# POLITECNICO DI TORINO

Master's Degree in Cybersecurity


Master's Degree Thesis

# Fault Injection Analysis of Automotive Benchmarks using HPC-Based Monitoring


| Supervisors | Candidate |
| --- | --- |
| Prof. Stefano DI CARLO | Amalia Vittoria MONTEMURRO |
| Prof. Alessandro SAVINO | |
| Doctor. Enrico MAGLIANO | |


October 2025

# Abstract

The downscaling of electronic components in terms of voltage and physical dimensions has enabled the development of modern high-performance microprocessors, which are increasingly adopted in safety-critical applications. However, their reduced dimensions have increased the sensitivity to radiation. Radiation-induced soft errors have become a key threat in terms of reliability in safety-critical real-time embedded systems (SACRES). A common technique used to enhance the hardware reliability is N-Modular Redundancy (NMR). With the growing complexity of modern microprocessors, this solution has become unaffordable. The development of lighter alternatives for implementing hardware/software error detection mechanisms has become a crucial area of research.

This thesis investigates the effects of fault injections on automotive benchmarks to assess the reliability of the target setup, a crucial step for studying new, lighter robustness solutions. The selected benchmarks are taken from the EEMBC MultiBench™ Multicore Benchmark Suite to strengthen the representativeness of the experimental environment. The strength of this approach lies in the fact that MultiBench combines a wide variety of application-specific workloads with the EEMBC Multi-Instance Test Harness (MITH), which is compatible and portable across most multicore processors and operating systems. The benchmarks are executed in a real-time scenario under FreeRTOS, exploiting its scheduling algorithm to enable task concurrency, on the Xilinx Pynq-Z2 board. With this setup, benchmarks are scheduled concurrently and executed across multiple runs of the system, resulting in different demos. Faults are injected into both memory and registers via host, and their effects are analyzed by leveraging Hardware Performance Counters (HPCs) to monitor micro-architectural events. To maximize the information gathered, each fault is executed multiple times so that its impact can be observed across different monitored events.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

**a2time**
: Angle to Time Conversion

**AI**
: Artificial Intelligence

**API**
: Application Programming Interface

**ASIL**
: Automotive Safety Integrity Levels

**BSP**
: Board Support Packages

**BG**
: Background

**BRAM**
: Block Random Access Memory

**CAN**
: Controller Area Network

**canrdr01**
: CAN Remote Data Request

**CCR**
: Cycle Count Register

**CIA**

Confidentiality, Integrity and Availability

**CPU**

Central Processing Unit

**CRC**

Cyclic Redundancy Check

**CSMS**

Cybersecurity Management System

**CSV**

Comma-Separated Values

**ECC**

Error Correction Code

**EEMBC**

Embedded Microprocessor Benchmark Consortium

**ELF**

Executable and Linkable Format

**EMFI**

Electromagnetic Fault Injection

**FIB**

Focused Ion Beam

**FG**

Foreground

**FPGA**

Field-Programmable Gate Array

**GPIO**

General-Purpose Input/Output

**HFI**

Hardware Fault Injection

**HDMI**

High-Definition Multimedia Interface

**HPC**

Hardware Performance Counter

**ISR**

Interrupt Service Routine

**ISO**

International Organization for Standardization

**JTAG**

Joint Test Action Group

**LFI**

Laser Fault Injection

**LED**

Light Emitting Diode

**LU**

Lower-Upper

**matrix01**

Matrix Arithmetic

**MCR**

Move to Coprocessor

**MBU**

Multi-Bit Data Upset

**MITH**

Multi-Instance Test Harness

**MRC**

Move from Coprocessor

**NIST**

National Institute of Standards and Technology

**NMR**

N-Modular Redundancy

**OS**

Operating System

**OTA**

Over-The-Air

**PC**

Personal Computer

**PC**

Program Counter

**PCA**

Principal Component Analysis

**PMCR**

Performance Monitor Control Register

**PMCNTENSET**

Performance Monitor Count Enable Set

**PMU**

Perfomance Monitoring Unit

**PMSELR**

Performance Monitors Event Counter Selection Register

**PMXEVCNTR**

Performance Monitors Event Count Register

**PMXEVTYPER**

Performance Monitors Event Type Select Register

**Pmod**

Peripheral Module

**POSIX**

Portable Operating System Interface

**PRNG**

Pseudorandom Number Generator

**PYNQ**

Python on Zynq

**QEMU**

Quick Emulator

**RAM**

Random Access Memory

**RDR**

Remote Data Request

**rspeed01**

Road Speed Calculation

**RTOS**

Real-Time Operating System

**SACRES**

Safety-Critical Real-Time Embedded Systems

**SBU**

Single-Bit Data Upset

**SDK**

Software Development Kit

**SDC**

Silent Data Corruption

**SECDED**

Single Error Correction/Double Error Detection

**SEU**

Single Event Upset

**SFI**

Software Fault Injection

**SMOTE**

Synthetic Minority Over-sampling Technique

**SoC**

System on a Chip

**SRAM**

Static Random-Access Memory

**SUMS**

Software Update Management System

**TARA**

Threat Analysis and Risk Assessment

**TCB**

Task Control Block

**TDC**

Top Dead Center

**TCL**

Tool Command Language

**TMR**

Triple Modular Redundancy

**t-SNE**

t-distributed Stochastic Neighbor Embedding

**UNECE**

United Nations Economic Commission for Europe

**VHDL**

VHSIC Hardware Description Language

**VLSI**

Very-large-scale integration

**V&V**

Verification and Validation

**XSCT**

Xilinx Software Command Line Tool

# Chapter 1

# Introduction

## 1.1 Motivation

In recent years, several international standards for security and safety have been integrated into the traditional lifecycle development process. In this context, the automotive industry is among the sectors that have experienced the most significant impact. Some key standards are outlined below:

- **ISO/SAE 21434:2021 — Road Vehicles: Cybersecurity Engineering** [**1**]: It establishes a framework for cybersecurity management throughout the product lifecycle. It also addresses the problem of standardizing roles and responsibilities during product development and emphasizes the importance of executive management's commitment to cybersecurity by providing specific guidelines. A new introduction of this regulation is the Threat Analysis and Risk Assessment, which provides a systematic approach to assess the cybersecurity risks.

- **UNECE Regulation No. 155 (R155)**[**2**]: This regulation focuses on the vehicle lifecycle, including the supply chain. Suppliers should adopt the *secure by design* approach, including cybersecurity management from the early stages of development. It also requires the implementation of a Cybersecurity Management System to ensure protection against cyber threats.

- **UNECE Regulation No. 156 (R156)**[**3**]: It is complementary to R155, and the purpose of this regulation is to ensure secure software updates throughout

the vehicle's lifetime. It requires the deployment of Software Update Management System which ensures that software updates are delivered securely, authenticated, and traceable. This is particularly challenging but necessary, given the increasing reliance on Over-The-Air updates in modern vehicles.

- **ISO/SAE 24089:2023 — Road Vehicles: Software Update Engineering** [4]: It complements UNECE-R156 by defining processes for planning, developing, verifying, validating and distributing software updates. By applying the standard, manufacturers can minimize risks introduced by frequent software changes.

Current standards and regulations emphasize the need for safety and security in modern vehicles: to address this necessity, Verification and Validation techniques and fault injection methods are widely used [5, 6]. These methods help evaluate safety in critical systems, and systematic test case analysis is essential for identifying and addressing potential failures.

The growing demand for improved reliability analysis and fault detection in the automotive sector drives the research and development presented in this thesis.

## 1.2 Verification and Validation

V&V are two complementary processes used to determine whether a product, service, or system satisfies its requirements and fulfills its intended purpose [7]. A key aspect of this process is that verification and validation should be considered as *independent* activities, meaning that they are ideally performed by an external entity rather than by the development team itself [8, 9]. Although the terms are often used interchangeably in practice, they describe distinct objectives and rely on different methodologies:

- **Verification**: it focuses on checking that the system has been built according to its design specifications. It can be performed at different stages of the development cycle, using procedures appropriate to each phase. Importantly, verification is not limited to the final stages; on the contrary, it should be applied as early as possible. From both a security and an economic perspective, detecting flaws in the early phases is crucial, since correcting them later may become significantly more costly and time-consuming [10].
- **Validation**: it refers to the set of activities used to demonstrate that the system, once implemented, actually meets the needs and expectations of the

end users in its operational environment. It includes the testing and validation of the actual product [10].

As demonstrated by the literature [5, 6], fault injections can be used in the V&V process of embedded automotive software as they enable assessing system reliability, evaluating fault tolerance, and identifying vulnerabilities.

## 1.3 Thesis Overview

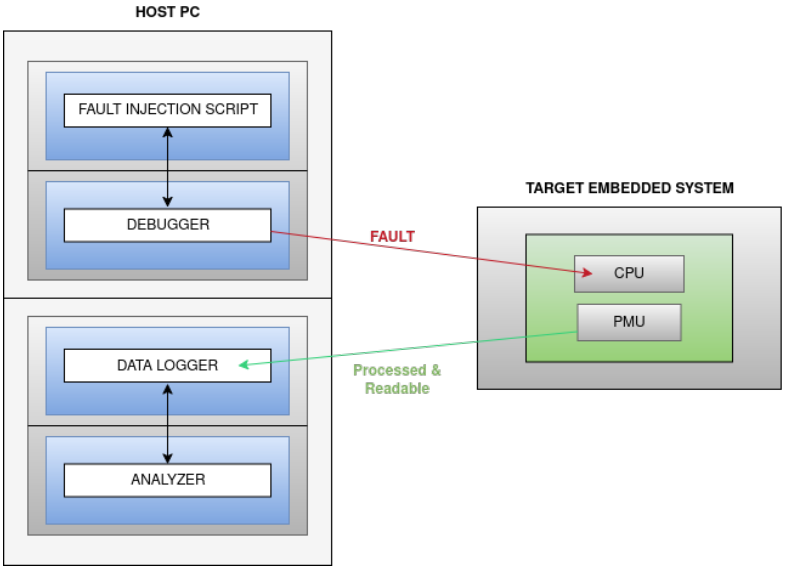


**Figure 1.1:** Experimental setup: fault injection via JTAG and PMU data acquisition on the PYNQ-Z2.

This thesis investigates the effects of fault injections on a target system and provides a comprehensive reliability analysis using automotive benchmarks from the Embedded Microprocessor Benchmark Consortium MultiBench™ Multicore Benchmark Suite[11]. Building on the work of [12, 13], which introduced a fault injection framework for Safety-Critical Real-Time Embedded Systems applications, this thesis enforces the usage of HPC-based monitoring and proposes an environment as close as possible to a real-time scenario. The environment is built by developing test cases in which multiple automotive benchmarks are executed, using specific scheduling algorithms provided by FreeRTOS. This setup is fundamental, since automotive systems perform several tasks in parallel. The correct profiling of

the target system is evaluated using the Hardware Performance Counters special-purpose registers, which track architectural events in the modern microprocessor. These registers are collected from the Perfomance Monitoring Unit using the developed fault injector. The ARM Cortex-A9 PMU provides six counters to gather statistics on the operation of the processor and memory system, and each counter can count any of the 168 events available [14]. This approach is portable and innovative thanks to the fault injector that can collect valuable data to enable better anomaly detection: common examples of trackable events are *cache misses*, *instruction executed* or *branch misprediction*. Although the proposed framework is highly modular and generalizable, the target system tested during the thesis is an Xilinx Pynq-Z2 board, built around the Xilinx Zynq 7020 System on a Chip.

### 1.3.1 The basic workflow

The workflow adopted consists of two different phases:

- The workload is first executed without any fault injection. The results of this fault-free run (called *golden run*) are stored and later used as a reference for comparison with subsequent executions;
- The workload is then re-executed on the benchmark selected as the injection target. Faults are injected through the board's debug interface, and multiple *repetitions* and *iterations* (the difference is explained in chapter 4) are performed in order to collect the architectural events required for the analysis.

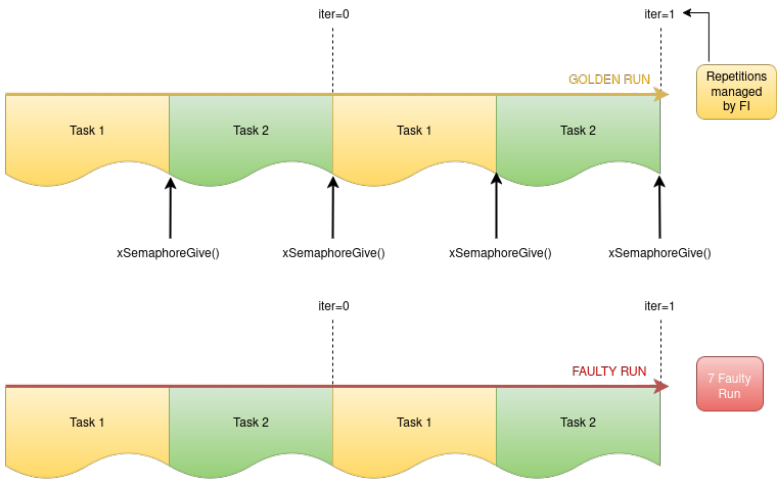


**Figure 1.2:** Basic workflow proposed in the experiments

The setup phase is managed by the fault injector, implemented as a Python script, which allows precise targeting of the injections. In practice, this means that faults can be injected randomly into one of the two benchmarks executed, and their effects can then be evaluated. After executing the campaigns, the collected data are transformed into a format suitable for machine learning, which can aid in training models for fault detection purposes.

## 1.4 Structure of the Thesis

The thesis is organized into the following chapters:

### Chapter 1 — Introduction

This chapter introduces the context of the research, outlines the motivations and objectives, and describes the specific contributions made to the existing framework.

### Chapter 2 — Background

This chapter provides an overview of the theoretical background necessary for this research, including definitions of fault, error, and failure, as well as a taxonomy of fault injection techniques. It also reviews the state of the art, with particular emphasis on the framework proposed in [12, 13], which serves as the foundation for the developments presented in this thesis.

### Chapter 3 — Target Platform

Describes the tools and technologies used during this thesis. It provides architectural information on the target platform and its main components and also addresses the key features of Real-Time Operating System.

### Chapter 4 — Implementation Details

This chapter presents the design and implementation of the fault injection campaigns, including the system architecture, HPC-based monitoring integration, and

key improvements over previous work.

## Chapter 5 — Experimental Setup and Evaluation

This chapter is divided into two parts: the first part presents the setup and results of the fault injection campaigns, while the second part covers the preprocessing steps and data visualization of the resulting dataset.

## Chapter 6 — Conclusions and Future Work

This chapter summarizes the personal contributions to the framework used, discusses the main findings, and outlines possible directions for future research, particularly regarding the extension of fault injection techniques and the application of machine learning for anomaly detection.

# Chapter 2

# Background

This chapter provides an overview of the main concepts required to understand the choices behind the implementation and the terminology used in this thesis. It discusses a structured taxonomy that positions fault injection as a central concept for dependability evaluation and then provides details about the framework developed in [12, 13] that is utilized in this thesis.

## 2.1 Dependability

Dependability [15, 16] can be considered as the system's ability to perform its required function consistently and continuously, and it refers to the operational requirement of a system. Although often associated with modern computing, this is an old concept that first appeared in 1822 in the context of Babbage's Calculating Engine[17], even before the first generation of electronic computers. Some of the definitions of dependability are provided as follows:

- **IFIP WG 10.4 [18]**: "*[..] the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers [..]*";
- **IEC IEV [19]**: "*Dependability (is) the collective term used to describe the availability performance and its influencing factors: reliability performance, maintainability performance, and maintenance support performance*";
- **Laprie [20]**: "*Trustworthiness of a computer system such that reliance can be placed on the service it delivers to the user*".

As can be understood by these definitions, the property of dependability is highly related to different attributes such as **reliability**, **availability**, **safety**, and **security**.

### 2.1.1 A taxonomy of Dependability

A detailed taxonomy of the fundamental concepts of dependability was presented by Avizienis et al. in 2001 [15]. According to the authors, dependability is structured into three main components: **threats**, **means**, and **attributes**.

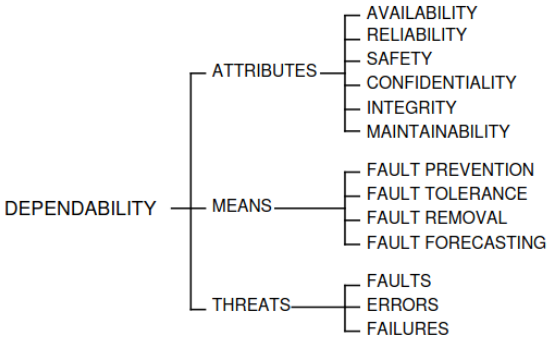


**Figure 2.1:** The dependability tree from [15]

#### Attributes

The dependability tree identifies several key attributes: **availability**, **reliability**, **safety**, **confidentiality**, **integrity**, and **maintainability**. These attributes can be interpreted as the properties that characterize a system from the standpoint of dependability. It is important to note that the Confidentiality, Integrity and Availability triad - although fundamental to information security - is not sufficient by itself to guarantee dependability. Dependability should be understood as a probabilistic and inherently uncertain property, due to the possible presence of faults and their propagation into errors and failures [15].

#### Means

Avizienis et al.[15] also introduce the *means* to achieve dependability, introducing the families of techniques and methods employed to mitigate the impact of faults

on a system[21]:

- **Fault Prevention**: techniques aimed at preventing the occurrence or introduction of faults. These approaches can substantially reduce the likelihood of faults, but often involve significant cost and effort [22]. Examples are rigorous development processes, design reviews, or shielding.

- **Fault Tolerance**: techniques that enable the system to deliver correct service in the presence of faults. This is particularly crucial in SACRES, where service continuity must be preserved despite errors [22]. Examples are redundancy, error detection, and recovery.

- **Fault Removal**: techniques to reduce the *number* or the *severity* of faults, through verification, testing, debugging, and corrective maintenance [22].

- **Fault Forecasting**: techniques to estimate the current number of faults, their future incidence, and their potential consequences [22]. Typical examples are: reliability modeling, statistical testing, and field data analysis.

### Threats

A *threat* can be defined as a potential danger to a computer system that may result in the interception, alteration, obstruction, or destruction of computational resources, as well as any other form of service disruption [23, 24]. When a threat damages a system asset, thereby preventing the system from delivering the expected service, a **failure** occurs. An **error** is defined as the part of the system state that may lead to a subsequent failure, whereas a **fault** constitutes the underlying cause of the error. A fault may be *active* when it actually produces an error, or *dormant* when it remains latent without immediate effect [20, 15, 25].

Because dependability reflects a computer system's ability to deliver a justifiably trusted service, it is essential to analyze both the causes of faults and their potential consequences to detect and mitigate anomalies. Furthermore, the concept of fault is central in legislative and regulatory contexts. International standards, such as **ISO 26262** [26], explicitly define frameworks and requirements to mitigate the presence and impact of faults in safety-critical systems.

## 2.2 Faults in ISO 26262

Although first published in 2011 and revised in 2018, ISO 26262 remains particularly significant, as it requires organizations to provide *sufficient evidence* of compliance with safety requirements, and fault injection techniques can be used effectively to offer such evidence. According to *ISO document 10303-226* [27], a fault is defined as an abnormal condition or defect at the component, equipment, or subsystem level which may lead to a failure. A fault is active when it causes an *error*; otherwise, it is *dormant*. In this definition, there are some key concepts linked to faults:

- Failure: the event in which the service delivered by the system deviates from the expected correct one;

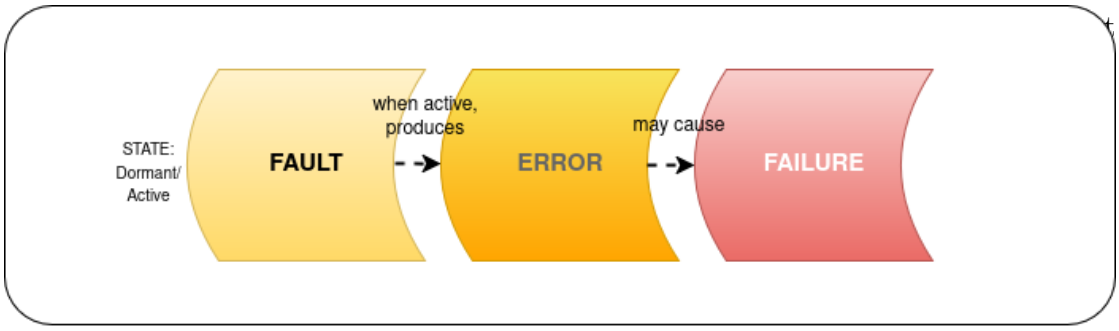


**Figure 2.2:** Relationship between Fault, Error, and Failure

A fault occurring on a target device does not always result in an error; often, faults remain inactive and have no observable effects. When a fault causes a temporary error, it is classified as a *soft error*.

### 2.2.1 Soft Errors

Soft errors are transient faults that occur in electronic devices due to external or internal factors, causing temporary changes in the device's state or behavior. They can reverse or flip the data state of a memory cell, register, latch, or flip-flop; however, these disturbances do not permanently damage the circuit, which can still correctly store data once new values are written [28, 29].

Despite the transient nature of the fault, *soft errors* are becoming a concern in modern microprocessors, due to the increased sensitivity of smaller transistor

geometries and lower operating voltages. Under these conditions, circuits become more vulnerable to perturbations such as cosmic radiation, electrical noise and interference, as well as variability or defects introduced during manufacturing [30].

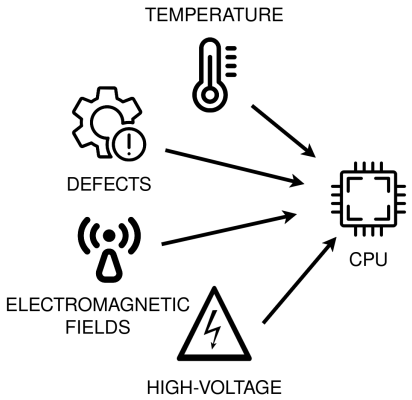


**Figure 2.3:** Causes of soft errors in modern microprocessors

Soft errors include Single Event Upsets, which can be further categorized into Single-Bit and Multi-Bit Data Upsets.

### Single-Bit Data Upset

A Single-Bit Data Upset occurs when a high-energy particle, such as a cosmic ray or a neutron, strikes a sensitive region of a semiconductor device, altering the charge stored in a memory cell or flip-flop. The event changes the logical state of the storage element, from '0' to '1' or vice versa. In sequential logic, such as flip-flops, the upset manifests itself as a single erroneous bit, which is usually overwritten during the next valid clock cycle. While in many non-critical applications the effect of an SBU may be negligible, in safety-critical or mission-critical systems even a single incorrect bit can compromise correctness [31].

In Field-Programmable Gate Array, SBU may occur both in user data and in the (typically SRAM-based) configuration memory. In the latter case, a flipped configuration bit can alter the programmed functionality, leading to persistent misbehavior or the propagation of corrupted data. Mitigation commonly combines *configuration memory scrubbing*[32] (periodic readback and correction) with Error Correction Code [31, 33, 34].

#### Multi-Bit Data Upsets

Multi-Bit Data Upset arises when a single high-energy particle simultaneously affects multiple adjacent storage elements, causing two or more bits within a memory word or logic element to flip [35]. MBUs are particularly critical in memories where physically adjacent cells map to the same logical word, since they can produce errors that exceed the correction capability of standard ECC, resulting in uncorrectable errors. The spatial correlation of MBUs is highly dependent on the device layout: for instance, in SRAM memories, if the cells of a logical word are placed close together, the likelihood of uncorrectable MBUs increases. Conversely, modern FPGA architectures often physically separate the SRAM cells belonging to the same logical word, thereby reducing the probability that MBUs will corrupt multiple bits in a way that cannot be mitigated[36, 31].

Mitigation strategies for MBUs include the use of interleaved memory architectures (to physically separate adjacent bits of the same logical word physically), more advanced ECC schemes capable of detecting and correcting multiple-bit errors, as well as architectural hardening techniques at the FPGA design level[31, 36, 37].

## 2.3 Mitigation Techniques

### 2.3.1 Classical Approaches: TMR and ECC

The primary mitigation techniques discussed in the literature include the following:

- Triple Modular Redundancy (TMR);
- Error Correction Code (ECC).

TMR is a fault-tolerant approach based on **N-Modular Redundancy**, in which three identical logic circuits are used to compute a specified Boolean function. Each circuit receives the same input data, and their outputs are compared through a majority-voting mechanism. In this way, even if one of the three circuits produces an incorrect result, the other two can correct it, ensuring that the fault is effectively masked. TMR technique is particularly effective in protecting against errors and introduces a small delay in the circuit. However, its main drawback is that it requires triplicate logic and additional circuitry, which results in more than three

times the original area [38].

ECC represents a set of techniques for detecting and correcting errors: the detection is enabled by adding redundant parity bits to the data. One of the most well-known examples is the family of Hamming Codes, which belongs to the linear error-correcting codes. The standard Hamming code can only detect and correct a single-bit error. Still, there are more powerful versions in which additional parity bits are added, a technique known as *Single Error Correction/Double Error Detection* (SECDED). ECC RAM commonly uses Hamming codes with SECDED to automatically correct single-bit errors and raise an alert when double-bit errors occur [39, 40].

### 2.3.2 AI-Based Detection

More recently, AI-based approaches have been explored to enhance fault detection and prediction. Machine learning models trained on historical data have shown promising results in identifying patterns associated with potential failures, thereby improving the reliability and resilience of SACRES [41, 42].

## 2.4 Fault Injections Techniques

Different fault injection techniques have been introduced in the literature. The following sections analyze the most common approaches [43].

### 2.4.1 Hardware Fault Injection

Hardware Fault Injection involves techniques that deliberately introduce faults into the physical components of an electronic system to study their effects on system behavior. Several surveys and experimental studies [44, 45] group HFI methods into two broad categories:

- **Contact (direct) HFI**: the injector physically interfaces with the target to perturb its electrical conditions (e.g., via pin-level probes or by inserting the device into a manipulable socket); such methods enable relatively precise timing control but often require physical access to the target.

- **Non-contact (indirect) HFI**: the injector acts from outside the package using external stimuli—such as heavy-ion radiation or electromagnetic fields—to provoke faults without direct electrical contact.

Another common classification in the literature distinguishes techniques by their invasiveness, defined as the degree of physical alteration or risk of damage to the target: non-invasive, semi-invasive, and invasive. The following list summarizes the most common HFI techniques, with all information sourced from [46]:

- **Clock glitching (non-invasive)**: the system clock is disturbed to create timing violations that may cause registers to capture incorrect values. Effects are typically non-deterministic, making it challenging to target precise internal locations.
- **Voltage glitching (non-invasive)**: the supply voltage is momentarily perturbed (spikes or drops), altering logic timing and potentially inducing incorrect computations. As with clock glitching, reproducibility and spatial targeting are limited.
- **Electromagnetic Fault Injection (non-invasive)**: focused electromagnetic pulses are applied externally to induce currents or voltage transients in on-chip circuits. EMFI can produce effects similar to voltage glitches, but, due to field dispersion, precise spatial targeting is challenging.
- **Laser Fault Injection (semi-invasive)**: a laser beam is employed to inject localized energy into the die (after decapsulation or package thinning), producing controlled bit flips or timing faults. LFI enables finer spatial targeting compared to non-invasive techniques while remaining less destructive than invasive editing.
- **Focused Ion Beam editing (invasive)**: semiconductor-editing equipment is used to physically modify the chip (e.g., cut/add interconnects) to create permanent alterations or to expose internal nodes for probing. FIB is highly targeted and powerful, but destructive and complex to perform.

Each technique represents a trade-off between cost, potential damage to the target system and the reliability of the obtained results. Non-invasive methods are generally easier to deploy and less destructive, but they provide limited spatial precision. Semi-invasive and invasive techniques, on the other hand, enable higher targeting accuracy at the expense of greater complexity, higher costs, and an increased risk of permanent modification.

Overall, HFI techniques yield highly reliable results because tests are conducted directly on the physical device under realistic operating conditions. This makes them particularly suitable for diagnostic purposes and for validating detection mechanisms. However, hardware-based fault injection is costly, requires specialized expertise, and carries the risk of permanently damaging the device under test. Additionally, many techniques have limited portability across different architectures, restricting their applicability in heterogeneous embedded environments.

### 2.4.2 Software Fault Injection

SFI, on the other hand, introduces faults into a running software system. This technique is highly flexible: faults can be injected at multiple levels, including memory, registers, and even the operating system, which is otherwise difficult to analyze [43]. Although not considered intrusive, in real-time systems, where precise timing is essential for correct operation, software-level injections may disrupt normal system activity [47].

SFI techniques can be categorized by considering the trigger mechanism that produces the artificially generated fault or error, or the injection times. In this way, it is possible to distinguish according to [45, 48]:

- **Injection Strategy**
  - **Time-based:** Faults are injected at predetermined time intervals. This technique is easy to implement non-intrusively, but limited in flexibility.
  - **Location-based:** Faulty values are written into predefined memory locations. Suitable for memory corruption studies, but prevents dynamic control of the fault load.
  - **Execution-driven:** Faults are triggered dynamically depending on the control flow. This approach allows realistic and complex fault models but does not apply to black-box applications.
- **Injection Timing**
  - **Before runtime:** The program is modified before execution (e.g., source code mutation to insert software bugs).
  - **During runtime:** Faults are injected while the program is executing.
  - **At loading time:** Faults are introduced when external components are loaded, for example, via dynamic library binding or dependency injection.

The benefits of using software fault injection are numerous. They can be categorized as a flexible approach that enables extensive experimentation without the need for specialized hardware, eliminating the risk of damaging physical devices. Furthermore, it is cost-effective and can be easily extended to support new types of faults. The main drawbacks of software fault injection complement the strengths of hardware-based methods. Specifically, software-based approaches often offer limited observability and coverage, and it is more challenging to induce permanent faults.

#### Tools

In the literature, there are different tools that can be used to inject faults at the software level:

- **BOND** [49], a software-based fault injection system for COTS applications;
- **Xception** [50], a software-implemented fault injection tool for dependability analysis;
- **MAFALDA** [51], a fault injection environment for real-time COTS microkernel based systems ;
- **DOCTOR** [52], an integrated software fault injection environment .

Among the tools presented, *Xception* is the most interesting one for the thesis, because it exploits the debug unit of modern processors to inject faults. In the literature, there are few examples of SFI based on the debug unit, and a more precise approach will be presented in the next sections.

### 2.4.3 Other Categorization

The distinction between hardware-based and software-based fault injection alone is not sufficient. Fault injection techniques can also be categorized by the development phase in which they are applied [46]:

- **Simulation-Based Fault Injection**: faults are injected into high-level system models (often described in VHDL). This approach is particularly powerful

because it enables dependability evaluation in the early stages of development, as soon as a model is available.

- **Emulation-Based Fault Injection**: faults are introduced into an emulated environment rather than into real hardware. This approach relies on simulators such as QEMU or on FPGA-based platforms, and can reproduce scenarios like bit flips in memory or registers. It allows the analysis of system behavior under fault conditions without requiring a physical prototype.
- **Hybrid Fault Injection**: combines hardware and software-based techniques to leverage the advantages of both approaches.

### 2.4.4 Classification of Faults

The observable effects of a fault can vary significantly and are typically classified as follows [53]:

- **Crash**: the program terminates abnormally;
- **Hangs**: the program enters in an unresponsive state;
- **Reboots**: the operating system reboots;
- **Silent Data Corruption** (SDC): the program continues to run, but the result of the computation is incorrect;
- **Benign**: the program outputs the correct result, and the fault does not affect the outcome.

Among these outcomes, detecting SDCs is particularly challenging: for this reason, different detection mechanisms have been developed and analyzed in the literature.

## 2.5 Related Works

Dependability evaluation constitutes a crucial step in the design of SACRES, as widely demonstrated in the scientific literature. Among the various approaches proposed in recent years, the framework presented in [12] is particularly relevant and forms the basis of the evaluation conducted in this thesis. The authors introduce a fault injection framework specifically designed for the reliability assessment of

SACRES. This framework aims to reproduce soft errors induced by environmental disturbances and analyze their effects through machine learning algorithms to enable the detection of malfunctions in the system's architecture.

### 2.5.1 High-Level Details

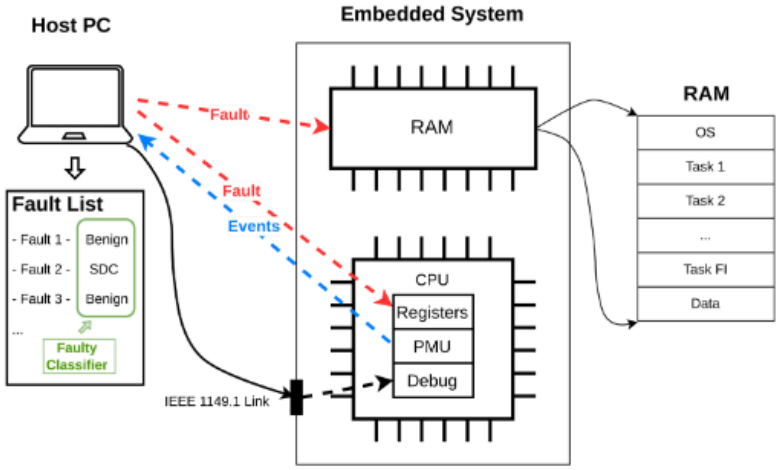


**Figure 2.4:** High-Level architecture of the fault injection framework presented in [12]

Figure 2.4 illustrates the high-level architecture of the proposed framework, which consists of two primary components: a Host PC and the embedded system under test. The framework employs a SFI mechanism based on the microprocessor's debug unit. While this method has been infrequently reported in the literature, its main advantage is the ability to inject faults at multiple levels, such as CPU registers, memory, and the operating system, with minimal overhead since execution must be halted for fault injection. In this context, the debug unit was the only viable option. In contrast to alternative techniques, this approach does not require code recompilation between executions following fault injection, as the code remains unchanged. Since the primary objective is to generate a dataset for machine learning-based assessment, this method is the most practical solution. Additionally, this approach guarantees experiment repeatability, which is critical for collecting architectural event data during execution (this requirement will be addressed in a subsequent section). The CPU debug unit operates in external mode using the JTAG protocol, which enables control of the debug phase of the target system from an external host and permits modification of CPU registers and memory.

The framework extends the traditional SBU approach introduced in [13] to support MBU. A key feature is the integration of the processor's PMU, which enables the collection of HPCs during execution. This capability allows the framework to combine functional correctness checks with a detailed analysis of micro-architectural events recorded at runtime.

The Host PC coordinates the fault injection campaigns through a Python-based script that interacts with the Xilinx Software Command Line Tool. On the target side, a Xilinx Zynq SoC running FreeRTOS executes the workload compiled into a ELF binary. The faults are injected either by selecting faults from a predefined list of location–time tuples (e.g., provided in a CSV file) or by generating them randomly at runtime. To ensure reproducible results, the random seeds used to initiate execution, as well as the golden outputs obtained in the fault-free runs are stored in the FPGA memory. Fault injection outcomes are classified according to the standard dependability taxonomies into Benign, SDCs, or crash/hangs.

### 2.5.2 Experimental Setup and Workflow

The experimental evaluation was carried out on a Pynq-Z2 board featuring a dual-core ARM Cortex-A9 and running FreeRTOS v.10. Benchmarks were selected from the MiBench suite, chosen for their diversity across application domains such as **Automotive**, **Networking**, **Security**, **Office**, **Telecommunications**, and **Industrial Control**. The subset considered includes:

- **SHA**, a hashing algorithm producing a 160-bit digest, widely used in cryptographic applications;
- **Dijkstra**, an implementation of the shortest-path algorithm on graphs;
- **Quicksort**, a sorting algorithm applied to arrays of strings;
- **Rijndael**, better known as AES, a block cipher for secure data encryption;
- **BasicMath**, which performs arithmetic operations often lacking hardware acceleration in embedded CPUs;
- **Stringsearch**, which performs pattern matching within a text.

These benchmarks were evaluated both individually and within multitask workloads managed by the FreeRTOS scheduler, resulting in 23 injection campaigns that covered injections into memory, CPU registers, and Program Counter.

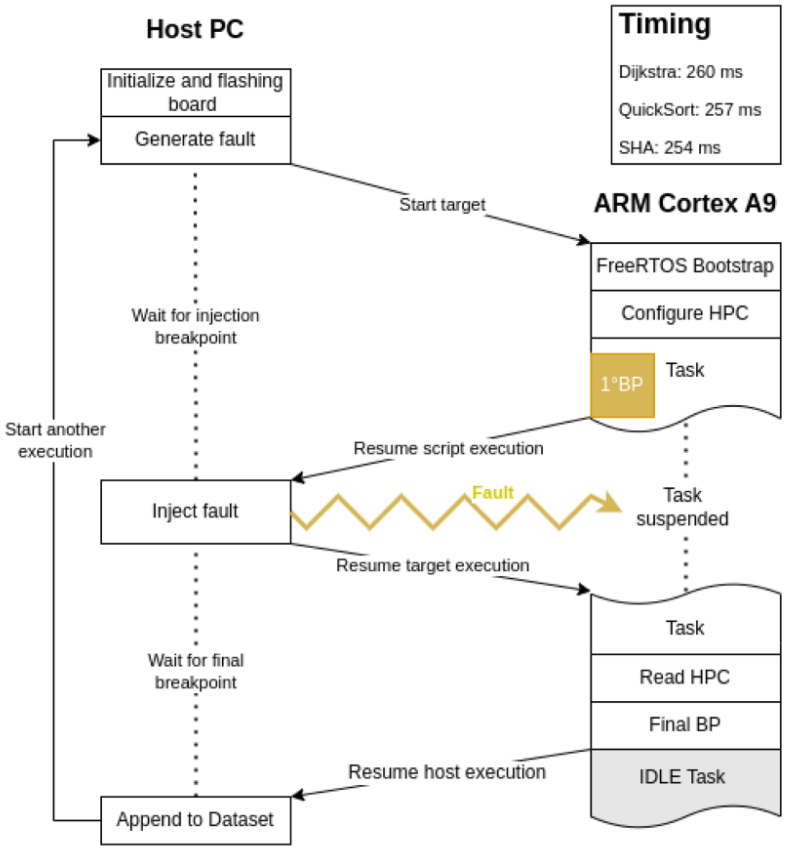


**Figure 2.5:** Software execution flow taken from [12]

The workflow, illustrated in Figure 2.5, is structured in two main phases. First, during the *golden run*, the host boots the target, supplies random input data to ensure *reproducibility*, and collects the *golden output*, used later for comparison. In the *faulty run*, the host activates an injection loop: for each iteration, it specifies the events to monitor, sets the injection breakpoint, and resumes execution. When the breakpoint is reached, faults are injected and execution continues until the end. The final output is then compared with the golden reference. If outputs coincide, the run is classified as *Benign*; if they differ, it is classified as *SDC*; and if execution fails to terminate within the timeout, it is labeled as *Crash/Hang*. The timeout is carefully calibrated to account for both OS boot and benchmark execution time, with additional margin to distinguish between minor timing deviations and genuine failures.

### 2.5.3 Results

The main findings of the experimental campaigns can be summarized as follows:

- Faults injected into memory are mostly benign, owing to the large address space that reduces the likelihood of affecting critical regions;
- CPU register injections exhibit a higher susceptibility to *SDCs*, particularly in cryptographic workloads where intermediate values are more sensitive;
- PC injections are especially critical, frequently resulting in system crashes or significantly increasing the SDC rate.

In addition, HPC profiling demonstrated its effectiveness in distinguishing benign executions from faulty ones, thus confirming the potential of performance counters as indicators for the development of fault detection mechanisms in SACRES.

# Chapter 3

# Target Platform

This chapter introduces the platform and tools used to conduct this thesis. It opens with a description of the target system employed in the experimental work and highlights the principal features of FreeRTOS, the Real-Time Operating System under which the benchmarks are executed.

## 3.1 Target Hardware Platform

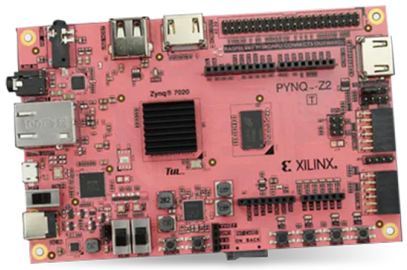


**Figure 3.1:** Pynq Z2 board [54]

The target hardware platform used for the experimental setup is a Pynq-Z2[54] 3.1, which is a FPGA development board, based on ZYNQ XC7Z020. It supports Python on Zynq (PYNQ, an open-source framework from Xilinx that enables the customization of Xilinx ZYNQ SoCs without fully rely on the design programming

logic circuit. PYNQ-Z2 board integrates Ethernet, HDMI Input/Output, MIC Input, Audio Output, Arduino interface, Raspberry Pi interface, 2 Pmod, user LED, push-button and switch. It is designed to be easily extensible with Pmod, Arduino, and peripherals, as well as general purpose General-Purpose Input/Output pins.

### 3.1.1 The Cortex-A9 processor

The processor exhibited by the board is an ARM Cortex-A9[55, 56]: it belongs to the Cortex-A series, which can support complex OS and multiple user applications, available in different smartphones and digital devices. It is effective and powerful, suitable for a large number of applications and constitutes the perfect choice for applications where energy efficiency is necessary.

ARM Cortex-A9 is a 32 bit multi-core processor, based on ARMv7 architecture and on the implementation of *Thumb-2* instruction set, which reduces the size of programs with little impact on performance. Moreover, it presents the following features:

- High-Efficiency Superscalar Pipeline;
- NEON Media Processing Engine;
- Optimized Level 1 Caches;
- Thumb-2 Technology;
- Floating-point unit.

Among these interesting characteristics, the PMU available is the most relevant one. In general, PMU is commonly used for performance analysis and debugging, providing a wide range of *events* that can be monitored for performance profiling. In general, most of the events provided are common, however, each processor can define its own specific events.

In the case of Cortex A-9, as already reported in the introductory part, according to the Technical Reference Manual [55], PMU provides six counters to gather statistics on the operation of the processor and the memory system. Counters can be accessed from the internal CP15 interface as well as from the DAP interface. A clear overview of the registers are reported in the table below:

**Table 3.1:** Cortex-A9 PMU Registers

| Name | Type | Description |
|---|---|---|
| PMXEVCNTR0–5 | RW | Event Counter Register |
| PMCCNTR | RW | Cycle Count Register |
| 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 |

Among them, the most interesting ones are the following:

- **Performance Monitor Control Register** (PMCR): which enables/disables the PMU, controls reset of counters and sets some global modes;
- **Event Counter** (PMC): a set of 32-bit counters that can be programmed to track specific event, typically we have 6 events counters and 1 cycle counter;
- **Cycle Counter** (CCNT): a dedicated counter for counting CPU clock cycles;
- **Performance Monitors Event Type Select Register** (PMXEVTYPERn): each counter has a register that defines which event it should monitor.
- **PMU Interrupts**: Counters can trigger an interrupt on overflow.

Event Selection Registers are very useful because they enable the selection of the Performance Monitor Count Register to Count. They can be accessed in this way:

```
MRC p15, 0,<Rd>, c9, c13, 1 ; Read PMXEVTYPER Register
MCR p15, 0,<Rd>, c9, c13, 1 ; Write PMXEVTYPER Register
```

More details will be provided in the later chapters, for managing and configuring the PMU.

## 3.2 Real-Time Operating System

RTOS is an operating system specifically designed to support real-time computing applications, where tasks must meet strict deadlines. Due to its characteristics, an RTOS is commonly used in embedded systems and automotive context where being *event-driven* and *preemptive* is important, meaning that the scheduler can assign different priorities to tasks and preempt lower-priority tasks in favor of higher-priority ones [57, 58, 59].

For the proposed framework, the choice fell on RTOS according to several factors:

- **Faithfulness to the automotive context**: in this scenario, RTOS are largely used and in most of the cases they represent the only choice.
- **Reproducibility for profiling**: an RTOS is easy and predictable, so that reproducibility can be ensured to reply the experiments and collect as much information as possible for a correct profiling.

### 3.2.1 Tasks

Real-time applications are typically structured into *periodic tasks*, which can be defined as a sequence of instructions designed to perform specific functions. Tasks are managed and scheduled by the operating system and, depending on the criticality of the functions they implement, can be assigned different priorities: tasks with higher priority are executed before those with lower priority. In general, each task comprises an initialization phase followed by an *infinite loop*, within which the required functions are repeatedly executed.

When an operating system is capable of managing and executing multiple tasks concurrently, this capability is referred to as multitasking. In the case of a RTOS, true parallel execution of tasks on a single-core processor is not possible. Instead, the RTOS employs scheduling algorithms to allocate CPU time among the available tasks. A **schedule** specifies the strategy adopted by the operating system to assign resources and determine the execution order of tasks. Each task can exist in one of four states: **ready**, **running**, **suspended**, or **blocked**. The scheduler within the RTOS manages the transitions between these states according to the adopted scheduling policy.

Two main types of schedulers can be distinguished in RTOS: **preemptive** and **cooperative**. A preemptive scheduler allows a task to be interrupted during its execution if a task with higher priority becomes ready, ensuring that the most critical task is executed immediately. Conversely, a cooperative scheduler does not permit preemption: even if a higher-priority task is ready, it must wait until the currently running lower-priority task has completed its execution before being scheduled.

#### Scheduling Algorithms

As previously introduced, a **scheduling algorithm** is defined as a set of rules that determine, at any point in time, the order in which tasks are executed. One of the simplest approaches is *static scheduling*, where decisions are based on fixed parameters assigned to tasks prior to their activation. Other relevant examples include:

- **Timeline scheduling**: tasks are organized within a predefined timeframe, called the *major cycle*. Each task is executed during its dedicated *minor cycle*. This is one of the simplest and most deterministic scheduling strategies.
- **Round-Robin scheduling**: ensures fairness by allocating fixed time slices to tasks belonging to the same priority level. Once a task exhausts its time slice, the scheduler moves to the next task in the queue.
- **Rate-Monotonic Scheduling**: a fixed-priority scheduling algorithm in which tasks are assigned priorities based on their periods: the shorter the period, the higher the priority. It is optimal among static-priority scheduling policies.
- **Earliest Deadline First**: a dynamic-priority scheduling algorithm in which tasks are scheduled according to their absolute deadlines. This approach is particularly suitable for real-time systems, where meeting strict timing constraints is critical.

### 3.2.2 FreeRTOS

FreeRTOS is a widely adopted open-source OS designed for embedded systems [60]. It is particularly suitable for microcontrollers and applications characterized by limited memory resources and constrained computational capabilities. A basic FreeRTOS application is typically structured around tasks, which represent

independent units of execution. Two types of tasks can be distinguished: the *standard tasks*, defined by the user to perform specific application functions, and the *idle task*, which is automatically created by the kernel and executed whenever no other tasks are ready to run.

In the case of standard tasks, the user must specify several parameters at creation time, including a unique name for identification, the task function to be executed, its priority, and the stack size. Task priority is expressed as a positive integer value, ranging from 0 (the lowest priority, typically reserved for the idle task) to `configMAX_PRIORITIES - 1`. The constant `configMAX_PRIORITIES` is defined in the configuration file `FreeRTOSConfig.h` and determines the maximum number of priority levels available within the system.

The scheduler of FreeRTOS supports different policies for task execution. By default, the scheduler operates in a preemptive mode with time slicing, but there is also the possibility to disable timeslicing or configure a cooperative scheduling approach. Regardless of the chosen policy, when two or more tasks share the same priority level, the *Round-Robin* mechanism is applied to determine their execution order.

```
1 /* FreeRTOSConfig.h */
2 #define configUSE_PREEMPTION    1     // preemption enabled
3 #define configUSE_TIME_SLICING  1     // time-slicing enabled
```

### 3.2.3 FreeRTOS Fundamentals

FreeRTOS provides a set of primitives for synchronization and communication between tasks. These mechanisms are essential to coordinate concurrent activities, manage access to shared resources, and synchronize tasks with interrupts.

#### Binary Semaphores and Mutexes

A binary semaphore is a synchronization primitive that can assume only two values, typically 0 or 1. It is often used to signal events, for example from an Interrupt Service Routine to a task. In FreeRTOS, when multiple tasks attempt to take the same semaphore, the scheduler ensures that the highest-priority task succeeds first. The API also allows specifying a *block time*, which is the maximum number of

system ticks a task will remain in the *Blocked* state while waiting to acquire the semaphore.

A mutex (mutual exclusion object) is similar to a binary semaphore but includes a *priority inheritance* mechanism. This feature helps mitigate the problem of *priority inversion*, ensuring that lower-priority tasks holding a mutex do not unduly delay higher-priority tasks.

### Task Notifications

In addition to semaphores, FreeRTOS provides lightweight synchronization through *task notifications*. A task notification can be considered as a binary or counting semaphore built directly into each TCB. This approach reduces RAM consumption and improves performance. Functions such as `ulTaskNotifyTake()` or `ulTaskNotifyTakeIndexed()` are used in place of `xSemaphoreTake()`, with the parameter `xClearOnExit` set to `pdTRUE` to emulate the binary semaphore behavior.

### The RTOS Tick

When a block time is specified (for example, when a task is put to sleep), FreeRTOS measures the waiting period using the system *tick*. A timer interrupt, known as the *RTOS tick interrupt*, periodically increments the kernel tick count. This allows the kernel to measure time with a resolution determined by the tick frequency. Each time the tick count is incremented, the scheduler verifies whether it is time to unblock any tasks whose delay periods have expired.

### The `tasks.c` File

The file `tasks.c` is the core of the FreeRTOS kernel. It implements task management and scheduling, including the creation and deletion of tasks, context switching, and state transitions. Moreover, it provides the implementation of TCB and defines the mechanisms for handling priorities and scheduling policies.

# Chapter 4

# Implementation Details

After reviewing the literature on fault injection techniques, this chapter describes the implementation of the proposed framework. It first presents the benchmarks used in the single-workload campaigns and then proceeds describing the multi-workload campaign.

## 4.1 AutoBench - Automotive Industrial Benchmarking

The benchmarks selected to carry out the experiments and ensure an environment as close as possible to the real one are provided by the *Embedded Microprocessor Benchmark Consortium* [61]. AutoBench is suite of benchmarks created to evaluate the performance of microprocessors and microcontrollers in automotive, industrial and general-purpose applications. It consists of sixteen benchmarks, including **basic automotive algorithms** and **signal processing algorithms**: a brief description of the algorithm is proposed in appendix.

The choice fell on AutoBench because it is one of the most adopted automotive benchmark suites, used in both academia and industry. It integrates a set of automotive workloads with the EEMBC Multi-Instance Test Harness, a framework designed to ensure portability across a wide range of multicore processors and operating systems. MITH employs a POSIX-compliant, thread-based API to provide a common programming model: benchmarks interact with the harness through an abstraction layer, which offers a flexible interface for testing heterogeneous,

thread-enabled workloads.

Each benchmark is composed of three main elements: **workload**, **dataset**, and a **Cyclic Redundancy Check**. The workload corresponds to the algorithm under test, which may be implemented as single-threaded or multi-threaded. The dataset consists of the input data supplied to the workload, typically available in two different sizes (4MB and 4KB). Finally, the CRC is used to verify the correctness of the computation by comparing the produced output against a reference signature.

Among the sixteen algorithms available in the EEMBC AutoBench suite, the following benchmarks were selected to be included in the framework: *Angle to Time Conversion*, *CAN Remote Data Request*, *Matrix Arithmetic*, and *Road Speed Calculation*. These benchmarks were chosen because, on one side, they reflect typical computational patterns found in automotive applications and provide sufficiently complex scenarios to emulate realistic operating conditions, including communication between interfaces. On the other, they are diverse enough to enable a comprehensive analysis of different system aspects, ranging from arithmetic-intensive workloads to communication and control-oriented tasks. Although all four benchmarks were integrated, the experiments carried out focused exclusively on *a2time* and *rspeed01*.

The selected benchmarks were integrated in two configurations: single and multi-workload setup. The single-workload campaign refers to the execution of a single workload (in our case either a2time or rspeed01); while multi-workload campaign refers to the execution of multiple workloads (both a2time and rspeed01), leveraging the FreeRTOS scheduler. For the single-workload, the existing framework was just extended to include new test cases and to enable the fault injector to target the newly added tasks. For the multi-workload configuration, it was necessary to create a scheduling infrastructure that manages the chosen benchmarks concurrently under FreeRTOS, while ensuring *reproducibility* and *randomness* typical of the existing framework. In the experiments, we considered two concurrent tasks and injected faults into one of them, but the infrastructure can easily be extended to include more than two tasks.

### 4.1.1 Benchmark Structures

Each AutoBench benchmark follows a standardized execution flow, structured around a set of core functions. This standardization is particularly convenient because it allows to understand the workflow of data around the execution and the

basic operation performed by the algorithm. The set of core functions are provided below: the * can be substitute with the name of the benchmark that is executed.

- **define_params_***: configures the general parameters of the benchmark, such as the dataset to be used and the reference CRC associated with that dataset. The input file containing the dataset is read and stored into RAM. This function is called once before any execution, regardless of the number of runs performed.

- **bmark_init_***: initializes the parameters for the current run, allocates the memory required for the output file, and resets all counters. This function must be called at the beginning of each new iteration.

- **t_run_test_***: implements the core benchmark algorithm. It executes the workload in a loop, repeating the computations across multiple iterations without reinitializing the parameters. Results are overwritten in the output buffer at each iteration.

- **bmark_verify_***: computes the CRC of the output data on the fly and compares it with the reference CRC provided during parameter definition. The function returns a boolean value indicating whether the results are correct.

- **bmark_fini_***: finalizes the benchmark run by releasing the memory allocated for the output file, effectively performing the inverse operation of bmark_init_*.

- **bmark_clean_***: completely clears the memory and resets the variables used by the benchmark, performing the inverse operation of define_params_*.

To satisfy the requirements needed to carry out the experiments, some of the functions were slightly modified, but they do not change the overall structure.

### 4.1.2 Benchmark Descriptions

The description of the operation performed by the algorithms integrated in the framework are reported as follows. Details are directly taken from the EEMBC AutoBench Data Book [62].

### Angle to Time Conversion

This benchmark models an embedded automotive application in which the processor processes input from a toothed wheel mounted on the engine crankshaft. Each tooth produces a pulse, and the time interval between consecutive pulses provides a measure of the crankshaft's angular velocity (engine speed). From this information, the benchmark estimates the crankshaft position relative to the *Top Dead Center* reference, computes the instantaneous engine speed, and converts the tooth pulses into an accurate crankshaft angle expressed in linear time from TDC.

At the start of each iteration, the kernel reads a counter value from the input dataset and subtracts the previous value to obtain the elapsed time between two consecutive teeth. As long as TDC is not detected, the tooth counter is incremented to track progress through the revolution. When the counter matches the firing angle associated with a cylinder, the benchmark simulates an ignition event by generating the corresponding firing time. Upon detecting the next TDC, the tooth counter is reset and the cycle repeats.

The a2time benchmark can be classified as **CPU-centric**. Its workload is dominated by arithmetic and logical operations rather than memory accesses. The algorithm performs continuous numerical conversions from crankshaft angle to linear time, involving subtraction, multiplication, and division on a small set of variables.

### CAN Remote Data Request

This benchmark emulates an automotive communication workload in which nodes connected through a Controller Area Network exchange messages. The scenario modeled is the reception of a Remote Data Request message, which is broadcast to all nodes. Each node inspects the message identifier to determine whether it is responsible for the requested data. If so, the node retrieves the appropriate information and places it back on the network to be received by the requester.

At the kernel level, the benchmark processes messages from a simulated receive buffer by checking their identifier fields. Messages that are not relevant are discarded, while relevant ones are either stored locally or, in the case of an RDR message, used to fetch the corresponding data. The retrieved data is then written into a simulated transmit buffer, ready to be sent back across the network to the original requester.

### Matrix Arithmetic

This benchmark represents an automotive and industrial workload characterized by intensive matrix arithmetic. The kernel operates on $n \times n$ input matrices, performing an Lower-Upper decomposition as its primary computation. In addition, it evaluates the determinant of the input matrix and computes a cross product with a second matrix, thereby stressing the system with a mix of linear algebra operations: these simple calculations are nowadays the basic foundations of *AI*.

### Road Speed Calculation

This benchmark models an automotive workload in which the processor periodically computes vehicle speed from differences between timer counter values. To mitigate noise, the raw values are filtered, and the algorithm must also handle corner cases such as counter rollover, abrupt variations in measurements, or the absence of increments when the speed is zero (to avoid indefinite waiting).

The benchmark combines arithmetic operations with control-flow routines. The arithmetic part relies on basic operations (addition, subtraction, multiplication, and division), which may represent a performance bottleneck on low-end microcontrollers. Conversely, in more advanced processors, efficiency is influenced not only by raw arithmetic throughput but also by pipeline behavior, since a considerable number of compare and branch instructions are executed. Processors that balance both aspects tend to achieve the best performance on this workload.

## 4.2 Integration into the single workload campaigns

To correctly integrate the benchmarks into the proposed framework, both *randomness* and *reproducibility* must be ensured. Randomness permits to increase the coverage during fault-injection campaigns, while reproducibility allows replaying the same fault across iterations to collect the architectural events. In our setup, inputs are sampled uniformly using `rand()` and the seed is derived from the clock cycle count or Cycle Count Register value to achieve randomness. On the other side, reproducibility is ensured because each seed used for the input generation is stored in a dedicated memory region separated from the main memory, that remains intact also after the reboot. Moreover, for AutoBench benchmarks that

provide a dataset, the stored seed is also used to deterministically select dataset indices, ensuring identical elements are replayed across experiments.

The interaction and the organization between host and target is consistently divided in *golden run* and *faulty run*. The phases are reported as follows:

- The host initiates the campaign by rebooting the target system to ensure consistency and that injections begin from a clean, deterministic state. In respect to the previous framework, the boot procedure includes additional checks, since injecting faults into a corrupted environment may produce unpredictable behavior or cause crashes.
- Then, the host configures the target for the injection by generating a fault list composed of tuples that identify the injection location (address and register). Injections may target memory locations or CPU registers; each specified location also indicates the particular bit position to be flipped.
- After the execution of the workload on the target system, a final breakpoint is established, necessary to read the HPCs from the PMU.

Once the workload is completed, the resulting golden output is saved for reference. During the faulty run, the previously stored inputs are read to execute the workload under injected faults. The faulty output is compared against the golden reference and the result is classified in *benign*, *SDC* or *crash/hang*.

The integration of new benchmarks in the proposed structure was necessary to extend the dataset previously obtained with new data from the Autobench benchmarks. Only in this way, it was possible to learn about the effectiveness and usefulness of the data collected employing the new test cases. Furthermore, the inclusion of benchmarks more faithful to the automotive context has allowed us to improve some practical aspects of the implementation that in the past had not had a great influence, but which would certainly have affected the performance of multiworkload-campaigns.

## 4.3 Multi-workload campaigns

The execution of multiple workloads within the same campaign was realized thanks to the development of a new scheduling infrastructure. The infrastructure was designed to be highly customizable, supporting more than two concurrent workloads

in future tests, and to remain controllable by the fault injector even though the target is rebooted at each injected fault. Customizability is achieved through the configuration data structures `MultiBenchCfg_t` and `MultiBenchCfgTLS_t` (more details about the code are provided in the next chapter), which provide a convenient and extensible way to set benchmark parameters. Controllability to repeated reboots is ensured by storing runtime configuration and control data in BRAM, this enables the injector to preserve and reapply settings across iterations. Another critical requirement was randomness, achieved by extending the previously described approach to the multi-workload setting. In this case, a per-benchmark local seed is derived from the iteration number and the benchmark index, ensuring deterministic reproducibility for each benchmark, while randomness is preserved because `rand()` is still used.

### 4.3.1 High-level overview of MultiBench

The main components of a MultiBench campaigns are the following:

- `benchList[]`: an array of pointers to benchmark wrapper functions that determines which benchmark each task executes.
- `demo(int golden)`: the demo entry point. It initializes the `MultiBenchCfg_t` parameters by reading values written by the fault injector in BRAM and then calls `runMultiBenchmark(&cfg)`.
- `runMultiBenchmark(&cfg)`: performs sanity checks, allocates PMU vectors, creates semaphores used to orchestrate execution, spawns the worker tasks, and iterates the campaign for the number of iterations specified by the fault injector.
- `multiBenchmarkTask()`: waits for a notification to start, sets the execution context, runs the assigned benchmark, reads the PMU when the task is the injection target, stores the execution result, resets the PMU buffers, and releases the semaphore.

#### Configuration structure

The infrastructure relies on several ad-hoc *struct*s. The `MultiBenchCfg_t` configuration structure holds the global parameters of a campaign: `benchList`, the number of tasks (used for sanity checks), the global seed (provided by the fault

injector and used as the source for per-task local seeds), the number of iterations (the number of golden-run repetitions; one of these is randomly chosen as the injection target), and the target injection index (which indicates whether task 1 or task 2 will be the injection target). The `WorkerParam_t` structure contains each task's local parameters, including a pointer to the global configuration, the task's benchmark index, a local seed, the number of iterations already executed, a semaphore used for synchronization, and the task handle.

### 4.3.2 Global Structure for HPC collection and task management

A common feature to preserve also in the multibench campaigns is the reproducibility, which is necessary to collect HPC meaningful for the analysis. In this case, since just one workload is target of the injections, the infrastructure allows to collect just the HPC of interests. In future works, it will be possible to interconnect the execution flows of the two benchmarks to emulate a real-time scenario, and to collect the HPCs of both tasks in order to analyze how a fault in one workload affects the other and to investigate their mutual interactions. From a practical point of view, the collection of HPC and the association of the gathered data with benchmark tasks, it was necessary to define some global variables: `g_numTask` (the number of tasks), `g_pcVector` and `g_numTaskVector`. These variables are inizialized in the `multibench` file, but used internally in FreeRTOS `tasks.c`. In the operational flow, `runMultiBenchmark` initializes `g_numTask` and allocates `g_pcVector` (one slot per task): after creating the tasks, task numbers are stored in `g_numTaskVector`. During the execution, PMU collection is integrated into the scheduler routines: an initial PMU snapshot is taken at scheduler start using the function (`readPMU(pc_init_state)`), and on each task *switch-out* a final snapshot (`readPMU(pc_fin_state)`) is read and the difference is accumulated into `g_pcVector[index]`. To quickly determine whether the current TCB corresponds to a benchmark task, the helper function `check_currentTCB_is_bench_task` compares `uxTaskGetTaskNumber(pxCurrentTCB)` against the entries in `g_numTaskVector`; if not found it returns `-1` and no collection is performed.

#### Advantages

This approach is, in practice, the only workable solution, since only the operating system has precise knowledge of when context switches occur. Moreover, other advantages are reported below:

- By extending the FreeRTOS kernel, benchmark binaries are freed from duplicated collection code that would otherwise need to be present across different benchmarks. In SACRES applications, where the resources allocated are quite limited, this requirement should be enforced.

- Integrating PMU monitoring into the scheduler makes it straightforward to correlate counter values with iterations, seeds, and injection events, which is particularly useful for machine learning-based analysis and post hoc investigations.

- The scheme scales naturally to multiple tasks: accumulators are allocated per task and data collection can be limited to the injected (target) task, reducing overhead.

However, it is also important to highlight the timing overhead introduced by this approach, since PMU reads and delta computation may add latency to context switches. In order to limit the effect on task timing, the number of architectural events to track are limited, so that a limited overhead is introduced.

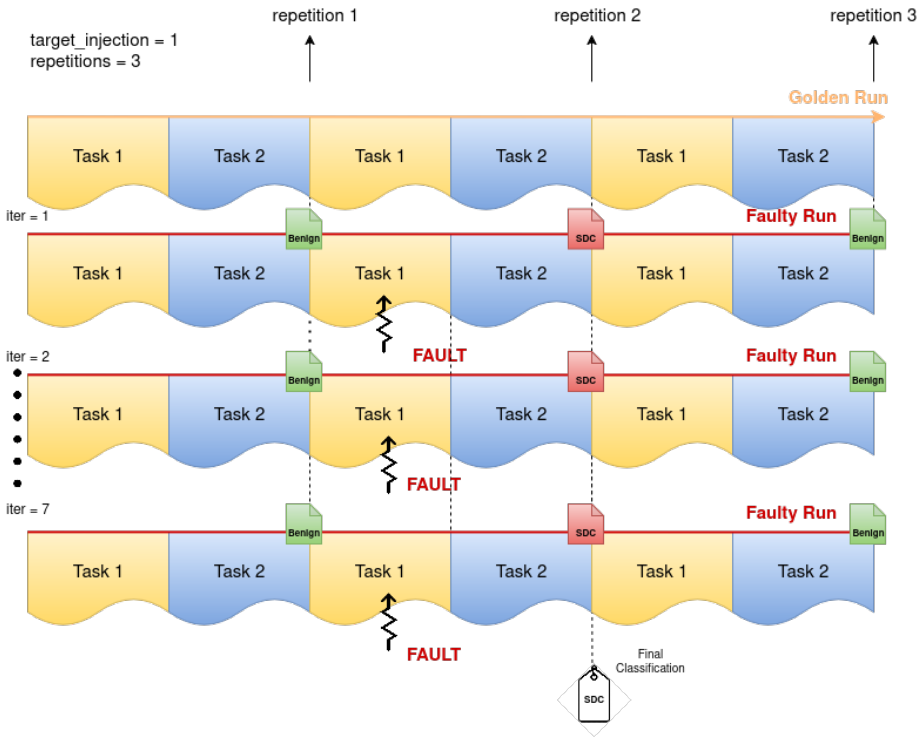


**Figure 4.1:** Multi-workload campaigns workflow

### 4.3.3 Timeline-based Multibench Scheduler

The multi-workload campaigns included also the realization of a timeline-based multibench routine, based on the execution of sets of Foreground workloads within predefined time windows and Background tasks, executed only in the slots left idle by the FG windows. The aim of this approach was to introduce precise timing of workload windows, a strict requirements of RTOS. The main components of the approach are:

- **FG worker tasks**: Each FG task waits for a scheduler notification to start its window, executes the benchmark and collects the PMU data in the case of the target of the injections. Then classifies the results and perform a busy-waits until the absolute end of the FG window.
- **BG worker tasks**: BG tasks are low-priority tasks that run only when no FG window is active in the same partition. BG tasks are suspended/resumed by the timeline scheduler.
- **Timeline Scheduler**: this is an high-priority task that manages major cycle and timeline events, notifies the FG tasks to start and stop, and suspends/resumes BG tasks accordingly.

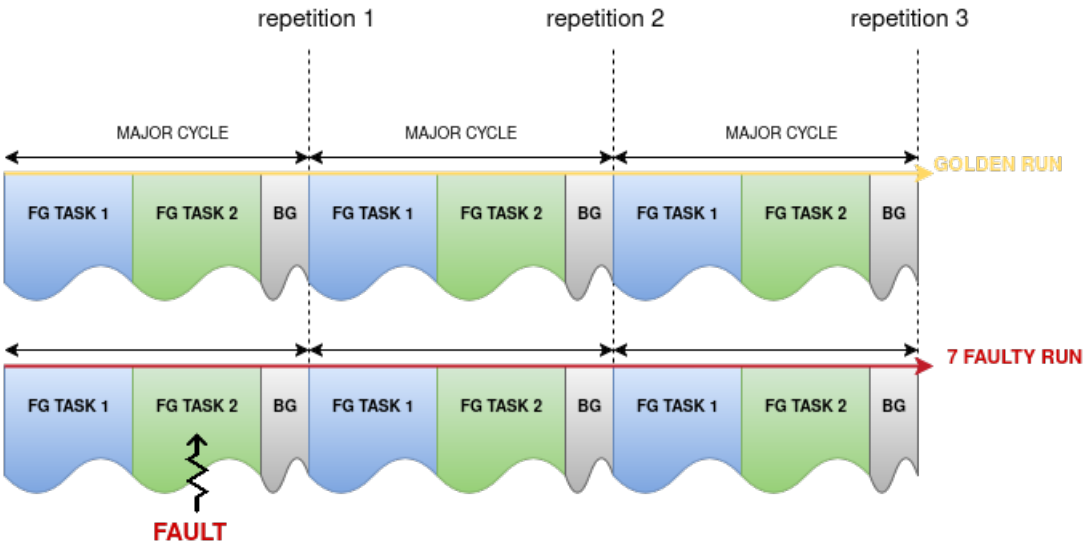


**Figure 4.2:** Timeline Scheduling workflow

As in the multibench campaigns described above, a series of data and configuration

structures where realized:

- `MultiBenchCfgTLS_t *cfg`: timeline configuration (major period, number of major cycles, global seed, etc.).
- `xFGTasksInfo[] / xBGTasksInfo[]`: per-window and per-BG task descriptors (start, stop offsets, partition, stack depth, parameters pointer, name).
- `TimelineEvent` array `events`: built at startup with two events per FG window (start and stop), then sorted by time.
- `sFGHandles`, `sBGHandles`: runtime arrays of created TaskHandle_t for FG/BG tasks.
- `g_numTask`, `g_pcVector`: shared globals used for PMU accumulation.
- `g_slotEndTick[]`: per-FG index absolute end tick used by FG tasks to busy-wait until window end.

**Considerations**

Timeline scheduling is a simple and general approach for RTOS workloads, and was necessary to characterize time–driven campaigns. This enriches the framework with data from a schedule where timing is explicitly enforced.

### 4.3.4 Fault Injector

Introducing multi–bench campaigns required changes to the existing infrastructure of the fault injector. A key clarification is the distinction between *repetitions* and *iterations*:

- **Repetition** ($r$): the number of times the same set of tasks is executed within one run. One randomly chosen repetition is the one where the fault is injected.
- **Iteration** ($i$): the number of times the same *faulty run* is repeated to collect different architectural events.

Given this terminology, the workflow is the following:

- **Golden run**: $R$ repetitions of the task set without injection are executed and the and the outputs are collected as reference.
- **Faulty runs**: The faulty run is performed $I$ times (iterations) and each faulty run consists of $R$ repetitions. One repetition $r^* \in \{1, \ldots, R\}$ is randomly chosen for the injection, while the other repetitions are executed without changes. After each repetition, the outcome is classified as *Benign*, *SDC*, or *Crash/Hang*.

In practice, $I$ iterations of the faulty run and, within each iteration, $R$ repetitions of the task set with different input (with one injected repetition) are performed. Final classification uses majority voting over the $I$ iterations for each repetition $r$:

$$\text{Final}(r) = \begin{cases} \text{Crash/Hang}, & \text{if } N_{\text{Crash}}(r) > 0, \\ \text{SDC}, & \text{if } N_{\text{SDC}}(r) > \left\lfloor \frac{I}{2} \right\rfloor, \\ \text{Benign}, & \text{otherwise}, \end{cases}$$

where $N_{\text{SDC}}(r)$ and $N_{\text{Crash}}(r)$ are the counts over iterations. In our typical configuration $I = 7$ (events collection), so "more than three" SDCs across iterations yields an overall SDC; ties resolve to *Benign*. It is important to notice that, even if a majority voting system is introduced, the property of reproducibility should ensure that the same fault repeated across multiple iterations produces the same outcome.

# Chapter 5

# Experimental Setup and Evaluation

This chapter begins by detailing the implementation of the experiments, then proceeds to address the preprocessing phase and the data visualization techniques used to enable a better assessment of the results.

## 5.1 Campaign Setup

The complete experimental setup used for the fault-injection campaigns consisted of a host PC running *Ubuntu Linux* with the *Xilinx Vitis IDE*. The host is responsible for building the FreeRTOS kernel and benchmark binaries, flashing them into the target PYNQ-Z2 board, controlling execution via the debugger, and collecting the execution logs and results.

### 5.1.1 Configuration of the target

The benchmarks are executed through FreeRTOS as tasks, which are created using the function `xTaskCreate()`. This function allows specifying different parameters (see A.1) and allocates the TCB and the task stack from the FreeRTOS heap. After the task creation, the application calls `vTaskStartScheduler()`, which performs

the kernel initialization, creates the Idle task, and starts executing the highest-priority **ready** task. As already mentioned previously, a task in FreeRTOS is implemented as an *infinite loop*, which implies that if it does return, this usually indicates a configuration or resource problem. If the compilation goes correctly, this results in a ELF file that can be flashed on the board, completing the setup of the target.

**Listing 5.1:** Task Creation in FreeRTOS

```
int main( void )
{
#if ( configSUPPORT_STATIC_ALLOCATION == 0 )

    /* Create the Fault Injector task */
    xTaskCreate(     faultInjectorTask,
                     ( const char * ) "FI",
                     FI_STACK_SIZE,
                     NULL,
                     FI_PRIORITY,
                     &xFITask );

#endif

    /* Start the tasks and timer running. */
    vTaskStartScheduler();
    for( ;; );
}

static void faultInjectorTask(void *pvParameters)
{
    if (feature == 0) {      // events as features
        if (golden_run == 0) {
            confPMU(c);
        }
        targetTask(golden_run, num_benchmark);
    }
    vTaskDelete(NULL);
}
```

### 5.1.2 Configuration of the Fault Injector

The configuration of the host PC also involves arranging the fault injector, the component responsible for controlling execution and injecting faults within the framework. This is realized through XSCT, which is an interactive and scriptable command-line interface to Xilinx Software Development Kit that also includes

Vitis. XSCT is based on the Tool Command Language which allows to create and configure hardware, generating Board Support Packages and application projects, and producing flash boot images. The choice fell on XSCT [13, 12] because it exposes commands for controlling the debug session of a running application and for inspecting both hardware and software state. To automate XSCT, it was necessary to use the module *pexpect*, which enables a Python script to spawn child processes, interact with them, and react to expected patterns in their output: the usage here allows driving XSCT from Python. The fragment below presents the code used to prepare the board and flash the ELF file, along with additional checks to ensure a clear and uncorrupted environment.

**Listing 5.2:** Python snippet automating board reset and flashing through XSCT

```
xsct = pexpect.spawn("xsct")     #spawn the xsct terminal
xsct.expect("xsct%")             #wait the xsct terminal to be ready
print(xsct.before.decode())
xsct.sendline("rst -processor -clear-registers")
xsct.sendline("rst -system")
xsct.sendline("rst -srst")
xsct.sendline("rst -por")
xsct.sendline("rst -dap")
xsct.sendline("disconnect")
xsct.sendline("source ./init.tcl")
xsct.expect(".*Successfully downloaded.*") #wait until the board is
    flashed
```

**Listing 5.3:** Fault Injector code

```
# loop over faults
for i in range(int(num_of_fault)):
    print("Fault num:", i)
    rand_iter_inj = random.randint(0, num_of_rep-1)

    # prepare golden run, flash board, set MMIO...

    # determine number of runs per fault
    num_of_run = 7
    if feature == "memory": num_of_run = 1
    elif feature == "all":  num_of_run = 8
    if checkpointing:       num_of_run = 1

    result = [[0] * num_of_rep for _ in range(num_of_run)]

    # runs (e.g., one per PMU-event-group)
    for y in range(num_of_run):
        xsct.sendline("mwr 0x40000000 0x0")
        xsct.sendline("mwr 0x40000000 " + str(int(y+1)))

```

```
21          if rand_iter_inj == 0:
22              xsct.sendline("bpadd " + str(rand_bp_pos))
23              num_bp_remove += 1
24
25          xsct.sendline("con -addr 0x00100000")
26
27          # repetitions per run (one of these may host the injection)
28          for r in range(num_of_rep):
29              if rand_iter_inj == r:
30                  xsct.expect(".*Breakpoint.*")              # stop on
    injection bp
31                  # perform bit-flip(s) here
32                  xsct.sendline("bpremove " + str(num_bp_remove))
33                  xsct.sendline("con")
34
35              try:
36                  xsct.expect(".*Breakpoint.*")    # wait for final/
    iter breakpoint
37              except:
38                  crash = True
39                  break
40
41              # read/aggregate result for this repetition
```

From the snippet above, several aspects need to be considered. The host-side script is organized as nested loops that set the parameters for each injection: in particular, the host configures the total number of faults to inject, which in our case was typically between 3 000 and 5 000 faults per campaign. For each fault, the script also selects a repetition index: this repetition number determines on which run the injector will perform the bit flip and the observed outcome is attributed to that specific repetition (this mechanism is used in the multi-workload campaigns). Finally, the number of runs per fault is set according to the number of events to track; as described in the previous chapter and better described in the following section, the number is set to seven runs per fault to cover the selected set of architectural events while remaining feasible in terms of overall execution time.

To collect meaningful architectural events, it is necessary to detect the precise end of the benchmark execution, but in FreeRTOS, when no other ready tasks exist, the kernel continues to schedule the *Idle* task. For this reason, in the setup, it was necessary to add a **final breakpoint** at the end of the task (typically on a *xil_printf*). During each run, XSCT inserts this breakpoint, resumes the target, and waits until the breakpoint is hit (line 36 of the above snippet). When the program counter reaches the final breakpoint, the debugger halts the core, and read the PMU counter values. Then resets and reconfigures the counters.

**Listing 5.4:** Final Classification in the Fault Injector

```
#final classification of the repetition

xsct.sendline("mrd 0x40000008")      #read the result of the
    comparison
xsct.expect(".*40000008: *")
value = xsct.readline().decode()

if crash:
    print("--Crash/Hangs--")
    result[y][r] = -1
else:
    if value[len(value)-5] == '1':
        result[y][r] = 1
        print("--Benign--")
    else:
        result[y][r] = 0
        print("--SDC--")

if (y == (num_of_run-1)):

    result_final = result_checker(result, num_of_run, r)
    if result_final == 1:
        print("FINAL--Benign--")
        f.write("benign\n")
    elif result_final == 0:
        print("FINAL--SDC--")
        f.write("SDC\n")
    elif result_final == -1:
        print("FINAL--Crash/Hangs--")
        f.write("crash/hangs\n")

xsct.sendline("mwr 0x40000008 0x1") #Reset the value

def result_checker(result, num_of_run, r):
    sdc = 0
    benign = 0
    crash = 0

    for y in range(num_of_run):
        if (result[y][r] == 0):
            sdc += 1
        elif (result[y][r] == 1):
            benign += 1
        elif (result[y][r] == -1):
            crash += 1
    if sdc > benign and sdc > crash:
        return 0    #return SDC
    elif benign > sdc and benign > crash:
```

```
48          return 1    #return Benign
49      else:
50          return -1   #return crash/hangs
```

In the multi-workload campaigns, at the end of the last run, `result_checker(result, N_run, r)` performs a majority vote over the set classification and the final label for the fault at repetition $r$ is the class with the largest count.

### 5.1.3 PMU Configuration

The target architecture of the embedded system exposes a large set of architectural events that can be profiled; however, only six PMU hardware counters are available concurrently. For this reason, to provide full coverage, each fault must be executed multiple times, reconfiguring the PMU so that different subsets of events are counted in each run. However, if all the *168* events of interest are covered, the replication factor equals:

$$\text{replications per fault} = \frac{168}{6} = 28.$$

For this reason, based on prior experiments [13, 12], the monitored architectural events is set to *42*, bringing the replication factor down to:

$$\text{replications per fault} = \frac{42}{6} = 7.$$

which explains the number of run needed of each fault and used by the fault injector.

The configuration of PMU also involved the creation of functions that enable reading the events that, in the specific run, the PMU has to track. To manage the PMU registers, it is necessary to use the ARM coprocessor interface. Writes are performed with the *Move to Coprocessor* instruction and reads with *Move from Coprocessor*. These instructions are written using `asm volatile`, which embeds the assembly and prevents the compiler from reordering or removing it.

Configuration proceeds as follows: (i) enable user-mode access to the PMU; (ii) program the Performance Monitor Control Register to enable the event counters and reset them; (iii) enable counting in the Performance Monitor Count Enable Set register. Then, for each hardware counter, select it in the Performance Monitors Event Counter Selection Register and assign the event code in the Performance Monitors Event Type Select Register. The read phase reflects the configuration

one: for each counter, select it with Performance Monitors Event Counter Selection Register, read back the programmed event from Performance Monitors Event Type Select Register if needed, and load the count value from the Performance Monitors Event Count Register using MRC. Then, the script prints the event identifier and the value obtained.

**Listing 5.5:** PMU Configuration

```
/* Function to configure the Performance Monitoring Unit (PMU) to track specific
 * architectural events. The parameter 'c' selects the batch (1..7), each batch
 * programming 6 events into the counters.
 */
void confPMU(int c){
    c = c - 1;
    // Traceable events (42)
    int events[42] =
    {1,3,4,5,6,7,10,12,13,15,16,17,18,23,80,96,97,98,101,102,103,
    104,108,109,110,112,113,114,115,118,119,120,129,131,133,138,139,
    140, 142,144,145,146};

    // Enable user-mode access to performance counters
    asm volatile ("MCR p15, 0, %0, C9, C14, 0\n\t" :: "r"(1));
    // Enable and reset event counters in PMCR
    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));

    // Program the six performance counters
    for(int i = 0; i < 6; i++){
        // Select counter in PMSELR
        asm volatile ("MCR p15, 0, %0, c9, c12, 5\t\n" :: "r"(i));
        // Select event in PMXEVTYPER
        if(c < 7)
            asm volatile ("MCR p15, 0, %0, C9, C13, 1" :: "r"(events[i + c*6]));
    }
}

/* Function to read the six performance counters (PC).
 * If 'pc_vett' is NULL, values are printed; otherwise stored in the array.
 */
void readPMU(int *pc_vett){
    unsigned int counter_value; // current counter value
    unsigned int evn_type;      // event number
```

```
35
36      for(int i = 0; i < 6; i++){
37          // Select counter in PMSELR
38          asm volatile ("MCR p15, 0, %0, C9, C12, 5" :: "r"(i));
39          // Read event type in PMXEVTYPER
40          asm volatile ("MRC p15, 0, %0, C9, C13, 1" :"=r"(evn_type));
41          // Read counter in PMXEVCNTR
42          asm volatile ("MRC p15, 0, %0, C9, C13, 2" :"=r"(
    counter_value));
43
44          if(pc_vett == NULL){
45              xil_printf("%d: %d\n", evn_type, counter_value); // read
    by sniffer
46          }else{
47              pc_vett[i] = counter_value;
48          }
49      }
50  }
```

## 5.2 Dataset extraction and statistics for single-workload campaigns

The results obtained from the execution of the single-workload campaigns, targeting respectively the CPU registers of rspeed01 and a2time for the injections, were collected in two different raw files named `single_rspeed` and `single_a2time`. These files contain the total number of faults injected, the register and position of the injection, the location (LOC), and the architectural events collected during the run. Each raw log was parsed with the same preprocessing procedure to produce a CSV file. Labels were normalized into three classes: **benign**, **SDC**, and **crash**; entries labeled as `crash/hangs` in the raw log were normalized to `crash`.

The main statistics for the extracted datasets are reported in Table 5.1.

**Table 5.1:** Dataset statistics extracted from `single_a2time` and `single_rspeed`.

| Category | `single_a2time` | `single_rspeed` |
|---|---|---|
| Total records | 10 355 | 11 393 |
| Benign | 9 568 (92.39%) | 10 692 (93.85%) |
| Crash (incl. hangs) | 720 (6.96%) | 685 (6.01%) |
| SDC | 52 (0.50%) | 15 (0.13%) |

From the statistics, both benchmarks exhibit a very large majority of *benign* executions, with *SDC* being the less frequent outcome. The higher *SDC* count observed in *a2time* compared to *rspeed01* can be explained directly from their code structure. In *a2time*, the pipeline is timing-sensitive around TDC detection: the TDC margin and the moving average (`deltaTimeAvg`) create windows in which a transient fault can perturb timing/angle and *nonlinearly* propagate to the final CRC. On the other side, *rspeed01* applies filtering and fail-safe logic (clamping unrealistic periods, rejecting large jumps, and zeroing speed on persistent anomalies), strategy that allows to be less vulnerable to the faults injected.

```
/* TDC detection and speed update */
if (pulseDeltaTime1 > (TDC_TEETH * deltaTimeAvg1 * TDC_MARGIN)) {
    tdcTime1 = rotationTime1;
    rotationTime1 = 0;
    engineSpeed1 = RPM_SCALE_FACTOR / tdcTime1;
    toothCount1 = 0;
}

deltaTimeAccum1 += pulseDeltaTime1;
if ((toothCount1 > 0) && (toothCount1 % (params->tonewheelTeeth /
    CYLINDERS) == 0)) {
    deltaTimeAvg1 = deltaTimeAccum1 / (params->tonewheelTeeth /
    CYLINDERS);
    deltaTimeAccum1 = 0;
}

firingTime1 = ((FIRE1_ANGLE - angle1) * tdcTime1 / TENTH_DEGREES) +
    params->angleCounter;

*params->RAMfilePtr = firingTime1;
params->RAMfilePtr++;
tcdef->CRC = Calc_crc32(firingTime1, tcdef->CRC);
```

The code provided shows that `engineSpeed1` depends inversely on `tdcTime1`, and `firingTime1` combines the estimated angle and timing through multiplications and additions. Small perturbations in counters (`rotationTime1`, `angleCounter`) or averages (`deltaTimeAvg1`) can produce deviations in the CRC sequence, which can explain the higher likelihood of SDCs. Although there is increased sensitivity to failures, the incidence of SDCs remains minimal.

```
if (toothDeltaTime1 < MIN_TOOTH_TIME)
    toothDeltaTime1 = toothDeltaTimeLast1;
if (toothDeltaTime1 > 4 * toothDeltaTimeLast1)
```

```
4       toothDeltaTime1 = toothDeltaTimeLast1;
5
6   toothTimeAccum1 += toothDeltaTime1;
7   toothCount1++;
8
9   if (toothCount1 >= params->tonewheelTeeth / 2) {
10      if (toothTimeAccum1 > MAX_TOOTH_TIME * params->tonewheelTeeth /
        2) {
11          roadSpeed1 = 0;
12      } else {
13          roadSpeed1 = SPEEDO_SCALE_FACTOR / (toothTimeAccum1 / params
        ->tonewheelTeeth * 2);
14          toothCount1 = 0;
15          toothTimeAccum1 = 0;
16      }
17  }
18  *params->RAMfilePtr = roadSpeed1;
19  params->RAMfilePtr++;
20  tcdef->CRC = Calc_crc32((e_u32)roadSpeed1, tcdef->CRC);
```

As shown in the code provided, the checks performed by *rspeed01* can mask the effect of faults before the CRC is updated, which leads to fewer SDCs.

## 5.3 Dataset extraction and statistics for multi-workload campaigns

Also in the case of multi-workload campaigns, multiple tests were performed, targeting respectively a2time and rspeed01 for the injections. The total amount of injected faults for each campaign was 12,153 for a2time and 12,258 for rspeed01. It is essential to consider that in these campaigns, multiple repetitions are executed, while only one repetition is actually targeted for fault injection. In our tests, we chose five repetitions per fault, and one of them was randomly selected for injection.

**Table 5.2:** Fault-level effectiveness summary for a2time and rspeed01.

| | a2time | | rspeed01 | |
|---|---|---|---|---|
| **Metric** | **Count** | **Percentage** | **Count** | **Percentage** |
| Total faults | 12,153 | 100.00% | 12,258 | 100.00% |
| SDCs | 3,044 | 25.05% | 870 | 7.10% |
| Benign | 9,109 | 74.95% | 11,388 | 92.90% |

The increased complexity of the infrastructure led to a higher SDC rate compared to single-workload experiments. This suggests that task interference and stricter timing constraints amplify fault propagation.

## 5.4 Preprocessing

The data collected from the campaigns were processed to generate a final dataset required for training machine learning algorithms. Two datasets were created from the single-workload campaigns and two from the multi-workload campaigns. The total number of samples obtained from the single-workload campaigns was 10 355 for a2time and 11 393 for rspeed01. In the multi-workload campaigns, 12 153 samples were collected for a2time and 12 258 for rspeed01. The features extracted for all datasets included `fault_id`, `reg`, `rep`, `pos`, `loc`, `res`, and the event identifier, resulting in a total of 42 events.

**Table 5.3:** Dataset

| fault_id | reg | pos | loc | e1 | e3 | e4 | e5 | ··· | e144 | e145 | e146 | label |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | r2 | 27 | 1244085 | 90 | 63774 | 1 | 43400 | ··· | 12 | 12 | 0 | benign |
| 1 | r1 | 19 | 1239281 | 91 | 63814 | 1 | 43474 | ··· | 12 | 12 | 0 | crash |
| 2 | r0 | 13 | 1243166 | 89 | 63829 | 1 | 43424 | ··· | 12 | 12 | 0 | crash |

As already mentioned in the previous sections, the obtained datasets are highly unbalanced, with most faults resulting in benign execution. In machine learning, such unbalance poses challenges, including misleading accuracy when the majority class dominates predictions. For instance, if benign outcomes are prevalent, a model may achieve high accuracy by primarily predicting this class, yet fail to detect SDCs outcomes effectively. Accuracy alone is often insufficient for evaluating unbalanced datasets: in this context, it is necessary to assess metrics like precision, recall, F1-score and confusion matrices, which provide a clearer assessment of minority class performance.

To address this issue, three primary categories of solutions are commonly employed: *data-based*, *algorithm-based*, and *tuning-based* approaches. Data-based approaches involve modifying the dataset to achieve a more balanced class distribution, thereby enabling the model to learn representative patterns from both classes. One such technique is *undersampling*, which reduces the size of the majority class by randomly removing samples. This adjustment allows the model to allocate equal attention to both majority and minority classes, reducing bias and improving

minority class learning. Conversely, *oversampling* increases the number of minority class examples, either by duplicating existing data or generating synthetic samples, to match the majority class size.

To demonstrate the potential of the proposed framework despite existing challenges, the solution utilizes a data-driven approach by applying the *Synthetic Minority Over-sampling Technique*(SMOTE). SMOTE addresses class unbalance by generating synthetic data points for the minority class through interpolation between existing samples and their nearest neighbors. This method introduces variation and avoids simple duplication, thereby enhancing model learning. The technique operates as follows:

- Identifies the k nearest neighbours for each minority class instance;
- Randomly selects one neighbour and generates a synthetic sample by interpolating between the original instance and the selected neighbour.

This method was selected because, compared to alternative approaches, it reduces the risk of overfitting by generating new synthetic data. Additionally, expanding the minority class enables the model to learn more effectively from these instances. However, due to the highly unbalanced nature of the datasets in this case, this solution remains suboptimal.

### 5.4.1 Dimensionality Reduction and Data Visualization

To facilitate interpretation of the results, both *t-distributed Stochastic Neighbor Embedding* (t-SNE) and *Principal Component Analysis*(PCA) were applied. Although both techniques are used for dimensionality reduction, they differ in several key aspects:

- t-SNE is an unsupervised, non-linear dimensionality reduction technique for exploring and visualizing high-dimensional data;
- PCA is a statistical technique that transforms high-dimensional data into a low-dimensional form while preserving as much variance as possible.

PCA is more effective for linear data, whereas t-SNE can identify clusters and structures that linear methods may overlook by preserving pairwise similarities between data points. Both are dimensionality reduction techniques and can be used complementarily:

- PCA is commonly used for dimensionality reduction and feature extraction, which is always a preprocessing step for machine learning models;

- t-SNE is commonly used for data visualization.

In this analysis, PCA was implemented as a preprocessing step, and the cumulative variance curve was plotted to assess the principal components.

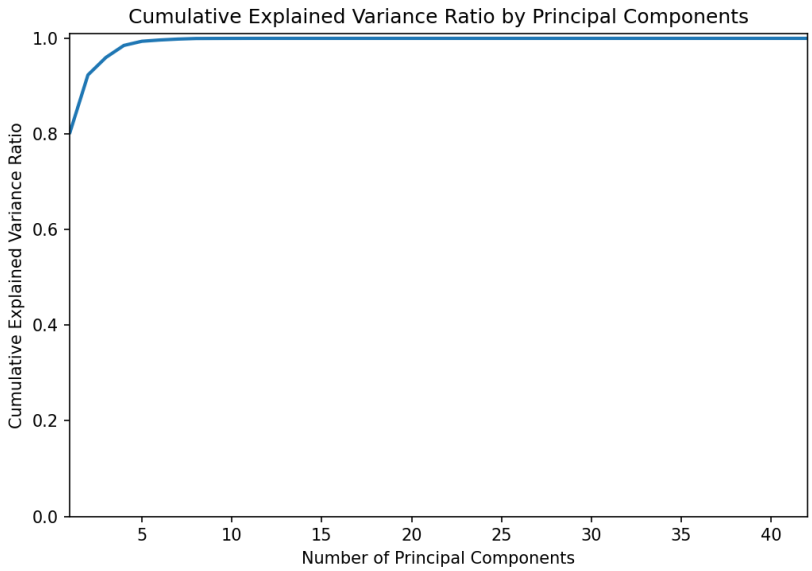


**Figure 5.1:** Cumulative Explained Variance Ratio by Principal Components for a2time

PCA cumulative variance curve indicates that the first principal components account for a substantial proportion of the total variance, although the presence of few features. Subsequently, t-SNE was implemented.

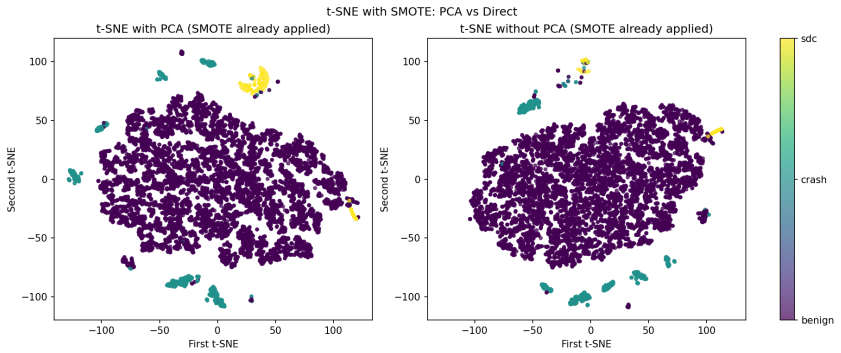


**Figure 5.2:** t-SNE for a2time

The t-SNE plots compare two configurations: t-SNE run on data after PCA (left) and t-SNE run directly on the balanced dataset. In both views, benign samples form a dense, central cluster; crash samples appear as more distinct clusters at the periphery; SDCs remain sparse and partially overlap with crash clusters.

It is possible to conclude that, although SDCs remain rare and overlapping, preprocessing leads to tighter groupings and fewer outliers. This local structure improvement may provide more consistent training signals and enhance the model's ability to recognize SDC behavior, even in the presence of such a challenging dataset.

# Chapter 6

# Conclusions and Future Work

This thesis presented a fault injection analysis of automotive benchmarks using HPC-based monitoring, building upon the previously developed framework described in [13, 12]. The existing framework comprised a host PC and a target embedded system, with faults injected through the debug unit. The research conducted in this thesis was organized into three primary components:

- **Inclusion of new benchmarks**: First, the benchmark suite was expanded to include additional automotive sector benchmarks, thereby enhancing the acquisition of HPC statistics representative of the real-time automotive environments. These benchmarks were modified to meet the requirements of the selected framework, ensuring both randomness and reproducibility. For this reason, new functions were implemented to generate random seeds, particularly in multi-workload scenarios, where each task required a unique yet reproducible seed. Then, benchmarks were executed using the developed multi-workload infrastructure.

- **Fault Injection Campaigns**: After implementing these new features, fault injection campaigns were conducted to collect the necessary statistics and HPC data for evaluation.

- **Preprocessing and Evaluation**: After data collection, the results were formatted for machine learning algorithms to facilitate a clear assessment of system reliability. The results of the fault injection campaign indicated

that the majority of faults resulted in benign outcomes. In single-workload campaigns, faults did not significantly affect overall results. However, in multi-workload campaigns, a greater number of SDCs was detected, attributable to the increased complexity of the proposed scheduling infrastructure. This process resulted in the creation of multiple datasets suitable for training new models, thereby improving the accuracy and precision in the classification of SDCs.

The development of this thesis involved several challenges, each of which was successfully addressed:

- Working with real hardware involves inherent risks. To provide a comprehensive analysis based on HPC monitoring, extensive testing was required. This approach increased the risk of register corruption, which could compromise test results.
- Achieving sufficient coverage required a broad set of tests, which created significant timing challenges. Collecting all necessary data took several weeks.
- A further challenge involved data collection and the creation of a new dataset, which was heavily imbalanced with mostly benign outcomes and few SDCs. This imbalance may hinder future efforts to develop models for accurate SDC classification.

The solutions proposed in the thesis aim to address these challenges by offering a flexible and realistic approach. While this thesis has addressed numerous challenges in real-time scenarios, several areas remain for future research and development to further enhance the framework:

- Inclusion and testing of additional automotive benchmarks could provide new data and statistics, thereby ensuring a more representative experimental environment.
- Further investigation of the FreeRTOS scheduling algorithm may enrich the framework and enable more precise evaluations using a time-driven approach.
- Exploring the connections between different benchmarks, particularly where outputs are correlated, could enrich the analysis of fault effects based on these interconnections.
- Deep learning approaches could also be considered and the results evaluated.

In conclusion, this thesis has addressed several challenges and improved the existing framework, thereby contributing to the development of next-generation resilient hardware.

# Appendix A

# Specifications

**Table A.1:** Common FreeRTOS APIs for Task Management and Synchronization

| **API** | **Description** |
|---|---|
| `xTaskCreate()` | Creates a new task with specified function, priority, and stack size. |
| `vTaskDelete()` | Deletes a task. |
| `vTaskDelay()` | Places a task into the Blocked state for a number of ticks. |
| `xSemaphoreCreateBinary()` | Creates a binary semaphore. |
| `xSemaphoreTake()` | Attempts to take (acquire) a semaphore, with optional block time. |
| `xSemaphoreGive()` | Releases (signals) a semaphore. |
| `xTaskNotify()` | Sends a task notification. |
| `ulTaskNotifyTake()` | Waits for (and clears) a task notification, emulating a binary semaphore. |

| **Benchmark** | **Description** |
|---|---|
| Angle to Time Conversion | Simulates reading toothed-wheel pulses on an engine crankshaft to detect TDC, compute engine speed, and convert tooth events into precise crankshaft angle expressed in linear time from TDC. |
| Basic Integer and Floating Point | Exercises basic numeric capability by computing $\arctan(x)$ via a rational polynomial (telescoping series) over a constrained input domain. |
| Bit Manipulation | Models a character display pipeline heavy in bit logic: shifting chars into a line buffer, mapping via a character ROM, and expanding to pixels in a display buffer. |
| Cache Buster | Stresses systems without cache/locality: pointer-driven control flow intentionally thwarts code/data locality to highlight non-cache, look-ahead execution performance. |
| CAN Remote Data Request | Emulates a CAN network node set handling Remote Data Request (RDR) frames: check message IDs, gather associated data, and queue replies for transmission. |
| Fast Fourier Transform | Computes a radix-2 decimation-in-frequency FFT on complex inputs, then forms the power spectrum of the signal. |
| Finite Impulse Response Filter | Performs fixed-point FIR filtering (16/32-bit), e.g., high/low-pass filters processing input sample streams. |
| Inverse Discrete Cosine Transform | Executes an iDCT on input blocks using 64-bit integer arithmetic, representative of video/graphics/image-processing kernels. |
| Inverse Fast Fourier Transform | Performs a radix-2 decimation-in-frequency inverse FFT (complex inputs) to synthesize time-domain signals from spectra (e.g., noise-cancellation use cases). |
| Infinite Impulse Response Filter | Implements a Direct-Form II cascaded biquad IIR (16/32-bit fixed-point) with binary comparators—exercising MACs, rounding, and feedback dynamics. |
| Matrix Arithmetic | Runs LU decomposition on $n \times n$ matrices, computes the determinant, and performs a cross-product with a second matrix. |
| Pointer Chasing | Uses a doubly-linked list and searches for many tokens across the list, recording traversal steps—stressing pointer manipulation and memory latency. |
| Pulse Width Modulation | Simulates H-bridge/stepper control: generates PWM + phase signals, updates motor position toward a commanded setpoint, and checks completion each PWM cycle. |
| Road Speed Calculation | Repeatedly derives vehicle speed from timer counts with filtering, handling counter rollover and abrupt changes; mixes arithmetic with intensive flow control. |
| Table Lookup and Interpolation | Retrieves values from 2D/3D calibration tables (e.g., ignition angle from load/speed) using bilinear interpolation instead of costly full computations. |
| Tooth to Spark | ECU-style fuel and ignition timing: determines run/start states and adjusts injector duration and spark timing per engine speed/load and other conditions on each pass. |

# Appendix B

# Implementation Code

```
/*waits for notification, runs its benchmark, reads PMU if targeted,
writes result flag, and signals completion via a counting semaphore*/
static void multiBenchmarkTask(void *pv)
{
    WorkerParam_t *p = (WorkerParam_t *)pv;
    const MultiBenchCfg_t *cfg = p->cfg;

    for (;;) {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        multibenchSetContext(p->idx, p->iter);
        int ok = cfg->benchList[p->idx](p->local_golden);

        if (cfg->target_injection == p->idx)
            readPMU_multi(g_pcVector[p->idx], p->iter);

        ((volatile char*)0x40000008)[0] = ok ? 1 : 0;
        reset_vettPMU(g_pcVector[p->idx]);
        xSemaphoreGive(p->barrier);
    }
}

/* creates tasks once, then for each iteration notifies all tasks,
passes per-task seeds, and waits on a counting barrier*/
void runMultiBenchmark(const MultiBenchCfg_t *cfg)
{
    g_numTask     = cfg->numTasks;
    g_pcVector    = pvPortMalloc(cfg->numTasks * sizeof *g_pcVector);
    barrier       = xSemaphoreCreateCounting(cfg->numTasks, 0);

```

```
30      for (int i = 0; i < g_numTask; ++i) {
31          params[i] = (WorkerParam_t){
32          .cfg=cfg,
33          .idx=i,
34          .iter=0,
35          .barrier=barrier
36      };
37
38      xTaskCreate(multiBenchmarkTask, name, 1024, &params[i],
    tskIDLE_PRIORITY+1, &params[i].handleTCB);
39      vTaskSetTaskNumber(params[i].handleTCB, (i+10));
40      g_numTaskVector[i] = uxTaskGetTaskNumber(params[i].handleTCB);
41      }
42
43      for (int iter = 0; iter < cfg->numIterations; ++iter) {
44          for (int t = 0; t < g_numTask; ++t) {
45              params[t].iter = iter;
46              params[t].local_golden = getLocalSeed(cfg->global_seed, t
    , iter);
47              xTaskNotifyGive(params[t].handleTCB);
48          }
49          for (int t = 0; t < g_numTask; ++t)
50              xSemaphoreTake(barrier, portMAX_DELAY);
51      }
52  }
53
54  /*reads repetitions and target from BRAM, fills cfg, and runs*/
55  static BenchWrap_t benchList[] = { a2time_wrap, rspeed_wrap };
56
57  void demo(int golden)
58  {
59      MultiBenchCfg_t cfg = {
60          .benchList = benchList,
61          .numTasks  = sizeof benchList / sizeof benchList[0],
62          .global_seed = golden,
63          .numIterations = ((volatile char*)0x40000018)[0],  //
    repetitions from FI
64          .target_injection = ((volatile char*)0x4000001C)[0]// target
    wrap index
65      };
66      runMultiBenchmark(&cfg);
67  }
```

```
1   /*read repetitions/target from FI (BRAM), fill cfg, run timeline.*/
2
3   static BenchWrap_t benchList[] = { a2time_wrap, rspeed_wrap };
4   static MultiBenchCfgTLS_t g_cfg_tls = {
```

```
    .benchList = benchList,
    .numTasks = sizeof benchList / sizeof benchList[0],
    .global_seed = 0,
    .numIterations = 0,
    .majorPeriodTicks = 100,
    .numMajorCycles = 10,
    .target_injection = 0
};

void demo_TLS(int golden)
{
    volatile uint8_t *repetitions = (volatile uint8_t *)0x40000018u;
    g_cfg_tls.global_seed    = golden;
    g_cfg_tls.numIterations = *repetitions;
    g_cfg_tls.target_injection = ((volatile uint8_t*)0x4000001Cu)[0];
    runMultiBench_Timeline(&g_cfg_tls);
}

/*runs inside scheduled windows, reads PMU if targeted,
writes the outcome flag, then sleeps until window end*/
static void foregroundWorkerTask(void *pv)
{
    WorkerParam_t *p = (WorkerParam_t *)pv;
    for (;;) {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        multibenchSetContext(p->idx, p->iter);
        int ok = p->cfg->benchList[p->idx](p->local_golden);
        if (p->cfg->target_injection == p->idx)
            readPMU_multi(g_pcVector[p->idx], p->iter);
        ((volatile char*)0x40000008)[0] = ok ? 1 : 0;
        reset_vettPMU(g_pcVector[p->idx]);
        busyWaitTicks(g_slotEndTick[p->idx], 0);
    }
}

/*builds start/stop events from FG windows,
then iterates major cycles, notifying FG and gating BG per partition
   */
static void vTaskStartTimelineScheduler(void *pv)
{
    const MultiBenchCfgTLS_t *cfg = (const MultiBenchCfgTLS_t *)pv;
    const TickType_t majorTicks = cfg->majorPeriodTicks;
    const int numMaj = cfg->numMajorCycles;

    /*
        build and sort events (start/stop for each FG window) */

    TickType_t majorStart = xTaskGetTickCount();

```

```
53      for (;;) {                                      // continuous timeline
54          for (int mc = 0; mc < numMaj; ++mc) {
55              TickType_t wake = majorStart, last = 0;
56
57              for (int e = 0; e < evCount; ++e) {
58                  TickType_t t = events[e].time;
59                  if (t > last) { vTaskDelayUntil(&wake, t - last);
    last = t; }
60
61                  int fi = events[e].index, part = xFGTasksInfo[fi].
    partition;
62                  if (events[e].kind == +1) {                  // FG
    window starts
63                      if (part > 0) sPartitionFGActive[part] = 1u;
64                      WorkerParam_t *p = (WorkerParam_t *)xFGTasksInfo[
    fi].pvParameters;
65                      if (p) {
66                          p->iter = mc;
67                          p->local_golden = getLocalSeed(cfg->
    global_seed, p->idx, mc);
68                          g_slotEndTick[p->idx] = majorStart +
    xFGTasksInfo[fi].stop;
69                          xTaskNotifyGive(sFGHandles[fi]);     // notify
     FG start
70                      }
71                      for (int i = 0; i < numBGTasks; ++i)      //
    suspend BG in part
72                          if (xBGTasksInfo[i].partition == part &&
    sBGHandles[i])
73                              vTaskSuspend(sBGHandles[i]);
74                  } else {                                     // FG
    window ends
75                      if (part > 0) sPartitionFGActive[part] = 0u;
76                      for (int i = 0; i < numBGTasks; ++i)      //
    resume BG in part
77                          if (xBGTasksInfo[i].partition == part &&
    sBGHandles[i]) {
78                              vTaskResume(sBGHandles[i]);
79                              xTaskNotifyGive(sBGHandles[i]);
80                          }
81                  }
82              }
83              TickType_t elapsed = xTaskGetTickCount() - majorStart;
84              if (elapsed < majorTicks) vTaskDelay(majorTicks - elapsed
    );
85              majorStart += majorTicks;                        // next
    major cycle
86          }
87      }
```

```
88 }
89
90 /* init per-task params, create FG/BG tasks and timeline task*/
91 void runMultiBench_Timeline(MultiBenchCfgTLS_t *cfg)
92 {
93     if (!cfg || cfg->numTasks <= 0 || cfg->numTasks > NUM_TASKS)
       return;
94
95     g_numTask  = cfg->numTasks;
96     g_pcVector = pvPortMalloc(cfg->numTasks * sizeof *g_pcVector);
97
98     for (int i = 0; i < cfg->numTasks; ++i) {
99         params[i] = (WorkerParam_t){
100        .cfg=(const MultiBenchCfg_t*)cfg,
101        .idx=i,
102        .iter=0,
103        .local_golden=cfg->global_seed, .barrier=NULL };
104    }
105    /*
106        creation of the FG, BG and scheduler
107                                                */
108 }
```

# Bibliography


[1] ISO/SAE. *ISO/SAE 21434:2021 — Road vehicles — Cybersecurity engineering*. https://www.iso.org/standard/70918.html. International Standard, Geneva: International Organization for Standardization. 2021 (cit. on p. 1).

[2] *Regulation No. 155 — Cybersecurity and Cybersecurity Management System*. World Forum for Harmonization of Vehicle Regulations (WP.29). United Nations Economic Commission for Europe (UNECE), 2021. URL: https://unece.org/transport/documents/2021/03/standards/uniform-provisions-approval-vehicles-regard-cyber-security-and (cit. on p. 1).

[3] *Regulation No. 156 — Software Update and Software Update Management System*. World Forum for Harmonization of Vehicle Regulations (WP.29). United Nations Economic Commission for Europe (UNECE), 2021. URL: https://unece.org/transport/documents/2021/03/standards/uniform-provisions-approval-vehicles-regard-software-update-and (cit. on p. 1).

[4] *ISO/SAE 24089:2023 — Road Vehicles — Software Update Engineering*. International Standard, International Organization for Standardization. Geneva: ISO/SAE, 2023. URL: https://www.iso.org/standard/79090.html (cit. on p. 2).

[5] Ludovic Pintard, Michel Leeman, Abdelillah Ymlahi-Ouazzani, Jean-Charles Fabre, Karama Kanoun, and Matthieu Roy. «Using fault injection to verify an autosar application according to the ISO 26262». In: *SAE 2015 World Congress & Exhibition*. SAE International. 2015 (cit. on pp. 2, 3).

[6] Jihyun Park and Byoungju Choi. «ASFIT: AUTOSAR-based software fault injection test for vehicles». In: *Electronics* 9.5 (2020), p. 850 (cit. on pp. 2, 3).

[7] *IEEE Standard for System, Software, and Hardware Verification and Validation*. DOI: 10.1109/IEEESTD.2016.7473044. New York, USA: Institute of Electrical and Electronics Engineers (IEEE), 2016. URL: https://standards.ieee.org/ieee/1012/7324/ (cit. on p. 2).

[8] Renato Rafael Arcanjo, LEG Martins, and DLG Fernandes. «Verification and Validation of Embedded Software in an Automotive Context: A Systematic Literature Review». In: *ResearchGate, October* (2023) (cit. on p. 2).

[9] Water Test Network / Interreg North-West Europe. *Guide to Verification, Validation and Certification*. Online PDF. URL: https://vb.nweurope.eu/media/9617/guide-to-verification_validation_certification.pdf (cit. on p. 2).

[10] David Chenho Kung and Hong Zhu. *Software Verification and Validation.* 2008 (cit. on pp. 2, 3).

[11] EEMBC. *EEMBC MultiBench™ Multicore Benchmark Suite*. https://www.eembc.org/multibench/. 2009 (cit. on p. 3).

[12] Enrico Magliano, Alessandro Savino, and Stefano Di Carlo. «Real-time Embedded System Fault Injector Framework for Micro-architectural State Based Reliability Assessment». In: *Journal of Electronic Testing* (2025), pp. 1–16 (cit. on pp. 3, 5, 7, 17, 18, 20, 43, 46, 55).

[13] Enrico Magliano, Alessio Carpegna, Alessadro Savino, and Stefano Di Carlo. «A micro architectural events aware real-time embedded system fault injector». In: *2024 IEEE 25th Latin American Test Symposium (LATS)*. IEEE. 2024, pp. 1–6 (cit. on pp. 3, 5, 7, 19, 43, 46, 55).

[14] *ARM Architecture Reference Manual: Performance Monitoring Unit*. ARM Ltd. 2025 (cit. on p. 4).

[15] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. «Fundamental concepts of dependability». In: *Technical Report Series-University of Newcastle upon Tyne Computing Science* (2001) (cit. on pp. 7–9).

[16] Peter Tröger. *Dependability Attributes*. Lecture notes / PDF. Dependable Systems Course, HPI / University, 2014. URL: https://osm.hpi.de/depend/2014/05_dep_attributes.pdf (cit. on p. 7).

[17] Wikipedia contributors. *Difference engine — Wikipedia, The Free Encyclopedia*. 2025. URL: https://en.wikipedia.org/w/index.php?title=Difference_engine&oldid=1313632957 (cit. on p. 7).

[18] IFIP Working Group 10.4. *IFIP Working Group 10.4*. https://www.dependability.org/?page_id=265. 2025 (cit. on p. 7).

[19] IEC. *International Electrotechnical Vocabulary – Dependability*. IEC 60050-192:2015, Clause 192-01-22. Available at: https://www.electropedia.org/iev/iev.nsf/display?openform&ievref=192-01-22. 2015 (cit. on p. 7).

[20] Jean-Claude Laprie. «Dependability: Basic concepts and terminology». In: *Dependability: Basic Concepts and Terminology: In English, French, German, Italian and Japanese*. Springer, 1992, pp. 3–245 (cit. on pp. 7, 9).

[21] Deniz Kasap, Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. «Micro-Architectural features as soft-error markers in embedded safety-critical systems: preliminary study». In: *2023 IEEE European Test Symposium (ETS)*. IEEE. 2023, pp. 1–5 (cit. on p. 9).

[22] Goutam Kumar Saha. «Software Fault Avoidance». In: *IEEE Reliability Society Newsletter Vol* 57 (2011), p. 25 (cit. on p. 9).

[23] *Securing Small-Business and Home Internet of Things (IoT) Devices: Mitigating Network-Based Attacks Using Manufacturer Usage Description*. Special Publication NIST SP 1800-15B. Gaithersburg, MD, USA: National Institute of Standards and Technology (NIST), 2019. URL: https://doi.org/10.6028/NIST.SP.1800-15 (cit. on p. 9).

[24] *Minimum Security Requirements for Federal Information and Information Systems*. Federal Information Processing Standards Publication FIPS 200. Gaithersburg, MD, USA: National Institute of Standards and Technology (NIST), 2006. URL: https://doi.org/10.6028/NIST.FIPS.200 (cit. on p. 9).

[25] IEC. *International Electrotechnical Vocabulary (IEV) – Part 192: Dependability*. IEC 60050-192:2015. Available at: https://www.electropedia.org/iev/iev.nsf/display?openform&ievref=192-01-22. 2015 (cit. on p. 9).

[26] *ISO 26262: Road vehicles – Functional safety*. Second edition. International Organization for Standardization, 2018 (cit. on p. 9).

[27] ISO/TC 184/SC 4. *Industrial automation systems and integration — Product data representation and exchange — Part 226: Application protocol: Ship mechanical systems*. Application protocol (AP 226) ISO 10303-226. Working draft / WD referenced in on-line catalogs. International Organization for Standardization (ISO), 2001. URL: https://genorma.com/en/standards/iso-wd-10303-226 (cit. on p. 10).

[28] Robert Baumann. «Soft errors in advanced computer systems». In: *IEEE design & test of computers* 22.3 (2005), pp. 258–266 (cit. on p. 10).

[29] Yousun Ko and Bernd Burgstaller. «BEC: Bit-Level Static Analysis for Reliability against Soft Errors». In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2024, pp. 283–295 (cit. on p. 10).

[30] Saad Memon, Rafal Graczyk, Tomasz Rajkowski, Jan Swakon, Damian Wrobel, Sebastian Kusyk, Seth Roffe, and Mike Papadakis. «When Radiation Meets Linux: Analyzing Soft Errors in Linux on COTS SoCs under Proton Irradiation». In: *arXiv preprint arXiv:2503.03722* (2025) (cit. on p. 11).

[31] *Single Event Effects – A Comparison of Configuration Upsets and Data Upsets*. White Paper WP0203. White Paper. Microsemi Corporation, 2002 (cit. on pp. 11, 12).

[32] Melanie Berg, C. Poivey, D. Petrick, D. Espinosa, Austin Lesea, K. LaBel, M. Friendlich, H. Kim, and Anthony Phan. «Effectiveness of internal vs. external SEU scrubbing mitigation strategies in a Xilinx FPGA: Design, test, and analysis». In: *2007 9th European Conference on Radiation and Its Effects on Components and Systems*. 2007, pp. 1–8. DOI: 10.1109/RADECS.2007.5205603 (cit. on p. 11).

[33] James F Ziegler and William A Lanford. «Effect of cosmic rays on computer memories». In: *Science* 206.4420 (1979), pp. 776–788 (cit. on p. 11).

[34] Massimo Violante, Luca Sterpone, M Ceschia, D Bortolato, Paolo Bernardi, M Sonza Reorda, and A Paccagnella. «Simulation-based analysis of SEU effects in SRAM-based FPGAs». In: *IEEE Transactions on Nuclear Science* 51.6 (2004), pp. 3354–3359 (cit. on p. 11).

[35] European Cooperation for Space Standardization. *multiple bit upset (MBU)*. ECSS. 2025. URL: https://ecss.nl/item/?glossary_id=74 (cit. on p. 12).

[36] Gustavo Neuberger, Fernanda De Lima, Luigi Carro, and Ricardo Reis. «A multiple bit upset tolerant SRAM memory». In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 8.4 (2003), pp. 577–590 (cit. on p. 12).

[37] Swagata Mandal, Sreetama Sarkar, Wong Ming Ming, Anupam Chattopadhyay, and Amlan Chakrabarti. «Criticality aware soft error mitigation in the configuration memory of SRAM based FPGA». In: *arXiv preprint arXiv:1810.09661* (2018) (cit. on p. 12).

[38] Wikipedia contributors. *Triple modular redundancy — Wikipedia, The Free Encyclopedia*. 2025. URL: https://en.wikipedia.org/w/index.php?title=Triple_modular_redundancy&oldid=1304120120 (cit. on p. 13).

[39] Wikipedia contributors. *Error correction code — Wikipedia, The Free Encyclopedia*. 2025. URL: https://en.wikipedia.org/w/index.php?title=Error_correction_code&oldid=1311490145 (cit. on p. 13).

[40] Jafar Vafaei, Omid Akbari, Muhammad Shafique, and Christian Hochberger. «X-rel: Energy-efficient and low-overhead approximate reliability framework for error-tolerant applications deployed in critical systems». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 31.7 (2023), pp. 1051–1064 (cit. on p. 13).

[41] Behzad Ghasemi Parvin and Leila Ghasemi Parvin. «Applications of artificial intelligence in fault detection and prediction in technical systems». In: June 2023. DOI: 10.6084/m9.figshare.25180289 (cit. on p. 13).

[42] Pragya Dhungana, Rupesh Kumar Singh, and Hariom Dhungana. «Machine learning model for fault detection in safety critical system». In: *International Conference on Applications in Electronics Pervading Industry, Environment and Society*. Springer. 2023, pp. 499–507 (cit. on p. 13).

[43] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. «Fault injection techniques and tools». In: *Computer* 30.4 (1997), pp. 75–82. DOI: 10.1109/2.585157 (cit. on pp. 13, 15).

[44] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. «A survey on fault injection techniques». In: *Int. Arab J. Inf. Technol.* 1.2 (2004), pp. 171–186 (cit. on p. 13).

[45] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. «Fault injection techniques and tools». In: *Computer* 30.4 (1997), pp. 75–82 (cit. on pp. 13, 15).

[46] Amit Mazumder Shuvo, Tao Zhang, Farimah Farahmandi, and Mark Tehranipoor. «A comprehensive survey on non-invasive fault injection attacks». In: *Cryptology ePrint Archive* (2023) (cit. on pp. 14, 16).

[47] Alfredo Benso and Paolo Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Jan. 2003 (cit. on p. 15).

[48] Lena Feinbube, Lukas Pirl, and Andreas Polze. «Software Fault Injection: A Practical Perspective». In: *Dependability Engineering*. Ed. by Fausto Pedro García Márquez and Mayorkinos Papaelias. London: IntechOpen, 2017. Chap. 4. DOI: 10.5772/intechopen.70427. URL: https://doi.org/10.5772/intechopen.70427 (cit. on p. 15).

[49] A. Baldini, A. Benso, S. Chiusano, and P. Prinetto. «"BOND": An interposition agents based fault injector for Windows NT». In: *Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. 2000, pp. 387–395. DOI: 10.1109/DFTVS.2000.887179 (cit. on p. 16).

[50] Ricardo Maia, Luis Henriques, Ricardo Barbosa, Diamantino Costa, and Henrique Madeira. «Xception fault injection and robustness testing framework: a case-study of testing RTEMS». In: *Workshop de Testes e Tolerância a Falhas (WTF)*. SBC. 2005, pp. 61–72 (cit. on p. 16).

[51] J-C Fabre, Manuel Rodriguez, Jean Arlat, and J-M Sizun. «Building dependable COTS microkernel-based systems using MAFALDA». In: *Proceedings. 2000 Pacific Rim International Symposium on Dependable Computing*. IEEE. 2000, pp. 85–92 (cit. on p. 16).

[52] Seungjae Han, Kang G Shin, and Harold A Rosenberg. «Doctor: An integrated software fault injection environment for distributed real-time systems». In: *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*. IEEE. 1995, pp. 204–213 (cit. on p. 16).

[53] Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. «The soft error problem: An architectural perspective». In: *11th International Symposium on High-Performance Computer Architecture*. IEEE. 2005, pp. 243–247 (cit. on p. 17).

[54] TUL PYNQ. *PYNQ-Z2 User Manual, Version 1.0*. TUL Corporation. 2018. URL: https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf (cit. on p. 22).

[55] EMCelettronica. *Processore ARM Cortex-A9*. https://it.emcelettronica.com/processore-arm-cortex-a9 (cit. on p. 23).

[56] Wikipedia contributors. *ARM Cortex-A9 — Wikipedia, The Free Encyclopedia*. 2025. URL: https://en.wikipedia.org/w/index.php?title=ARM_Cortex-A9&oldid=1304461228 (cit. on p. 23).

[57] Vedant Rokad. *Fundamentals of RTOS*. URL: https://medium.com/@rokadvedant03/fundamentals-of-rtos-41e8eaef3a7e (cit. on p. 25).

[58] Wikipedia contributors. *Real-time operating system. Wikipedia*, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Real-time_operating_system (cit. on p. 25).

[59] *European Commission Research Participant Portal: Guidance Document*. Tech. rep. European Commission, 2018. URL: https://ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5c07e1a67&appId=PPGMS (cit. on p. 25).

[60] Amazon Web Services, Inc. *FreeRTOS Official Documentation*. Available online. 2025. URL: https://www.freertos.org/ (cit. on p. 26).

[61] Embedded Microprocessor Benchmark Consortium (EEMBC). *EEMBC Autobench*. 2025. URL: https://www.eembc.org/autobench/ (cit. on p. 29).

[62] *AutoBench Benchmark Suite: Datasheet*. Tech. rep. Embedded Microprocessor Benchmark Consortium (EEMBC), 2001. URL: https://www.eembc.org/techlit/datasheets/autobench_db.pdf (cit. on p. 31).