($P < 0.05$) the correlation coefficient is called statistically significant.

This suggests we may benefit from using PCA to re-map data in a lower dimension space to remove the uncorrelated features and to reduce the number of parameters to estimate. It also suggests that models that use diagonal covariance matrix (Naive Bayes assumption) should perform significantly worse than other models.



(a) QSort **(b)** Dijkstra **(c)** SHA

Figure 4.3: Heatmaps of the Pearson correlations. The darkest regions represent the features more correlated (The features are over the two axis)

It is also possible to compute the correlation between the features and the label. In this case, a greater correlation between the features and the labels is reflected in a greater accuracy in the classification. Furthermore, the plots of the Gaussianized features, figure 4.2, clearly present some regions without overlapping the two categories. This should also affect the correlation between those more discriminant features and the label, presenting a higher correlation value.

Table 4.2: Pearson correlation values between most discriminant features and the label separated by benchmark, ordered from the most negative correlated to the most positive correlated

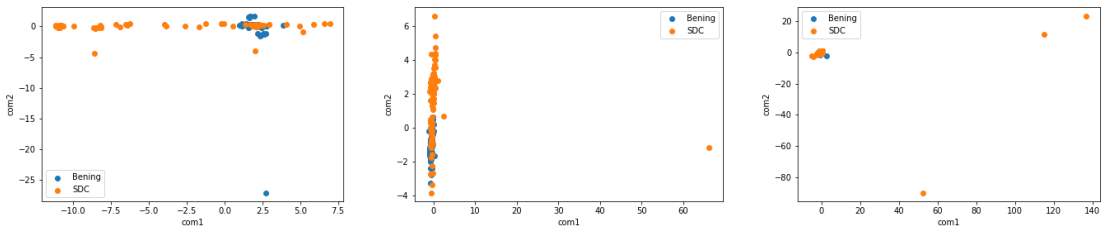
Event	**Pearson coefficient**	**P-value**
	Dijkstra	
9	-0.556	1.381e-23
2	-0.407	8.317e-13
27	-0.405	1.023e-12
...	...	...
18	0.020	0.736
6	0.084	0.157
3	0.348	1.520e-09
	QSort	
25	-0.557	3.431e-17
21	-0.529	2.194e-15
11	-0.509	3.263e-14
...	...	...
39	0.719	0.318
33	0.451	3.794-11
38	0.475	2.336e-12
	SHA	
12	-0.851	0.0
8	-0.780	0.0
4	-0.534	4.598e-172
...	...	...
34	0.127	7.779e-15
38	0.164	1.395e-15
5	0.255	6.032e-36

The correlations reported in table 4.2 prove the hypothesis that a significant correlation exists between the features, with less overlapping between the two categories and the label. See appendix A for the event description.

The last analysis performed over the three benchmark datasets is the one of the scatter plots. A scatter plot uses dots to represent values for two different numeric variables.
Our data are represented with more than two variables, to be precise, with 42 numerical features (architectural events). Principal Component Analysis is performed to reduce the data dimension from 42 to 2.

The Principal Component Analysis (PCA) is an unsupervised learning technique for reducing the dimensionality of data, increasing interpretability, and at the same time minimizing information loss.
It helps to find the most significant features in a dataset. For our purpose, it makes data easy to plot in a 2D scatter plot. In general, it is also useful because working with high-dimensional data is time-consuming, and often, machine learning models seem to overfit and reduce the ability to generalize.
Thanks to the PCA, the data are represented by the projection over the N principal component, where N are parameters that represent the numbers of dimensions after the reduction, in our case two. The Principal Component is a straight line that captures most of the variance of the data. They have a direction and magnitude. Principal components are orthogonal projections (perpendicular) of data onto lower-dimensional space.

The figures 4.4 shows the scatter plot of the z-normalize dataset after applying 2D PCA, separated by categories.
They show the distribution of the SDC in general more scattered with respect to the benign and we a significant presence of outliers.

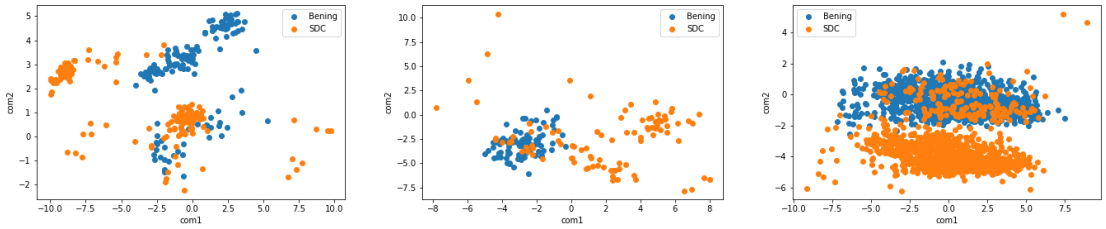


(a) Dijkstra **(b)** QSort **(c)** SHA

Figure 4.4: Scatter plots of the three datasets, after applying two dimensional PCA over z-normalized data.

This can be explained: we aspect this behavior for the reasons that bit flipping brings to a SDC has a high potential of in-deterministic behavior and consequently a significant alteration of the architectural events.

The scatter plotted over the gaussianized data are more scattered and more legible.



(a) Dijkstra **(b)** QSort **(c)** SHA

Figure 4.5: Scatter plots of the three datasets, after applying two dimensional PCA over gaussianized data.

In general, the two categories are not completely overlapping but not completely separated. In fact, they present the same region in common. This region, where the two categories are overlapped, they are regions hard to classify. For this region, by a trained model over those data, we did not expect a high accuracy.

4.2 Classifier

A classifier in machine learning is an algorithm that aims to categorize data into a set of classes. There are both supervised and unsupervised classifiers.

- **Unsupervised** machine learning classifiers are fed only with unlabeled datasets. They classify according to pattern recognition or structures and anomalies in the data
- **Supervised** machine learning classifiers are trained with labeled datasets, from which they learn according to predetermined categories.

Depending on the task and the data, there are different typologies of classification algorithms.

Decision Tree is a supervised machine learning algorithm used to build models like trees to classify data into hierarchical categories tree.

Naive Bayes classifiers are probabilistic algorithms that calculate the probability of each category for a given data point, then output for the highest probability:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)} \tag{4.4}$$

For each point, this equation 4.4 calculates the probability that a data point belongs within a certain category. Examples of this are Gaussian Classifier or Logistic Regression.

K-Nearest Neighbors is a pattern recognition algorithm that learns from training data points by calculating how they correspond to each other in n-dimensional space. K-NN places a given point within a category by calculating its nearest neighbor.

Support Vector Machines (SVM) classify data within finite degrees of polarity. Practically, SVM assigns a hyperplane that best separates the two categories. The best hyperplane is the one with the largest distance between each category.

Artificial Neural Networks are designed to work much like the human brain does. They belong to deep learning models that require vast amounts of training data. There are a variety of artificial neural networks, including convolutional, recurrent, and feed-forward.

Now, to choose the correct machine learning model, we have to start considering our goal. We have to perform a binary classification to classify our three

dataset samples into two categories (benign and SDC). Due to the fact that we have labeled datasets, we can exploit supervised techniques such as the Naive Bayes classifier or a Neural Network. Furthermore, our features are continuous variables that a Naive Bayes model can simply manage without requiring more complex Neural network models.
From the data analysis, we figured out that our data fit well in Gaussian distribution if properly Gaussianized. This suggests that a Naive Bayes Gaussian model that classifies data based on Gauss distribution could bring a good classification accuracy.

4.2.1 Gaussian Classifier

Using Bayes'theorem, we can derive a generative Gaussian classifier. These kinds of classifiers are simple, intuitive, and interpretable. The Gaussian/normal distribution is one of the most commonly present in nature. The model assumes that the continuous-valued features for each class follow a Gaussian (normal) distribution
The Gaussian classifier is built on Bayes' theorem, which is a fundamental concept in probability theory. It relates the probability of a particular event happening based on prior knowledge of conditions that might be related to the event.
The Naive assumption of feature independence. It assumes that the features used for classification are conditionally independent given the class label. This simplifies the model but may not hold true for all datasets. To build the model, two sets of parameters for each class need to be estimated: the mean (μ) and the variance (σ^2).
To classify a new data point, the Gaussian classifier calculates the likelihood of the data point's feature values under each class's Gaussian distribution. It then combines this likelihood with the prior probabilities of the classes using Bayes' theorem to determine the most likely class for the data point.

It also performs well with small datasets. This point is fundamental to our work because due to the high unbalancing of our two categories, An imbalance occurs when one or more classes have very low proportions in the training data as compared to the other classes, see table 4.3 for our categories proportion. Our data presents a severe class imbalance due to the fact that the occurrence of Silent Data Corruption is a rarer event compared to benign occurrence. The minority class is harder to predict because there are few examples of this class by definition. This means it is more challenging for a model to learn the characteristics of examples from this class. Also, the abundance of examples from the majority class (or classes) can swamp the minority class.

Table 4.3: Breakdown of the various categories percentage separated by benchmark

Benchmark	Benign	SDC
SHA	88.7%	11.3%
QSort	97.3%	2.7%
Dijkstra	95.6%	4.4%

For example, a model trained with 99% of the major class and 1% of the minority class can train the model with the bias of classifying all the samples belonging to the major class, with an accuracy of 99% that actually is very high. However, often, the goal is to detect these rare events. In SACRES, this is fundamental. It is more relevant to identify a potentially harmful SDC instead of classifying all as benign and missing the SDC.

For this purpose, we have to introduce two new metrics to use with accuracy:

- **Precision**: measures the accuracy of positive predictions, indicating how many of the predicted positive cases were actually true positives.
- **Recall**: measures the ability to identify all relevant instances, indicating the ratio of true positives to all actual positives.

They are useful metrics in unbalanced datasets. They are represented by the following equations:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.5}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.6}$$

Where:

TP (True Positives) : Correctly predicted positive instances.
FP (False Positives) : Incorrectly predicted positive instances.
FN (False Negatives) : Missed positive instances.
TN (True Negatives) : Correctly predicted negative instances.

Returning to our task, where the positive class is the SDC, the most relevant to detect and the minority, and the negative one is benign, our goal is to maximize the recall to increase the detection of all possible SDC. At the expense of a lowering of the precision.

Practically, this means that it is better to misclassify a benign respect than misclassify a SDC.

To address the problem of the unbalanced dataset, many techniques are possible, like resampling the dataset, balanced bagging classifier, or moving the threshold. We chose the simplest one: undersampling the database. Basically, we train our Gaussian classifier over a dataset properly filtered to randomly remove a portion of the majority class samples to re-balance the dataset.

Now that we have chosen a proper Gaussian classifier and we have a balanced dataset, the dataset is divided into a training dataset and into a testing dataset, following these proportions:

- **2/3** for the Train dataset.
- **1/3** for the Test dataset.

After training the classifier and testing it with the proper balanced dataset, these are the results in terms of accuracy:

Table 4.4: Accuracy results of the Gaussian Classifier for a balanced datasets, made use both Z-normalized data and gaussianized data.

Benchmark	**Z-Norm**	**Gaussianized**
Dijkstrs	75%	70%
QSort	72%	74%
SHA	77%	72%

Despite what was hypothesized before, the Gaussianization preprocess worsen the results, a part for the qsort dataset, but where the improvement is very limited. In general, the accuracy results obtained from the balanced datasets show that a trained Gaussian classifier is able to learn how to classify those samples, even if the accuracies are not very high, they are significantly above the 50% threshold, the percentage of a model that classify completely random.

For what concern the recall and the precision. the results are shown in table 4.5. As mentioned before, for our task the recall metric is very significant.

Table 4.5: Recall and Precision compute for every benchmarks for both z-normalized data and gaussianized data

Benchmark	**Recall**	**Precision**
	Z-Normalized	
Dijkstra	54.1%	93.8%
QSort	47.2%	94.4%
SHA	54.1%	99.0%
	Gaussianized	
Dijkstra	42.3%	94.7%
QSort	58.3%	84.0%
SHA	59.2%	79.9%

Unfortunately, the results show a poor recall against good precision. This means that our classifier is good at classifying our negative class, the Benign, but not the positive one, the SDC. This could be caused by the fact that benign distribution is less scattered and consequentially easier to model. For this reason, the data in the overlapped region (Region Hard to detect) are classified as Benign due to the high density of Benign in those regions. The recall results are slightly better with the Gaussianized data.

All the code needed to preprocess the dataset, train, test the model, and compute the metrics is written in Python, with the use of various libraries, like numpy, matplotlib, sklearn, and scipy.

Chapter 5

Conclusion

This thesis work aims to improve the Trustworthiness and the resilience of the microprocessor to soft errors caused by faults, applicable in the real scenario through Artificial Intelligence.

As explained in this report, the basic idea is to use the events collected by the Performance Monitoring Unit (PMU) to train a machine learning model able to detect when a fault causes a soft error. In particular, this work is focused on the binary classification task, considering only the Benign and Silent Data Corruption (SDC) outcomes of a fault. Considering as fault one randomly bit flipping in the CPU registers.

Due to our goal, the work needs to address two problems:

1. Fault Injector able to simulate real faults.

2. Machine Learning Model to detect SDC.

Starting with the Fault Injector, many possibilities were considered, from the HWIFI by means of specific hardware to a more sophisticated SWIFI approach with software emulates. In the end, we decided to realize a Software-Implemented Fault Injection (SWIFI) by using the debugger. We have largely explored this approach, creating a script in Python able to control the debug process of the benchmark executed on the embedded target system.

Basically, the script uses the XSCT commands (3.3) to stop the execution using the debugger, then uses access randomly to a register to perform a bit-flipping, and at the end, control the output to label the execution and collect the performance counter. This approach gives has the full controllability and repeatability of the fault injection process, giving the possibility to create a large dataset of

executions in a relative short time.

The second part of this thesis is about an analysis of machine learning models applied to the classification of the architectural events collected in the first part. The primary objective was to explore the feasibility of using machine learning techniques to detect SDC occurrences within the context of different benchmark datasets.

The analysis began with a correlation study, which revealed significant correlations between certain features and the classification label. These correlations supported the hypothesis that distinct architectural events have less overlap between categories, making them potentially valuable for classification.
Furthermore, Principal Component Analysis (PCA) allows for visualization in 2D scatter plots. These scatter plots highlighted the challenges of classifying SDC instances, as there was substantial overlap between the SDC and benign regions. This overlap, referred to as the "Region Hard to Detect," presented a significant challenge for accurate classification.

The results indicated that the Gaussian Classifier achieved an acceptable accuracy. However, it showed a trade-off between precision and recall, favoring precision. This meant that the model was more successful at correctly classifying benign instances but struggled to detect SDC instances effectively.
The evaluation of the model highlighted the challenge of detecting rare events like SDC in an imbalanced dataset. Future work could explore additional techniques for improving recall while maintaining a satisfactory level of precision. This may include experimenting with more complex classifiers and exploring different features.

In conclusion, this thesis contributes to the understanding of the use of the debugger to inject faults in a real embedded system and apply machine learning to the detection of SDC in architectural events. While the results suggest potential for improvement, they underscore the complexities of the task and the need for further research in this area.

Appendix A

Arm Architectural Events

Event Number	Description
0	Level 1 instruction cache refill
1	Level 1 data cache refill
2	Level 1 data cache access
3	Level 1 data TLB refill
4	Instruction architecturally executed, Condition code check pass, load
5	Instruction architecturally executed, Condition code check pass, store
6	Instruction architecturally executed, Condition code check pass, exception return
7	Instruction architecturally executed, Condition code check pass, Software change of the PC
8	Branch Instruction architecturally executed, immediate
9	Instruction architecturally executed, Condition code check pass, unaligned load or store
10	Branch instruction Speculatively executed, mispredicted or not predicted
11	Cycle
12	Predictable branch instruction Speculatively executed

13	Level 2 data cache refill
14	Level 2 data cache access, read
15	Bus access, read
16	Bus access, write
17	Bus access, Normal, Cacheable, Shareable
18	Bus access, peripheral
19	Data memory access, read
20	Data memory access, write
21	Unaligned access, read
22	Exclusive operation Speculatively executed, Load-Exclusive
23	Exclusive operation Speculatively executed, Store-Exclusive pass
24	Exclusive operation Speculatively executed, Store-Exclusive fail
25	Operation speculatively executed, load
26	Operation speculatively executed, store
27	Operation speculatively executed, load or store
28	Operation speculatively executed, integer data processing
29	Operation speculatively executed, Software change of the PC
30	Operation speculatively executed, Cryptographic instruction
31	Branch Speculatively executed, immediate branch
32	Exception taken, other synchronous
33	Exception taken, Instruction Abort
34	85 (missing)
35	Exception taken, Hypervisor Call
36	Exception taken, Instruction Abort not Taken locally
37	Exception taken, Data Abort or SError not Taken locally
38	Exception taken, IRQ not Taken locally

39	Release consistency operation Speculatively executed, Load-Acquire
40	Release consistency operation Speculatively executed, Store-Release
41	92 (missing)

Bibliography

[1] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. «Using heavy-ion radiation to validate fault-handling mechanisms». In: *IEEE Micro* 14.1 (1994), pp. 8–23. DOI: 10.1109/40.259894 (cit. on p. 7).

[2] Athanasios Chatzidimitriou, Pablo Bodmann, George Papadimitriou, Dimitris Gizopoulos, and Paolo Rech. «Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments». In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019, pp. 26–38. DOI: 10.1109/DSN.2019.00018 (cit. on pp. 8, 13).

[3] Qiang Guan, Nathan DeBardeleben, Sean Blanchard, and Song Fu. «F-SEFI: A Fine-grained Soft Error Fault Injection Tool for Profiling Application Vulnerability». In: May 2014. DOI: 10.1109/IPDPS.2014.128 (cit. on p. 9).

[4] Ying Wang, Jian Dong, Sen Zhang, and Decheng Zuo. «B-SEFI: A Binary Level Soft Error Fault Injection Tool». In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2019, pp. 235–241. DOI: 10.1109/QRS-C.2019.00053 (cit. on p. 9).

[5] Ang Jin, Jianhui Jiang, Jiawei Hu, and Jungang Lou. «A PIN-Based Dynamic Software Fault Injection System». In: Nov. 2008, pp. 2160–2167. DOI: 10.1109/ICYCS.2008.329 (cit. on p. 9).

[6] Deniz Kasap, Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. *Micro-Architectural features as soft-error induced fault executions markers in embedded safety-critical systems: a preliminary study*. 2023. arXiv: 2211.13010 [cs.AR] (cit. on pp. 10, 14, 18).

[7] G. Yalcin, O. S. Unsal, A. Cristal, and M. Valero. «FIMSIM: A fault injection infrastructure for microarchitectural simulators». eng. In: *2011 IEEE 29th International Conference on Computer Design (ICCD)*. IEEE, 2011, pp. 431–432. ISBN: 9781457719530 (cit. on p. 10).

[8] Nathan Binkert et al. «The Gem5 Simulator». In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi.org/10.1145/2024716.2024718 (cit. on pp. 10, 13, 18).

[9] Michał Mosdorf and Janusz Sosnowski. «Fault injection in embedded systems using GNU debugger». In: *Pomiary Automatyka Kontrola* 57 (Jan. 2011), pp. 825–827 (cit. on p. 10).

[10] Felipe Rocha da Rosa, Rafael Garibotti, Luciano Ost, and Ricardo Reis. «Using Machine Learning Techniques to Evaluate Multicore Soft Error Reliability». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.6 (2019), pp. 2151–2164. DOI: 10.1109/TCSI.2019.2906155 (cit. on pp. 12, 15).

[11] Andreas Löfwenmark and Simin Nadjm-Tehrani. «Fault and timing analysis in critical multi-core systems: A survey with an avionics perspective». In: *Journal of Systems Architecture* 87 (2018), pp. 1–11. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2018.04.001. URL: https://www.sciencedirect.com/science/article/pii/S1383762117304903 (cit. on p. 12).

[12] Felipe da Rosa, Vitor Bandeira, Ricardo Reis, and Luciano Ost. «Extensive Evaluation of Programming Models and ISAs Impact on Multicore So Error Reliability». In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465855 (cit. on p. 12).

[13] Athanasios Chatzidimitriou and Dimitris Gizopoulos. «Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy». In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2016, pp. 69–78. DOI: 10.1109/ISPASS.2016.7482075 (cit. on p. 13).

[14] Wilfried Oed. «Cray Y-MP C90: System features and early benchmark results». In: *Parallel Computing* 18.8 (1992), pp. 947–954. ISSN: 0167-8191. DOI: https://doi.org/10.1016/0167-8191(92)90039-A. URL: https://www.sciencedirect.com/science/article/pii/016781919290039A (cit. on p. 15).

[15] G.D. Robinson. «Why 1149.1 (JTAG) really works». In: *Proceedings of ELECTRO '94*. 1994, pp. 749–754. DOI: 10.1109/ELECTR.1994.472649 (cit. on p. 20).

[16] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. «MiBench: A free, commercially representative embedded benchmark suite». In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739 (cit. on p. 25).