POLITECNICO DI TORINO

Master Degree in Electronic Engineering

Master Degree Thesis

Fault injection techniques for Real-Time Operating Systems

Supervisor
Prof. Maurizio Rebaudengo

Candidate
Dario Mamone - s242209

CNRS - LIRMM - Université de Montpellier
Supervising professor
Prof. Alberto Bosio

Academic Year 2017-2018

Acknowledgments

I would like to thank professor Maurizio Rebaudengo for the constant support, for the possibility to have a study experience abroad and, furthermore, for having introduced me to the world of research.

I wish to thank professor Alberto Bosio as well for all the suggestions provided for the development of this work and for his constant availability during my stay in Montpellier.

Abstract

When an electronic system stops working properly, the causes of such malfunctioning could be due to a human factor or to the external environment. If the former ones can be discarded, then it is highly probable that the system encountered an error as consequence of high energy particles which stroke the hardware causing a permanent or transient damage: in the former case, the whole system is definitively harmed and only a physical substitution of the broken circuit can solve the problem; in the latter case, instead, power cycling would be a sufficient solution. In both circumstances, if the system must be always active and respond respecting well defined deadlines, such misbehaviors can lead to catastrophic consequences; in the worst case, the element to be substituted cannot be even accessed: this is the case of automotive, avionic and aerospace applications.
In order to avoid these scenarios, the system must be extensively tested and then strengthen where it showed high sensitivity to random variation in signals and data: fault injection is exactly that technique which allows to spot vulnerabilities in a system, highlighting those parts which need to be hardened.

This work aims to investigate the effects of Single Event Upset (SEU), caused usually by high energy particles, in Real-Time Operating Systems (RTOS) specifically developed for embedded solutions, analyzing the consequences of faults injected on most relevant data of the operating system itself. SEU effects are simulated using a prototyping board designed by STMicroelectronics running FreeRTOS as embedded OS; parameters of the injection are sent to the hardware from a host computer, which automatizes the process. Fault injection campaigns are performed on various parts of the OS and information about the status of the system are extracted after the injection.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Nowadays, embedded systems are used in a huge amount of fields, from medicine to automotive, from consumer electronics to avionics and aerospace, from security and access control systems to biology and so on. In all these cases the system must meet both required and desirable features chosen at design time and/or imposed by a standard, according to its *mission*. In any case, a desirable property for every system is a high *dependability*: this can be achieved with a deep analysis of the system, targeted to identify weaknesses and then with the implementation of some techniques which allow to mitigate or completely remove them. However, extensive testing phases require money and time, delaying the entrance of the product on the market and increasing the final per-unit price: for these reasons an optimal trade-off must be found during the design phase so that the product final price does not exceed the target one and, at the same time, the system can still work with the desired quality level.
Dependability can be reduced in many ways: without taking into consideration errors due to design, hardware and software bugs, problems occurred during fabrication and intentional tampering, there are many other external events that can affect this property during the lifetime of the application; some of them are due to the interaction of the circuit with the surrounding environment and this could cause problems like memory bit-flip, signal degradation, data loss, permanent damage of the physical circuit.
As modern electronics is becoming more and more complex, with even more strict specifications - operations must be done in a fast, correct and safe way, ensuring the continuous stability of the application - and at the same time many hardware platforms support the integration of an OS, it is a good practice to surround the application with an operating system, which preferably includes some kind of robustness, even if resources are poor: the overhead added is minimal if compared to the advantages it brings. If some timing constraints have to be respected, a RTOS can be considered as possible solution. However, the RTOS itself introduces a new place where weaknesses could manifest their dangerousness, so it is important to characterize its behavior in presence of faults and eventually to implement some techniques to prevent, remove or tolerate them: this requires the development of new testing systems aimed at measuring the RTOS robustness, spotting all its vulnerabilities and characterizing the behavior of the application when such faults occur.

1.2 State of the art

Fault injection has a very long tradition in the world of testing: many injection systems, in fact, have already been developed to test generic RAMs, FPGAs, ASICs, or even ISA-based circuits exploiting different techniques. Many of these testing environments have been created to evaluate the dependability, to identify vulnerabilities, or, in the case of software-based testing, to perform an analysis of the system under test using a more portable and easier to implement solution. Some examples of the most famous software-based injection environments are FERRARI[1], Xception[2], FTAPE[3].
However, this thesis has the aim to develop a system to inject faults in OS-related variables in order to test its robustness: up to now, it seems that fault injection in operating systems has been done only in those parts of OSs that are exposed the most, but it was never performed extensively in internal data used exactly by a RTOS; some of these past works, in fact, allow to perform tests on OSs whose code is partially or completely closed source, reducing drastically places where it is possible to inject. In the following sections the most relevant found documents concerning the topic of fault injection in OSs are briefly described.

1 - A Generic Fault Injection Framework for the Android OS
[4] This work is a bachelor thesis made at Technischen Universität Darmstadt with the aim to develop an injection framework for Android. Two types of injections are performed: the first one is based on the modification of the operating system code before compiling it; the second one instead requires the presence of an external actor able to manage the injection remotely. The device under test, a virtual machine, remains for the whole duration of the campaigns attached to the host computer exploiting a TCP communication, while GDB (GNU Debugger) is used to perform the injection. Injections are done in the driver calls and during memory accesses as soon as a predefined breakpoint inserted in the code through GDB is hit: in the former case, when a driver call occurs, parameters passed to it are modified on the fly; in the latter case instead, content of hardware registers accessed by the kernel when trying to read or write a datum in the memory is subjected to injection: they are data memory and address registers integrated in the load-store unit of the architecture.

2 - FIFA: A Kernel-Level Fault Injection Framework for ARM-based Embedded Linux System
[5] This work is the most similar to this thesis but shows anyway some relevant differences. This time the injection is done on a Linux OS running on ARM hardware and managed by a host computer that communicates with the device under test through a serial port. The injection can be done exploiting hardware breakpoints provided by ARM or using KGDB remotely: in all cases the memory location to be tested is accessed when the kernel executes a particular part of the code, injecting so in a non defined time instant.

3 - An RTOS-based Fault Injection Simulator for Embedded Processors
[6] The fault injector presented in this work aims to study the effects of some injections in the system in order to tamper it. More precisely, injections are done in some memory locations containing results of some encryption and decryption processes to perform a Differential Fault Attack (DFA): this kind of cryptanalysis technique allows to find relevant information about a cryptographic key doing some injections during the encryption of a clear text and then comparing its encrypted version with the fault-free one. In this

case the targets of the injection are generic memory cells containing user data and not system data.

4 - Robustness Evaluation of Operating Systems
[7] In this doctoral thesis a system able to inject faults in Windows CE is presented. Again interface between the user and kernel of the operating system is selected as target of the injection: three drivers (serial port, network card and flash card reader drivers) are subjected to tests. Anyway, also in this case, the internal data of the OS are not put under test because they are hidden to the user.

1.3 Structure of the thesis

The thesis has been written so that, for each chapter, all the required knowledge have been already given, at least generically, by the previous ones. Some chapters contain very detailed descriptions, for example, of some parts of the developed injection environment or of the operating system used as target of the experiments: sometimes the discussion could seem too complicated, but, in these cases, it was necessary to report all the information to provide a complete overview of the topic.

Chapter 1 - Introduction
The current chapter contains the motivation of this work and the state of the art of fault injection techniques on operating system.

Chapter 2 - Physics of Single Event Effects
This chapter explains briefly the physics behind SEE and provides a classification scheme.

Chapter 3 - Fault injection
The topic of fault injection is introduced starting from the more generic concept of dependability, providing then descriptions concerning the various injection techniques.

Chapter 4 - Real-Time Operating Systems
Real-Time Operating Systems are analyzed from a generic point of view; FreeRTOS most important data and kernel operations are then studied, providing detailed descriptions of those parts which are most relevant for this work.

Chapter 5 - Fault injection Environment
A very detailed description of the developed fault injection environment is provided; here it is possible to find a sufficient amount of information to learn how to use the injector.

Chapter 6 - Experimental environment
Important definitions related to identified misbehaviors classes and to quantities defined to present results in a compact way are provided here.

Chapter 7 - Benchmarks under test
In this chapter the Automotive suite of benchmarks provided by EEMBC® is described: such programs are used as standards to give scientific relevance to injection experiments.

Chapter 8 - Experimental results
Results related to all experimental campaigns performed are reported, with a detailed description of the behavior of the system in all the relevant cases in which the OS has shown a misbehavior.

Chapter 9 - Conclusions
A brief summary of the behavior of the system under test is given, with some suggestions for future improvements of the tested RTOS.

1.4 Hardware and software used

In order to provide a summary and to introduce all software tools and hardware platforms used to develop the work, a quick list is written below.

- **STM32F3DISCOVERY** This is a prototyping board produced by STMicroelectronics with integrated programmer/debugger; this platform has been chosen because it is cheap and complete. Moreover all STM32 devices are well supported by guides, tutorials and a very active community.
- **Eclipse Neon** This is a well-known IDE written in Java, complete, free and easy to use.
- **AC6 workbench for STM32** This additional package for Eclipse contains the cross compiler *arm-none-eabi*, a debugger for STM32 devices and the code of FreeRTOS; it can be downloaded from an Eclipse repository maintained by ac6-tools.com.
- **STM32F3 HAL** Hardware Abstraction Layer have been used as they partially remove the complexity of low-level programming and allow to speed up the development.
- **FreeRTOS v9.0.0** Lightweight, complete and open source Real-Time Operating System specifically designed for embedded systems with poor resources.
- **EEMBC® Automotive suite v2.0** Set of official benchmarks provided by EEMBC® containing different algorithms used in automotive field. This suite has been chosen because it provides benchmarks that are widely used as standard programs for performance tests under UNIX/Linux system; they have already been ported to other minor hardware platforms.
- **Python 2.7** Interpreted scripting language developed by Guido van Rossum. It has been used to develop the host-side injection manager and the parser used to analyze the produced data after each injection campaign. Three additional packages are required for these scripts to work: *numpy*, *matplotlib* and *serial*.

Chapter 2

Physics of Single Event Effects

2.1 Introduction

A *Single Event Effect* (SEE) is the electrical noise induced in a circuit by a natural phenomenon that is external to the circuit itself. SEEs are caused by *ionizing particles* coming from space or Earth which collide with electronic devices: they may come from deep space (cosmic and gamma rays), Sun (solar wind), magnetosphere (van Allen belts) and Earth's crust (from naturally radioactive materials). Those ionizing particles can be both massive (heavy ions, protons, neutrons, electrons) or massless (photons); furthermore, the incriminated particles causing SEE can directly impact onto the device or they can create a secondary cascade of particles when entering the atmosphere.
SEEs are dangerous because they may lead to misbehaviors or ruptures and their incidence becomes even bigger because of the continuous scaling of technology. Their severity can be analyzed taking into account the consequences of their impact on the circuit and the way in which its mission is harmed. The consequences of a SEE and the probability that a phenomenon has visible effects on a circuit can be estimated by reproducing the phenomenon with testing techniques described in the next chapter (physical, logical or simulation-based injection), then, with the results of these tests it is possible to know which are the most sensitive parts of the system, to classify the misbehaviors and eventually to develop new methods to solve or at least to reduce some vulnerabilities.
Notoriously, on-board electronics used in avionics and aerospace are the most subjected because they work at high altitudes or even outside the atmosphere, where there is no protection against these phenomena, even though it is thought that also many sudden failures in electronic devices working ashore (consumer electronics, industrial applications, automotive circuits) are caused by ionizing particles that are able to reach the ground.

2.2 Cosmic rays

Cosmic rays are one of the first causes of SEEs in electronic devices and for this reason they have been analyzed in detail since 60s . Usually their origin can be studied identifying two actors: a particles source and an acceleration mechanism which actually makes them dangerous for electronics; such particles originate in supernovae, at the center of galaxies or in neutron stars with a high rotational frequency like pulsars. It is possible to identify two types of cosmic rays: primary and secondary.

Primary cosmic rays
Primary cosmic rays are composed by all those high energy particles coming from near or distant astronomical objects which interact with the electronics and directly cause SEEs; they are able to reach the atmosphere unperturbed, passing through its top part where the air density is very low (over ~40km of altitude) without being subjected to any major interaction. Primary cosmic rays are composed mainly by electrons and protons but sometimes nuclei of heavier elements can be observed too.

Secondary cosmic rays
Secondary cosmic rays are generated by the interaction of high energy particles belonging to primary cosmic rays with substances in the atmosphere, leading to the production of a cascade of other particles like pions, muons and kaons; 80% of all particles reaching the 0m level of altitude (sea level) is made of muons, which have a flux through a horizontal surface of 1 particle per $1cm^2$ in 1 minute.

2.3 SEE Classification

SEEs are classified according to the damage they cause and to the type of harming mechanism they induce. JEDEC standards identify many classes of SEE but in the following only the most relevant ones are described.

2.3.1 Destructive SEE

A destructive SEE (called *hard error* too) causes a permanent damage to a circuit: good solutions adopted to reduce or totally remove the impact of such error are based on adding some redundancy in the system at hardware level (system-level, block-level or gate level) or at software level (data redundancy, SEC-DED algorithms). If even the added redundancy does not have the expected protective effect and device breakage is not avoided, only its physical substitution can solve the problem.
Most relevant destructive SEEs are: Single Event Latchup (SEL), Single Event Burnout (SEB), Single Event Gate Rupture (SEGR) and Single Event Hard Error (SEHE).

Single Event Latchup (SEL)
Latchup is a type of SEE that can be triggered in any CMOS-based circuit because of its intrinsic structure: in fact, as depicted in figure 2.1, it is possible to spot a parasitic thyristor among the various wells. In this configuration, if a high energy ionizing particle goes through the n-well of the PMOS region, a high number of electron-hole pairs are generated, creating a flow that dissolves the charges in order to go back to equilibrium: the current follows the path from the n-well to the p-substrate, but this leads to a rising current flowing through the base of the parasitic PNP bipolar transistor which causes its collector current to rise and so the voltage across the resistance R_p to increase. This causes an increment of current flowing in the base of the NPN bipolar transistor which brings its collector current (a fraction of the same passing in the PNP gate) to rise again and provoking a loop mechanism that destroys the device. In some cases latchup is not necessarily destructive.

Single Event Burnout (SEB)
This event happens when a high kinetic energy particle strikes the device generating a high localized current spike: if this amount of current is not tolerated, the device breaks;

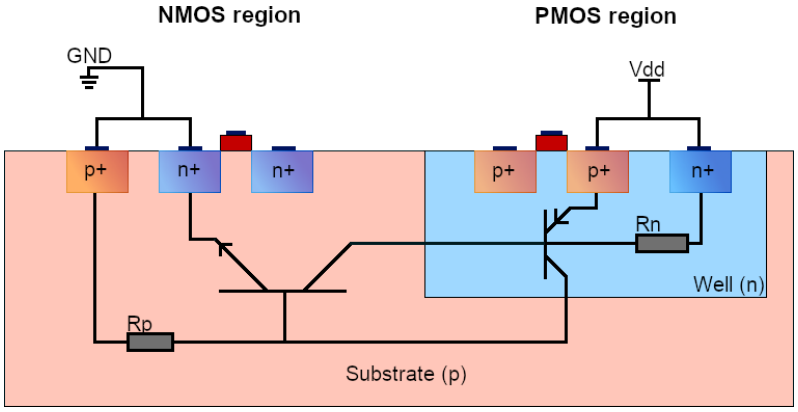

Figure 2.1: Latchup parasitic circuit

in this case breakage is due to a current exceeding the device absolute maximum ratings and not to internal mechanism triggered by the SEE. It may be caused by heavy ions, protons and neutrons.

Single Event Gate Rupture (SEGR)
Gate rupture is caused by heavy ions, protons or neutrons impacting against the device and thus creating a current path through the gate of a MOSFET; the gate is permanently damaged as consequence of this event, leading to a slightly higher current drained from the power line. Moreover the effect may vary according to the angle of incidence of the ion.

Single Event Hard Error (SEHE)
This error affects mainly SRAMs and it is manifested as stuck-at bit in the memory or in other parts of the device. It is thought that this effect is due to the deposition of a large charge in some isolated locations of the circuit which interacts statically with the surrounding integrated elements and that is not able to dissolve.

2.3.2 Non-destructive SEE

A non-destructive SEE (called *soft error* too) causes a transient error in the system that can lead to silent or catastrophic misbehaviors but it doesn't damage physically the circuit. The error is surely removed with power cycling. Most relevant non-destructive SEEs are: Single Event Upset (SEU), Single Event Functional Interrupt (SEFI) and Single Event Transient (SET).

Single Event Upset (SEU)
This is one of the most studied SEE because it is very common: it is caused by the interaction of a particle with a memory element and its consequent change of state (known as bit-flip too). Most important types of SEU are Single Bit Upset (SBU) and Multiple Bit Upset (MBU).

Single Event Functional Interrupt (SEFI)
This SEE can be detected when the device resets or locks-up indefinitely because of a change in an internal register: in some cases this event is classified as subset of SEU since it is usually caused by a flip in one or more bits in some control registers of the device.

Single Event Transient (SET)
This event is caused by a momentary voltage spike in a precise position of a circuit caused by a sudden event like a high energy particle hitting the device or a strong electromagnetic interference with a near source. If such transient value of the signal is retained by a memory element, a misbehavior could be seen.

Chapter 3

Fault injection

3.1 Dependability

Dependability is the property of an application to perform operations as expected, whenever it is required.
Dependability is made of *attributes*, *threats* and *means*. Attributes are a set of properties that describe the system from the dependability point of view; threats instead are those events that induce the system to not operate as expected; means are all those techniques and methods that a designer can exploit to avoid that faults cause system failures or at least to mitigate errors. Following concepts are intensively used when talking about dependability.

- **MTTF** Mean time to failure: if the system must work correctly for a very long time, it is important to consider this quantity. It represents the average time between the system start up and the instant when the first failure occurs.

- **MTTR** Mean time to repair: it is the average time required to an operator to repair the system, starting from the detection of the problem.

- **MTBF** Mean time between failures: this quantity expresses the average time between two consecutive failures.

- **Failure rate** This quantity is indicated with λ and is expressed in [FIT] (Faults In Time). It is used as parameter by the failure model adopted.

$$1FIT = \frac{1}{10^9} \quad \left[\frac{numberoffailures}{hours}\right] \tag{3.1}$$

3.1.1 Attributes

Dependability has many attributes, but the most important ones are 4, usually pointed with the acronym RAMS (Reliability, Availability, Maintainability and Safety). With the growth of the electronic industry and with the constant request of increasingly dependable systems, new attributes have been introduced in the last decades, like Security, Confidentiality, Integrity.

Reliability
Reliability is the probability that a system behaves correctly until a time t, starting from t_0 and assuming it worked well until t_0.
This attribute can be expressed analytically by a probability model that well describes the behavior of the system with the passing of the time. Such analytical function is continuous and real and it has two or more parameters: the independent variable (time), the failure rate and eventually other parameters required by the model.

$$R(t) = f(t, \lambda, ...) \tag{3.2}$$

Availability
Availability is the probability that a system behaves correctly at a generic time t. It is just the ratio between the time during which the system was available and the total time the system was active for (available or not).

$$A = \frac{up\_time}{up\_time + down\_time} \tag{3.3}$$

Maintainability
Maintainability is the probability that a broken system can be fixed before a time t. In this case one needs to know if the system is accessible to the operator (physically or remotely), if the failure can be repaired and only then one can estimate the average time required for the fixing.

Safety
Safety is the probability that a system correctly works or stops working without harming anything and anyone when a problem occurs. It is very difficult to estimate analytically this attribute because it is strictly related to the system itself, its specifications and the applicative field it was designed for. Safety can be expressed by words defining some behaviors the system must adopt when problems occur and then testing if it acts in the expected way by forcing those problem in the application. It is evident that in this case fault injection testing techniques are very useful.

3.1.2 Threats

Threats are all those events that lead a system to not operate as desired. They have several possible causes, be they internal or external to the device, intentional or accidental, permanent or temporary. Threats have a life cycle and for this reason three different stages are identified. When a phenomenon excites a particular combination of events in the device, it is possible that a part of the system shows a local behavior different from the expected one: in this case we talk of *fault*, whose effect is still restricted to a small area in the circuit and it is still impossible to know if such event will propagate in the circuit; if this happens and so the fault leads to a temporary change of the state of the system, such fault has turned into an *error*. If the error affects those parts of the system that are directly connected to the output, then the system explicitly shows a behavior different from the expected one: the error has become a *misbehavior*, called also *failure*. The possibility to observe a misbehavior on the output depends not only on the circuit itself but also on the combination of values applied to the input and on previous values stored in eventual memory elements.

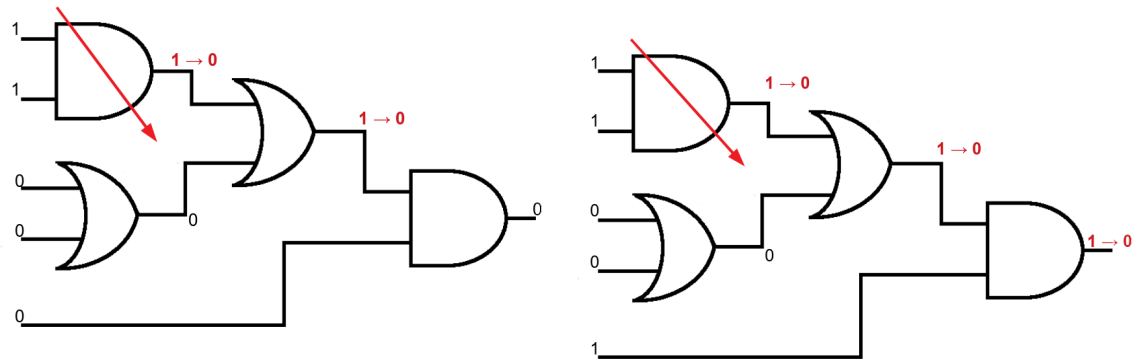

Figure 3.1: Examples of fault excited by a high energy particle in an AND gate. On the left the fault does not turn into a misbehavior while on the right a failure of the circuit can be observed.

3.1.3 Means

Dependability of a system can be increased by exploiting some techniques which reduce the effect of some faults or, in the best case, make the application completely insensitive to them. These methods are divided in 4 categories: fault prevention, fault removal, fault tolerance and fault forecasting.

Fault prevention
Fault prevention encases all those techniques aiming to strengthen the system so that it is less probable a fault is excited. An example of fault prevention method is the usage of hardened technology during the fabrication of a device: usually, consumer electronics is less stable, less safe and more prone to failures than other applications because a high dependability is not required. On the contrary, in other fields, it is fundamental to have a system which does not fail easily and for this reason new advanced (but expensive) technologies are used to implement the circuit. Radiation hardening is, for example, a set of techniques used extensively in avionics to reduce the impact of radiation and high energy particles on the electronics.

Fault removal
These methods require that the system contains a unit able to perform a status check of the system itself and eventually to remove the fault, possibly before it becomes a misbehavior.

Fault tolerance
In some cases it is mandatory that the system continues its operations, even if it has been degraded as result of a fault. A widely used solution to tolerate errors (so to identify, to correct them or both) is the implementation of redundancy, which can be done at different levels. Redundancy at system level is the most expensive solution but it is still commonly used in some safety-critical application where a secondary or even a tertiary computing system could be lifesaving in case of critical errors. Redundancy at hardware level (consisting in the implementation of more identical devices in the same system or units in the same device) is very efficient too and it ensures that the system works well in many cases even if a problem occurs, but it could be expensive and, very often, other relevant parameters like area and power consumption are affected negatively. Redund-

ancy can be implemented at software level too, duplicating variables, implementing error detection and possibly correction codes, doing stricter checks on system data.

Fault forecasting
This set of techniques requires to do an analysis of the application to spot the most sensitive locations and then to statistically estimate how many errors will occur and in which parts of the system.

3.2 Architecture of a fault injection system

When a product reaches the end of a specific design phase, the end of production, when it is pretty old or it must be tested constantly to ensure it works properly, it is mandatory to develop systems which can automatize the main testing operations and to check that all the desired features of the application are still present. This scenario shows how relevant testing operations are during the development of a new application: it has been estimated that companies operating in the world of electronics spend more or less 40% of the budget to perform all these tests, so testing has become not only an operation that must be done obligatorily to ensure that every single product has the required quality but also a big expenditure whose burden should be reduced as much as possible.
Aside all the common types of test a new application can be subjected to - like validation of specifications, design verification, verification testing, production testing (also called end-of-manufacturing test), burn-in and incoming inspection - a particularly interesting one is the the robustness test which usually exploits fault injection techniques to analyze the possible consequences of SEE on an electronic device, in order to characterize the system safety end eventually its availability and reliability.
When designing a new testing system for robustness, it is important to define a set of features that allow to setup the fault injection environment in an efficient way. All testing systems must work defining a set of fault lists, each one containing a set of faults where the injection has to be performed; then the engineer has to define one or more fault models which can efficiently simulate the physical phenomenon harming the hardware or software. Finally the architecture of the testing system must be designed: it is important to use two separated actors, a host machine and device under test, which communicate constantly; such separation is necessary because some injections could lead to unexpected repercussion on the testing system itself and if they are not well isolated one from the other, injection system could fail, producing wrong results. All these aspects are described in detail in the following sections.

3.2.1 Injection technique

Fault injection testing systems can perform the injection in different ways: if it is important to reproduce precisely the physical phenomenon which awakens a fault with a possible consequent failure, a physical approach can be used; however, nowadays, less invasive techniques are preferred: physical injections do not give accurate information about the consequences of the test. The three possible approaches are described below and table 3.1 summarizes their pros and cons.

Physical testing
A proper laboratory instrumentation like a LINAC (LINear ACcelerator) or a laser-based injector and a vacuum chamber is required; the cost of such equipment is very high and if

the hardware is an instruction set architecture executing a code and accessing a memory, it is quite difficult to inject in specific data both in the memory and in internal registers. Moreover, if the layout of the circuit is not known, it is necessary to reverse-engineering it to understand how the various parts (caches, registers, logical and arithmetic units, peripherals) are arranged on the die. The circuit is put inside a vacuum chamber so that the accelerated ions do not impact on other molecules present in the environment, then it is positioned under a pointing device; as soon as the accelerated particles reach the desired energy, the LINAC begins to hit the location pointed on the die. Results can be extracted at runtime or when the injection has finished. Usually random injections campaigns are performed and then results are extracted using statistics. This technique has been mainly used to characterize the behavior of simple devices like CMOS-based, logic circuits, FPGAs or RAMs.

Logical testing
It can be hardware-based or software-based. In the former case the circuit must include internally a fault-injection unit that can be enabled when a test has to be run; moreover a specific communication protocol must be implemented so that an eventual external machine can send injection parameters to the device under test. Such technique can be applied in a totally different way too: if the engineer is interested on the effect of external phenomena on the propagation of signals on a board, he can use an automatic system based on probes or nails able to force known values on traces of a PCB. In the first case a relevant overhead in terms of area is added while in the second case the cost of the test equipment could be prohibitive.
In the case of software-base injection (often referred to as SWIFI), instead, a software that has low-level access to the various parts of the system is launched and then it injects in the desired position. This technique leads to a real injection as well, but the timing of the application changes because of the time required by the injection routine to run.

Simulation
Adopting this solution, the test is not done using the hardware but the effects of the SEE are simulated using a description of the circuit and a simulation tool. This technique does not require the physical circuit to be tested but only its hardware description; moreover, it allows to test hardware very well but, in the case of software robustness testing using a bit flip model, it is still impossible to inject in a memory location containing the data to flip because usually, even in smaller instruction set architectures, it is difficult to identify deterministically such memory cell.

3.2.2 Fault model choice

A fault model is the model of a real fault caused by one or more physical phenomena using a simplified description, so that a testing system can easily reproduce the effect of the fault itself. For example, the *stuck-at* fault model can be used to model the effects of some physical events like electromigration, short circuit paths in the device between a signal line and a power supply line, electromagnetic interference across too close interconnections, gate rupture and so on. Another type of fault model is the so called *stuck-on*, which expands the descriptive power of the stuck-at, considering the circuit from the point of view of the single transistors. With the increase of the working frequency, *path delay* fault models have become also very important as they allow to describe erroneous events due to a too small clock period: this model can be used to characterize the various paths in

Technique	Advantages	Disvantages
Physical	- Physical phenomena causing the SEE are replicated - Possibility to physically inject in well-defined parts of the die	- Very high cost - Expert personnel required to use the equipment - Impossibility to inject in specific data - Necessary reverse engineering of the die if its layout is unknown
Logical Hardware based	- Very precise hardware-related injection - Good simulation of the physical phenomenon - Device under test not affected in terms of executing time by the operations of the injecting unit	- Implementation of an additional hardware unit - Definition of a communication system with a machine controlling the injection experiment - Eventually, additional pins to be added on the board required - Eventually, high overhead in terms of area
Logical Software based	- Very precise software-related injection - High level of freedom during the development of the injecting system - Low cost - High portability of the system: it is sufficient to adapt the code to the architecture of the microcontroller or microprocessor	- Relevant overhead in terms of code size added - Relevant overhead in terms of additional executing time
Simulation based	- Absolutely not invasive - No hardware resource needed (besides a computer to run the simulation) - High level of freedom during the development of the injecting system	- HDL description of the circuit rarely available - Cost of the simulation tool can be relevant

Table 3.1: Summary of injection techniques

a circuit or to simulate the presence of too slow paths in the hardware using a software approach.

3.2.3 Fault injection space

Another important thing to be defined is the injection space which is related to the injection technique adopted. It is a multi-dimensional space: a good example can be made of injection time, injection place and injection type, but many other parameters can be added.

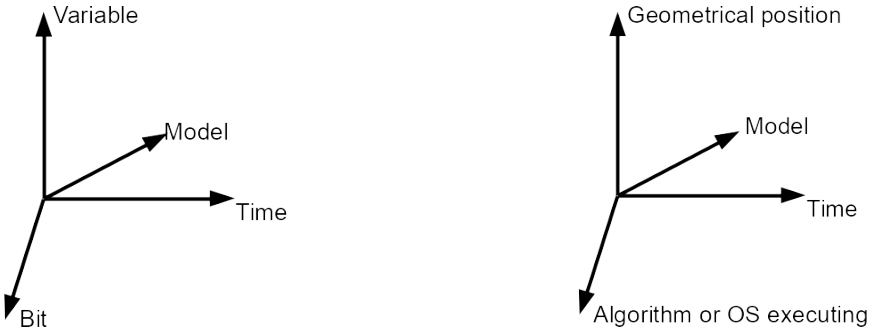

Figure 3.2: Diagrams of two possible injection spaces: on the left, an example of software-based injection in memory; on the right an example of physical injection in a die: geometrical position is actually a *(x,y)* tuple.

3.2.4 Fault lists definition

A fault list is a list of places, where the injection has to be done. Faults can be geometrical position in a circuit die (for physical tests), units in the integrated device or geometrical coordinates on a PCB (for hardware-based tests), variables and data in the memory (for software-based tests), signals values or memory elements content (for simulation tests): fault lists must be defined according to the injection technique adopted. It is used by the testing system to automatize the process and it can be built automatically or by hand. When the system under test is big and the number of faults in the fault lists is high, a process of *fault collapsing* must be done: it can be made according to some logical or boolean rules or analyzing the system; in fact, in the case of software test, some variables to inject in could have no direct consequence on the dependability of the application if they change their value (like debug variables and strings used for the user-machine communication). Fault collapsing allows to decrease the length of the fault list reducing test time.

3.2.5 Communication

An efficient communication mechanism must be used so that the host-machine can control the DUT. Such system can be developed *ad-hoc* and inserted in the device or on the board or it can be based on an existing unit or peripheral. An example of *ad-hoc* communication system is the JTAG protocol which must be appositely inserted in the device: such solution requires the whole design of the circuit to be reviewed so that a debug unit and eventually scan registers can be added; very high speed communication (and in some

cases even real-time tracing of the execution flow) between DUT and host is guaranteed. A communicating system based, instead, on an existing resource does not require any modification of the circuit; the only drawbacks are due to the fact that the used peripheral cannot be used by the application.

Chapter 4

Real-Time Operating Systems

4.1 RTOS common features

A Real-Time Operating System is an operating system able to perform desired operations in a precise amount of time, respecting well defined deadlines. In some cases, system reactivity and calculation times are very important for the whole application to work according to the specifications. Moreover, in safety-critical applications, it is important to avoid such deadlines to be violated: if this happen, the system could injure more or less severely other systems, people and objects.
When the application can tolerate some timing violations without messing up itself or the environment, we talk about *soft real-time* systems: an example is the keyboard input processing in a computer, as a quick response is preferred but not mandatory; on the contrary, if a missed deadline causes severe damages to itself or to the surrounding environment, we speak of *hard real-time* systems: an example is the real-time mechanism that injects fuel in the cylinders of an engine; in this case violations must be handled properly, in order to avoid as much as possible dangerous consequences. It is common to have, at the end of the system design, a mix of hard real-time, soft real-time and non real-time behaviors.
RTOSs are able to schedule concurrent operations belonging to different contexts in the form of *tasks* (or *processes*) and to switch among them in such a way that desired timing is respected. Each task can be in a defined state in every moment of its life and the programmer can partially choice how and when a task must change it; usually all operating systems (not only RTOSs) recognize three states for each task: the *ready state*, when the task is ready to be scheduled; the *running state* when the task has been switched in and it holds the core of the processor; the *waiting state* when, instead, the task is waiting for an event to happen. Sometimes two additional states are added and they are the *new state*, used to identify tasks that have been just created and the *deleted state*, when a task must be removed and it is waiting for the kernel to clear its stack and to free all the memory associated to it.
With the expression *context switching* we mean the mechanism that allows to run code from different contexts, and so to assign the core to tasks so that all of them are executed and their data are not mixed. A general scheme of states and transition mechanisms, valid for both common OSs and RTOSs, is shown in figure 4.1. However, in the case of RTOSs, when the system to be setup is complex, some calculations could lead to not satisfy time boundaries because of problems like loss of synchronization, deadlock, starvation or just because WCET (Worst Case Execution Time) for one or more portions of code has been

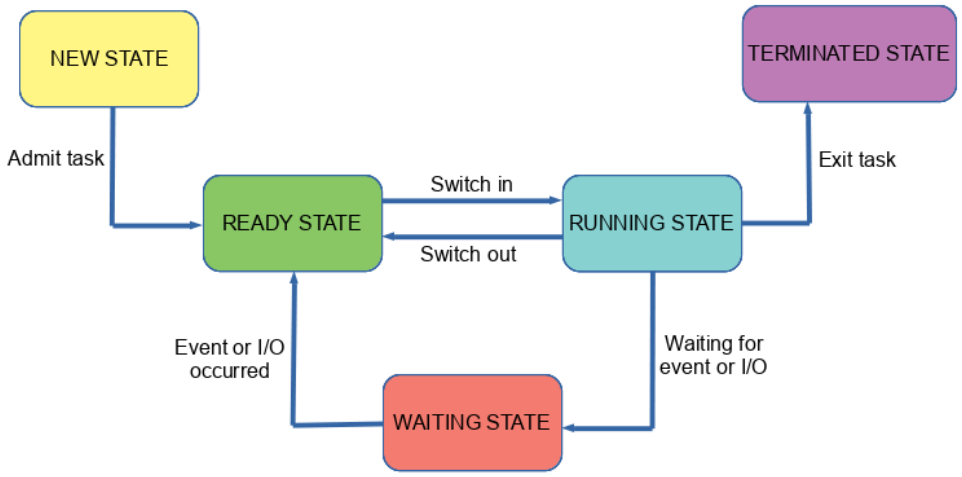

Figure 4.1: State transition diagram for a generic OS

miscalculated: in any case, it is the duty of the programmer to handle properly these events and to implement recovery mechanisms.
A RTOS can ensure the correct scheduling in real time by providing, mainly, three features: a real timing system, an efficient scheduling policy and possibility to delay or to stop execution of some tasks for a defined amount of real time.

4.1.1 System Tick

It is fundamental for a RTOS to have a real time reference able to keep the time for many operations made by the kernel and by the tasks: such element is a hardware timer and it can be a general purpose or a specific one, appositely inserted in the architecture as reference for the OS; this hardware unit can generate an interrupt on a regular basis, called *system tick* or not, according to the type of timing system the OS uses: in the first case we talk about *ticked* systems while in the second case such OSs are called *tickless*.

Ticked systems
System tick raises an interrupt on a regular basis and lets its handler to be executed: a check is performed on all the instantiated tasks to see if a context switch must be done and, if so, it switches out from the core the current operation and choices, if available, another one to be executed, according to the scheduling policy. A particular attention must be payed when defining the period of the system tick: in fact, if it is too high, the system could be not as fast as desired, producing slow responses to events that require a high reactivity; on the contrary, if this value is too low, the overhead due to the switching routine could impact negatively on the performance of the system. A value comprised between 1ms and 100ms is usually a good starting point to tune this parameter. Figures 4.2 and 4.3 underline negative aspects of a bad tuned system; in both cases the RTOS scheduler uses a round-robin policy. Anyway, it is important to highlight that the system tick is not the only way to produce a switch between operations, since other events can cause it, even if the tick didn't occur yet: examples are the mechanism used to move tasks from the ready state to the waiting state or the situation in which a task setups a

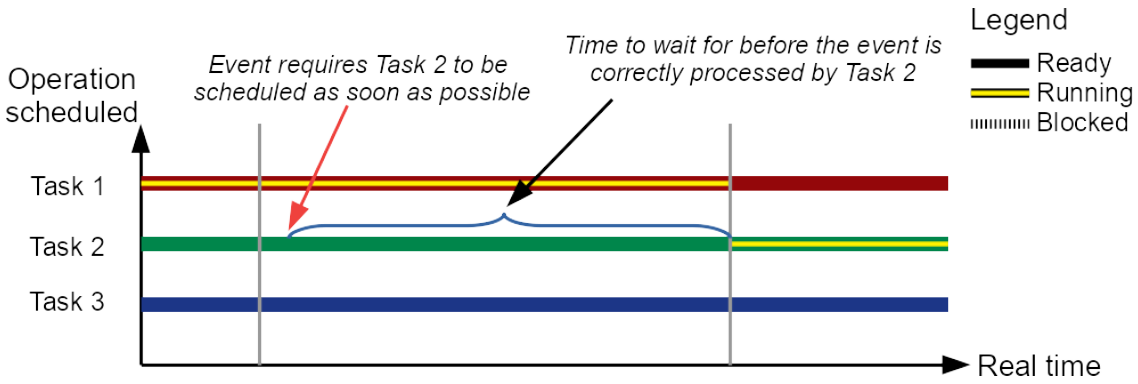

Figure 4.2: RTOS set with a low frequency system tick; its very low reactivity is highlighted.

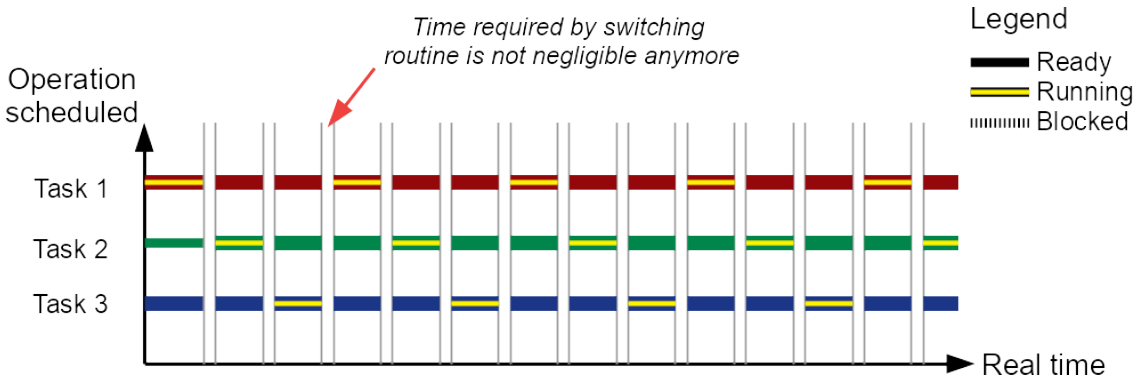

Figure 4.3: RTOS set with a high frequency system tick: time required by switching routine to perform operations is not negligible anymore if compared to useful time dedicated to real tasks.

timeout to wait for a busy resource.
In some cases, in ticked RTOSs, it can be comfortable to stop the tick generator when the core is idle and there is no task to be instantiated because they are all blocked/waiting. In this low-power mode the scheduler will be awakened by the first task entering again the ready state.

Tickless systems
Tickless systems can have a very low overhead during the execution because the context switch is done only when a new deadline is encountered: every time a task goes in a blocking/waiting state for a specified amount of time, it is added in an internal ordered queue with all the other processes waiting for a time-related event; then the hardware timer is set so that an interrupt is raised when the timeout occurs: the interrupt handler will perform the context switch. These RTOSs are more error prone because the programmer must pay more attention when defining deadlines and wake/sleep times for each task but kernel procedures are called only when needed and not on a regular basis.

4.1.2 Scheduler

The scheduler of a RTOS is that part of the kernel that must assign the core to the various tasks in turn, following the chosen scheduling policy. There are two possible approaches:

Preemptive scheduling This approach allows to switch the current task out even if it can still execute. This allows to give the core in a more fragmented way and ensure that, if the system is well designed, all instantiated tasks are executed. Prioritized preemption is a particular type of preemptive scheduling that exploits in a strict way the assignment of a priority: every time a new task is created, it is assigned a priority and then the scheduling is done taking in consideration such value. In general, when a task with higher priority with respect to the current task priority is ready to be executed, the latter is switched out and the core is given to the the former. Prioritized preemption is very useful when there are some tasks needing a higher attention to be executed primarily with respect to other less important ones.

Cooperative scheduling In this case, the core is assigned to a new task only when the current one terminates, when it goes in a blocked/waiting state or when it voluntarily switches out and so forces the scheduler to find a new candidate. If execution time required by all the calculations made by the various tasks can be properly estimated and a high level of freedom in the organization of the processes execution is preferable, the cooperative approach can be used.

Following figures represent 5 possible configurations of the RTOS scheduler: figure 4.4 shows the case of a tickless cooperative one - in this case no system tick is implemented; the programmer must calculate correctly the WCET of each task and then it must insert in all of them, in the correct position, a yield operation that allows the other tasks to run. Figure 4.5 instead is the case of a prioritized tickless cooperative RTOS where priority is used to choice the task to be run when the previous one yields the core. Figure 4.6 represents a scheduler with a defined custom algorithm - in this case scheduling is made at every system tick; many possible policies can be adopted, like round-robin, EDF (Earliest Deadline First), FCFS (First Come First Served) and so on. In figure 4.7 instead the scheduler is a prioritized preemptive one but without time slicing - with this solution the scheduler, at every system tick, tries to find one or more tasks with a priority higher than the current task priority and, if it exists, then it switches to one of them. Finally, figure 4.8 represents the behavior of a prioritized preemptive scheduler with round-robin time slicing.

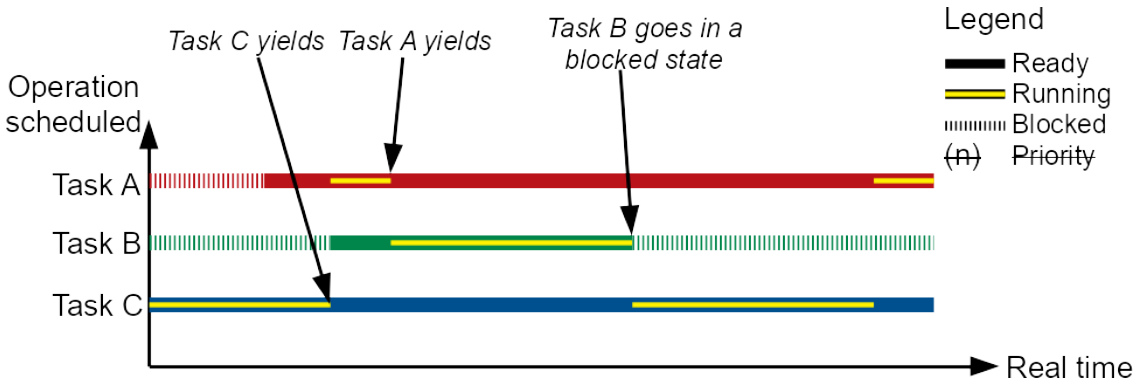

Figure 4.4: Behavior of a tickless cooperative RTOS: when a task releases the core or it goes in blocked state another ready task is scheduled, if available.

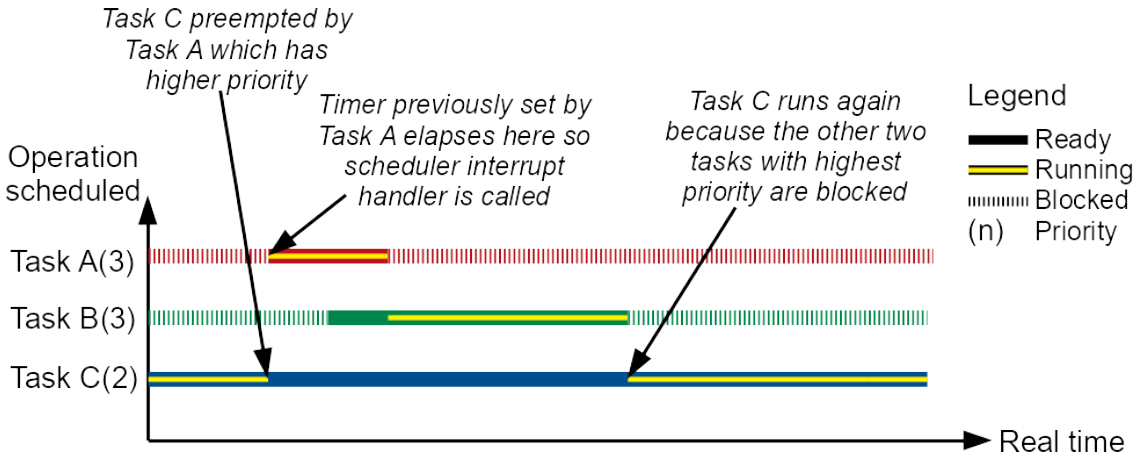

Figure 4.5: Behavior of a prioritized tickless cooperative RTOS: when a task releases the core or it goes in blocked state, the ready task with the highest priority is scheduled, if available.

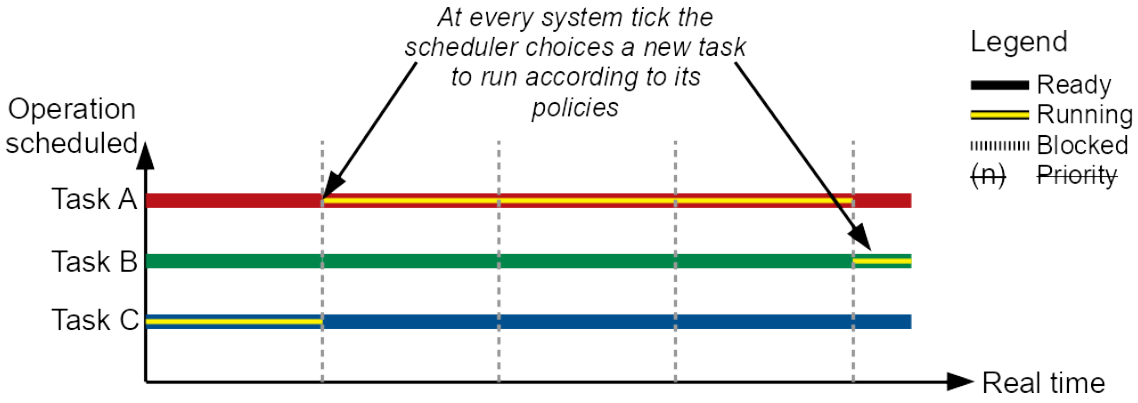

Figure 4.6: Behavior of a ticked RTOS with generic preemption: at every system tick a task is scheduled using a defined algorithm.

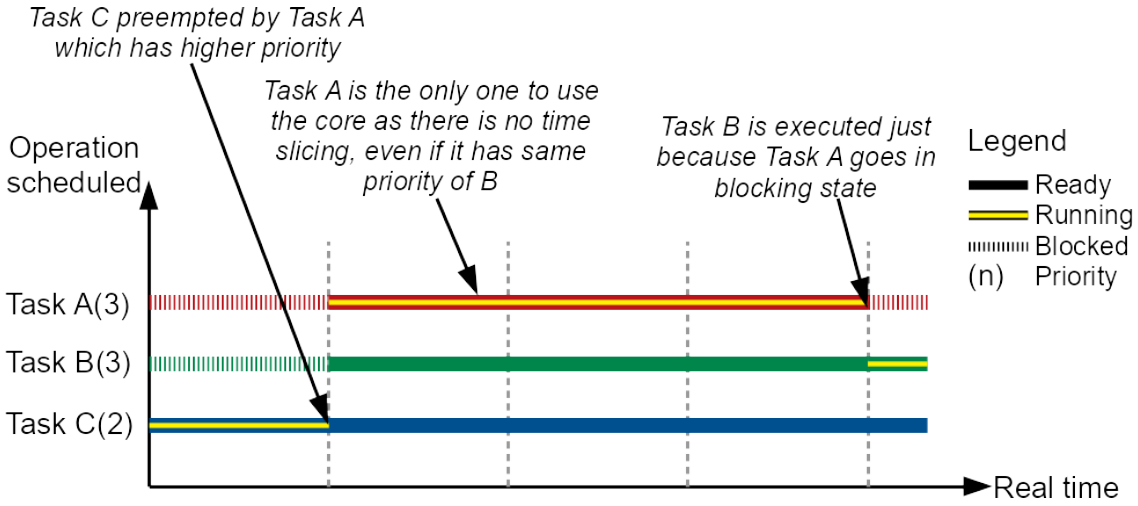

Figure 4.7: Behavior of a ticked RTOS with prioritized preemption but without time slicing. At every tick interrupt, if a new task with a priority higher than the current task priority is available, such task is switched in.

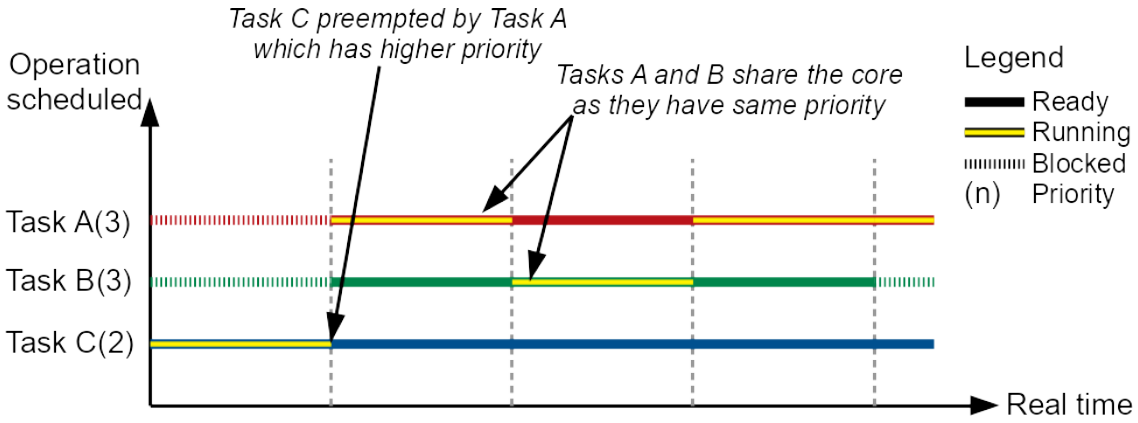

Figure 4.8: Behavior of a ticked RTOS with prioritized preemption and time slicing. At every tick interrupt, if a new task with a priority equal or higher than the current task priority is available, such task is switched in.

4.1.3 Execution delay and timeouts

In a common OS, in principle, it is possible to wait for a lot of time before an event occurs or a resource is released but in the world of RTOSs, as deadlines are often peremptory, it is important to define timeouts to avoid the system to wait for too much. Furthermore, delaying the execution of a portion of code of an arbitrary amount of real time could be convenient in some cases. For these reasons RTOSs provide some facilities like retardation of the execution of a task, suspension of the task and additional timeout parameters to be defined by the programmer for waiting operations.

4.2 FreeRTOS

Following section is related to FreeRTOS v9.0.0.
FreeRTOS is a Real-Time Operating System written in C and developed for embedded systems where other OSs won't fit the memory. It is a good choice also for its completeness, for the support provided by the community, the abundance of tutorials and guides and for its license: in fact, FreeRTOS is released under the MIT Open Source license which gives it gratuitousness and the possibility to use it in commercial-use applications, with no need to document it. However it is not covered by any warranty and special support: to obtain them, a commercial version called OpenRTOS™ must be bought. There exists another version called SafeRTOS™ that implements additional features like memory protection, forbids dynamic memory allocation and requires always deadlines to be specified in waiting operations in order to avoid infinite stalls or deadlocks.
To have a full comprehension of this OS, please consult the official FreeRTOS reference guide [12].

4.2.1 FreeRTOS properties

Before proceeding, FreeRTOS nomenclature is provided.

Application This term indicates the whole system running on the hardware platform, with the operating system properly set and all the tasks instantiated: each FreeRTOS project can be referred to by this term.

Task In FreeRTOS a task is the basic element to be executed in parallel with other ones. A FreeRTOS task should be thought more like a UNIX/Linux process than like a thread as they are not supported by this RTOS: each task has its own stack, it can access to the common heap and is not dependent on a father task.
Task Control Block The Task Control Block (TCB) is a data structure initialized for each task every time one is created. It contains all the necessary information about the task status, its stack position in the memory, mutexes and few other fields that can be used for debug.
Task Handle A task handle is a variable pointing to the first element of a TCB in the memory. It is used to address the task when necessary or to distinguish various tasks when doing some debug operations.
Idle task This task is always created with the priority set to 0 (the lowest available): when no other task is ready to run, this one is executed. It consists of just a loop, calling eventually a yield function that allows to force a switch, whenever the list of ready tasks is not empty. It is a good practice to not instantiate other tasks with 0 as priority.

From a technical point of view, FreeRTOS is a Real-Time Operating System with three possible scheduling policies, to be chosen at compile time: prioritized preemptive with time slicing (the most used), prioritized preemptive without time slicing and cooperative. In all these cases the system is ticked, even if it is possible to turn off the system tick source when the *tickless low-power mode* is activated: when the programmer already knows that FreeRTOS will pass a lot of time in the idle state with no task ready to be scheduled, it is possible to suspend the system tick until the next task to be awakened reaches the timeout. In this way the core can be put in a low power mode so that the interrupt handler is not executed at every tick produced by the timer.
Tasks can be in running, ready, delayed and suspended state; figure 4.9 shows the state transition diagram for this RTOS.

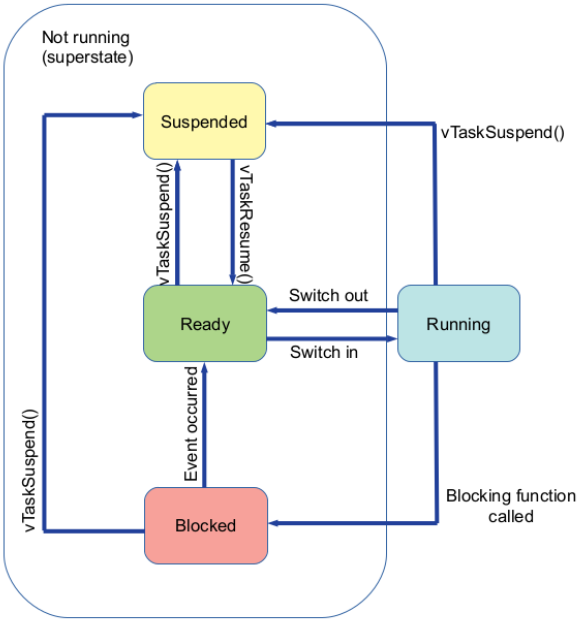

Figure 4.9: FreeRTOS state transition diagram

4.2.2 FreeRTOS files

The whole operating system is contained in few files: the most relevant ones are just 11 and each of them contains code for a fundamental part of the whole system; their content is described below.

FreeRTOSConfig.h
This file is the first thing to look at when approaching FreeRTOS for the first time: it contains some *#define*s that allow to adapt the behavior of the system to the programmer's needs. Every additional *#include*d header file that points to some modified parts of the system or that implements some features which are not provided by default must be added at the very end of this file.

FreeRTOS.h
In this file different parts of the system are made available or hidden to the programmer according to the settings written in FreeRTOSConfig.h file. Moreover, here one can find many definitions of dummy structures that are related to real objects used internally by the system: in order to avoid that the programmer accesses sensitive data, such dummy structures are used to obfuscate properly the code.

task.h / tasks.c
These two files (especially tasks.c) contain the kernel of FreeRTOS. In order to avoid the programmer to access sensitive data, they have a slightly different name.

queue.h / queue.c
In this file the structure of the queue is defined and many queue management functions are provided. It is important to note that this structure is used for semaphores and mutexes too, with some modifications.

list.h / list.c
In this file all functions related to the list structure are defined and implemented: all the common operations on lists can be done by calling such functions, which are available to the programmer.

semphr.h
This file contains some macros that allow to reuse the same functions and definitions of the queues for semaphores and mutexes: they are wrappers that call queues functions with predefined parameters, according to the type of object to be used.

port.c
In this file there are some routines and functions written at low-level strictly related to the hardware that is going to be used to run FreeRTOS. One can find here implementation of functions that allow to start the scheduler, to call the first task when the OS is ready to run, to end the scheduler and so on.

heap_*n*.c
FreeRTOS is provided with 5 heap management policies, each one contained in one of these files. Usually the number 4 is the preferred one.

4.2.3 FreeRTOS setup

One of the first things to do when designing a new FreeRTOS application is to define how the kernel must operate, which are the additional features to activate and setup the memory management. As already said, all these aspects can be set from the file *FreeRTOSConfig.h* by changing the values of the *#define*s; most important ones are reported below.

- **configUSE_PREEMPTION** If set to 1 the scheduler will act using a prioritized preemptive algorithm; if set to 0 the scheduler will be a cooperative one.
- **configUSE_TIME_SLICING** If set to 1 the scheduler will use time slicing, sharing the core among two or more tasks with the same highest priority; if set to 0 it will not act in this way.
- **configUSE_IDLE_HOOK** If set to 1 a custom function is called by the idle task every time it is executed. Such function is already defined as a hook but it must be implemented by the programmer: this is the best place where to put a sequence of operations to put the microcontroller in a sleep mode in order to save power. If set to 0, such hook is not available.
- **configUSE_TICK_HOOK** If set to 1 a tick hook is made visible to the programmer: similarly to the idle hook, the tick hook is a prototype function provided by FreeRTOS that must be implemented and that is called every time a tick occurs. Using this feature brings some advantages but could slow down the switch operation. Again, if set to 0 this feature is not available.
- **configTICK_RATE_HZ** Here the period of the hardware timer can be chosen. It is set to 1000Hz by default.
- **configMINIMAL_STACK_SIZE** Every time a new task is created, it is instantiated in the memory with a stack size as big as defined by the programmer at compile time. However, in order to reduce problems during the run, with this variable it is possible to set a minimum size for this memory region for all the tasks.
- **configTOTAL_HEAP_SIZE** FreeRTOS normally uses some dynamic memory functions to initialize idle task and few other features; however, if user code uses dynamic memory too, heap is exploited: the total amount of available memory treated as heap must be defined here and it must be sufficient to avoid the whole application to stall.
- **configUSE_TRACE_FACILITY** If set to 1, all the tracing macros present in the FreeRTOS code are available. This feature is useful when performance of the system must be estimated: it is made essentially of many macros accessible to the programmer that must be implemented; each macro is called by a particular kernel function (unaccessible in any other way) and allows to execute custom code useful to trace events and eventually to send them to a monitoring computer. If set to 0 this feature is completely deactivated.
- **configUSE_MUTEXES** If set to 1, mutexes can be used in the user code. In FreeRTOS mutexes and binary semaphores are not the same thing: while mutexes implement priority inheritance mechanism (very important to avoid deadlock in some subtle cases), binary semaphores do not. If set to 0 mutexes are not available but semaphores, both counting and binary, do.

4.2.4 FreeRTOS data and structures

In this section variables, structures and functions are summarily described: their names are the same used as targets for the injections and numbers are the same used as identifiers in the experimental results, in the following chapters. This section is fundamental to fully understand the results of the experiments.

Tasks lists

In FreeRTOS a list is a structure made of 3 fields; each element of the list is defined as list item and it is described by another structure of 5 fields. The generic list structure used by FreeRTOS is described below.

0. **uxNumberOfItems** This variable contains the number of items stored in the list.
1. **pxIndex** This field is a variable pointing to the last item inserted in the list.

2-4. **xListEnd** This object is defined as 'MiniListItem_t' and it is just a structure made of 3 fields used to recognize the end of the list.

The list item object (single element to be put in a list) is instead described below.

0. **xItemValue** This value is used to sort the list in a descending order.
1. **pxNext** This is a pointer to the next item inserted in the list.
2. **pxPrevious** This is a pointer to the previous item inserted in the list.
3. **pvOwner** It is a pointer to the owner of the list item, usually a TCB structure.
4. **pvContainer** It is a pointer to the list structure containing the list item.

The definitions of 'list' type and of 'list item' are important to understand some variables and fields structure in the next subsections.

FreeRTOS has 5 task lists: pxReadyTasksLists (this is actually a vector of lists, each one for a priority defined in the system), xDelayedTaskList1 (this list contains the delayed tasks), xDelayedTaskList2 (this list contains all the delayed tasks whose timeout overflowed the current tick count), xPendingReadyList (it contains the tasks that had to be moved to ready state while the scheduler was stopped) and xSuspendedTaskList (used to collect suspended tasks); the kernel continuously moves tasks in and out these lists when they change state. Figure 4.10 shows the structure of pxReadyTasksLists; all the other task lists have a structure similar to just one of the lists shown.

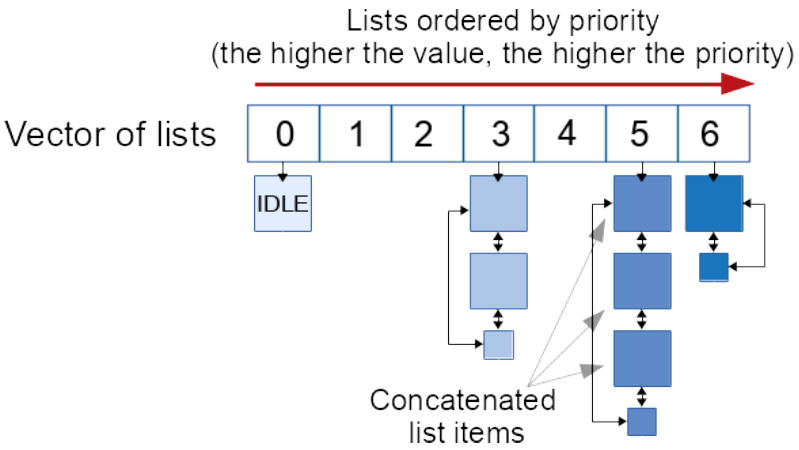

Figure 4.10: FreeRTOS pxReadyTasksLists vector

Global variables

Global variables are used by the kernel to perform operations, to take decisions and to know the state of the system every time the scheduler procedure is called. Relevant global variables identified are listed below.

0. **uxCurrentNumberOfTasks** This variable is constantly updated with the number of tasks running in the system. It is used by prvAddNewTaskToReadyList() to determine if a first initialization of a list is required: if so, prvInitializeTaskLists() is called and the lists pxReadyTasksLists, xDelayedTaskList1, xDelayedTaskList2, xPendingReadyList and xSuspendedTaskList are initialized. It is also used when a task is created, deleted or suspended: in particular, the function vTaskSuspend() uses it to determine if all the tasks instantiated have been suspended and if so, the OS links the pxCurrentTCB pointer to NULL.
1. **xTickCount** This value is set to zero when the scheduler is initialized and then is incremented in the function xTaskIncrementTick(): if this variable is bigger than xNextTaskUnblockTime and the pxDelayedTaskList is not empty, a task is removed from that list and is put into pxReadyTasksLists. This varible is fundamental when some tasks have been delayed: injecting here means that the system does not crash but the timing of the application could be heavily modified if any of the created tasks is in the suspended state.
2. **uxTopReadyPriority** Used to identify a ready task with the highest priority among all possible ready tasks.
3. **xSchedulerRunning** This variable is used by prvAddNewTaskToReadyList(): if the new created task has a priority higher than the task that is currently running, a switch is forced. In vTaskSuspend() it is used to take decisions when putting a new task in the xSuspendedTaskList.
4. **uxPendedTicks** This variable is used to keep count of the number of ticks passed while the scheduler was in suspended state.
5. **xYieldPending** This variable is set if a context switching is required for any reason (example: a task was put in the suspended state; now it is awakened because its timeout elapsed and it has a priority that is higher than the priority of the current task so a switch is required).
6. **xNumOfOverflows** Used by the mutex mechanism when a timeout is set for take and give operations and to keep trace of the overflows of the system tick timer (important to ensure correctness of timing).
7. **uxTaskNumber** It contains the number of tasks in the OS: when a task is removed, this value is incremented to force the regeneration of the list of active tasks.
8. **xNextTaskUnblockTime** This value is set to portMAX_DELAY when the scheduler is started. In xTaskIncrementTick() it is compared to xTickCount+1 and if the latter is bigger, the function removes the task from the list of delayed tasks. This variable is used in prvAddCurrentTaskToDelayedList() too: here it is set to a value equal to xTickCount+xTicksToWait, so that the scheduler, when xTaskIncrementTick() is called, will perform a check if there is any task to be resumed from delayed state.
9. **uxSchedulerSuspended** Used to take decisions when moving in/out tasks from the various lists and by the function xTaskIncrementTick(): if this variable is true, this function does almost nothing, otherwise, it checks for a new task to be switched in.

Task Control Block
The Task Control Block (TCB) is a data structure associated to each task created. It contains all the necessary information about the task itself.

0. **pxTopOfStack** This variable points to the top of the stack.
1. **uxPriority** This is the actual priority of the task exploited by the kernel to select the right ready task to be switched in.
2. **pxStack** This is a pointer to the base of the stack.
3. **uxTCBNumber** This variable is just a number given to the TCB to recognize it among the other TCBs.
4. **uxTaskNumber** This variable is similar to uxTCBNumber but it defines numerically the ID of the task.
5. **uxBasePriority** This is the base priority, used by the priority inheritance mechanism.
6. **uxMutexesHeld** Again this variable is exploited by the priority inheritance mechanism.
7. **ulNotifiedValue** This field is used by the notification mechanism (not considered in this work).
8. **ucNotifyState** This field is used by the notification mechanism (not considered in this work).

9-13. **xStateListItem** The container of this list item allows to understand which list the task belongs to (ready, suspended, delayed) and so which is its state; since this is a list item, it is actually made of 5 fields.

14-18. **xEventListItem** This item is inserted/removed from a list of a queue or mutex if the task uses them; since this is a list item, it is actually made of 5 fields.

Queue
The queue data structure is exploited when a queue, a generic semaphore or a mutex has to be created: the difference on their usage is related only to the way in which they are initialized and accessed by the provided functions.

0. **pcHead** It points to the head of the queue storage area.
1. **pcTail** It points to the tail of the queue storage area (equal to pcHead if the queue is used as mutex).
2. **pcWriteTo** This field points to the first free place in the storage area.
3. **u.pcReadFrom** This variable points to the last element queued.
4. **u.uxRecursiveCallCount** This field counts how many times the queue is recursively taken when the structure is used as a mutex (recursive take/give mechanism).

5-9. **xTasksWaitingToSend** List of tasks that are blocked waiting to post into the queue (give the mutex); since this is a list item, it is actually made of 5 fields.

10-14. **xTasksWaitingToReceive** List of tasks that are blocked waiting to read from the queue (take the mutex); since this is a list item, it is actually made of 5 fields.

15. **uxMessagesWaiting** This is the number of items actually in the queue (not the number of bytes).
16. **uxLength** This is the total amount of items hold in the queue.
17. **uxItemSize** This is the size of each item in the queue.
18. **cRxLock** Total number of items removed from the queue; this variable is used when the queue is accessed by an ISR.
19. **cTxLock** Total number of items put into the queue; this variable is used when the queue is accessed by an ISR.

20. **uxQueueNumber** Used for debug purposes (not considered in this work).
21. **ucQueueType** Used for debug purposes (not considered in this work).

4.2.5 FreeRTOS kernel

After the creation of the first tasks and an initial setup, FreeRTOS kernel works basically using an interrupt that allows to perform, if needed, a context switch to another task according to the selected scheduling policy: when the scheduler is started, the system tick timer is setup so that it generates an interrupt with the desired frequency. Every time such interrupt occurs, its handler is executed and a new ready task, if available, is selected for the switch-in; the actual context switching is made by another interrupt routine that is called (only if there is an available task) as soon as the ISR of the system tick is served: this interrupt is the so called PendSV, whose code is written in assembler (this routine is contained in the file port.c) and which saves the content of the registers in the stack. In the following sections the main operations done by the kernel are described.
In figures, names of FreeRTOS variables are highlighted in red, green boxes contain function names, yellow boxes conditionals and blue ones descriptions of algorithm.

Task creation
A task can be created during the setup of the application or at runtime. The way in which the task creation is managed changes slightly in these two cases: if the scheduler already started, a check is done to see if the new task has a priority higher than the priority of the current task and, if so, a pending switch is set. Figure 4.11 shows this operation in a detailed way.

Start of the scheduler
The scheduler is started as last operation after setup. It enables all the required interrupts, creates the IDLE task and setups the SVC (Supervisor Call) interrupt which is exploited to run the first task. Figure 4.12 shows operations in a detailed way.

Context switch
As already said, the scheduler performs context switching only when a tick interrupt occurs, choosing eventually the new task with a preemption/round-robin mechanism. Then, as shown at the end of figure 4.13, the PendSV interrupt bit is set if necessary, in order to actually do, in that case, a switch. Figure 4.14 shows summarily the operations made by the PendSV handler.

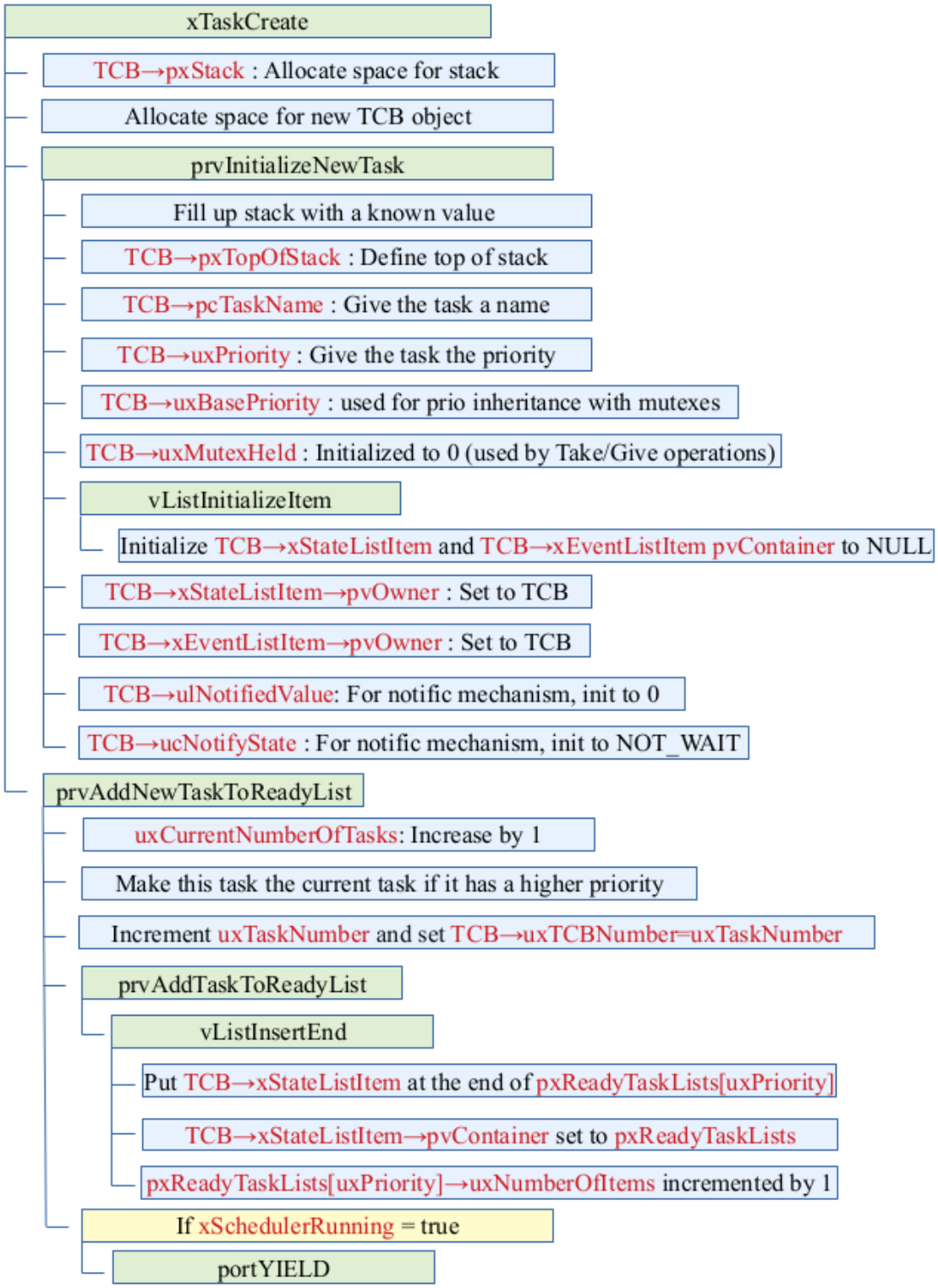

Figure 4.11: Kernel operations - Task creation

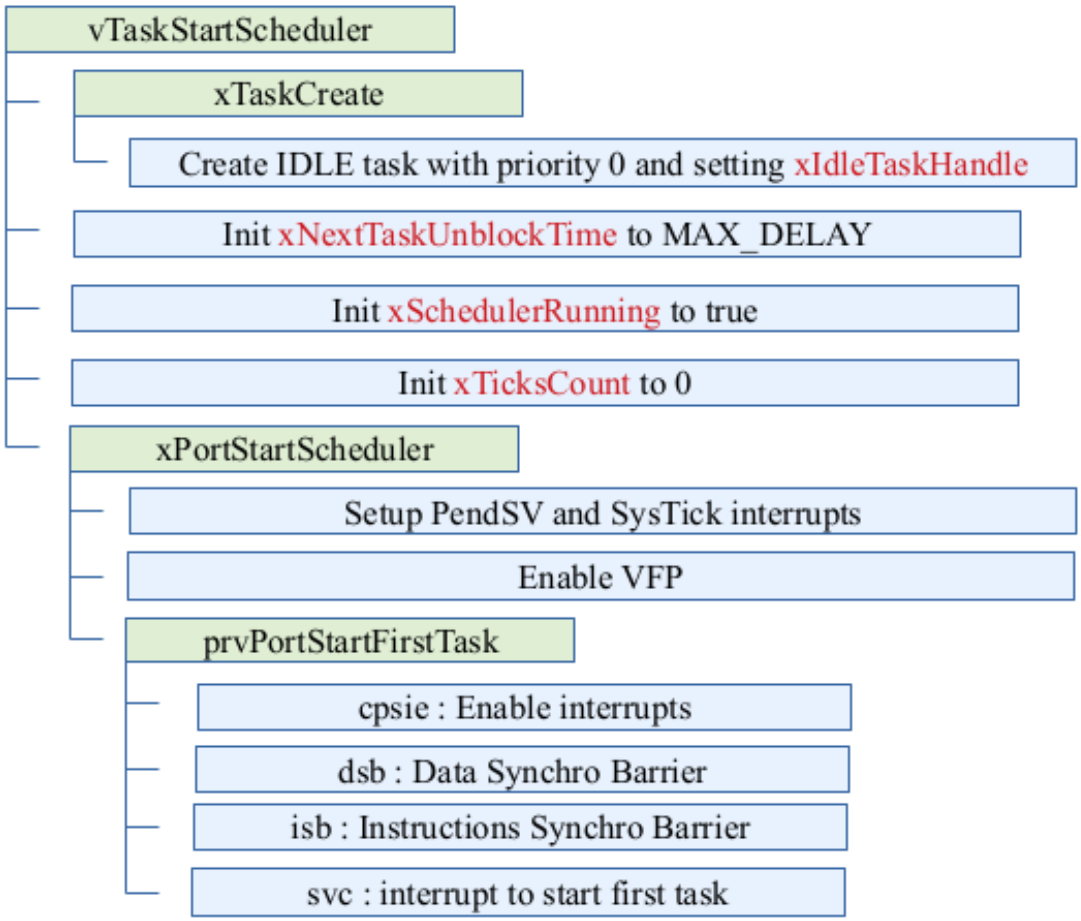

Figure 4.12: Kernel operations - Start of the scheduler

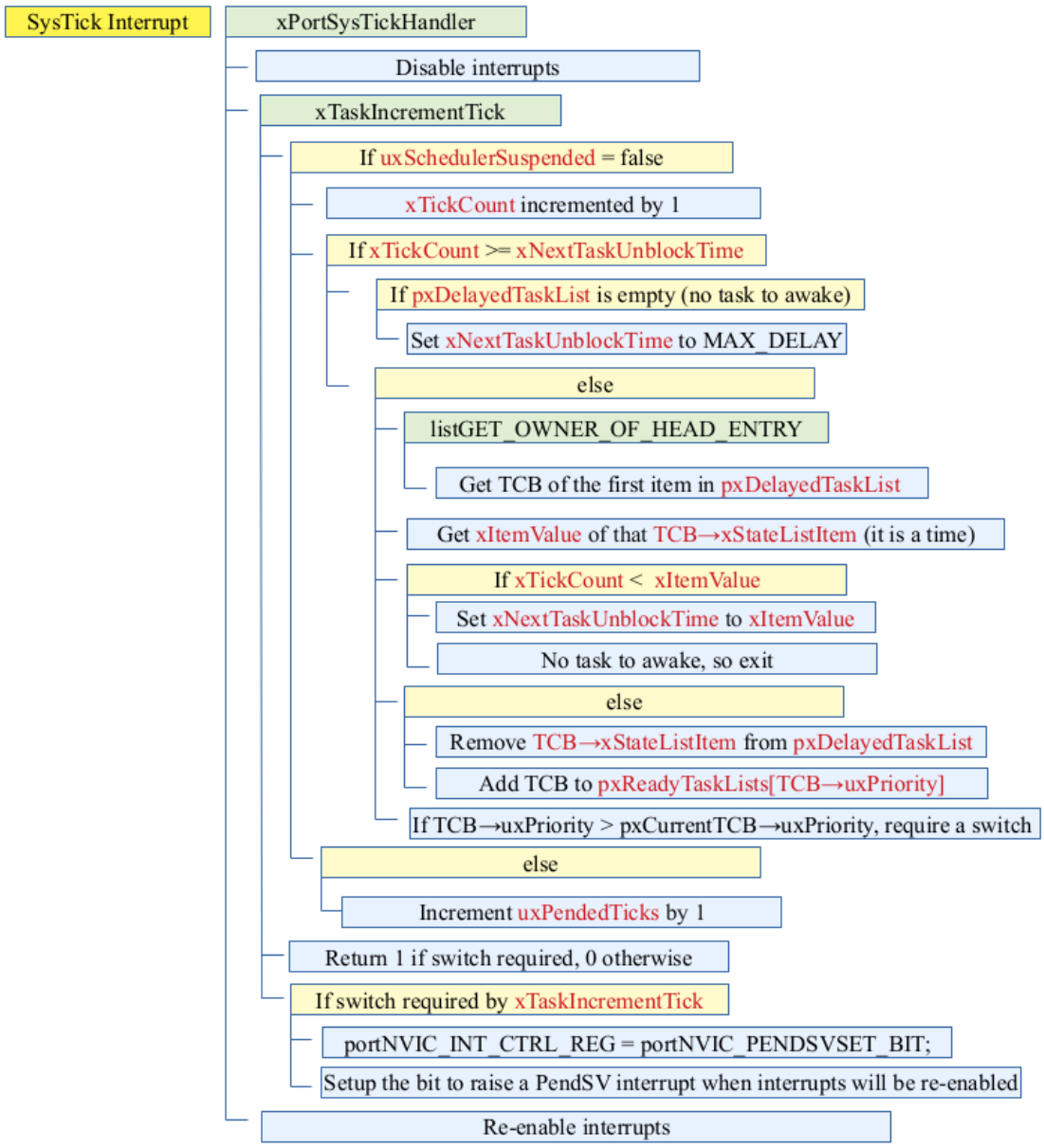

Figure 4.13: Kernel operations - Management of the system tick interrupt

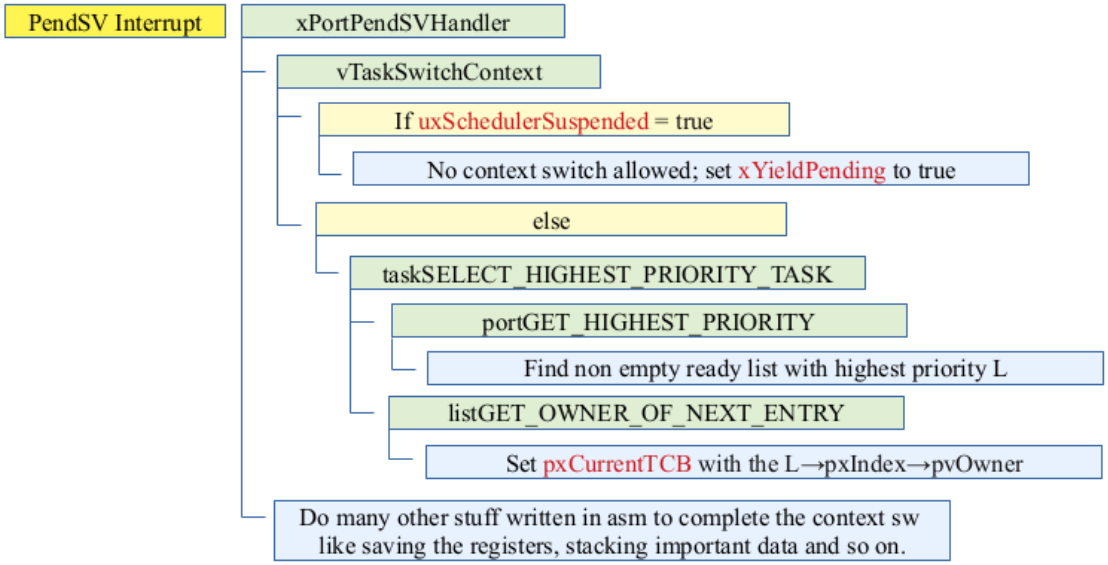

Figure 4.14: Kernel operations - Context switch done by PendSV interrupt routine

4.2.6 FreeRTOS mutexes

In FreeRTOS, mutexes are treated as a particular type of queues. Only mutexes are analyzed because they are the most interesting share control mechanism for this work.

Mutex creation
In FreeRTOS a mutex is a queue with 1 as number of items and 0 as item size. Initially it has no holder, so no task that has taken it and it is set as unlocked. Figure 4.15 shows this mechanism.

Mutex take operation
The 'take' (receive) operation requires an additional parameter, the tick count, that is the maximum number of ticks to wait for if the mutex is already taken; when this timeout elapses, the task goes on with its operations. As taking a mutex is a complex operation, additionally to figure 4.16, a pseudo-code listing is provided in appendix A.1.

Mutex give operation
The 'give' (send) operation is slighty simpler: the template is similar to the take function but some operations are not done because the give operation is made without a delay (to take a mutex instead the programmer can specify a timeout to wait for). Again, because of the complexity of this operation, pseudo-code is shown in appendix A.2, additionally to the figure 4.17.

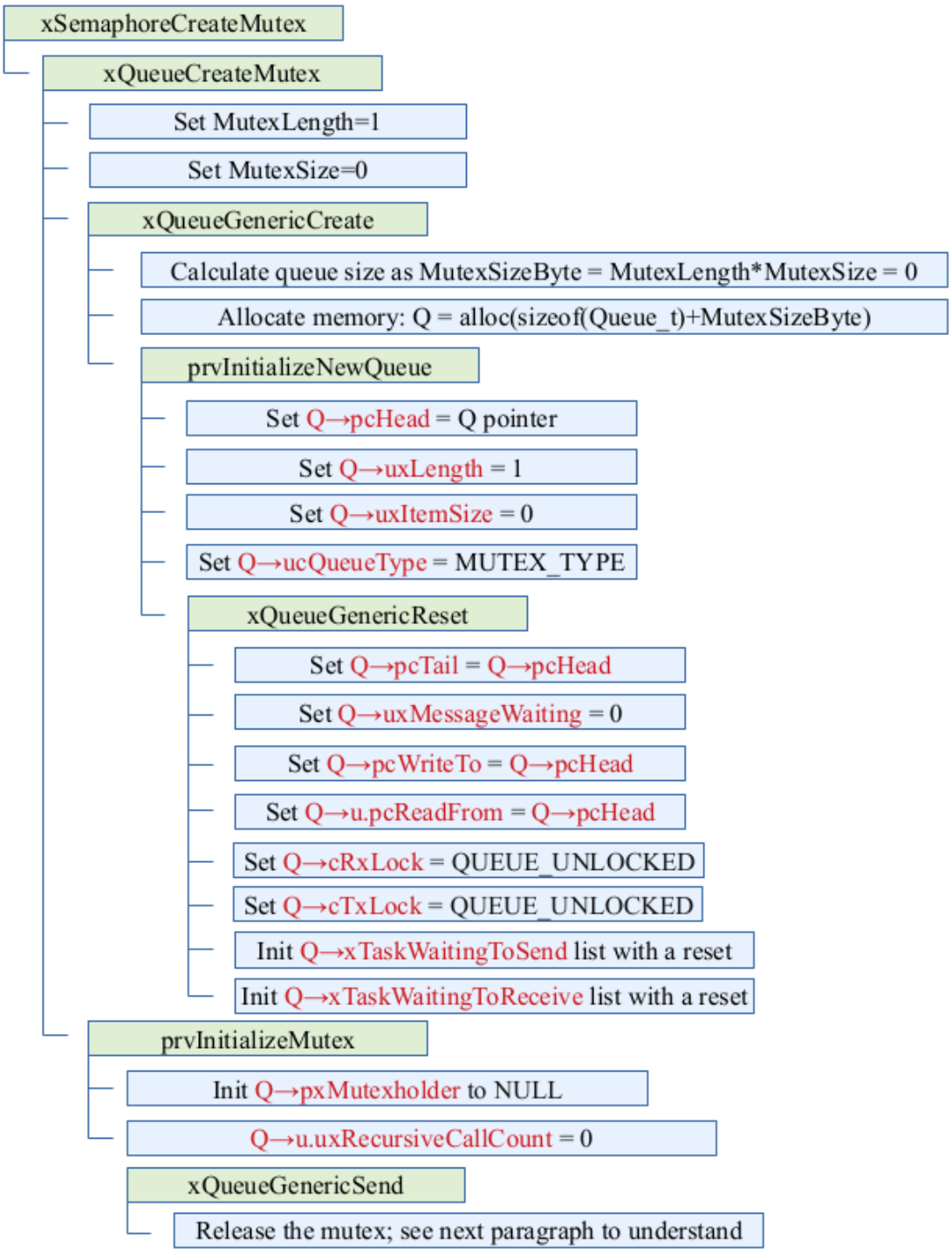

Figure 4.15: Mutex - Creation mechanism

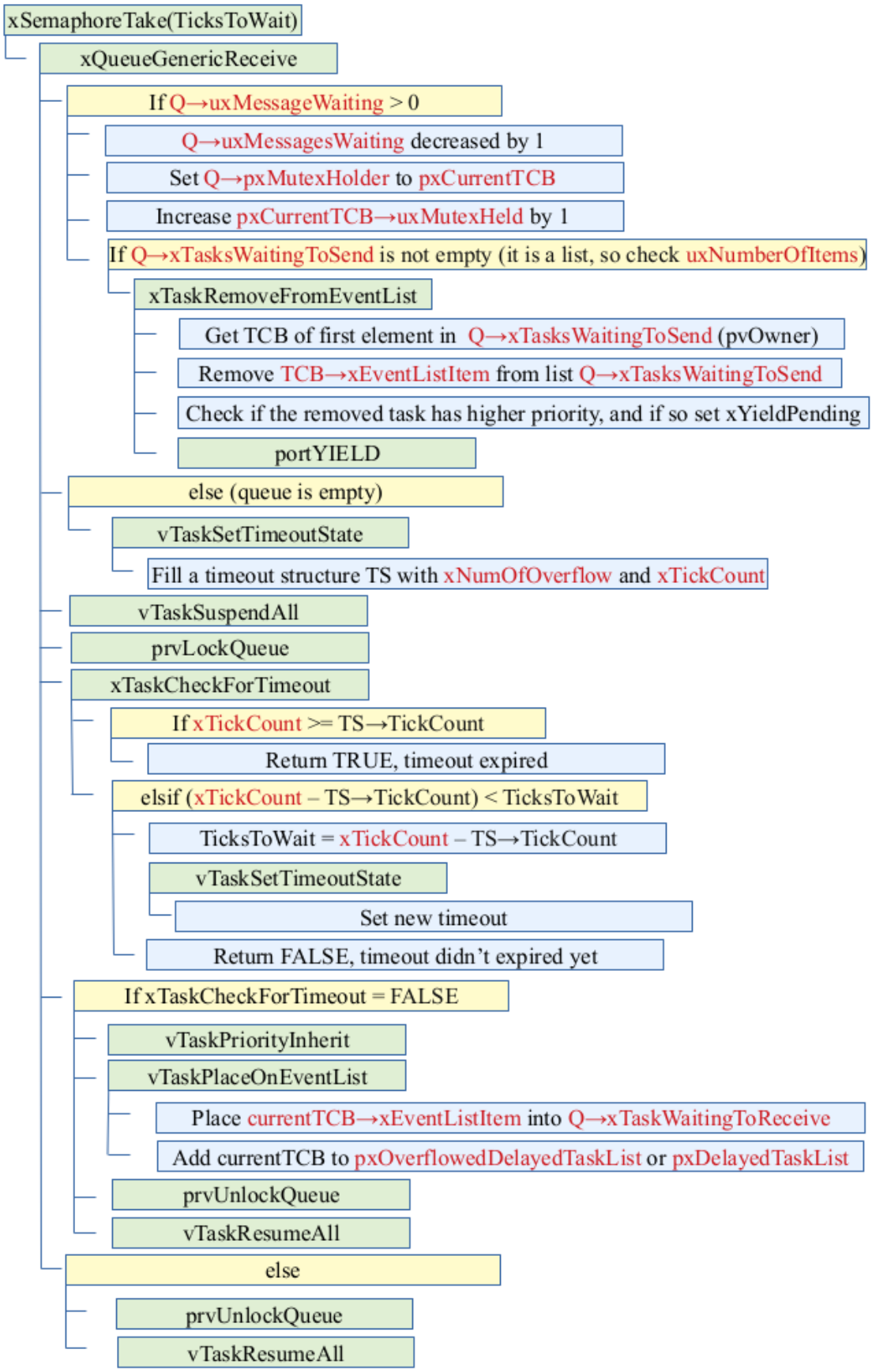

Figure 4.16: Mutex - Take mechanism

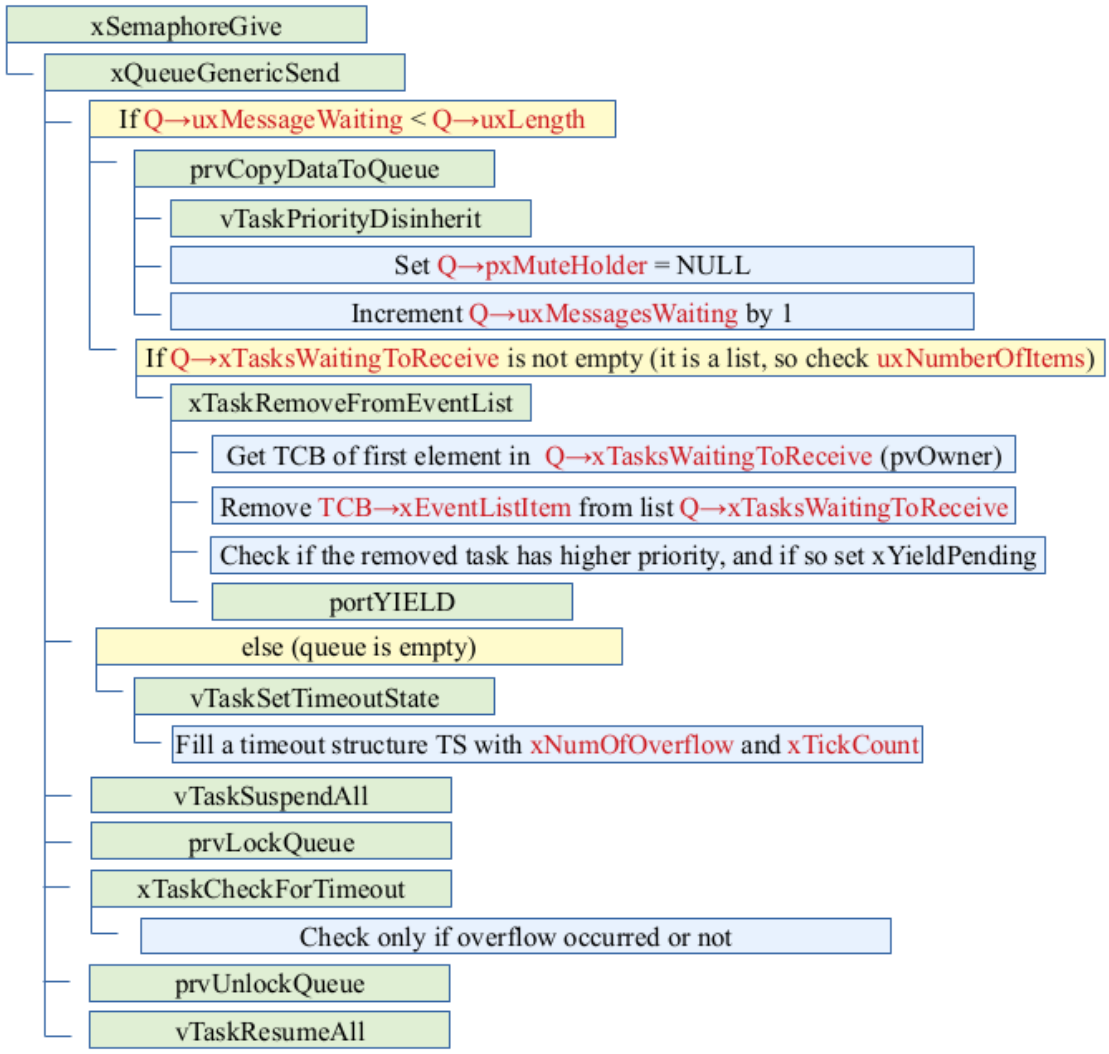

Figure 4.17: Mutex - Give mechanism

Chapter 5

Fault Injection Environment

5.1 Specifications

The aim of this work is to develop a *Fault Injection Environment* (FIE) able to reproduce the effects of SEU (in particular SBU and MBU) in the memory of the device under test (DUT), focusing only on main data structures and variables of a Real-Time Operating System and to trace the events so that they can be saved on a host computer and successively analyzed. The DUT must be chosen in order to be representative of a common platform used in embedded systems, with limited resources. The FIE must be able to perform automatically long injection campaigns after an initial configuration; moreover, it must be able to inject in given memory location, at given times and in the desired bit of the datum. Experiments must be repeatable in order to be relevant. The FIE must be as less invasive as possible. As results of the work, most sensitive sections of the RTOS must be identified.

5.2 Overview

The Fault Injection Environment is made of a board that acts as device under test (DUT) and a host machine working as platform that runs the injection campaign and saves results. From a generic point of view, the host-side program must send to the DUT a sequence of injection parameters as soon as the application starts, then the system is left free to run for a defined amount of time and finally the injection is performed on the desired datum. A resume routine is used by the DUT to send back to the host results of the injection. Figure 5.1 shows a top view scheme of the whole system.

FIE is composed by three parts: the first one is written for the DUT, it is architecture-dependent and it allows to communicate with the the host machine and to inject in the desired location; this part is called *FIEbrd* (that means 'FIE board') and it was developed entirely in C. The second one is a Python script called *FIEmon.py* (which stands for 'FIE monitor') that operates on the host-side, manages the injection campaign and saves results of the various experiments in a file; it is important to say that for each injection campaign two runs of the same algorithm must be done: the first one to produce a 'golden' file containing the results of the execution of the benchmarks without injection and the other one to get the outcomes of the real injection. Finally, there is another Python script, called *FIEparser.py*, that must be used to extract data from the injection campaign: it takes as input the two files previously created and performs some comparisons; this operation can be done at any time after the injection campaign is finished.

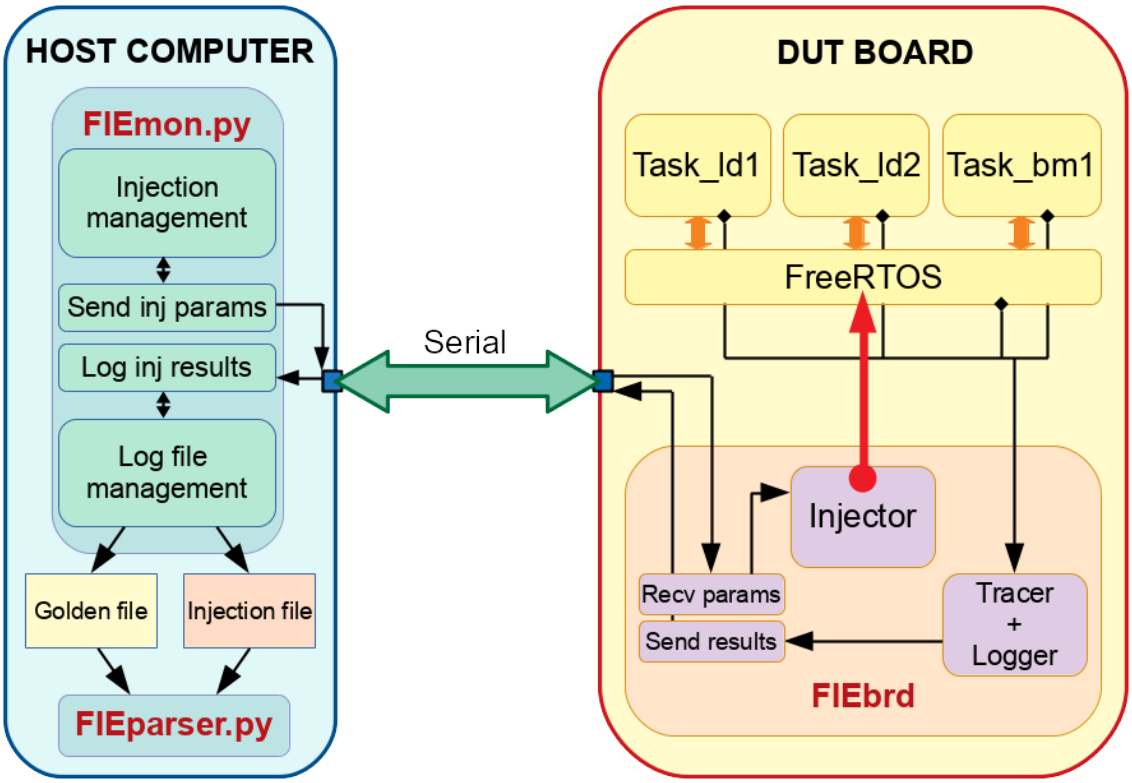

Figure 5.1: Top view of the injection system

The whole work has been divided into several steps: first of all it was necessary to port the EEMBC® Automotive suite for FreeRTOS, in order to see how and how many benchmarks could have been used; then a deep analysis of FreeRTOS has been done - it was fundamental to understand how its kernel works. The definition of the fault list followed as next step: candidate variables and data to inject in have been found among the most used and sensitive parts of the operating system, so various fault lists have been created, one for each type of OS resource. Next, the injection system has been written; it includes a tracing capability, useful to receive some information from the board about the operations performed by the kernel; these are the information logged and stored in a file on the host machine. As the quantity of data generated by the experiments is pretty high, the parser helps to do some automatic analysis on the information generated by the board and to print them both on a terminal and in the form of histograms. Finally, a lot of experiments have been done and results have been extracted with the help of the parser itself. Table 5.1 summarizes the properties of the Fault Injection Environment.

Just to summarize, a bit of nomenclature used from now on is provided in the following list.

Campaign This is a set of experiments made under well defined constraints and conditions. A campaign is made of two runs: the golden and the injection one and each run is made of one or more (even hundreds or thousands) experiments. When running a campaign, the two runs must do the same experiments exactly in the same order and under the same conditions.

Run The run is a set of experiments done sequentially and completely managed by the

Property	Value
Type of test	Robustness test
Technique used	Logical software-based fault injection
Target of injections	OS variables and structures
Event to simulate	Simulation of SEU (SBU and MBU)
Injection parameters	Bit, variable, time instant
OS target	FreeRTOS
Applicative used	EEMBC Automotive suite benchmarks
Actors	Board (DUT), host computer (manager)
Communication Host-DUT	USART

Table 5.1: Summary of Fault Injection Environment

host-side program FIEmon.py. The golden run is a sequence of experiments made without actually doing the injection: it is used to know which is the behavior of the system when no injection is done; it is fundamental to do it because it allows to have a reference to compare the degraded system to. The injection run is the sequence of experiments actually doing the injection: this is the only case where problems can really occur while the DUT executes the benchmark.

Experiment An experiment is the single iteration done during a run, with a real injection in the injection run or a simulated one in the golden run.

Fault list Called sometimes only 'list', this is a set of places where to inject. Each list contains all those faults that are strictly related to a part of the OS; it is made by many faults.

Fault A fault is the actual location where the injection is done. All faults have the same name of the variable instantiated in the memory. In this work a fault is called *locus* too.

5.2.1 Hardware

The FIE has been developed specifically for a board and tested on a host machine running a Linux-based operating system, but these hardwares can be easily changed by porting the code to other platforms.

DUT-side board
The STM32F3DISCOVERY board, like many other prototyping boards developed by STMicroelectronics®, is made of two parts: the top one includes the microcontroller STM32F103CBT6 which acts as programmer/debugger; the bottom, instead, contains the real microcontroller (STM32F303VCT6) to be programmed and used.
To work correctly with these boards it is necessary to install on the computer the ST-LINKv2 driver that allows to identify, once the device is connected, several peripheral: a mass storage device where one can upload its own .bin file containing the compiled program, a virtual serial port (marked under Linux with ttyACMn and under Window with COMn), a debugger port, a SCSI raw port and other additional features.

Host-side machine
The whole system has been developed and used on a commercial computer running Arch Linux. The fact that Python2.7, an interpreted language, has been used to develop the

scripts, removes the obstacle due to an eventual porting; in this way no translation is required since it is sufficient to install properly all the needed Python packages.

5.2.2 Operations

The STM32F3DISCOVERY board allows to send and receive data through a USART peripheral exploiting serial port virtualization. FIE exploits this feature so that host computer can send injection parameters to the DUT and receive back the outcome. It is important that the operations made by the DUT do not start until the host-side program is ready and synchronized: for this reason a synchronization mechanism is necessary. Figure 5.2 shows the sequence of operations done by both actors.

As soon as the executable is uploaded to the board, the DUT setups the clock of the

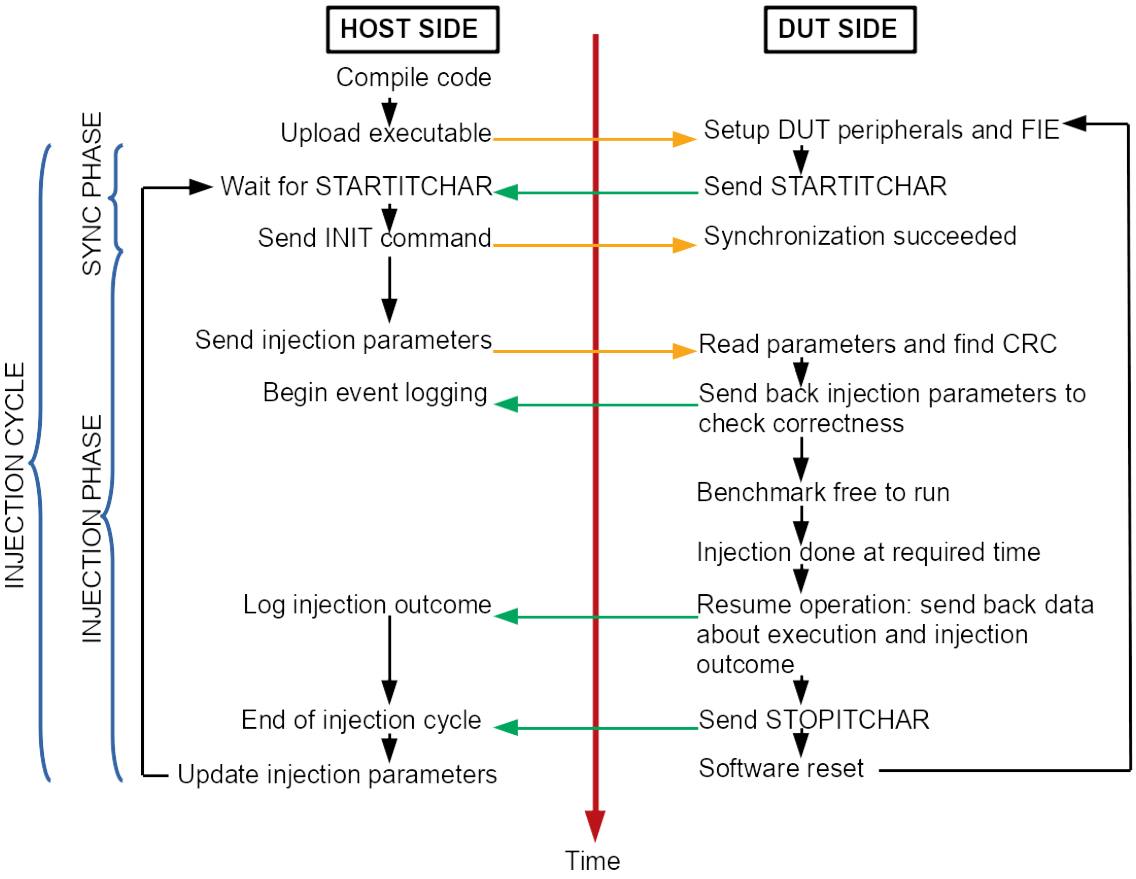

Figure 5.2: Sequence of steps made by host-side computer and DUT-side board. All communications are made using a USART peripheral.

microcontroller, the USART peripheral and the GPIO port connected to the LEDs on the board. FIE uses the some peripherals of the STM32F303VCT6 microcontroller to work and so it is important to develop applications that do not use them. Such peripherals are:

- *USART1* This peripheral is already connected to the programmer/debugger STLINK part of the board so all data written on this channel will be sent to the host computer over a virtual serial communication. It is used to receive injection parameters from the computer and to send back the outcome of the injection.
- *TIM2* This is a general purpose 32 bit timer. It is used to define a time instant in which injection is done. Interrupt generated by this peripheral (used in base count

mode) is activated, so that an ISR is called when the fixed period elapses. This timer will be called, from now on, 'injection timer'.

- *TIM3* This is a general purpose 16 bit timer. As the whole injection campaign must be automatized, it is important that, after some time following the injection, the DUT resets to proceed with the next experiment. This timer is set so that a fixed amount of time elapses after the injection is done and then microcontroller is reset. This timer interrupt too is enabled to put an end to the experiment and to proceed with the software reset. This timer will be called, from now on, 'resume timer'.
- *GPIOE* This is a GPIO port connected to 8 LEDs on the board. Two of them (PIN9 and PIN8) are turned on by the FIE to signal, respectively, at the instant when injection is done and when resume timer elapses its count, causing so software reset.

After the peripherals setup, the DUT sends a byte, called *STARTITCHAR* and representing a ASCII character, to the host computer, to inform that the experiment is ready to begin. Then the DUT starts to loop indefinitely waiting for a synchronization event coming from the computer: such even consists in the reception of a sequence of 4 bytes representing the *INIT* command that is sent back to the board by the computer as soon as the *STARTITCHAR* is received. When the USART receives all the four bytes, the DUT understands that the host is going to send injection parameters too, so it waits for them to arrive. According to the internal algorithm, the host choices such parameters and sends them over the serial connection in the form of string: all the parameters are padded by a TAB character. When all of them are received by the DUT, they are sent back to the host to allow the programmer to check visually if they are the correct ones. The injection timer period and prescaler registers are setup according to these parameters and its interrupt is activated: this peripheral is set so that an ISR is executed when their counter register reaches a value equal to the period. Now both FIE on the board and on the host machine wait for the benchmark application to perform some calculations: the host-side script waits to receive the results of the injection while the DUT-side injector waits for the injection timer (TIM2) to reach its timeout. When such event occurs, the injecting function is called and the injection is performed in the desired location: some checks are done before doing the injection because it can happen that the location is not available or not allocated, especially if it is a pointer. If the location exists, the injection is performed using the bitwise operation with a mask chosen at compile time and then the resume timer is setup. When also the resume timer reaches its timeout or if a crash occurs, a function is called to send to the computer the results of the injection, a *STOPITCHAR* to inform that the experiment finished and to do a software reset of the DUT so that it is ready for the next experiment. When the host machine receives the results, it logs them in a file and then updates the injection parameters. In the meantime, the DUT sends another *STARTITCHAR*: if the experiments have come to an end the computer-side script exits and the board will wait forever in the initial synchronization loop; on the contrary, if other experiments must be done, the cycle is repeated and new injection parameters are sent to the DUT.

5.3 Host-side FIEmon.py script

The Python2.7 FIEmon.py program is the host-side script able to manage the injection in 4 different ways and using 5 injection parameters. Operating modes are single injection (SIJ), normal injection (INJ), fine injection (DEP) and random injection (RAD). Injection parameters are the cycle, the bit where to inject, the fault where to inject, the value for

the period register of the timer and the value for the prescaler register of the timer. For each one of these parameters two variables are instantiated: a constant value, set by the programmer when configuring the campaign and a counter, used at runtime to keep track of the advancements of the experiments, up to the value set in the constant: each parameter is updated with a new value only when all the previous ones have reached their maximum assuring that experiments in every possible combination are done. A generic idea of the algorithm used to perform the injection is shown in figure 5.3.

Global variables that must be properly set before running the experiments campaign are instead described below.

- **STARTITCHAR** This character must be the same specified in the FIEbrd to begin the synchronization between host and DUT;
- **STOPTITCHAR** This character must be the same specified in the FIEbrd to signal the end of the experiment to the host machine, so that it can process the received results of the injection;
- **FILENAME** This variables is used just to specify the name of the file to save the results in. The format of the file name is not arbitrary and it must follow some rules so that it can work easily with the parser:

  $$[list\,name]\_[benchmark\,name]\_ < information > \_[run\,type]$$

  is the format, where all the various parts must not contain any space and they are divided by underscore. *[list name]* is used to specify the list where injection is done; *[benchmark name]* is instead the name of the benchmark used as workload for the DUT; *<information>* is any additional information which can be relevant to specify other conditions of the injection experiments; *[run type]* is finally a string which can be 'inj' or 'golden' and nothing else: it is used by the parser to recognize which of the two files given as input contains the golden run and which one the injection run. An example of good naming for two files generated during a campaign is: 'L2OSvars_a2time_8LSB_golden' and 'L2OSvars_a2time_8LSB_inj';
- **cFIE_ENABLED** The value assigned to this variable specifies which mode is used to perform the injection campaign. It is possible to use the 4 different modes, according to the necessity;
- **cFIE_LIST** This variable is used to specify in which of the 4 lists the injections must be done.

After the operating mode has been chosen assigning to the configuration variable cFIE_-ENABLED the correct value, the variables related to such mode must be set as well, as described in the following.

5.3.1 Single injection mode (SIJ)

In this mode the injection parameters are set just for one experiment: this means that the whole campaign will be made of 2 runs, each one of one experiment only. This mode must be used when a very precise injection in a defined moment and fault has to be done. The variables to be set in this mode are:

- **cFIE_INJ_CYCLE** In this mode this is a dummy variable because only one experiment is done in each run. It is set to 0 by default;
- **cFIE_INJ_BITN** This value allows to select in which bits the injection must be done. Attention must be payed to the size of the data to inject in: if bit set as target for the injection does not exist because the target data is shorter, then no injection is done and an error is reported. Bit count starts from 0;

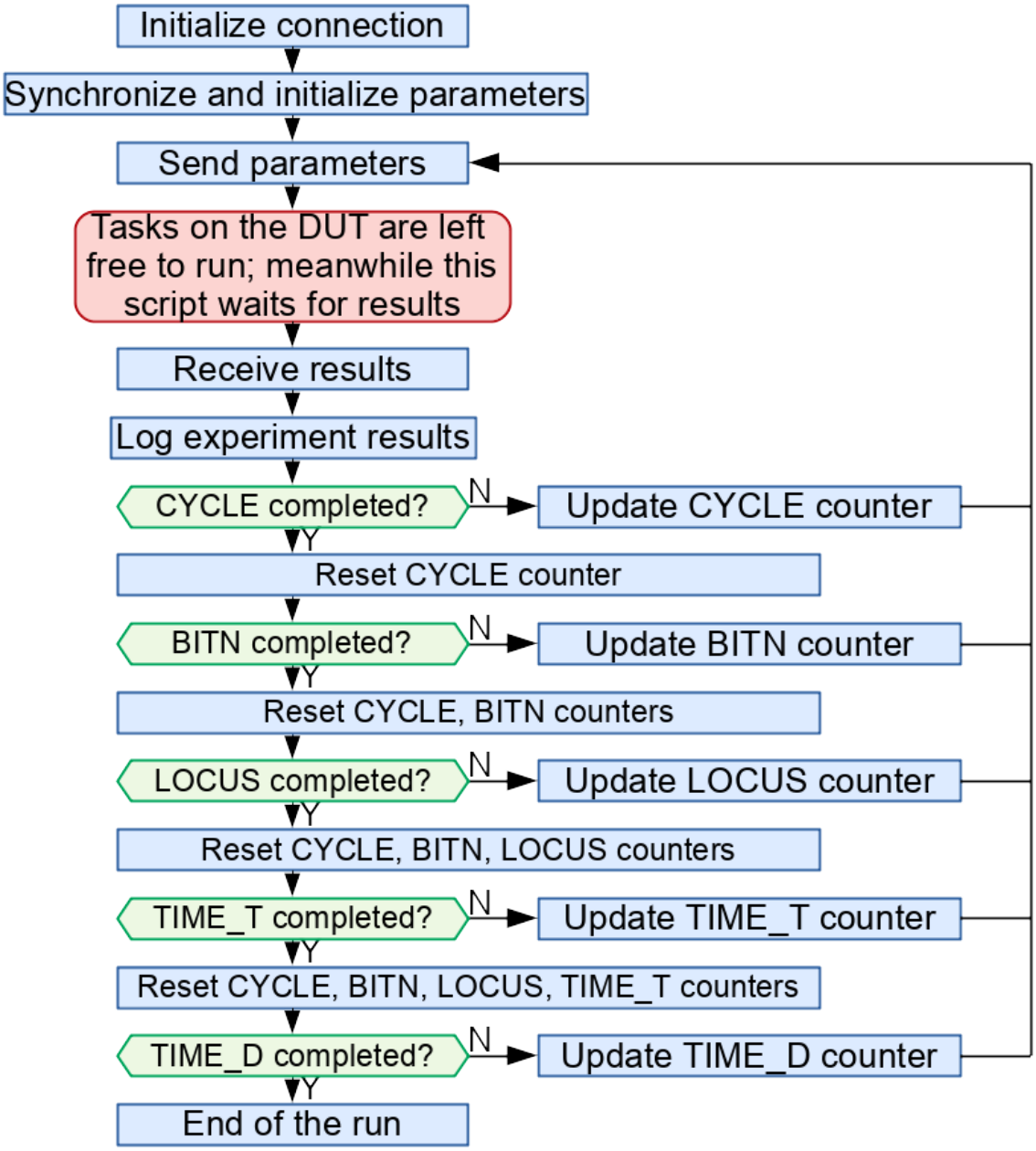

Figure 5.3: Generic flow diagram of the host-side algorithm

- **cFIE_INJ_LOCUS** This parameter says which is the fault to inject in among those present in the selected list. Such parameter in used in SIJ mode only since in the other ones injection is done automatically in all the faults of the selected list;
- **cFIE_INJ_TIME_T** This parameter is set to the desired value for the period of the timer of the DUT (TIM2). Values allowed are comprised between 0 and 65535;
- **cFIE_INJ_TIME_D** This parameter is set to the desired value for the prescaler of the timer of the DUT (TIM2). Values allowed are comprised between 0 and 65535.

5.3.2 Normal injection mode (INJ)

When using this mode it is possible to define a set of instants so that the injection is done, for each fault in the selected list, in all of them. This is pretty useful when one wants to do a campaign analyzing the effects of faults in many different moments.

- **cFIE_INJ_CYCLE** If this parameter is set to a value bigger than 0, each experiment on a fault is repeated more than one time always in the same conditions, leaving unchanged the other parameters. This can be useful to see if all injections in same conditions lead to same results;
- **cFIE_INJ_BITN** This parameter can be positive or negative: if it is positive, injection will be done starting from bit in position 0 up to the bit in the position specified; instead if it is negative, injection is done starting from MSB and going backward;
- **cFIE_INJ_PERIOD_VECT** This is a vector containing all the values for the period register of the injection timer (TIM2) to be sent during the run. Values allowed are comprised between 0 and 65535;
- **cFIE_INJ_TIME_T** This variable is the length (number of elements) of the cFIE_INJ_PERIOD_VECT vector;
- **cFIE_INJ_PRESC_VECT** This is a vector containing all the values for the prescaler register of the injection timer (TIM2) to be sent during the run. Values allowed are comprised between 0 and 65535;
- **cFIE_INJ_TIME_D** This variable is the length (number of elements) of the cFIE_INJ_PRESC_VECT vector.

5.3.3 Fine injection mode (DEP)

Sometimes it could be interesting to analyze the effect of the fault in close and subsequent time instants: to solve this the fine injection mode can be used; this mode allows to define in a 'fine grain' the instants where to inject: at each experiment in the same fault, the time will advance of a very small amount with respect to the previous injection instant.

- **cFIE_INJ_CYCLE** If this parameter is set to a value bigger than 0, each experiment on a fault is repeated more than one time always in the same conditions, leaving unchanged the other parameters. This can be useful to see if all injections in same conditions lead to same results;
- **cFIE_INJ_BITN** This parameter can be positive or negative: if it is positive, injection will be done starting from bit in position 0 up to the bit in the specified position; instead, if it is negative, injection is done starting from MSB and going backward;
- **cFIE_INJ_BASE_T** This value is used as base value for the period register of the injection timer (TIM2);

- **cFIE_INJ_TIME_T** This value is the maximum added to the base value: all values for the period register starting from cFIE_INJ_BASE_T up to cFIE_INJ_BASE_T+cFIE_INJ_TIME_T are used as parameters during the run;
- **cFIE_INJ_BASE_D** This value is used as base value for the prescaler register of the injection timer (TIM2);
- **cFIE_INJ_TIME_D** This value is the maximum added to the base value: all values for the prescaler register starting from cFIE_INJ_BASE_D up to cFIE_INJ_BASE_D+cFIE_INJ_TIME_D are used as parameters.

5.3.4 Random injection mode (RAD)

When the programmer has the time to perform a very high number of experiments in each campaign but he doesn't know when to inject, a random injection can be done: in this mode the injection times are chosen pseudo-randomly so that the runs can be repeated.

- **cFIE_INJ_CYCLE** In RAD mode this variable is very useful: in fact it allows to repeat the same injection in the same fault in many different instants chosen pseudo-randomly by the algorithm;
- **cFIE_INJ_BITN** This parameter can be positive or negative: if it is positive, injection will be done starting from bit in position 0 up to the bit in the position specified; instead if it is negative, injection is done starting from MSB and going backward;
- **cFIE_INJ_BASE_T** Like in DEP mode, this value is used as base value for the period register of the injection timer (TIM2);
- **cFIE_INJ_MOD_T** As RAD mode performs injections in pseudo-random times, a maximum value for such numbers is fixed with this variable;
- **cFIE_INJ_BASE_D** Like in DEP mode, this value is used to as base value for the prescaler register of the injection timer (TIM2);
- **cFIE_INJ_MOD_D** Same thing of cFIE_INJ_MOD_T but used, this time, for the prescaler register.

5.4 DUT-side FIEbrd system

The application on the board is made of some running tasks executing the user code and the FIEbrd itself. In all the applications, 3 FreeRTOS tasks are instantiated: two of them (called ld1 and ld2) just access a GPIO port and flip continuously the values of two pins, sharing a mutex; the third task (called bm1), instead, executes the actual EEMBC® benchmark. At the end of each iteration (tasks are always defined as infinite loop) all the three tasks go in delayed state for an arbitrary amount of time of the order of some milliseconds. Figure 5.4 shows which are the main operations done by the DUT.

The FIE system installed on the board is made of a set of files written starting from scratch and some modification to the code of FreeRTOS, necessary to extract some data usually not accessible to the programmer. The part of FIEbrd that is completely new is made of 5 main files while the rest is built up changing preexisting or auto-generated code used to setup the hardware of the microcontroller, the interrupt table and the RTOS itself. The content of the 5 files is illustrated below.

FIE_config.h
This file is made only of preprocessor macros (mainly *#defines*) used by the rest of the system as setup values. If nothing particular must be done, it is sufficient to regulate the

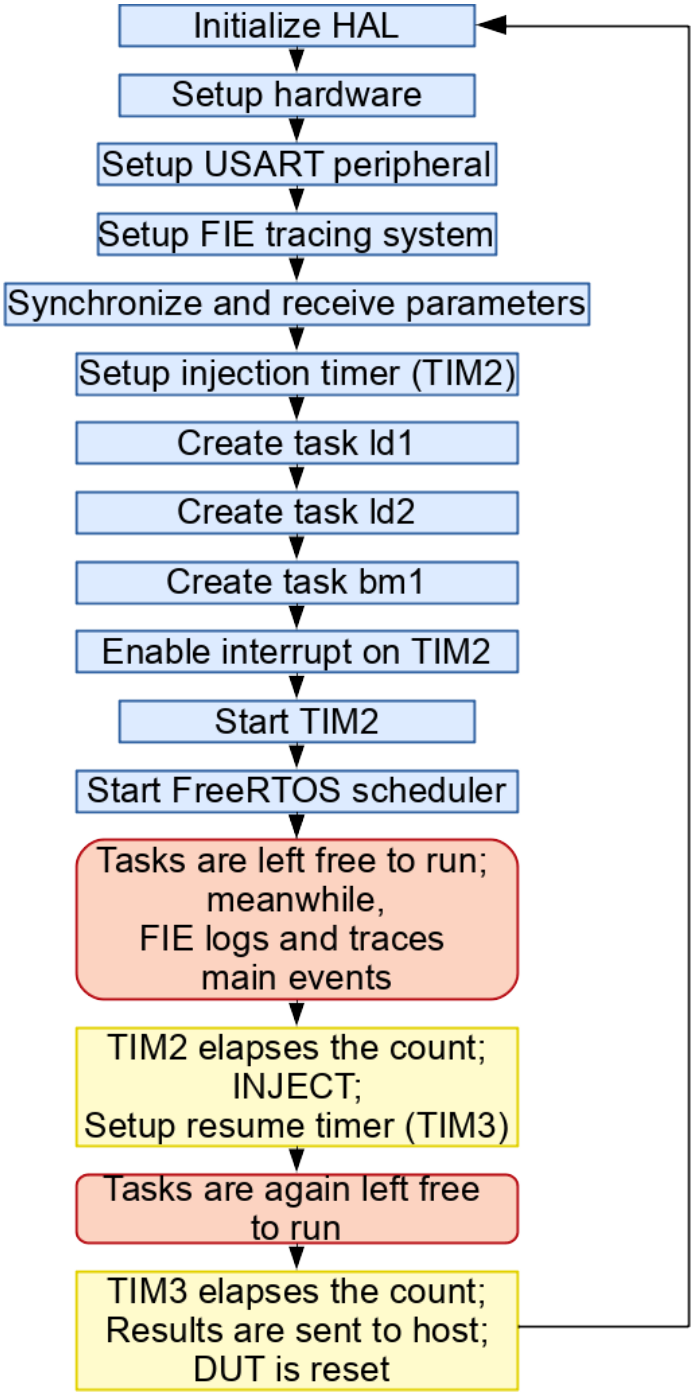

Figure 5.4: Generic flow diagram of the DUT-side sequence of operations

behavior of the injection system changing just these parameters.

- ▷ **cFIE_ENABLED** This parameter is used to activate or deactivate the FIE system; if this value is set to 0, then the whole FIE is excluded from the compilation, saving space and allowing the application to run normally. A value equal to 1 instead enables the injection system.
- ▷ **cFIE_INJECT_ENABLED** In order to generate both injection and golden report files, it is important to have a parameter that allows to enable or disable quickly the injection: setting this *#define* to 1 the injection is always performed, doing thus an injection run; if it is set to 0 instead injector is still called but injection is masked.
- ▷ **cFIE_PRECISEINJ_ENABLED** Sometimes it is important to perform the injection exactly when the system executes a portion of code: in order to support this feature, this configuration variable can be set to 1 and then the injector can be called in any part of the user code; when this feature is active, the system does not inject as soon as the injection timer (TIM2) reaches the timeout, but only after the end of the count of TIM2, when the injecting routine is explicitly called.
- ▷ **cFIE_TRACE_ENABLED** In order to trace system events, it is possible to exploit a tracing system integrated in the FIE. When this value is set to 1 some information about the execution of the instantiated tasks are sent to the host machine after the injection.
- ▷ **cFIE_TRACEPLUS_ENABLED** This is still a beta feature: it would like to emulate some other advanced tracers appositely developed for FreeRTOS but some additional effort must be done. As such function was not strictly necessary for this work, it has been partially implemented.
- ▷ **cFIE_INJ_MASK** This mask is the one used by the injector function to select which are the bits to be affected by the injection. Mixing this mask with the bit selection made during the setup of the host-side script allows to simulate every type of SEU, both SBU and MBU.
- ▷ **cFIE_OP** With this macro it is possible to choose the type of bit-level operation to perform when doing the injection. Three options are available: bit set (logic OR), bit reset (logic AND) and bit flip (logic XOR).
- ▷ **cFIE_BM_NUMBER** The amount of data sent to the host machine after the injection depends on how many tasks running in the system the programmer wants to trace. This variable contains exactly the number of tasks that must be monitored.
- ▷ **cFIE_LIST** With this variable the list where the injection must be done is selected: as the FIE to be designed must be as much low-size as possible, the code length of the injector function is reduced selecting only one fault list at a time.

FIE_environment.h and FIE_environment.c

These two files contain the core of the Fault Injection Environment; here all the hidden variables used by FreeRTOS are exported, the 4 fault lists are defined (only one at a time is actually compiled in order to reduce code size) and the various functions are defined and implemented. Moreover, some variables (both global and local) used by the FIE are defined: some of them must be correctly inserted in the user code so that a good logging of the events is performed. In the following all the functions of the FIE are described.

- ▷ **void FIE_start(void)** This function must be called in the main() immediately before the creation of the first task: it resets logging variables, synchronizes the DUT system with the host machine and allows to receive the injection parameters.
- ▷ **void FIE_stop(void)** This function is called when the experiment reaches its end to send back to the host machine the values of the logging variables, stored in a file as

result of the injection.

- ▷ **void FIE_timx_inj(void)** With this function the injection timer is setup with some of the injection parameters received by FIE_start(). This function must be called after FIE_start().
- ▷ **void FIE_timx_res(void)** Actually this function should not be used by the programmer because it is already called internally by the injection system: when the injection timer reaches its timeout, injection is performed and the resume timer is setup by means of this function.
- ▷ **void FIE_injector(void)** This is the real function used to perform the injection. According to the list chosen at compile time and the fault parameter received through the connection with the host machine, it does the injection in the desired datum. It is called by the timeout handler of the injection timer (TIM2).

Logging variables are those ones sent back to the host machine to keep track of system status, of its activity and of relevant events in the running tasks; their name are strictly related to the *ad-hoc* implementation of the benchmarks but they can be adapted easily.

- ▷ **FIE_REP_BM_CRC_OK** This variable must be properly updated in the user code. It is used to keep count of the number of times the benchmark made correct calculations: CRC is calculated on the output RAM file, compared to the given one and if they match, this variable is incremented by 1.
- ▷ **FIE_REP_BM_CRC_WRONG** This variable must be properly updated in the user code. It keeps trace of the number of times CRC was wrong.
- ▷ **FIE_REP_BM_MICROIT** This variable must be properly updated in the user code. It keeps trace of the number of microiterations done by the benchmark algorithm.
- ▷ **FIE_REP_BM_MACROIT** This variable must be properly updated in the user code. It keeps count of the number of iterations of infinite *while* loops of the task before TIM3 resume timer marks the end of the experiment.
- ▷ **FIE_REP_SYS_INJOK** As the injector shares the same memory of the user code, it could happen that the injection harms the FIE itself so an integrity check is done just before the results are sent to the host machine. A CRC is calculated among all the parameters and, if it is equal to the one calculated after their reception, it means that the injection system remained unharmed.
- ▷ **FIE_REP_SYS_INJ** In some cases it could happen that the fault to inject in is not available, especially if it is a variable that is still to be allocated. In such cases this parameter is set to 0 to inform the host that the injection was skipped.
- ▷ **FIE_REP_SYS_OK** This variable just says if, after the injection, the system crashed or not. It is usually 1 but, if an error handler is called before the resume timer (TIM3) reaches its timeout, it is set to 0 to highlight a crash.
- ▷ **FIE_REP_SYS_ETAF** The acronym ETAF means Elapsed Time After Fault: this is a numerical value that keeps count of the time (expressed in TIM3 ticks) elapsed between the injection and an eventual crash: if this value is 0 and FIE_REP_SYS_OK is 1, no crash occurred; if this variable is 0 and FIE_REP_SYS_OK is 0 then a crash occurred immediately after or even during the injection; if this variable is greater that 0 and FIE_REP_SYS_OK is 0 then the crash occurred after the injection and before the resume timer (TIM3) elapsed its count.

FIE_trc.h and FIE_trc.c

As already said, it could be useful to have a system deeply integrated in FreeRTOS able to trace its main events: this RTOS already provides some blank macros inside the kernel that must be implemented by the programmer, which allow to trace the main events of

the kernel itself. In the FIE such feature has been exploited adding a trace function: the number of 8 kernel events is continuously traced and stored in 8 tracing variables which are sent to the host machine when the experiment ends; in this way some more accurate comparisons about the execution can be done, basing the post campaign analysis not only on user code events but also on kernel events. The variables are:

- **FIE_TRC_SWIN** This tracing variable keeps count of all the times a task is switched in.
- **FIE_TRC_SWOUT** This tracing variable keeps count of all the times a task is switched out. At the end of each experiment this value should be equal to FIE_TRC_SWIN or different from it by only one unit.
- **FIE_TRC_MOVRDY** Every time a task is moved from the suspended or delayed list to the ready list this tracing variable is incremented by one.
- **FIE_TRC_DELAY** When a task calls the function vTaskDelay() to delay itself, this value is incremented by one.
- **FIE_TRC_SUSPEND** Like FIE_TRC_DELAY but this works for task suspension.
- **FIE_TRC_RESUME** Resume operation is the opposite of suspension: every time a task is resumed this variable is incremented by one.
- **FIE_TRC_QRECV** When a queue receives a new item or a semaphore or mutex is taken, this value is incremented by one.
- **FIE_TRC_QSEND** When a queue releases an item or a semaphore or mutex is given, this value is incremented by one.

5.5 FreeRTOS code modification

As FreeRTOS has been written so that many kernel data, variables, structures and functions are not available to the programmer, some modifications to the code were necessary in order to make them accessible.

FreeRTOSConfig.h
An additional *#define* has been added in order to activate/deactivate the DUT-side FIE. In this way it is very simple to switch from a normal FreeRTOS version to the modified one which supports the FIE. Such *#define* is *configFIE_ENABLED*; this name follows the FreeRTOS rules for the nomenclature. Moreover, at the end of this file, a header file containing all the function prototypes and the public data of the tracing system integrated in the FIE is included.

task.h
At the end of this file, the definition of the TCB structure is added: according to the value set in FreeRTOSConfig.h to *#define configFIE_ENABLED*, such definition can be available or not: if the FIE is enabled, such structure is visible to all the other files including FreeRTOS among the headers. More precisely, the structure called internally by the kernel tskTCB is defined in this file and not in tasks.c and it is renamed as FIE_TCB_t so that it can be used in other files; when the FIE is disabled, such structure is hidden and it is only defined in tasks.c so that it is not accessible anymore to the programmer. Moreover in this header all the functions extracting some data used by the kernel are provided: again, if FIE is enabled, these functions can be called to get the pointers to some kernel internal structure and variables that normally are not visible.

tasks.c
This is the place where the TCB structure is normally defined: such definition is automatically hidden if FIE is enabled and exported to task.h header file. Moreover, as many other variables are not accessible to the programmer because they are defined as static, when FIE is enabled, the same variables are defined in a normal way, allowing thus to export them easily. All the functions used to get the pointers to some internal data and structures defined in task.h are implemented at the end of this file.

queue.h
As the structure of the queue is used in many ways, it is not directly accessible to the programmer, so again, it is extracted and made available under the new type FIE_Queue_t. More specifically, such structure is used for queues, semaphores and mutexes: a wrapper made essentially of macro recalling queue functions with predefined parameters is used to allow its reuse as a semaphore.

Chapter 6

Experimental environment

6.1 Classes of misbehaviors

Every injection could lead to different types of (mis)behaviors of the system; however, only some of them can be actually identified and classified because of some limits in the tracing capabilities of the injection environment. Results have been divided into 4 categories: *crash*, *freeze*, *degradation* and *silent misbehavior*.

Crash
When a critical error occurs on the device under test, the internal reset handler is called in order to avoid further problems. In this case the misbehavior is classified as *crash*. A crash is identified when the main error handler is called forcing thus a software reset before the 'resume timer' reaches its timeout.

Freeze
A misbehavior is classified as *freeze* when the whole system stops working and does not respond to any regular input or event: only interrupts are executed, then, when the ISR returns, the system goes back to the frozen state. A freeze is identified when the whole system stops but it does not crash: simply no task is scheduled anymore.

Degradation
A misbehavior is classified as *degradation* when only a part of the system shows a behavior that is substantially different from the expected one: this means that only a part of the system is blocked or acts incorrectly during the execution while the rest is still able to run properly. A degradation is identified when the execution of one or more tasks is different from the expected one.

Silent misbehavior
This type of misbehavior happens when, after the injection, the system continues working in the expected way, without showing any appreciable difference with respect to the golden run.

6.2 Definitions

The parser performs all the operations required to categorize the misbehaviors as already said; however, in order to better understand the global results of a campaign, 4 quantities

have been defined: *consistency (C)*, *crash ratio (CR)*, *freeze ratio (FR)* and *degradation ratio (DR)*; they are all defined as ratios, expressed in percentage. They allow to get easily which were the effects of the injection in each fault of a list. Note that the sum of CR, FR and DR for each fault is always equal to 0 or 100.

Consistency
Consistency is defined, for each fault, as

$$C = \frac{Number\,of\,misbehaviors}{Number\,of\,injections} \tag{6.1}$$

It allows to understand how much that locus is sensitive to an injection, so how many times it produces a misbehavior when the fault is activated. If this value is 0 it means that the system is not sensitive to the injection in that particular locus at all. All the following numerical values must be always compared to this quantity as it allows to understand the overall incidence on the system.

Crash ratio
Crash ratio is defined, for each fault, as

$$CR = \frac{Number\,of\,crashes}{Number\,of\,misbehaviors} \tag{6.2}$$

It allows to understand which is the number of crashes over the total number of misbehavior shown by the fault. The higher this value is, the more the fault is a sensitive point of the RTOS.

Freeze ratio
Freeze ratio is defined, for each fault, as

$$FR = \frac{Number\,of\,freezes}{Number\,of\,misbehaviors} \tag{6.3}$$

Like the crash ratio, but it is the ratio between the number of freezes and the total number of misbehavior os a fault.

Degradation ratio
Degradation ratio is defined, for each fault, as

$$DR = \frac{Number\,of\,degradations}{Number\,of\,misbehaviors} \tag{6.4}$$

This is the percentage of degradations over the total number of misbehavior identified.

6.2.1 Fault lists

Fault lists used are 4 and they are made exactly of the same variables described in 4.2.4; they are reported below just for completeness.

List 1 - Global variables
0. uxCurrentNumberOfTasks
1. xTickCount
2. uxTopReadyPriority

3. xSchedulerRunning
4. uxPendedTicks
5. xYieldPending
6. xNumOfOverflows
7. uxTaskNumber
8. xNextTaskUnblockTime
9. uxSchedulerSuspended

List 2 - TCB data structure

0. pxTopOfStack
1. uxPriority
2. pxStack
3. uxTCBNumber
4. uxTaskNumber
5. uxBasePriority
6. uxMutexesHeld
7. ulNotifiedValue
8. ucNotifyState
9. xStateListItem.xItemValue
10. xStateListItem.pxNext
11. xStateListItem.pxPrevious
12. xStateListItem.pvOwner
13. xStateListItem.pvContainer
14. xEventListItem.xItemValue
15. xEventListItem.pxNext
16. xEventListItem.pxPrevious
17. xEventListItem.pvOwner
18. xEventListItem.pvContainer

List 3 - Task lists

0. uxNumberOfItems
1. pxIndex
2. xListEnd.xItemValue
3. xListEnd.pxNext
4. xListEnd.pxPrevious

List 4 - Mutex data structure

0. pcHead
1. pcTail
2. pcWriteTo
3. u.pcReadFrom
4. u.uxRecursiveCallCount
5. xTasksWaitingToSend.uxNumberOfItems
6. xTasksWaitingToSend.pxIndex
7. xTasksWaitingToSend.xListEnd.xItemValue
8. xTasksWaitingToSend.xListEnd.pxNext
9. xTasksWaitingToSend.xListEnd.pxPrevious
10. xTasksWaitingToReceive.uxNumberOfItems

11. xTasksWaitingToReceive.pxIndex
12. xTasksWaitingToReceive.xListEnd.xItemValue
13. xTasksWaitingToReceive.xListEnd.pxNext
14. xTasksWaitingToReceive.xListEnd.pxPrevious
15. uxMessagesWaiting
16. uxLength
17. uxItemSize
18. cRxLock
19. cTxLock
20. uxQueueNumber
21. ucQueueType

6.3 Host-side FIEparser.py script

The Python2.7 FIEparser.py program has to be used after the injection campaign is terminated and both the injection and golden files have been created. It aims at comparing the results of each experiment of the two runs in order to see if, under a particular combination of injection parameters, the operating system has shown a misbehavior of any kind.

6.3.1 Parsing algorithm

The parser exploits an empirical approach to categorize the results of the injections, based on the difference found between the injection run file and the golden run file. Logged data are written on a file, one text line per experiment, and they are made of three parts: injection parameters, benchmark results and number of iterations each task was executed for, general system and injection status. The format of the experiment outcome, saved on the log file in a single line, is shown below (here split in three lines). Meaning of names has been already explained in previous chapters.

$$[cycle]\,[bit]\,[locus]\,[timer\,period]\,[timer\,prescaler]$$

$$[BM\_CRCOK]\,[BM\_CRCWRONG]\,[BM\_MICROIT]\,[BM\_MACROIT]$$

$$[SYS\_INJOK]\,[SYS\_INJ]\,[SYS\_OK]\,[SYS\_ETAF]$$

Comparisons are made between benchmarks results and number of iterations to understand if, after the injection all the tasks are still working or not. It is important to highlight that very small differences in the execution of the tasks are not detected: if the number of times each instantiated task was scheduled is the same between the two experiments - under the same experimental conditions - in the injection and golden runs, the injection is classified as 'silent', even if a slight difference was actually present. In order to tune this diversity threshold, the script can receive as input an additional parameter called *tolerance*, which is used internally to discriminate between 'silent injection' and 'misbehavior': only if the difference between injection and golden experiment exceeds such value, result is classified as misbehavior.

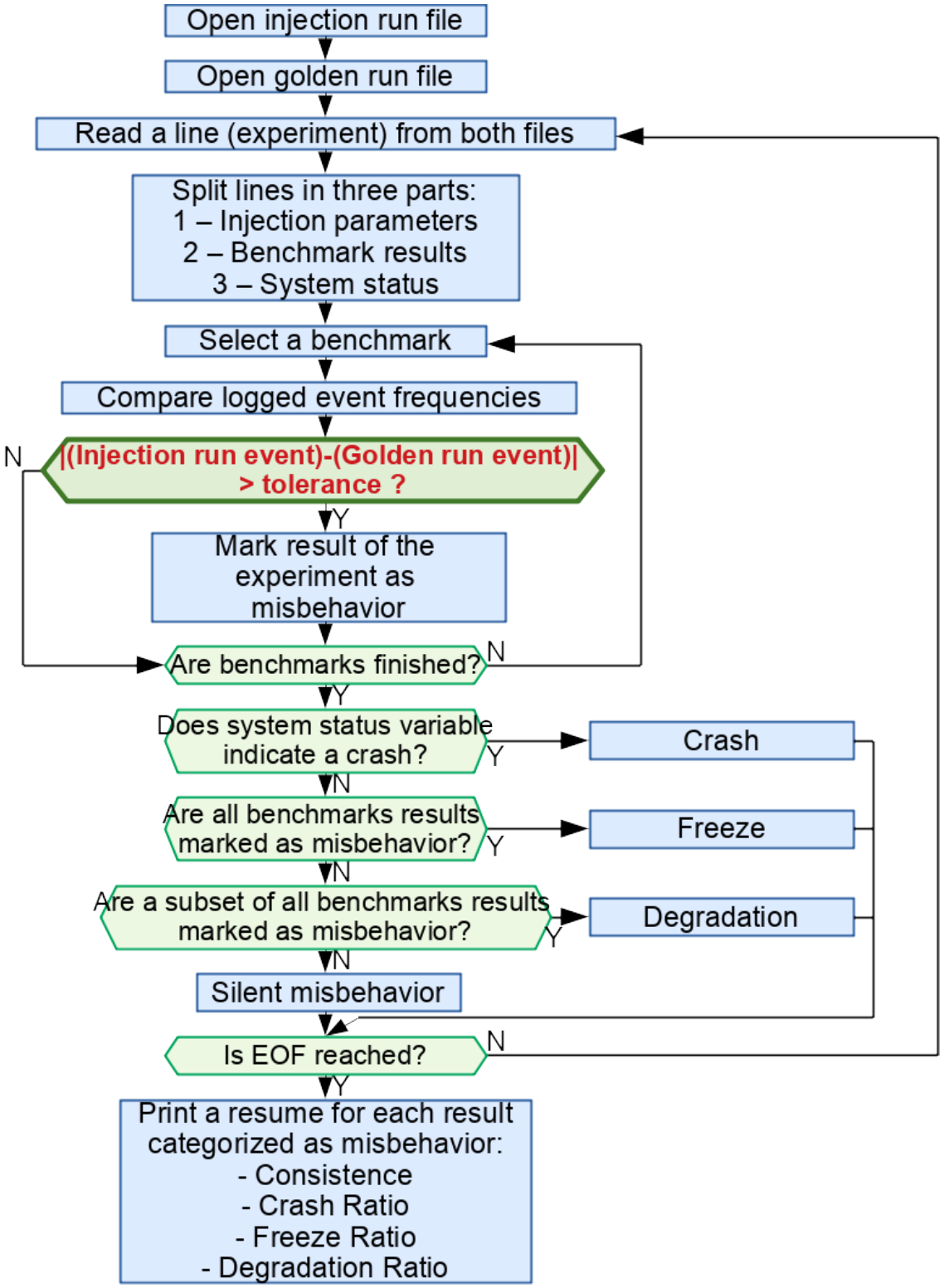

Figure 6.1: Generic flow diagram of the parsing algorithm used to extract results

Chapter 7

Benchmarks under test

In order to perform experiments using standard programs and so to ensure repeatability of the injections, EEMBC® Automotive suite [13] has been used: it is a set of benchmarks belonging to the Multibench™ suite that reproduce some very common calculations in the automotive field; they are thought to be used under an enclosing environment written specifically for UNIX/Linux systems called MITH (Multi-Instance Test Harness), that allows to instantiate a chosen subset of benchmarks among the given ones in the same run and to tune the parameters of the execution. These benchmarks have been written for multicore processors to test the scalability of the platform used, analyzing the distribution of the workload among the different cores and eventually helping to find bottlenecks in the schedule and in the execution; however, they have been used in a single core microcontroller because the performance analysis was not relevant for this work.
16 benchmarks were available: 13 of them have been ported successfully to the embedded environment whereas the remaining 3 have been discarded because of some difficulties encountered during the translation (benchmark requiring too much memory); MITH environment, instead, wasn't ported because it would have used too much memory.
Each benchmark is given with three elements, listed below.

Workload It is the algorithm executed, which can be single-thread or multi-thread.

Dataset Every benchmark is given with different set of input data, provided in the form of files, each one with a different number of input values. More or less all the benchmarks have a '4M' dataset (about 4MB of data) and a '4k' dataset (about 4kB of input data), even if, in some special cases, some of them use a reduced amount of input values.

CRC In order to check if the calculations have been done in the right way, various precalculated CRCs are provided, each one for a different dataset. These CRCs are included in the code and they are used by a 'check' function which performs the CRC calculation on the fly on the output values and compare it with the given one. Such function has to be called when the main algorithm ends.

All the benchmarks come with a sequence of functions that must be called in a defined order: their identifiers end with the name of the benchmark they are written for, but the operations done are similar in all the cases. The calling order is shown in the list below: the general benchmark name is replaced by an asterisk.

- **define_params_*** This function allows to setup the most general benchmark parameters like the dataset to be used and the correct given CRC related to that dataset. The input file containing such dataset is read and all the values are instantiated in a RAM file. Even if one wants to execute many times the benchmark, this function must be called only one time.

- **bmark_init_*** The parameters related to the current run of the benchmark are set: the memory required for the output file is allocated and all the counters are reset. If one wants to execute the whole benchmark many times, this function must be called every time the new iteration begins.
- **t_run_test_*** This is the function which actually contains the benchmark algorithm. It is made of a loop that allows to repeat the algorithm many times without having to clear and reinitialize the parameters: at every iteration it just repeats the calculations overwriting the results in the output file of the previous iteration.
- **bmark_verify_*** This function is the one charged to calculate the CRC on the fly and to compare it with the given one: it just returns a boolean value as result of the comparison.
- **bmark_fini_*** When the CRC has been calculated, this function must be called to free the memory allocated for the output file, preparing, eventually, the benchmark for another run. It is the opposite of *bmark_init_**.
- **bmark_clean_*** This function completely clears the memory and resets the variables used by the benchmark. It is the opposite of *define_params_**.

7.1 a2time - Angle to time conversion

This benchmark simulates a mechanism made of an engine with different cylinders (4, 6 or 8, to be chosen before compilation) with a crankshaft, a toothed reluctor wheel and a sensor able to generate a pulse every time it detects the passage a tooth: this type of mechanism is used to control the injection of fuel in the various cylinders and the subsequent spark. The wheel usually has one or more missing teeth that are used as reference point during the calculations. Differential angle between two subsequent points in the dataset is found (they are adjusted in case of overflow of the toothed wheel); then a counter keeping track of the shaft revolution is updated; when a cylinder finds an angle value that is the right one to perform ignition - this is called 'firing angle' - a 'fire' signal is asserted for that element of the engine. In the output file both engine speed and firing times for the various cylinders are stored.

7.2 aifftr - Fast Fourier Transform

This benchmark calculates the power spectrum of a time varying signal with the Fast Fourier Transform (FFT), by using a radix-2 decimation algorithm on complex input values. First of all a bit reversal vector is found and then applied to dataset points.

7.3 aifirf - Finite Impulse Response filter

This benchmark emulates a Finite Impulse Response filter; each input sample is given to two different pipelines that act one as low-pass filter and the other one as high-pass filter; their order is 35 in both cases. Their outputs are written back into the memory in an interleaved way.

7.4 aiifft - Inverse Fast Fourier Transform

This benchmark calculates the values assumed by a signal in the time domain starting from its spectrum and using the Inverse Fast Fourier Transform.

7.5 bitmnp - Bit manipulation

This benchmark simulates an algorithm that receives as input numbers in BCD format and shows them on a display: it updates the content of the screen by refreshing column by column. This algorithm uses intensively conditionals to stress the logical units of the hardware.

7.6 idctrn - Inverse Discrete Cosine Transform

This benchmark simulates the Inverse Discrete Cosine Transform widely used in digital graphics; it is applied to an input dataset representing a matrix of 64 bits values.

7.7 iirflt - Infinite Impulse Response filter

This benchmark implements a second order Infinite Impulse Response filter in the direct form II. Even in this case two filters operating on same input values are present in the algorithm, a low-pass and a high-pass filter.

7.8 matrix - Matrix arithmetic

This benchmark performs many calculations with matrix arithmetic. Specifically, LU decomposition of the matrix is done, then its determinant is calculated.

7.9 pntrch - Pointer chasing

This benchmark performs a linear search of a token in a doubly linked list. Each token to search for in the list is taken from the dataset and if it is found, then, the number of steps done through the list is saved in the output file stored in the memory and then next token is retrieved.

7.10 puwmod - Pulse Width Modulation

This benchmark simulates a unit controlling a H-bridge motor driver (figure 7.1) which can drive the motor both in clockwise and counterclockwise directions. The algorithm returns a set of output values that can be applied to two H-bridges, useful to control a bipolar stepper motor driver.

7.11 rspeed - Road speed calculation

This benchmark estimates the road speed by taking values from a digital timer and pulses from a tonewheel. Like *a2time* benchmark, the mechanism simulated is made of a shaft

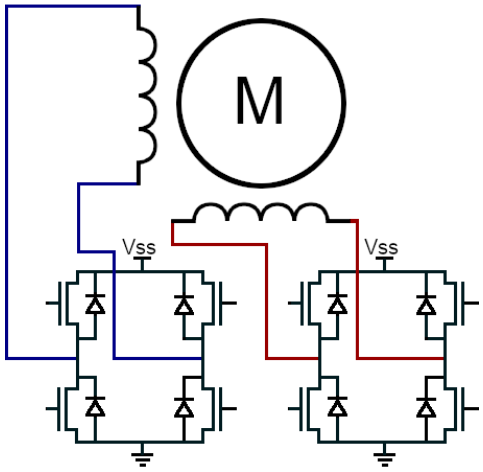

Figure 7.1: Bipolar stepper motor driver control circuit with connections to internal coils

that has a wheel and a tonewheel attached, with a sensor able to detect movement of the latter; when such event occurs, an internal cumulative variable is updated, adding to it the value of the timer and then, if required, the speed is calculated by dividing the number of teeth by the total time. When this is done, cumulative variable is reset and the algorithm is repeated with the next set of data coming from the timer. Speed calculation is done two times per revolution of the wheel.

7.12 tblook - Table lookup and interpolation

This benchmark simulates a solution that can be used in embedded systems when the chosen hardware has few RAM available: instead of saving all the samples coming from one or more resources (sensors, connections, calculations), only a subset of them is actually saved and then they are interpolated in order to find, approximately, missing points.

7.13 ttsprk - Tooth to spark algorithm

This benchmark simulates the regulation of the injection and ignition processes in an engine. In a control unit a tooth-to-spark algorithm controls the amount of fuel to inject in the cylinder, the mass of air to be used during the combustion and regulates the timing of the entire process according to operating conditions.

Chapter 8

Experimental results

8.1 Experiments summary

The Fault Injection Environment allows to perform a very high number of injections under many possible conditions and with different settings; in order to have a consistent amount of experiments which are statistically relevant and to not spend too much time to perform all the campaigns, only a subset of 3 applications, each one running one benchmark, have been extensively tested: *a2time*, *idctrn* and *tblook*; on the other benchmarks only a reduced number of tests have been conduced. Table 8.1 shows which are the experiments done, specifying for each benchmark the list and the bits where injection was performed. All experiments simulated the effect of SBUs, using the bit-flip model. Injections have been done at fixed time instants for 0 to 7 LSBs exploiting INJ mode of the FIE, while injections in MSB have been done at random times, setting the FIE to RAD mode: in the former case 50 time instants have been tested (chosen randomly but fixed for all the faults in the list) while in the second case 500 injections have been done at different random times for each fault in the selected list. Table 8.2 contains the total number of experiments done in the various lists, dividing between injections in 8 LSBs and in MSB. In the following sections results are analyzed and, for each relevant fault, an explanation of the observed behavior of the system is provided if possible. At the beginning of each section, a table is provided with all the faults that gave a result; some of them have a consistency close to 0% and for this reason an exact value is not provided but ~0% is used: such values are not considered relevant for the purposes of the discussion.
In following tables, 'CURTCB' stands for current task TCB, 'RDYTCB' for ready task TCB, 'DLDLST' for delayed task list and 'RDYLST' for ready tasks list.

NOTE: All the following results have been extracted using a value for the tolerance parameter equal to 10.

	List 1	**List 2 CURTCB**	**List 2 RDYTCB**	**List 3 DLDLST**	**List 3 RDYLST**	**List 4**
a2time	0-7,MSB	0-7,MSB	0-7,MSB	0-7,MSB	0-7,MSB	0-7,MSB
aifftr	0-7	0-7				
aifirf	0-7	0-7				
aiifft	0-7	0-7				
bitmnp	0-7	0-7				
idctrn	0-7,MSB	0-7,MSB	0-7,MSB	0-7,MSB	0-7,MSB	0-7,MSB
iirflt	0-7	0-7				
matrix	0-7	0-7				
pntrch	0-7	0-7				
puwmod	0-7	0-7				
rspeed	0-7	0-7				
tblook	0-7,MSB	0-7,MSB	0-7,MSB	0-7,MSB	0-7,MSB	0-7,MSB
ttsprk	0-7	0-7				

Table 8.1: Summary of experimental injection campaigns performed with target bits specified

	List 1	**List 2 CURTCB**	**List 2 RDYTCB**	**List 3 DLDLST**	**List 3 RDYLST**	**List 4**
0-7LSB	4000	7600	7600	2000	2000	8800
1MSB	5000	9500	9500	2500	2500	11000

Table 8.2: Number of experiments per list, divided by target bits

8.2 List 1 - Global FreeRTOS variables

8.2.1 Bits 0-7 injection results

Fault number	Fault name	Consistency
1	xTickCount	C<10%
2	uxTopReadyPriority	80%<C<90%
3	xSchedulerRunning	10%<C<20%
8	xNextTaskUnblockTime	C~0%
9	uxSchedulerSuspended	C=100%

Table 8.3: Faults producing misbehaviors for experiments in list 1, 0-7 LSB

Results obtained when injecting in 8LSB are more or less the same for all the benchmarks: faults which gave some results are 1,2,3,8 and 9, corresponding to *xTicksCount*, *uxTopReadyPriority*, *xSchedulerRunning*, *xNextTaskUnblockTime* and *uxSchedulerSuspended*. No crash occurred because when injecting in LSBs it is unlikely to modify sensitive data which are used, for example, to index elements in lists or vectors. All misbehaviors are recognized as freezes or degradations which means that global variables, in the case of slight modifications, are used mainly to regulate the scheduling process.
1) Experiments in locus 1 (*xTicksCount*) gave results only for injections in the 7th bit: such variable is used by the kernel in xTaskIncrementTick() to understand when a task must be moved back from the xDelayedTaskList1 list to the right pxReadyTasksLists (in order to awake it after the delay timeout elapsed), so it is easy to understand why it gives a misbehavior only when the injection is done on higher bits and such misbehaviors are all degradations (change of the tasks scheduling without any sudden error): such variable is compared to xNextTaskUnblockTime and, only if it is greater, the check on delayed tasks is done and its change leads to a different scheduling of the tasks.
2) Fault 2 (*uxTopReadyPriority*) causes mainly freezes and few degradations: this is due to the fact that such variable is used to select the right ready task, if any available, to be moved to running state among the ones in the pxReadyTasksLists vector, using the macro taskSELECT_HIGHEST_PRIORITY_TASK(); if this variable points erroneously to a position in the list where no task is available, system freezes in a loop thanks to a preventive configASSERT() check on the ready tasks list.
3) Variable related to fault 3 (*xSchedulerRunning*), instead, gave results only for injections in 1st LSB: it is used to know if the scheduler is suspended or not and if yes, no task is scheduled, so, when the injection is done on the 1st LSB (the only one used to understand the state of the scheduler), the whole system halts, producing a freeze.
8) This variable (*xNextTaskUnblockTime*) has a low consistency but such result depends heavily on the tolerance set to perform the post campaign analysis: decreasing such parameter, the consistency of such fault increases from ~0% up to a value comprised between 5% and 10%. This is due to the fact xNextTaskUnblockTime, when affected by an injection in LSB, changes the behavior of the system in a very slight way, modifying the scheduling sequence of tasks, and such difference could be not seen if tolerance is too high. The presence of such misbehavior is due to comparison made by the function xTaskIncrementTick() with xTicksCount, to understand if there is a delayed task that needs to be moved back to the ready list as its timeout has expired.

9) Fault 9 (*uxSchedulerSuspended*) always produces a freeze: this is due to the fact that this variable is always checked by a configASSERT() before it is used: in all cases, if its value is different from the expected one, the system is blocked by an infinite loop. Such checks are done in the function vTaskDelay(), in xTaskResumeAll() and in vTaskSuspend-All(): the first function is often used by the instantiated tasks, while the other twos are called internally by vTaskDelay() itself and by other debug functions, to retrieve system information, ensuring thus that a check on uxSchedulerSuspended is done in any case.

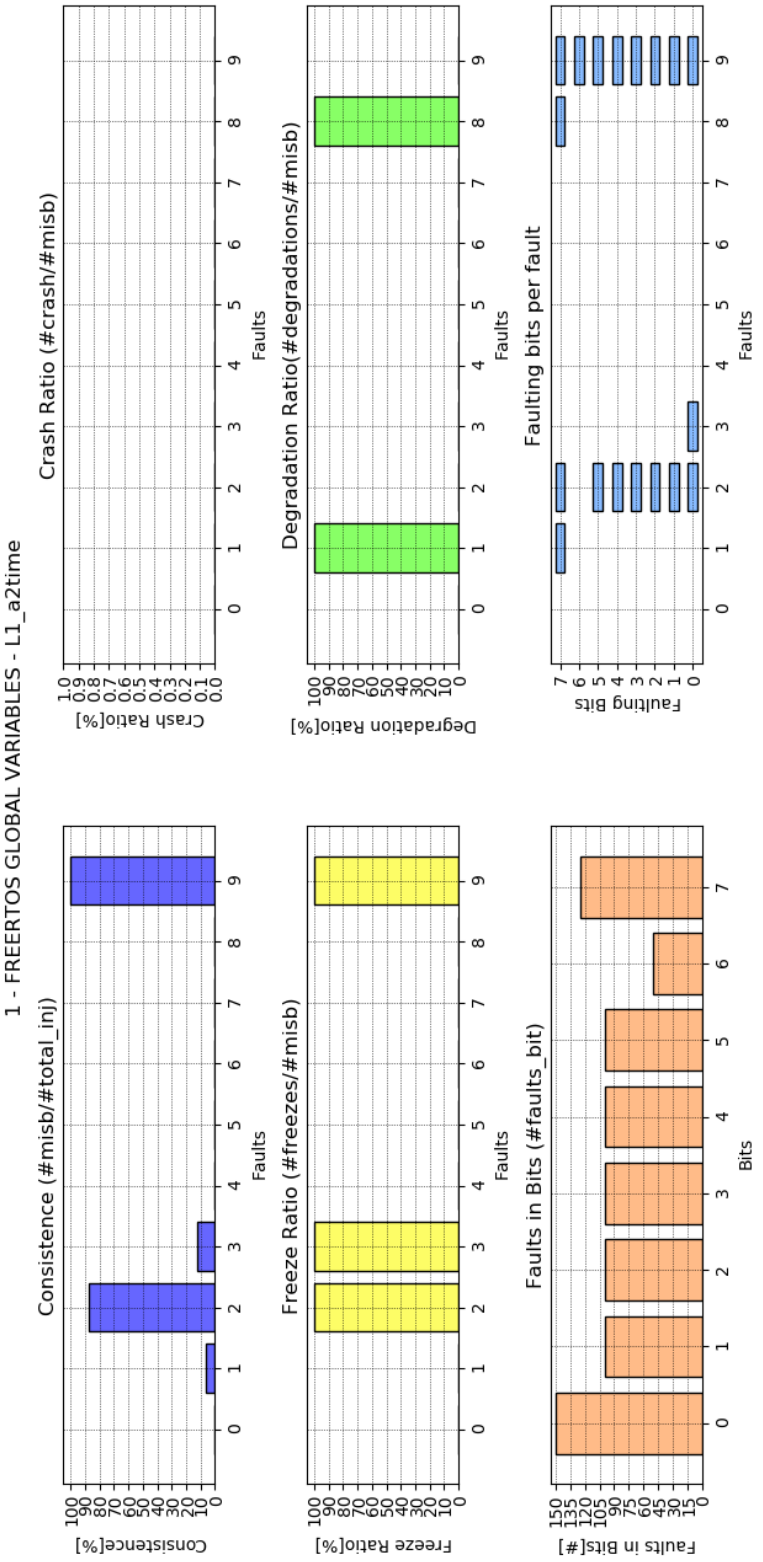

Figure 8.1: Results of injections in 8LSB, using a2time benchmark

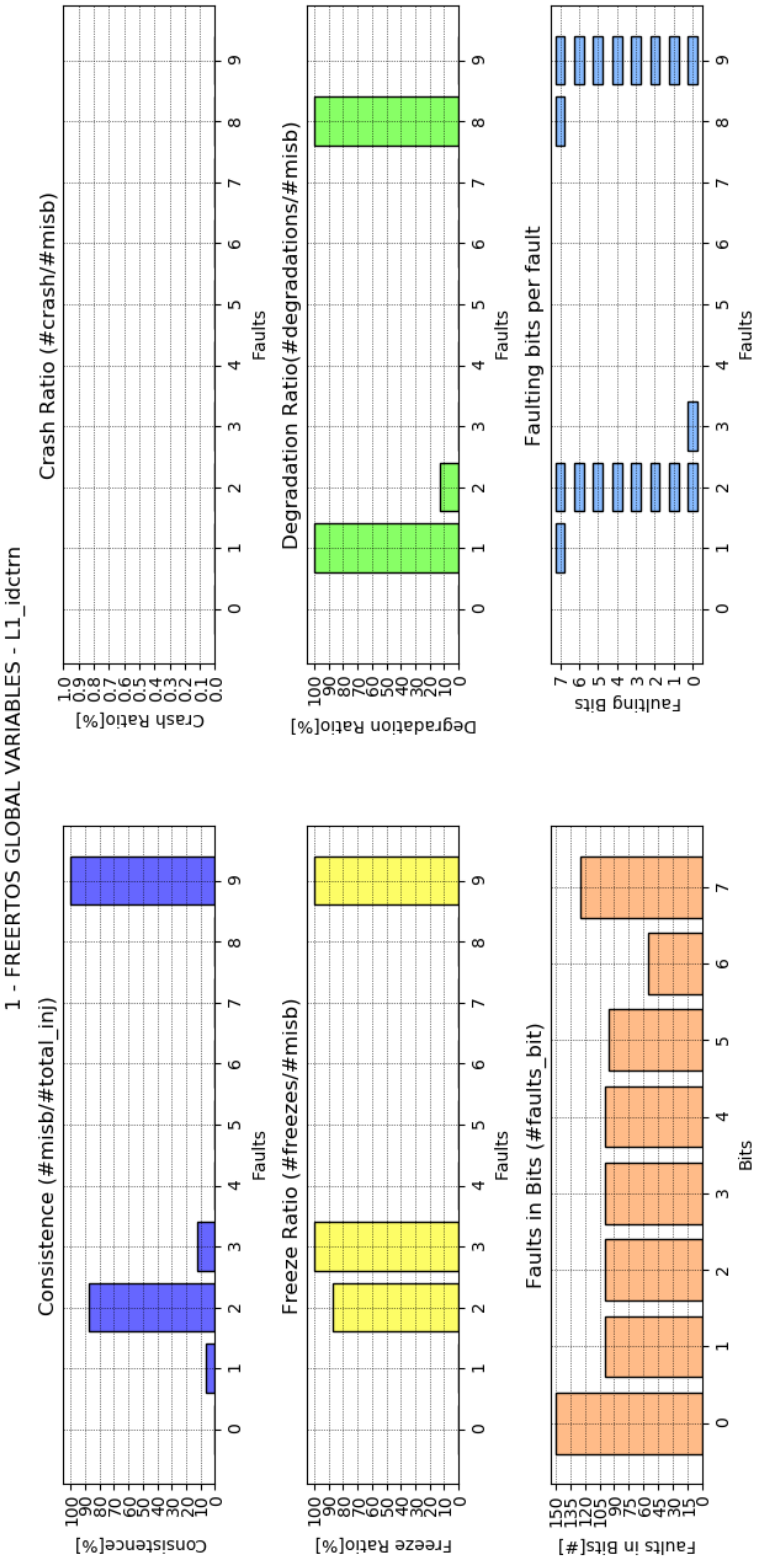

Figure 8.2: Results of injections in 8LSB, using idctrn benchmark

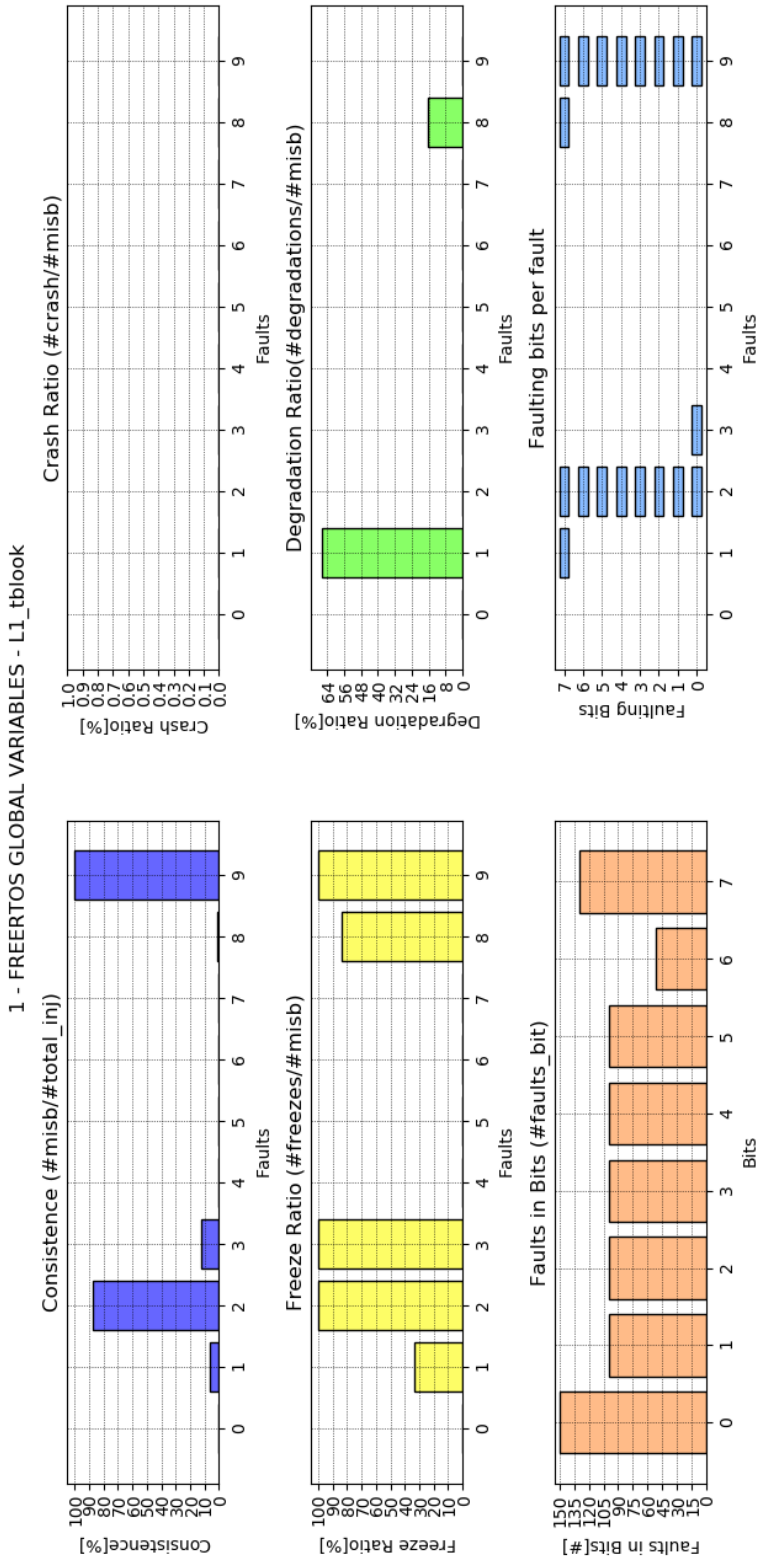

Figure 8.3: Results of injections in 8LSB, using tblook benchmark

8.2.2 MSB injection results

Fault number	Fault name	Consistency
2	uxTopReadyPriority	C=100%
4	uxPendedTicks	C=100%
8	xNextTaskUnblockTime	5%<C<50%
9	uxSchedulerSuspended	C=100%

Table 8.4: Faults producing misbehaviors for experiments in list 1, 1 MSB

When injecting on the MSB, instead, results are slightly different: misbehaviors in faults 2,8 and 9 are conserved, faults 1 and 3 disappear and fault 4 appears.
2) This time *uxTopReadyPriority* is heavily modified by the injection in the MSB, so the selection of the ready task with the highest priority in the vector pxReadyTasksLists is done trying to access to a completely wrong position in the memory: this leads to a crash.
4) Fault 4 is *uxPendedTicks* and injecting here always produces a freeze of the whole system: as said before, function xTaskIncrementTick() always checks if the scheduler is suspended or not; during a normal operation, if this time it is, the value of uxPendedTicks is incremented by one. When xTaskResumeAll() is called (by vTaskDelay() for example), all the pended ticks happened while the scheduler was stopped are solved; however, after the injection in the MSB, the value stored in uxPendedTicks becomes huge (flip is always made from 0 to 1 and this variable is unsigned) and so the whole system freezes because it is busy to solve, emptily, an enormous number of pended operations.
8) In MSB injections, *xNextTaskUnblockTime* fault becomes more relevant: this unsigned variable is compared to xTickCount in xTaskIncrementTick() and, only if it is greater than the tick count, the delayed task with the satisfied elapsed delay is moved back to the ready list. When the MSB of this variable is flipped (always from 0 to 1) and if all the tasks are already in the delayed state, no task is awaken because it would seem that no delayed task has reached its timeout delay for a long time; low consistency of this fault is due to the fact that if the injection happens when there is at least one task in the ready or running state, it will overwrite this variable as soon as it will call vTaskDelay(), nullifying the effect of the injection (this value is updated by the delaying function with the new value imposed by the task that is going to be retarded). It must be noted that, this time, unlike LSB experiments, all misbehaviors are system freezes and not degradations.
9) Fault 9 (*uxSchedulerSuspended*) has the same behavior with same explanation of the LSB experiments.

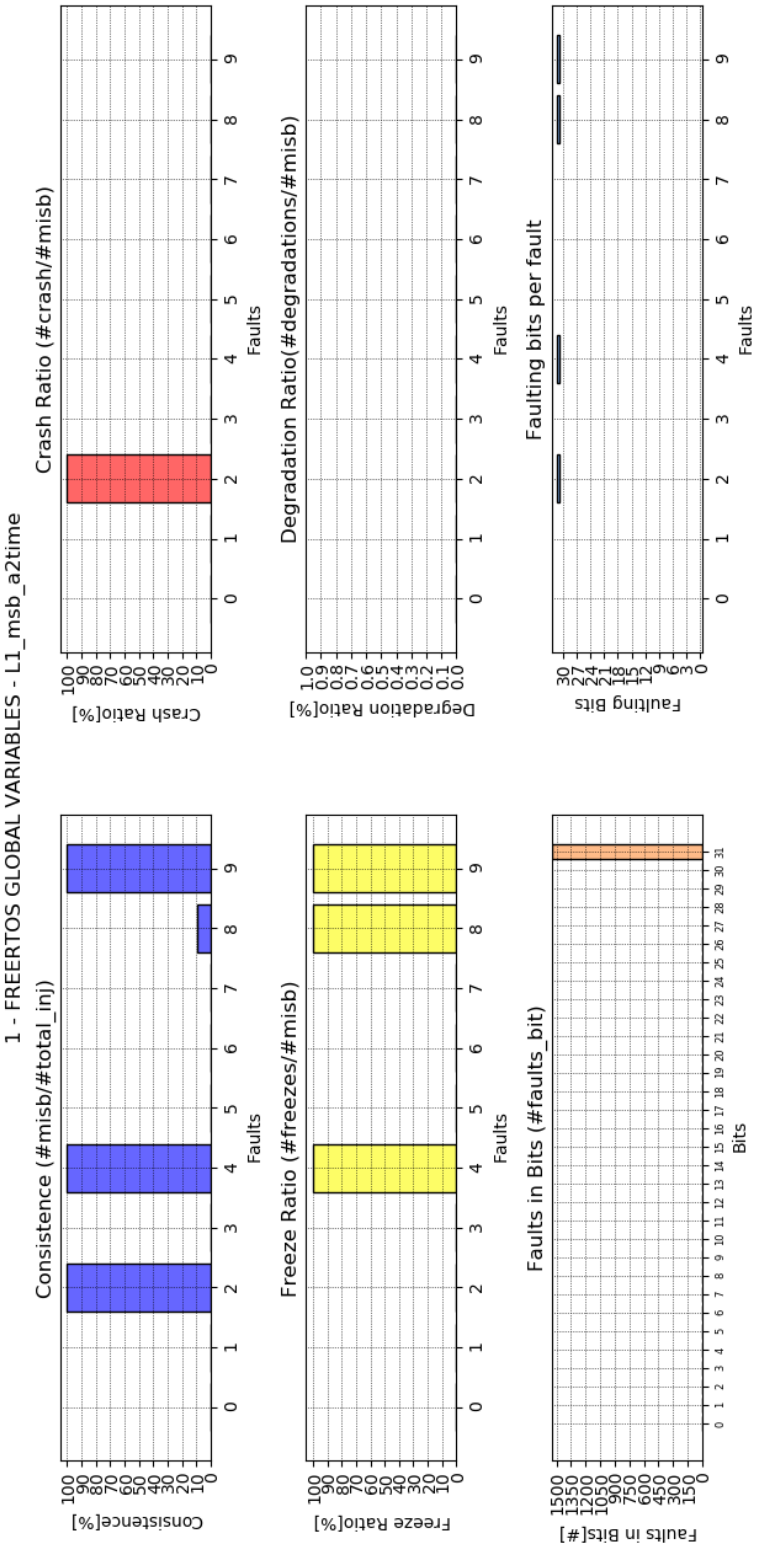

Figure 8.4: Results of injections in 1MSB, using a2time benchmark

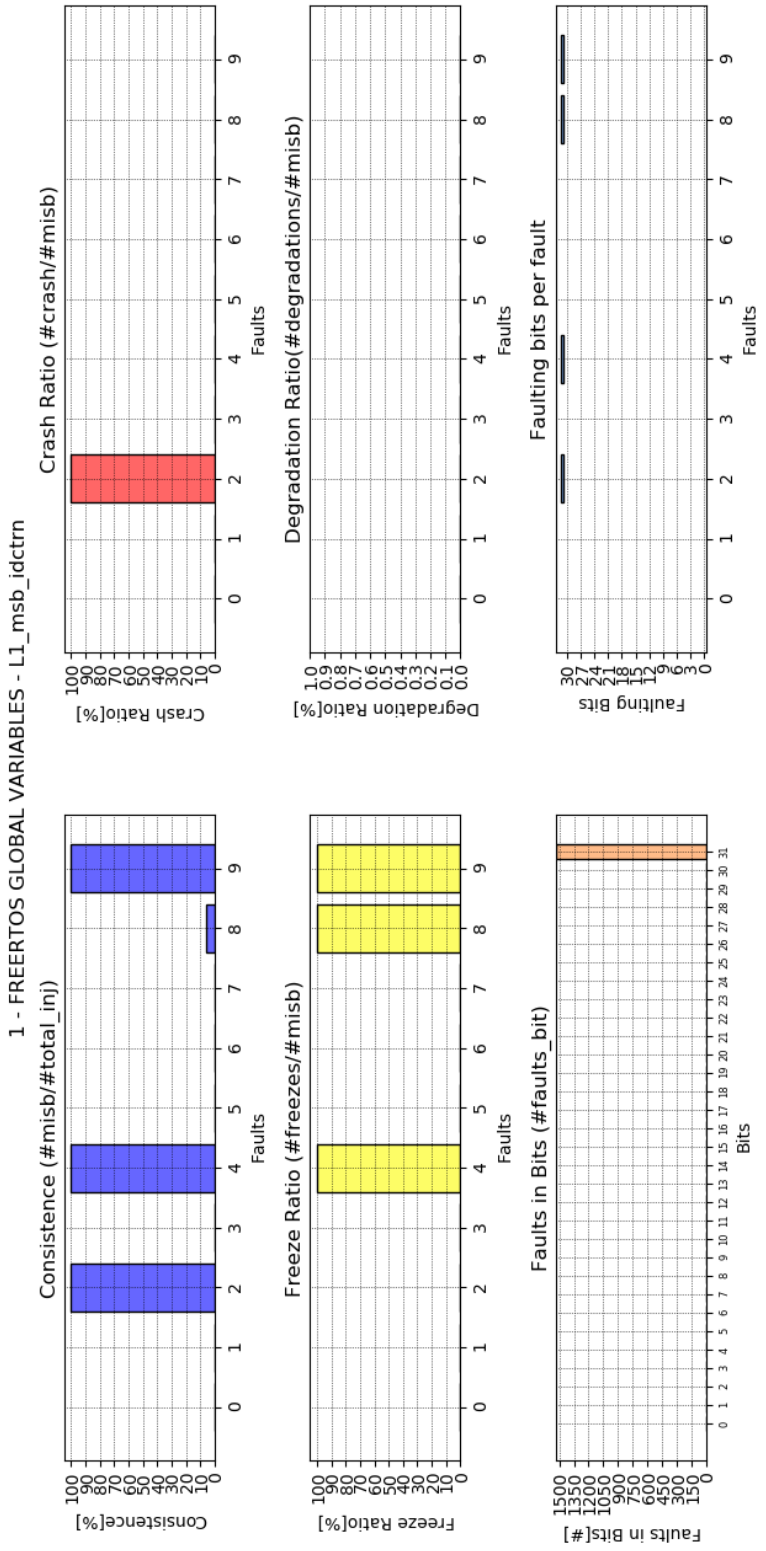

Figure 8.5: Results of injections in 1MSB, using idctrn benchmark

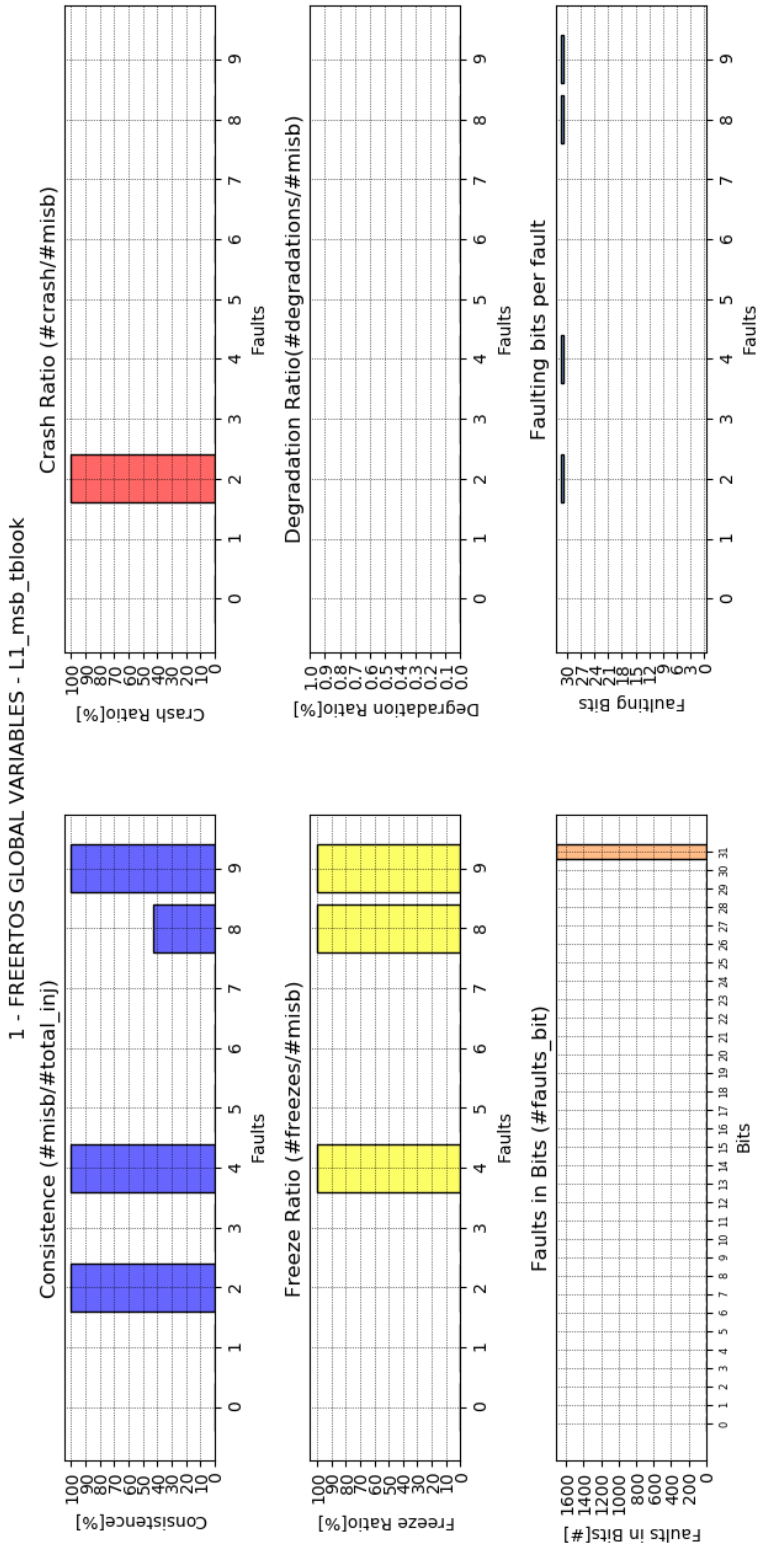

Figure 8.6: Results of injections in 1MSB, using tblook benchmark

8.3 List 2 - Current task TCB

8.3.1 Bits 0-7 injection results

Fault number	Fault name	Consistency
1	uxPriority	50%<C<90%
2	pxStack(*idctrn* only)	40%<C<50%
5	uxBasePriority(*idctrn* only)	30%<C<40%
6	uxMutexesHeld(*idctrn* only)	C~0%
10	xStateListItem.pxNext	C=100%
11	xStateListItem.pxPrevious	60%<C<100%
12	xStateListItem.pvOwner	C=100%
13	xStateListItem.pvContainer	60%<C<100%
18	xEventListItem.pvContainer	60%<C<100%

Table 8.5: Faults producing misbehaviors for experiments in list 2, current TCB, 0-7 LSB

Injections in LSBs gave relevant results in faults 1, 2, 5, 10, 11, 12, 13 and 18 (corresponding to variables *uxPriority*, *pxStack*, *uxBasePriority*, *xStateListItem.pxNext*, *StateListItem.pxPrevious*, *xStateListItem.pvOwner*, *xStateListItem.pvContainer* and *xEventListItem.pvContainer*).
1) Fault 1 (*uxPriority*) causes mainly freezes. This variable is used by the kernel when a task is moved back from the delayed state to the ready state, to know in which ready list it must be added among the possible priority-ordered lists. Injections in this location could lead to point to a wrong memory; moreover, if the value is wrong and the new required list is empty (because of the injection), a configASSERT() in taskSELECT_HIGHEST_PRIORITY_TASK() brings the system to an infinite loop: this explains the freezes.
2) Fault 2 (*pxStack*) produces misbehaviors only in the multithread application, showing only freezes: this happens because in this particular application some tasks, actually performing the benchmark calculations, are created and deleted at runtime, exploiting respectively the functions xTaskCreate() and vTaskDelete(); in order to completely remove such tasks from the system and to free their allocated memory, the latter function needs some information present in their TCBs, and one of these information is the base point of the allocated stack: when the stack, which is made of concatenated lists, is deallocated - using FreeRTOS and not standard C functions - two checks (on the block size and on the next free block status) are performed and if one of them fails the systems hangs in an configASSERT() loop.
5) Fault 5 (*uxBasePriority*) causes crashes, freezes and degradations, but only in multithread benchmark idctrn: this happens because, as said before, idctrn is a multithread benchmark which intensively exploits mutexes and in particular the priority inversion mechanism, in the case the various tasks sharing a mutex have a different priority. If this variable is changed by the injection, priority inversion is harmed leading to a wrong scheduling - after the inversion the restored priority is the wrong one and not the original one - and causing freezes and degradations. In other cases such injection leads to an erroneous access to the pxReadyTasksLists vector, since the restored priority contained in uxPriority is the wrong one.

10) *xStateListItem.pxNext* field is used by vTaskSwitchContext(), a fundamental function used by the kernel every time another task must be switched in: such function calls taskSELECT_HIGHEST_PRIORITY_TASK(), a macro used to select, if available, the ready task with the highest priority; listGET_OWNER_OF_NEXT_ENTRY() is the macro which actually changes the current running task with the new one, exploiting exactly xStateListItem.pxNext: injecting in this variable means that the current TCB variable will point to a non sense memory region (which should contain instead a TCB data structure), provoking thus a crash.
11) *xStateListItem.pxPrevious* is instead heavily used by vListInsertEnd() and uxListRemove(): in the first case the new item's pxPrevious is set to pxPrevious of the older item in the list, while, in the second function, pxNext->pxPrevious of the item to remove is set equal to pxPrevious, as in normal insert/remove operations. Changing this value with an injection leads mainly to crashes because the pointer, very often, points to a totally wrong position in memory.
12) The fault 12 (*xStateListItem.pvOwner*) is used, for example, in xTaskIncrementTick() and is accessed by listGET_OWNER_OF_HEAD_ENTRY() to know if there is a delayed task which needs to be moved to the ready list since its delay has expired. This variable is used by vTaskSwitchContext() too: while the variable related to fault 10 is accessed to get the next item in the list of ready tasks, this variable contains the pointer to the TCB of such 'next' item to be eventually switched in: again, as the switch function is called very often, it is obvious to see a consistency of 100% for this fault and a very high number of crashes.
13) *xStateListItem.pvContainer* is a field accessed by xTaskResumeAll() which, if there are some pending operations requiring a movement of the task from delayed to ready list, calls uxListRemove(): this function exploits pvContainer of the item to be removed in order to know the list it belongs to; if this value is different from the expected one, the index (value used by the list to surf among the list items) will continue to point to the item to be deleted, which is in a memory location that is not valid anymore.
18) Misbehavior in locus 18 (*xEventListItem.pvContainer*) can be explained in this way: when calling xTaskResumeAll(), xTaskIncrementTick() is called too and here a comparison between xTickCount and xNextTaskUnblockTime is done to know if there is a delayed task which needs to be awaken; pvContainer (that points to the list containing xEventListItem) is accessed to remove such task from the waiting list, but if it is injected, it will point to a completely wrong place, causing thus a crash.

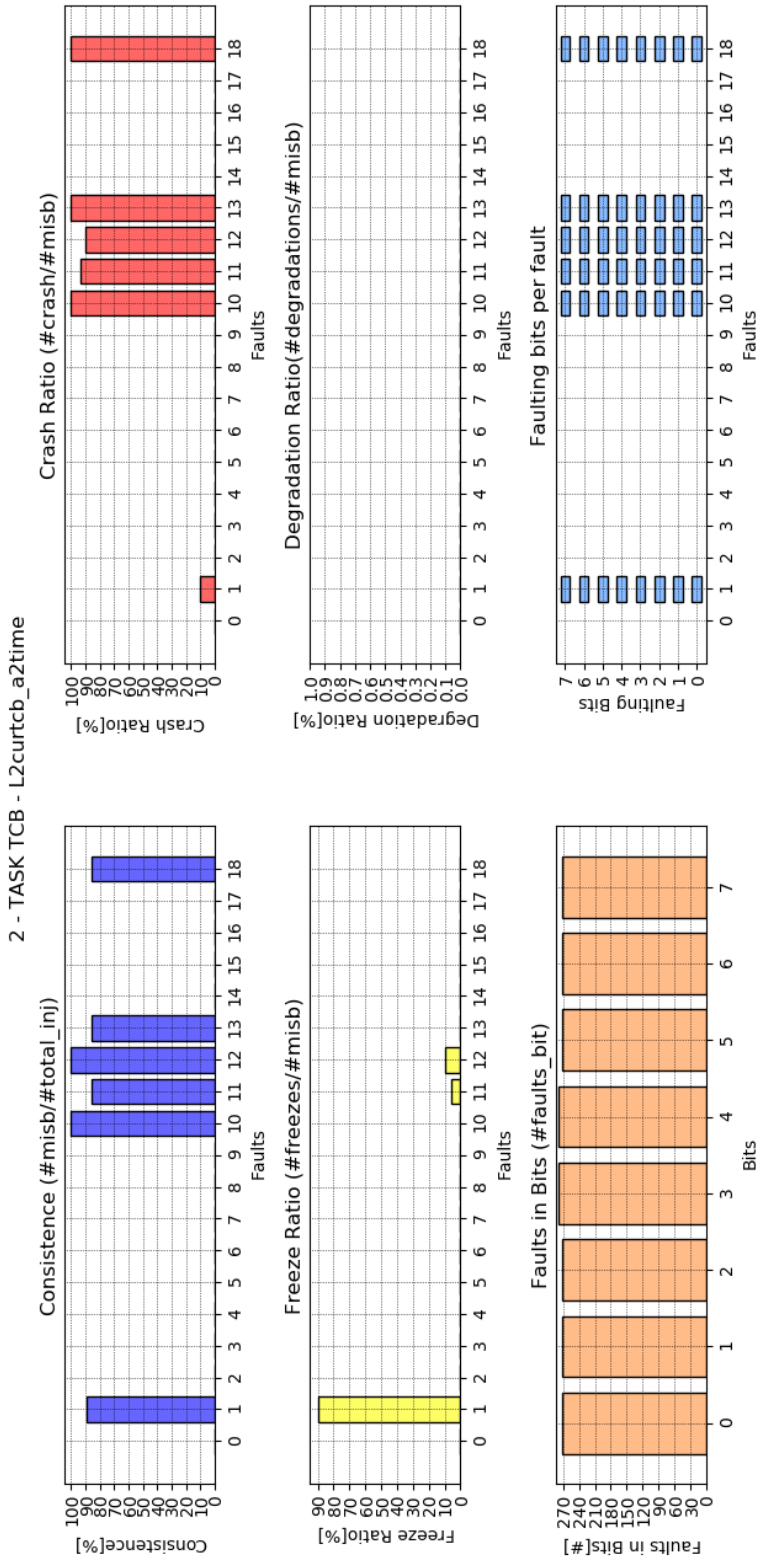

Figure 8.7: Results of injections in 8LSB, using a2time benchmark

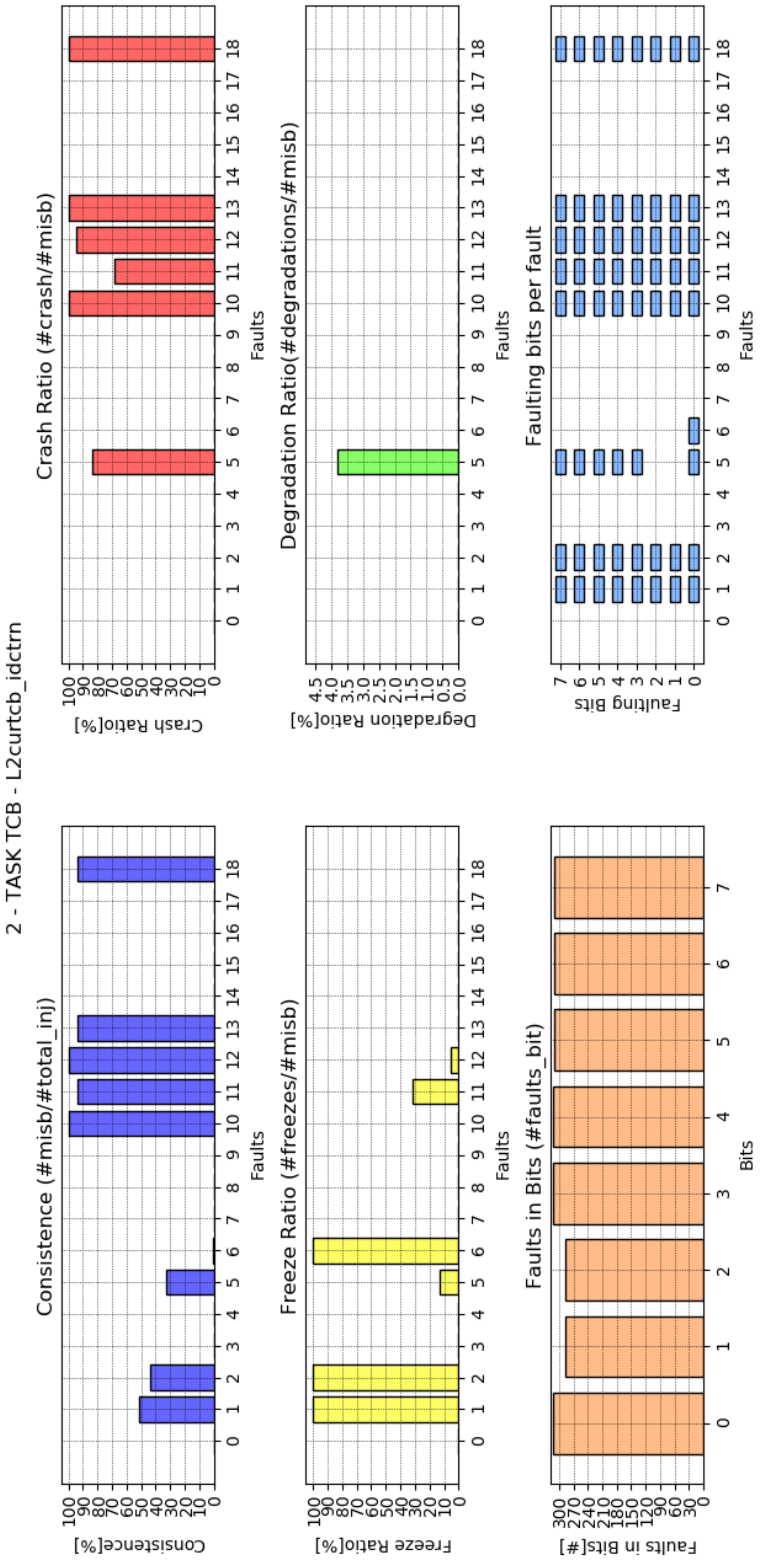

Figure 8.8: Results of injections in 8LSB, using idctrn benchmark

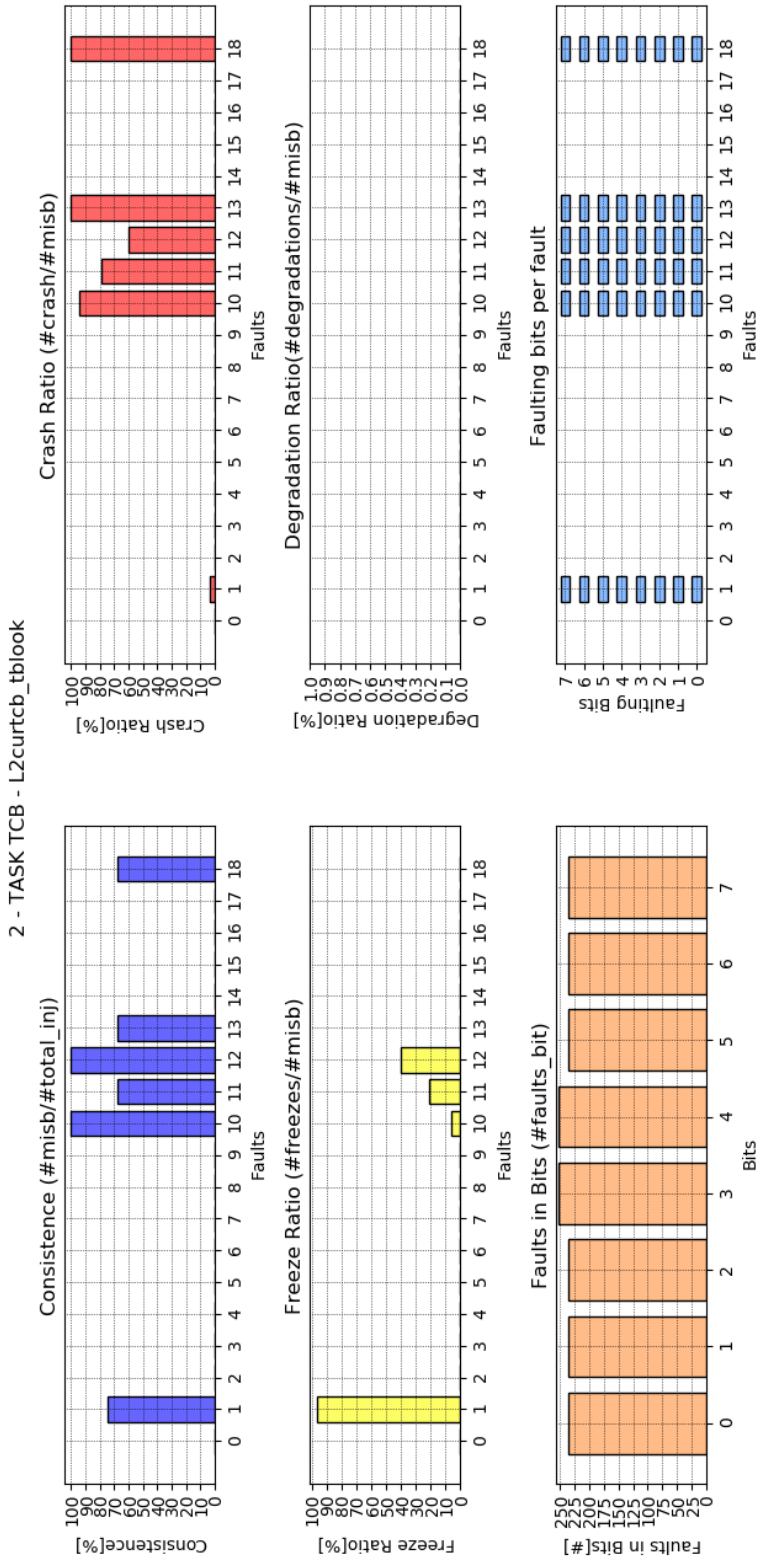

Figure 8.9: Results of injections in 8LSB, using tblook benchmark

8.3.2 MSB injection results

Fault number	Fault name	Consistency
0	pxTopOfStack(*idctrn* only)	C~0%
1	uxPriority	5%<C<70%
2	pxStack(*idctrn* only)	30%<C<40%
10	xStateListItem.pxNext	C=100%
11	xStateListItem.pxPrevious	60%<C<100%
12	xStateListItem.pvOwner	C=100%
13	xStateListItem.pvContainer	60%<C<100%
18	xEventListItem.pvContainer	60%<C<100%

Table 8.6: Faults producing misbehaviors for experiments in list 2, current TCB, 1 MSB

There are, again, visible similarities between the LSB and the MSB set of experiments: all the considerations provided for the LSB injections are still valid.
0) For the multithread benchmark idctrn fault 0 appears too (*pxTopOfStack*), producing crashes only: the other benchmark applications are not affected in this way because stack creation and deletion processes are not used at all.
10, 11, 12, 13, 18) Injections in MSB gives results that are similar to the ones got with experiments on LSBs: faults 10, 11, 12, 13 and 18 continue to have more or less the same behavior causing the 100% of crashes as, this time, the memory accessed by such pointer is absolutely invalid.

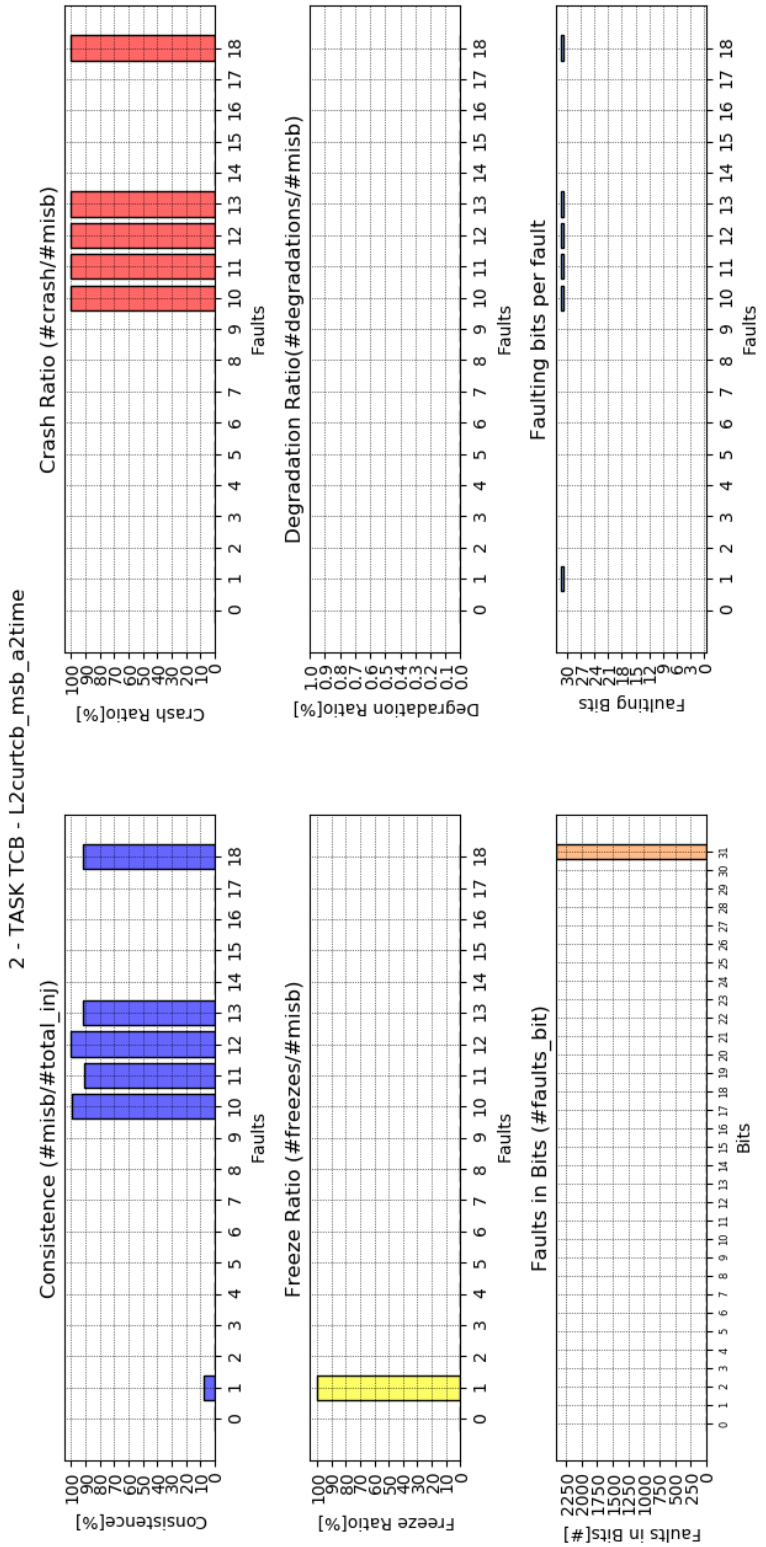

Figure 8.10: Results of injections in 1MSB, using a2time benchmark

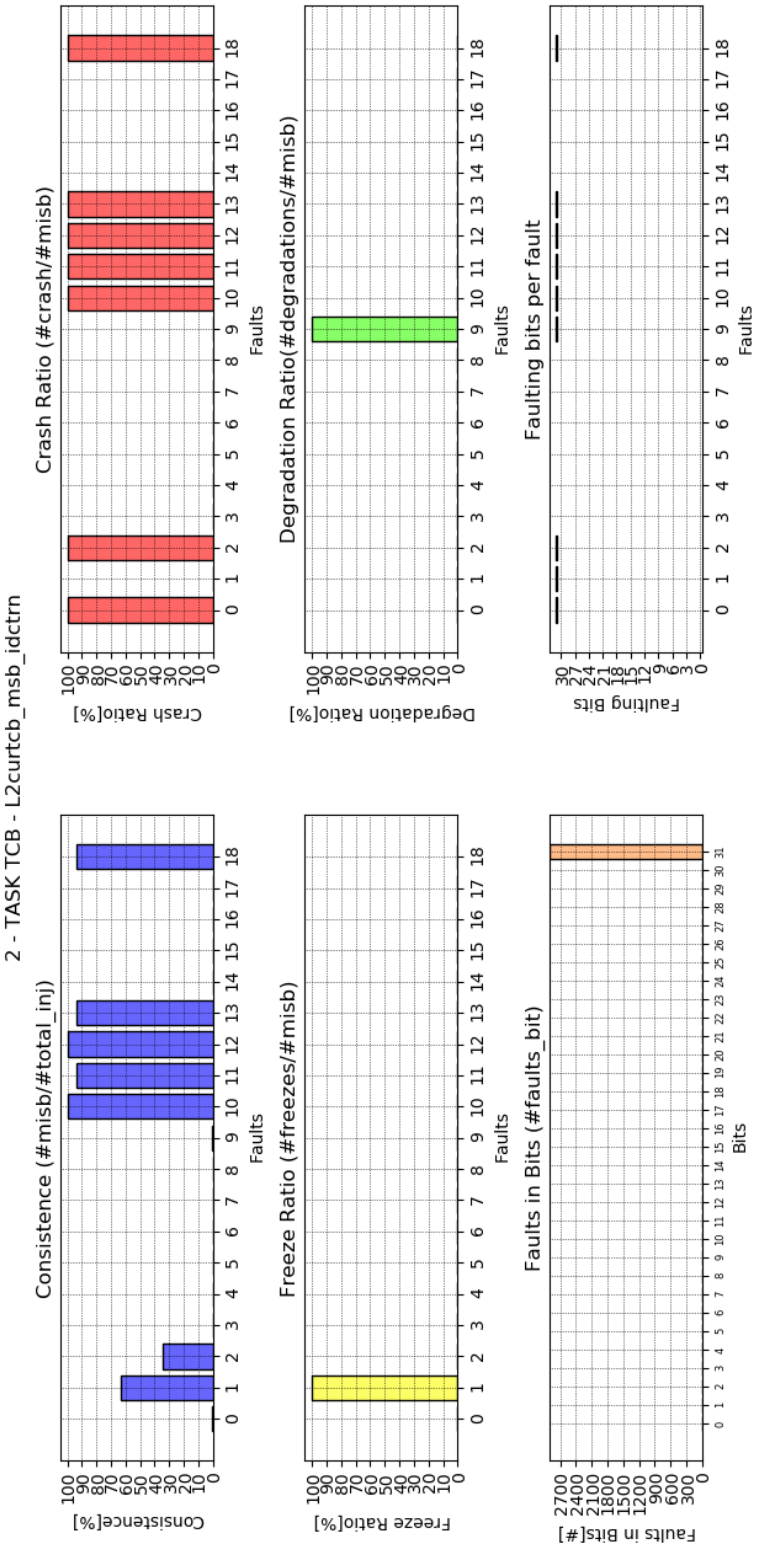

Figure 8.11: Results of injections in 1MSB, using idctrn benchmark

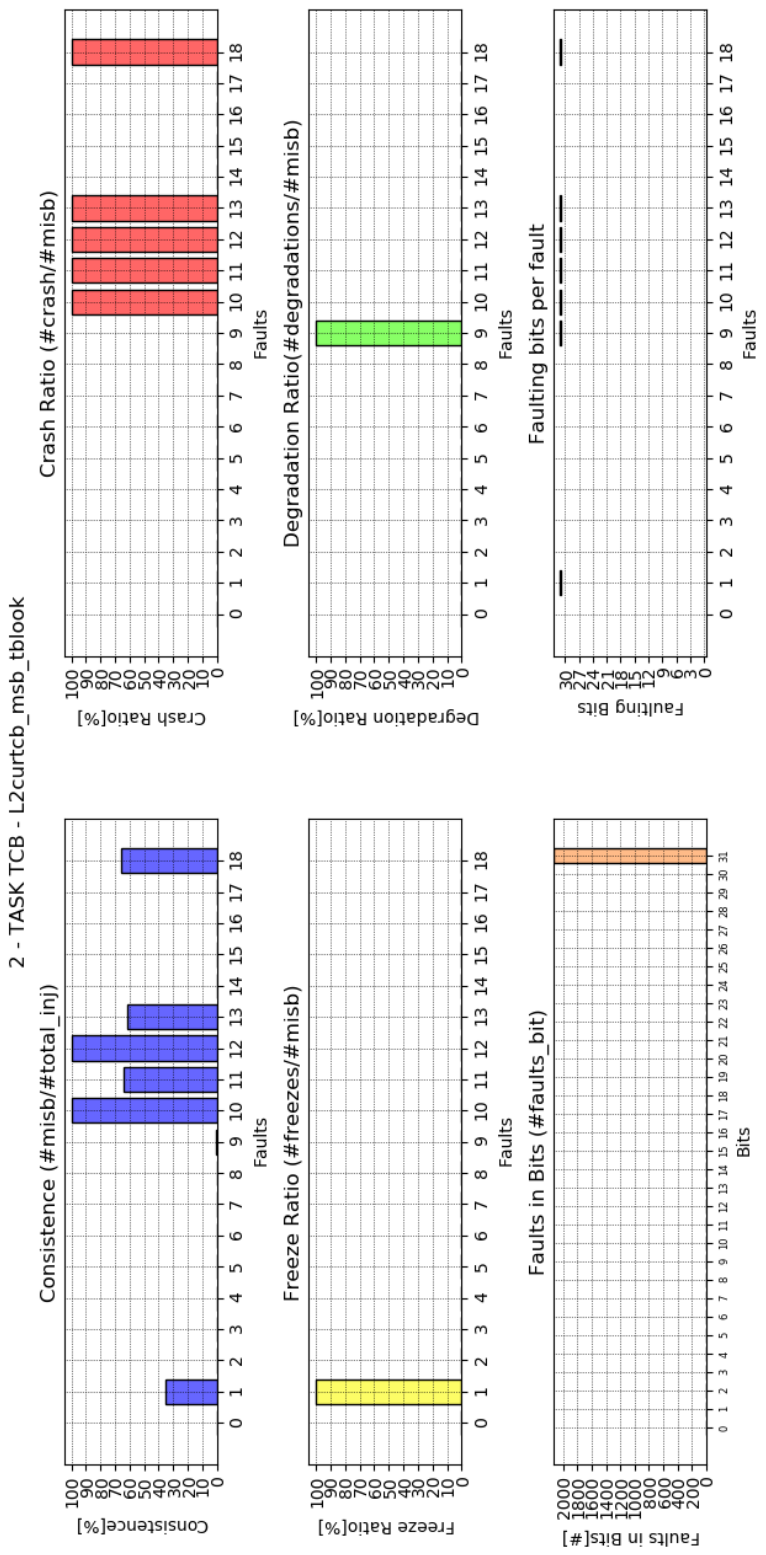

Figure 8.12: Results of injections in 1MSB, using tblook benchmark

8.4 List 2 - Ready task TCB

8.4.1 Bits 0-7 injection results

Fault number	Fault name	Consistency
1	uxPriority	2%<C<80%
2	pxStack(*idctrn* only)	35%<C<45%
5	uxBasePriority	2%<C<35%
10	xStateListItem.pxNext	35%<C<80%
11	xStateListItem.pxPrevious	35%<C<80%
12	xStateListItem.pvOwner	35%<C<80%
13	xStateListItem.pvContainer	35%<C<80%
18	xEventListItem.pvContainer	35%<C<80%

Table 8.7: Faults producing misbehaviors for experiments in list 2, ready TCB, 0-7 LSB

When injecting in ready task TCB, the number of injections done is actually lower than the total number of experiments: the FIE, in fact, has a system that is able to understand if the datum under test is present in memory or not, and, if not, experiment is aborted; TCBs belonging to ready tasks are not always instantiated in the ready tasks list (for example when all the created tasks are in the delayed state), so in some cases injection was not actually performed, obtaining thus a reduced consistency.
10, 11, 12, 13, 18) In any case, it is interesting to notice that faults 10, 11, 12, 13 and 18 (*xStateListItem.pxNext*, *xStateListItem.pxPrevious*, *xStateListItem.pvOwner*, *xStateListItem.pvContainer* and *xEventListItem.pvContainer*) have almost a fixed consistency and in the most of cases they produce a crash because they are pointers.
In idctrn, some other faults (0,1,3,4,6,7,8,9) appear with a fixed consistency: actually these faults could be left unconsidered, as they are caused by an injection made in a moment when the kernel was performing a sensitive operation; such behavior in fact was observed only when injecting in a particular time instant, with an unusual behavior in the DUT-side injector (system crashed before the injector could complete its operations, suggesting that injector interrupted a critical operation of the kernel since its ISR has maximum priority).

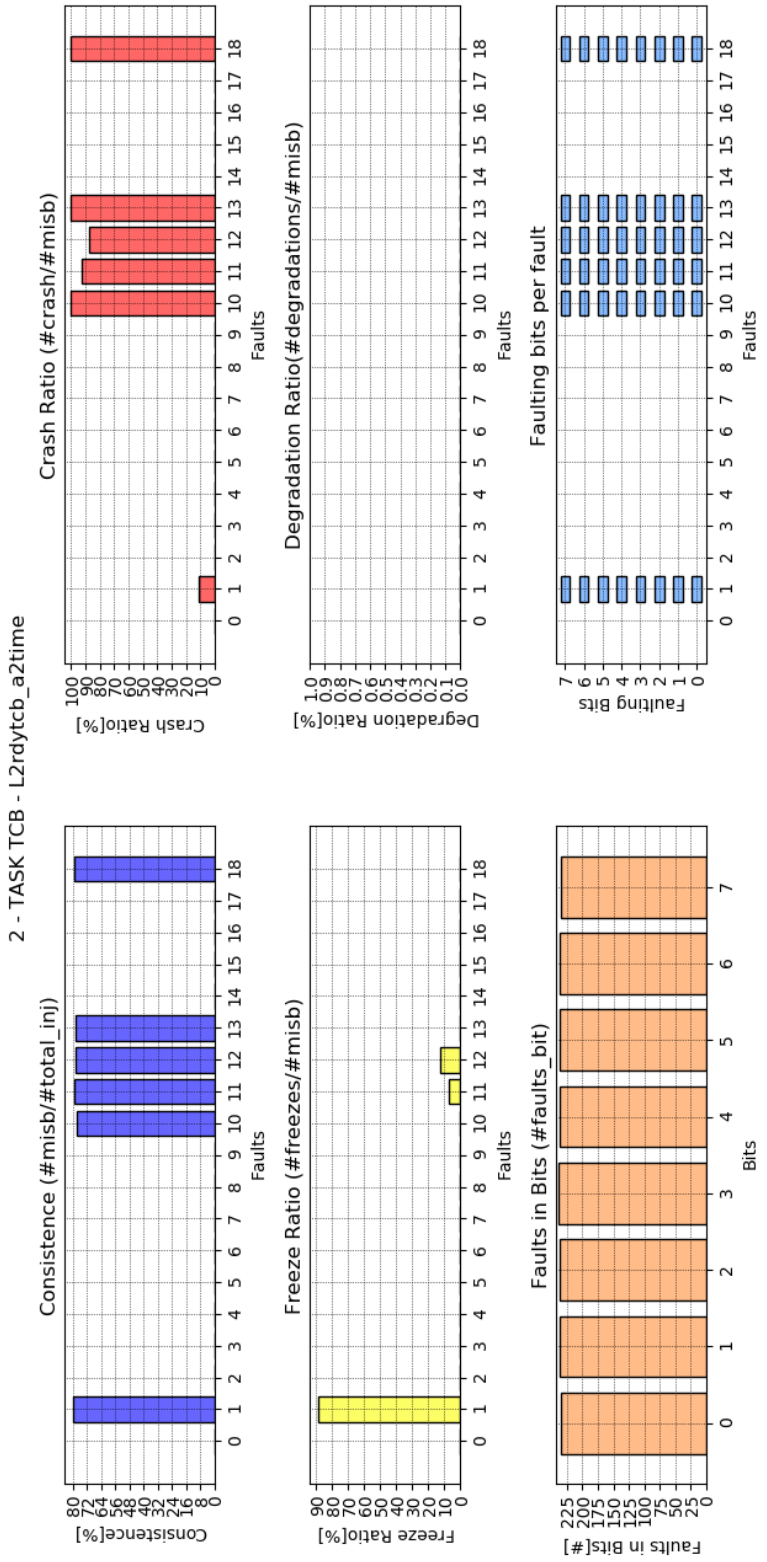

Figure 8.13: Results of injections in 8LSB, using a2time benchmark

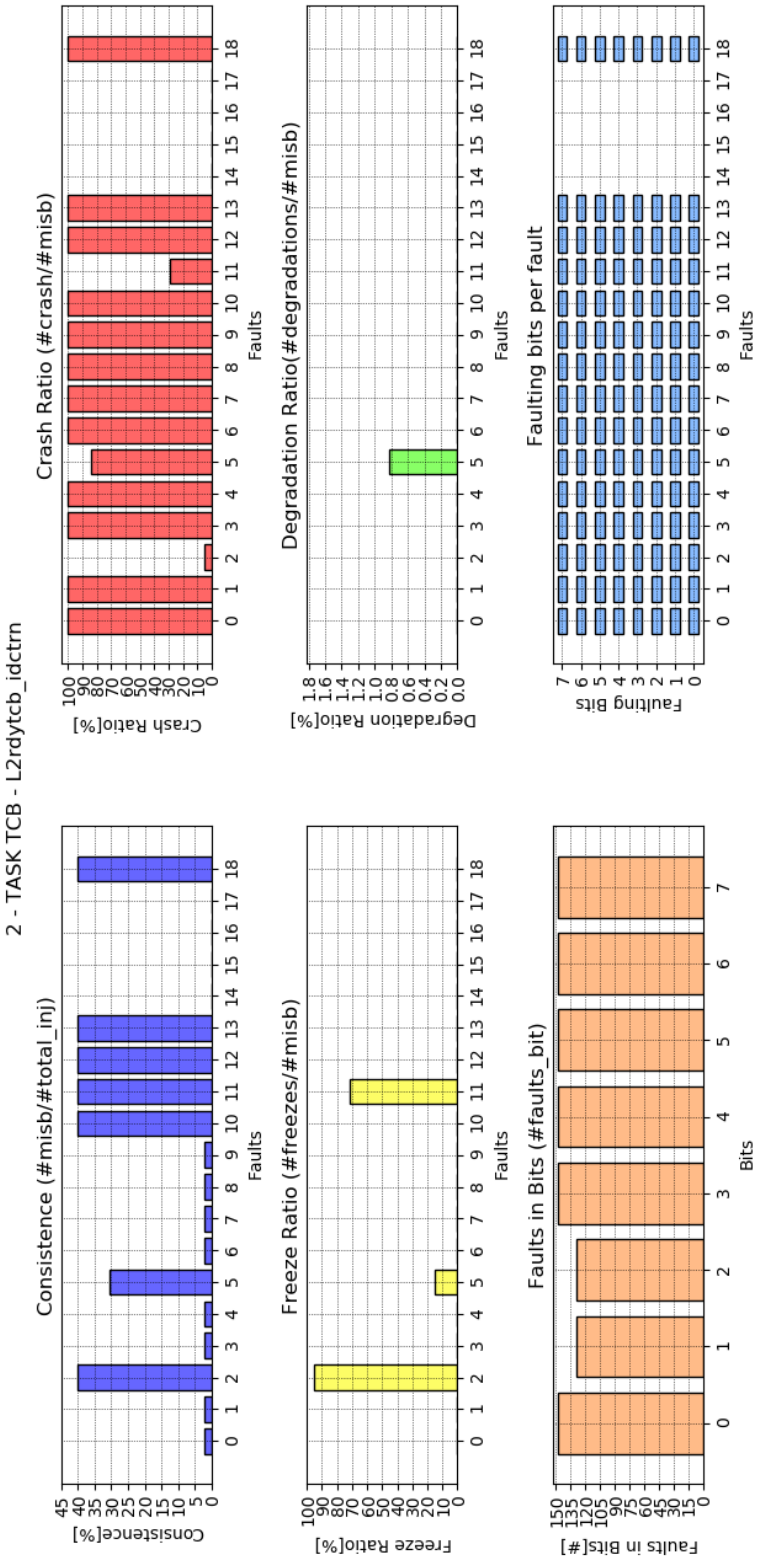

Figure 8.14: Results of injections in 8LSB, using idctrn benchmark

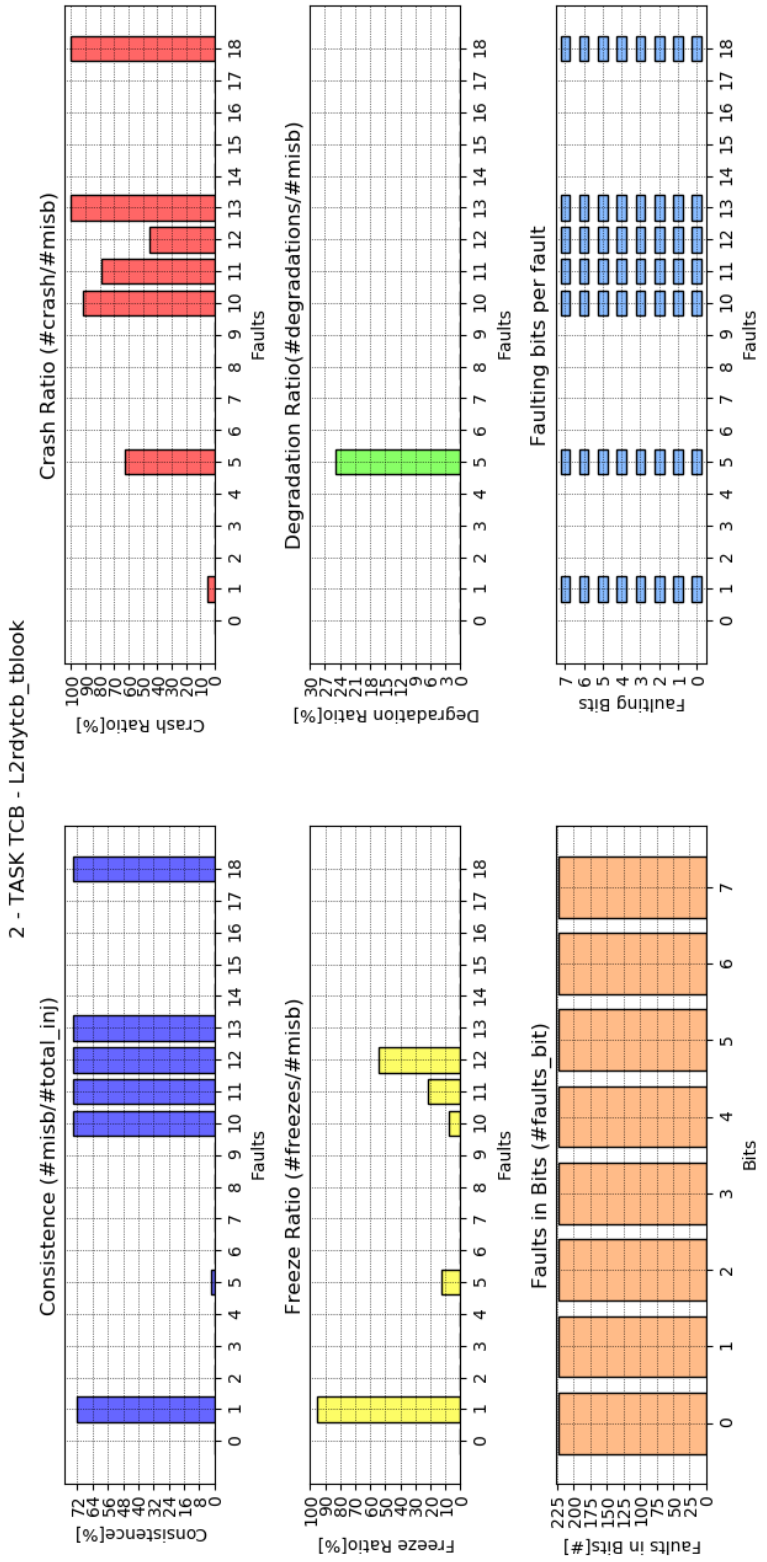

Figure 8.15: Results of injections in 8LSB, using tblook benchmark

8.4.2 MSB injection results

Fault number	Fault name	Consistency
0	pxTopOfStack	C~0%
1	uxPriority	C~0%
2	pxStack(idctrn only)	32%<C<36%
3	uxTCBNumber	C~0%
5	uxBasePriority	C~0%
6	uxMutexesHeld	C~0%
7	ulNotifiedValue	C~0%
8	ucNotifyState	C~0%
9	xStateListItem.xItemValue	C~0%
10	xStateListItem.pxNext	32%<C<100%
11	xStateListItem.pxPrevious	32%<C<100%
12	xStateListItem.pvOwner	32%<C<100%
13	xStateListItem.pvContainer	32%<C<100%
14	xEventListItem.xItemValue	C~0%
15	xEventListItem.pxNext	C~0%
16	xEventListItem.pxPrevious	C~0%
17	xEventListItem.pvOwner	C~0%
18	xEventListItem.pvContainer	32%<C<100%

Table 8.8: Faults producing misbehaviors for experiments in list 2, ready TCB, 1 MSB

Experiments on MSB of a ready task's TCB produced the most critical results: the usual pointers (10, 11, 12 and 13) always produce crashes but this time, for many other variables, even if consistency is low (in many cases it is lower than 1%), the common result is a crash too. The TCB of a ready task is a very sensitive part of FreeRTOS since it is surely accessed by the kernel when the task is selected to be switched in, so, the fault will manifest itself. In any case, most important results with a relevant consistency level, are still the faults 2, 10 ,11, 12, 13 and 18 as for the injections in current TCB.

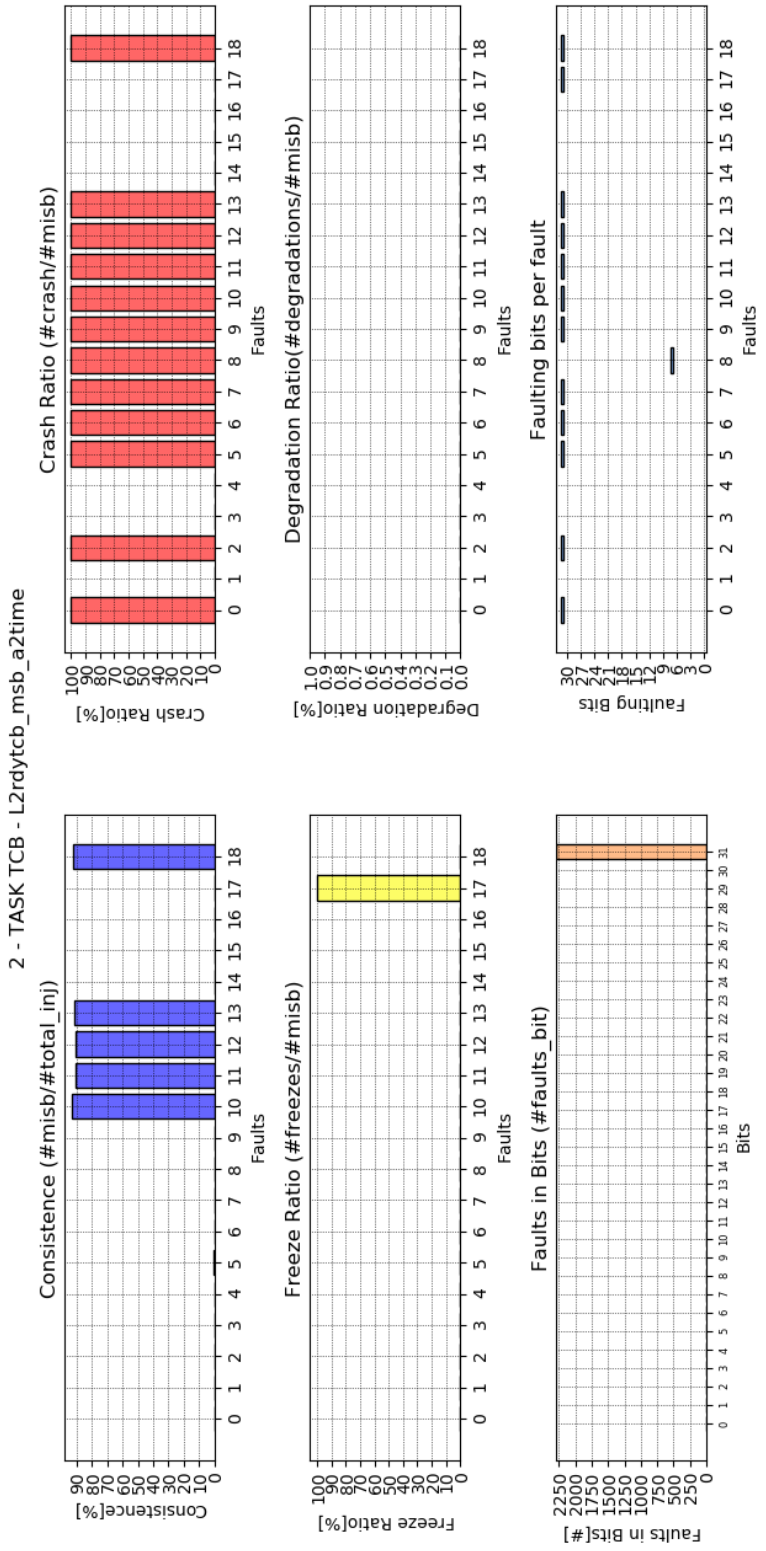

Figure 8.16: Results of injections in 1MSB, using a2time benchmark

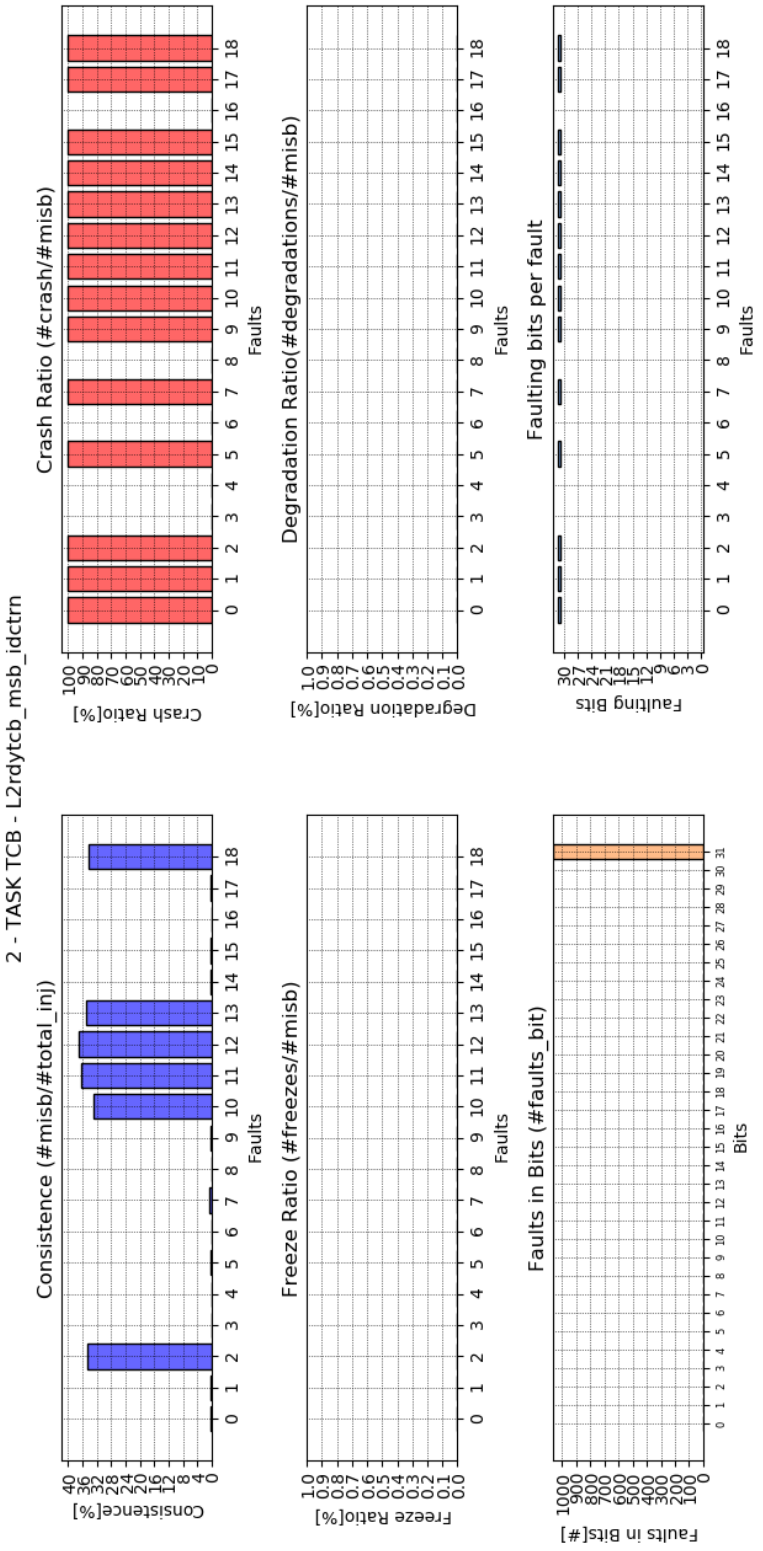

Figure 8.17: Results of injections in 1MSB, using idctrn benchmark

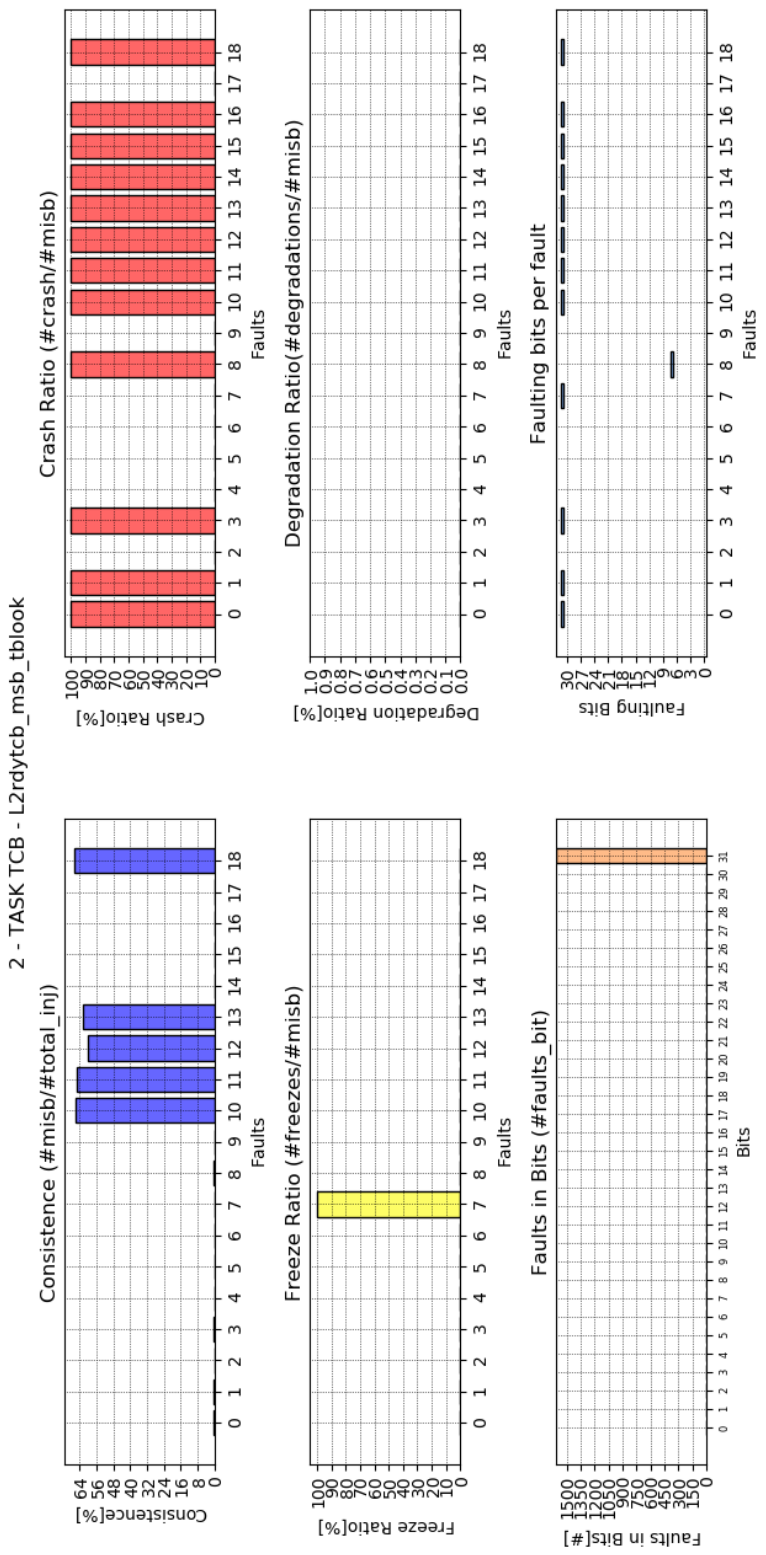

Figure 8.18: Results of injections in 1MSB, using tblook benchmark

8.5 List 3 - Ready tasks list

8.5.1 Bits 0-7 injection results

Fault number	Fault name	Consistency
0	uxNumberOfItems	C=100%
1	pxIndex	C=100%
4	xListEnd.pxPrevious	10%<C<60%

Table 8.9: Faults producing misbehaviors for experiments in list 3, ready tasks list, 0-7 LSB

This list gave some results for faults 0, 1 and 4, related to variables *uxNumberOfItems*, *pxIndex* and *xListEnd.pxPrevious*.
0) The fault 0 (*uxNumberOfItems*) is an integer containing the number of elements in the list: injecting here leads mainly to crashes. The consistency equal to 100% is justified by the fact that this variable is accessed very often by the kernel, in particular by the function vTaskSwitchContext(), which performs a check of the length of the ready list before another available ready task is switched in as running task: if this variable is equal to zero the system hangs in a configASSERT() (this happens because, in this case, the RTOS would expect to have a non-void ready task list during the context switching). Crashes are easily explainable referring to the fact that, if pxReadyTasksLists holds, actually, a number of tasks different from the value of uxNumberOfItems, a switch is done anyway, making the variable pxCurrentTCB to point to a non-sense position in memory.
1) The variable *pxIndex* related to fault 1 produces crashes too: this happens because this variable is a pointer to the last item inserted and when it is modified, kernel tries to access to a wrong memory region. Functions vListInsertEnd() and uxListRemove() access this parameter, respectively, to update pxNext and pxPrevious pointers of the new inserted item with the values of pxIndex and pxIndex->pxPrevious and to update it with the last item in the list, so with pxPrevious of the removed item. In both cases, after the injection such value is used before it is overwritten by the update and then it points to a wrong position in the memory.
4) The fault 4 (*xListEnd.pxPrevious*) is again a pointer which points to the last useful item in a list, just before the list end marker called xListEnd. Again, item insertion and deletion operations update this variable so that pxIndex always points to the end of the list (which is always a xListEnd structure). Function prvAddCurrentTaskToDelayedList() is used to move a task from the ready to the delayed task list and to do this the function vListInsert() is exploited: variable xListEnd.pxPrevious is used exactly to start the iteration during the insertion of the new item in the right position in xDelayedTaskList1 or xDelayedTaskList2; these two lists, as already explained, are used, respectively, when the wakeup time of the task does not overflow or overflows the tick counter. If the value is injected, the iteration across the list is made in a bad way.

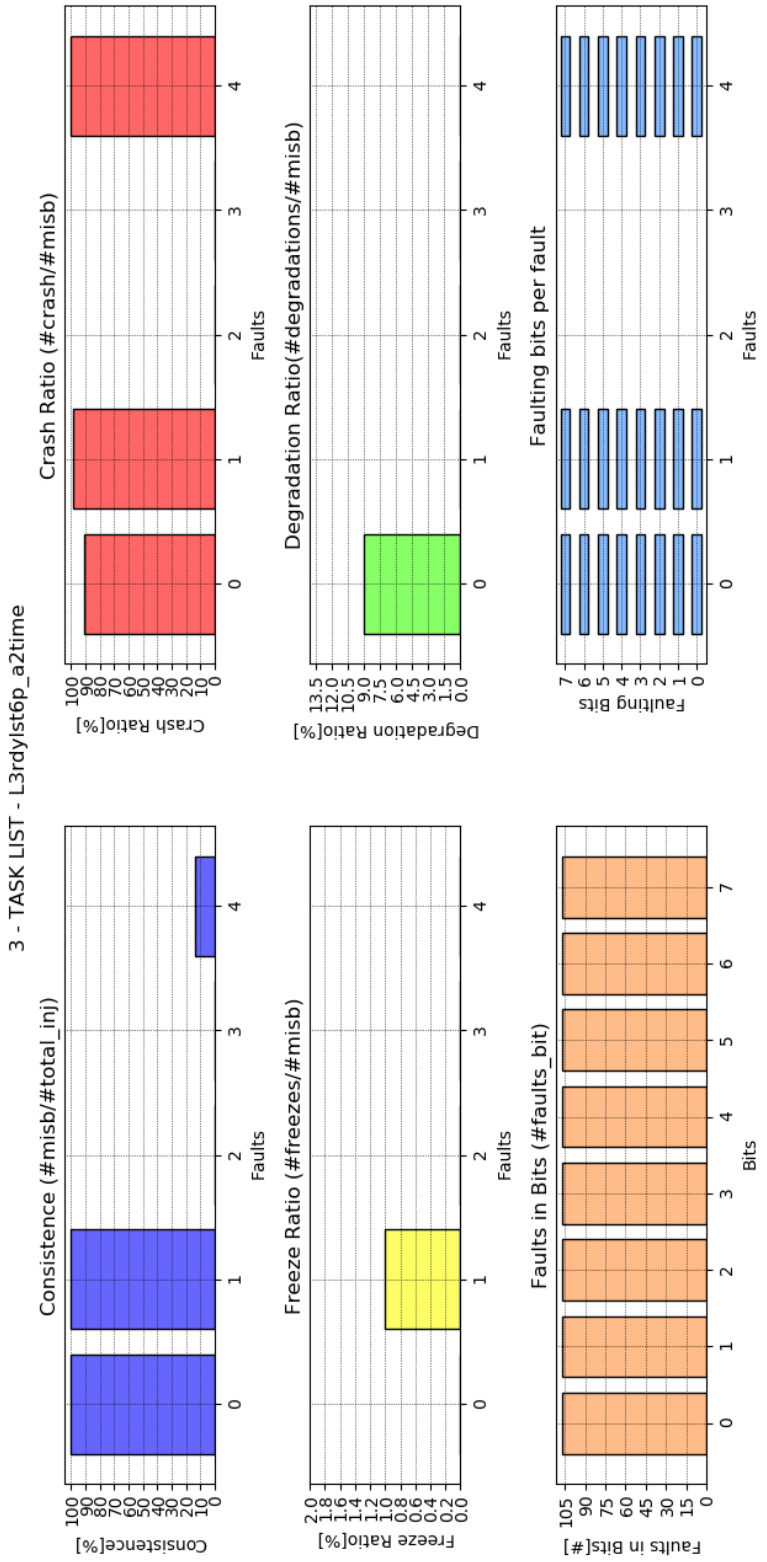

Figure 8.19: Results of injections in 8LSB, using a2time benchmark

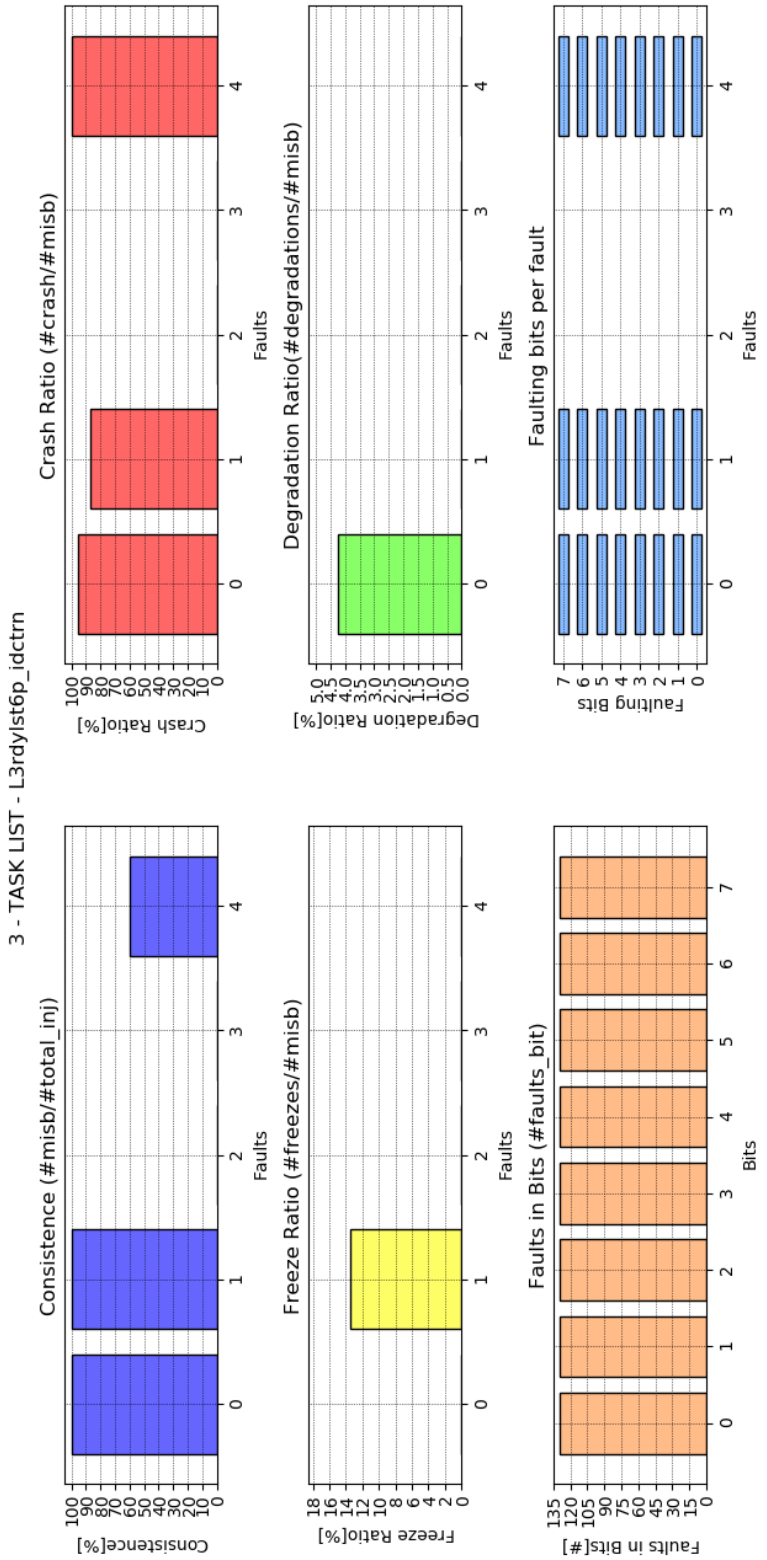

Figure 8.20: Results of injections in 8LSB, using idctrn benchmark

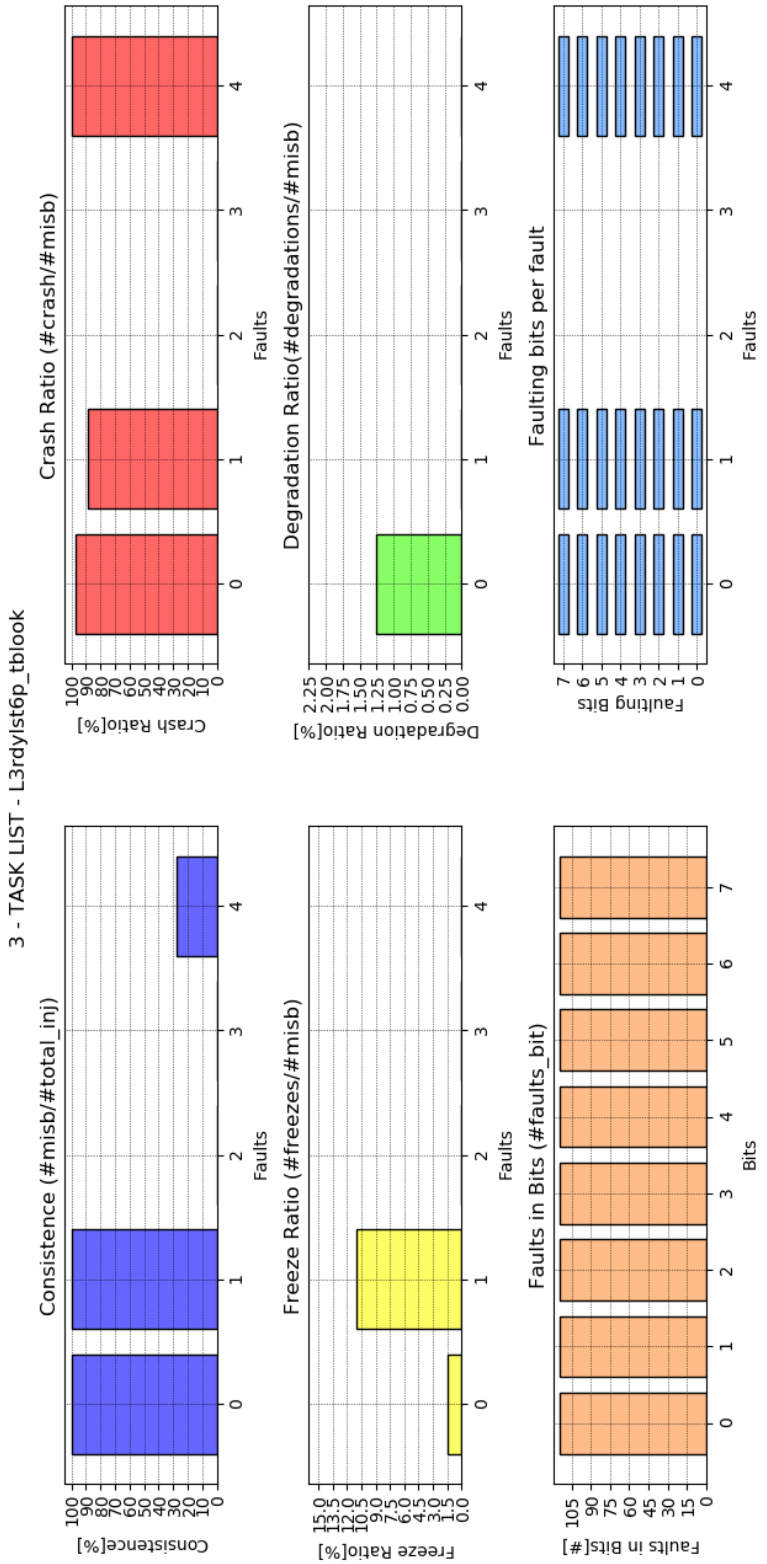

Figure 8.21: Results of injections in 8LSB, using tblook benchmark

8.5.2 MSB injection results

Fault number	Fault name	Consistency
0	uxNumberOfItems	C=100%
1	pxIndex	C=100%
3	xListEnd.pxNext	C~0%
4	xListEnd.pxPrevious	5%<C<70%

Table 8.10: Faults producing misbehaviors for experiments in list 3, ready tasks list, 1 MSB

Injections in MSB have very similar results with respect to the 8LSB case, with similar levels of consistency; however, crashes happens in 100% of cases for all experiments, due to the fact that pointers, this time, point to a completely wrong memory position, which probably is even outside the RAM.

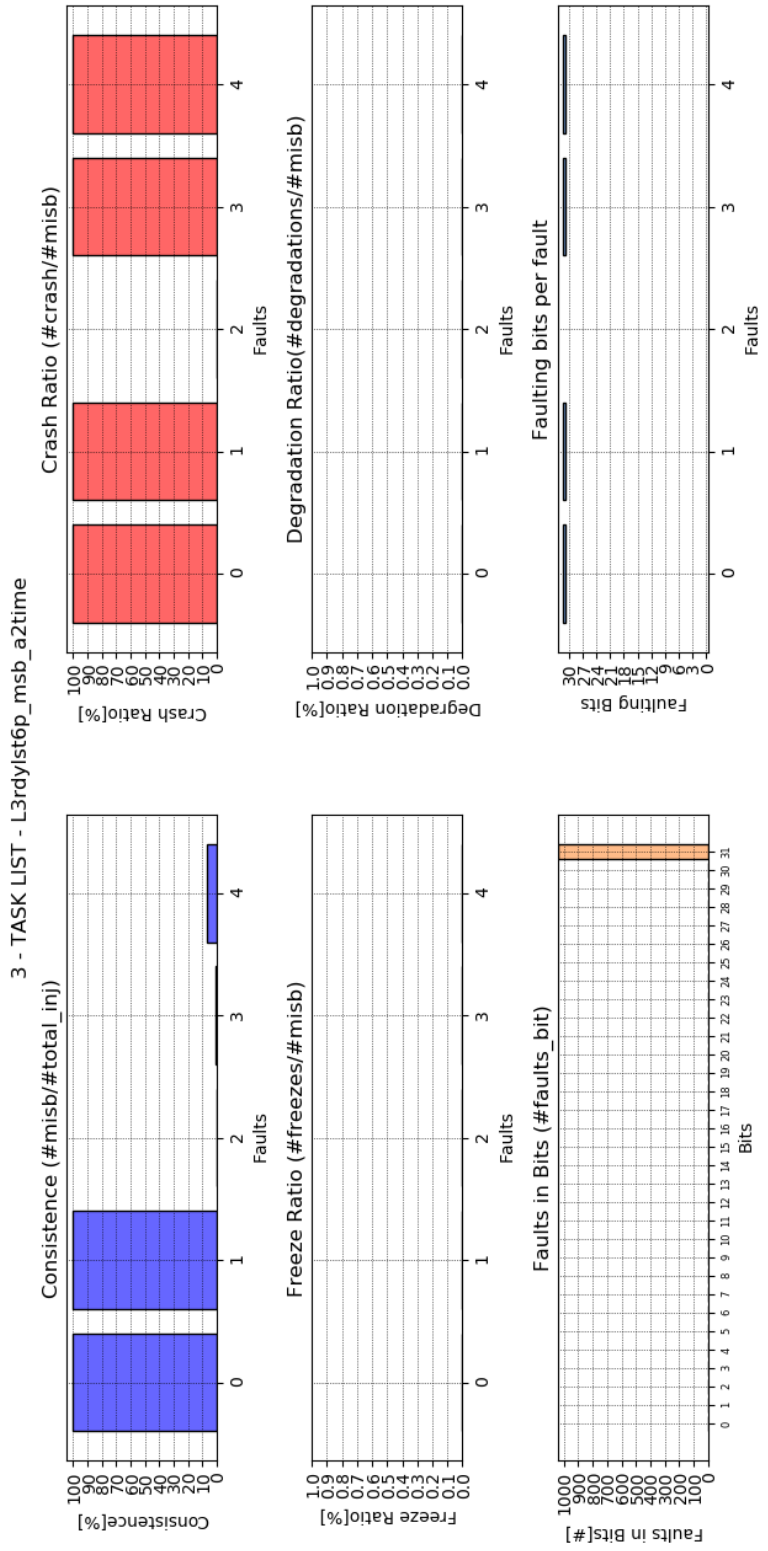

Figure 8.22: Results of injections in 1MSB, using a2time benchmark

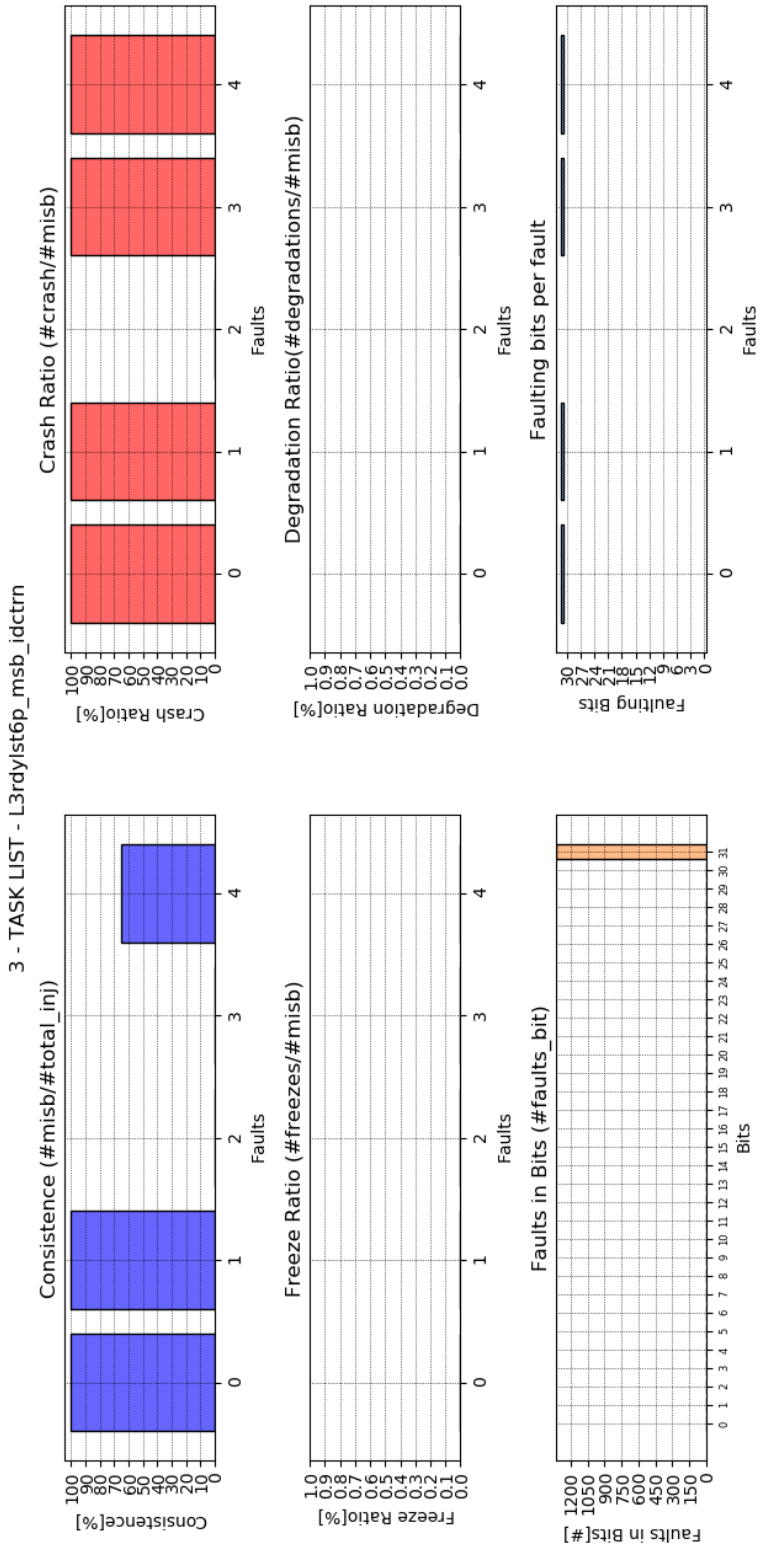

Figure 8.23: Results of injections in 1MSB, using idctrn benchmark

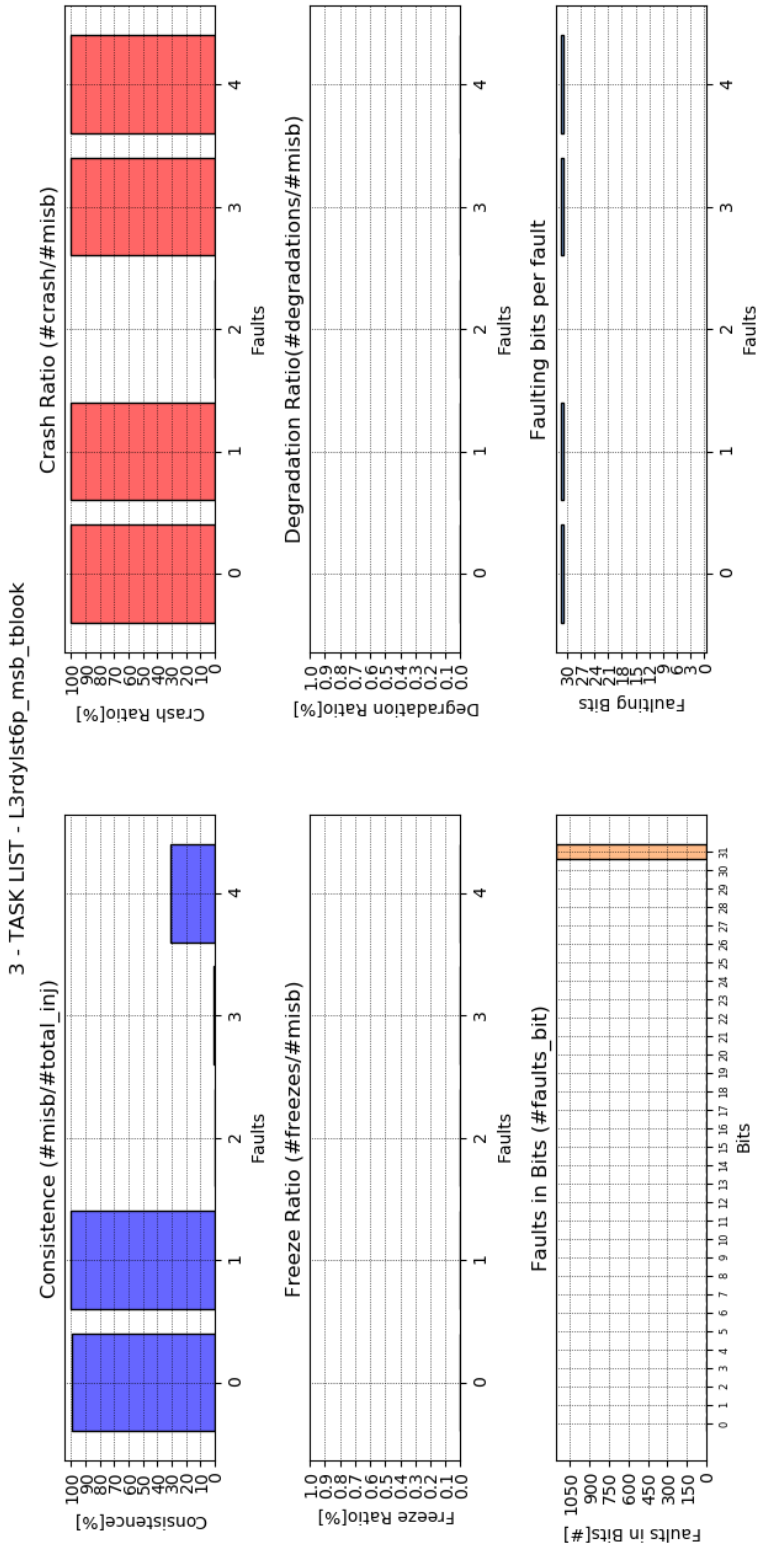

Figure 8.24: Results of injections in 1MSB, using tblook benchmark

8.6 List 3 - Delayed tasks list

8.6.1 Bits 0-7 injection results

Fault number	Fault name	Consistency
0	uxNumberOfItems	10%<C<20%
3	xListEnd.pxNext	C=100%

Table 8.11: Faults producing misbehaviors for experiments in list 3, delayed tasks list, 0-7 LSB

It is important to make a comparison with results of experiment on ready tasks list: this fault list produces a lower number of misbehaviors as fewer fields are used by the kernel. The overall number of crashes is lower: in any case they always happen only for pointers or pointer-related variables. Faults 0 and 3 (*uxNumberOfItems* and *xListEnd.pxNext*) give results.
0) Fault 0 (*uxNumberOfItems*) causes freezes and degradations because, when the number of items in the list is changed, some tasks are not moved back again to the ready state - as one can see from the following graphs, only 2 LSB give misbehaviors and the total number of instantiated tasks is 3 (idctrn has more instantiated tasks running in parallel as this is a multithread benchmark, but such tasks are never delayed so they never enter the list under test).
3) Fault 3 (*xListEnd.pxNext*) is used in delayed-to-ready state transition and it manifests a consistency of 100% because it is used in a complementary way with respect to *xListEnd.pxPrevious*, used instead in the case of ready-to-delayed state transition: prvAddTaskToReadyList() function, in fact, is called if there is a task which has reached its timeout delay and, if so, such task is moved back to ready state list using vListInsertEnd(), keeping the injected xListEnd.pxNext, which will be used then to surf the ready task list but that will point to a wrong next item.

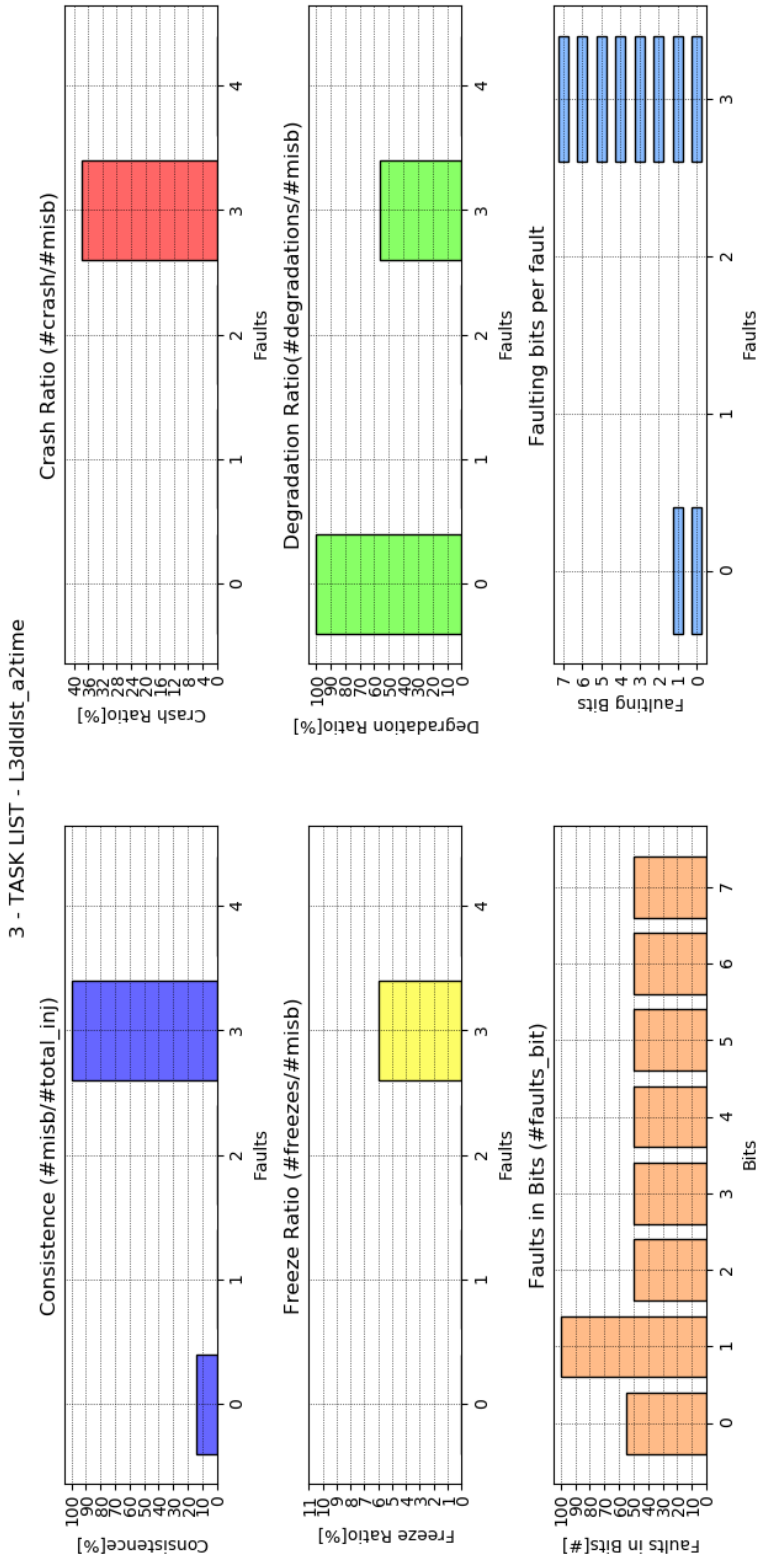

Figure 8.25: Results of injections in 8LSB, using a2time benchmark

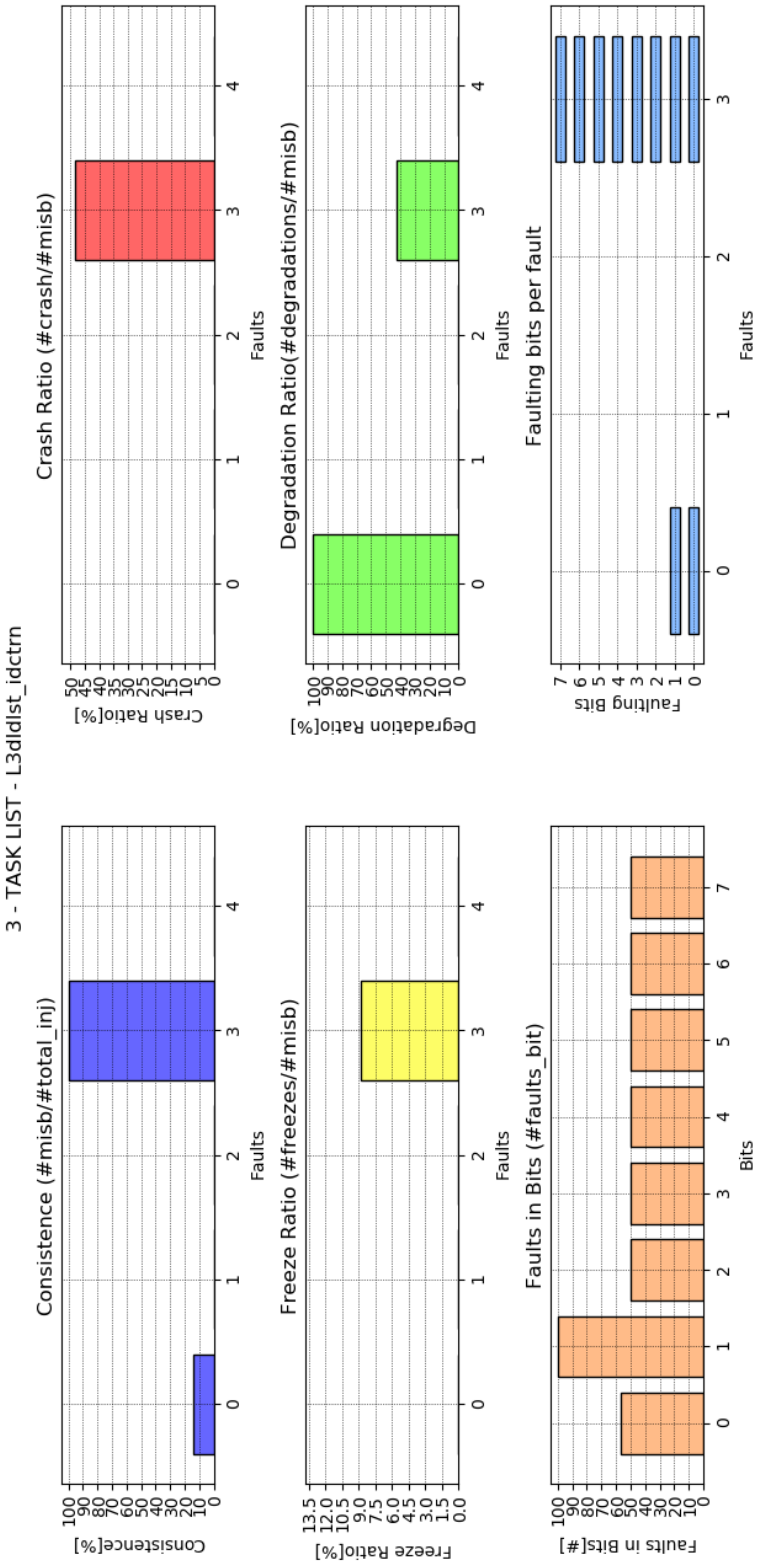

Figure 8.26: Results of injections in 8LSB, using idctrn benchmark

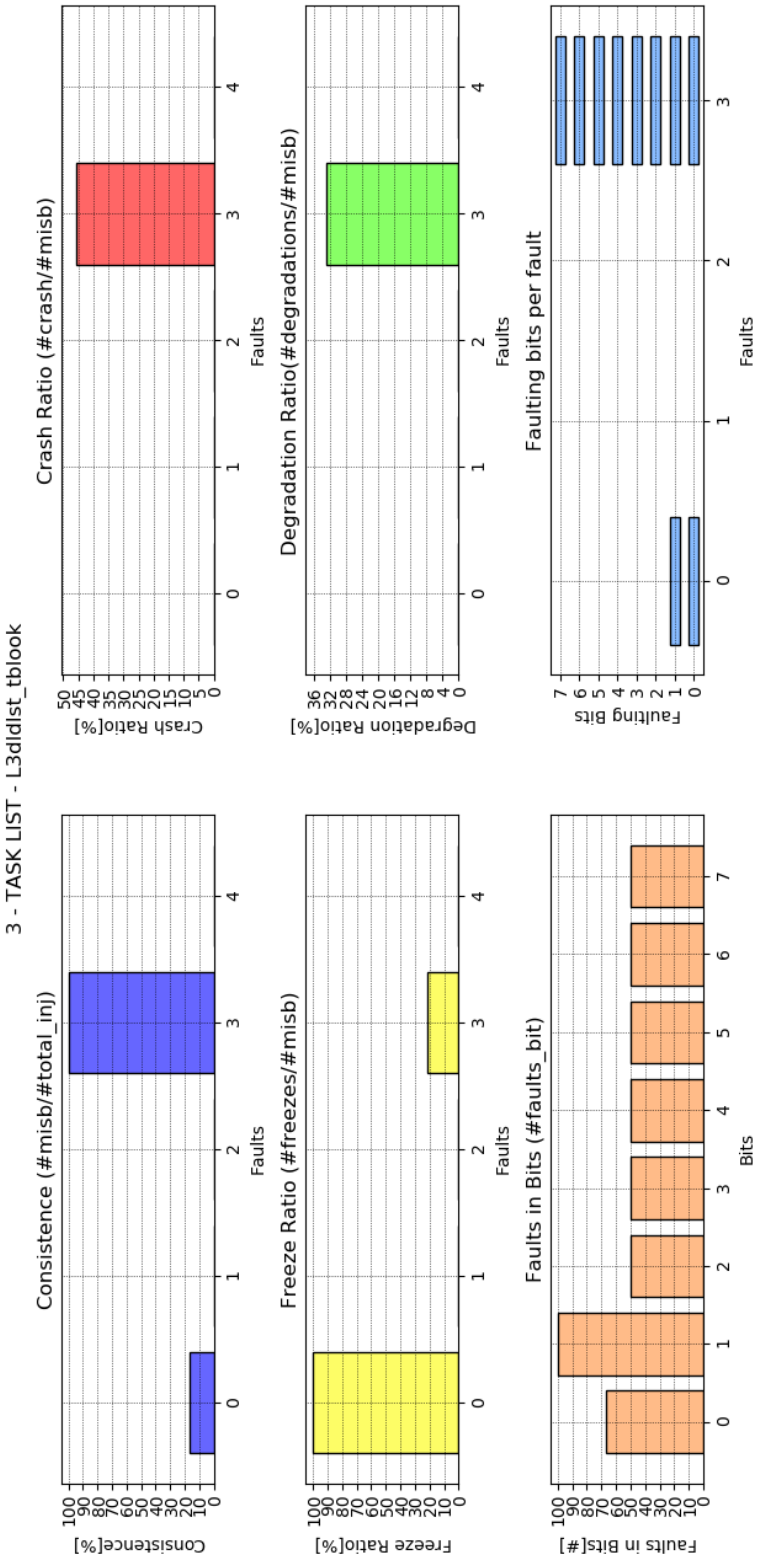

Figure 8.27: Results of injections in 8LSB, using tblook benchmark

8.6.2 MSB injection results

Fault number	Fault name	Consistency
3	xListEnd.pxNext	C=100%

Table 8.12: Faults producing misbehaviors for experiments in list 3, delayed tasks list, 1 MSB

3) Only fault 3 corresponding to *xListEnd.pxNext* is visible and, this time it causes always crashes: as said multiple times, MSB injections in pointers easily lead to completely wrong accesses to memory which immediately raises an exception and the consequent call of the ISR related to a default error.

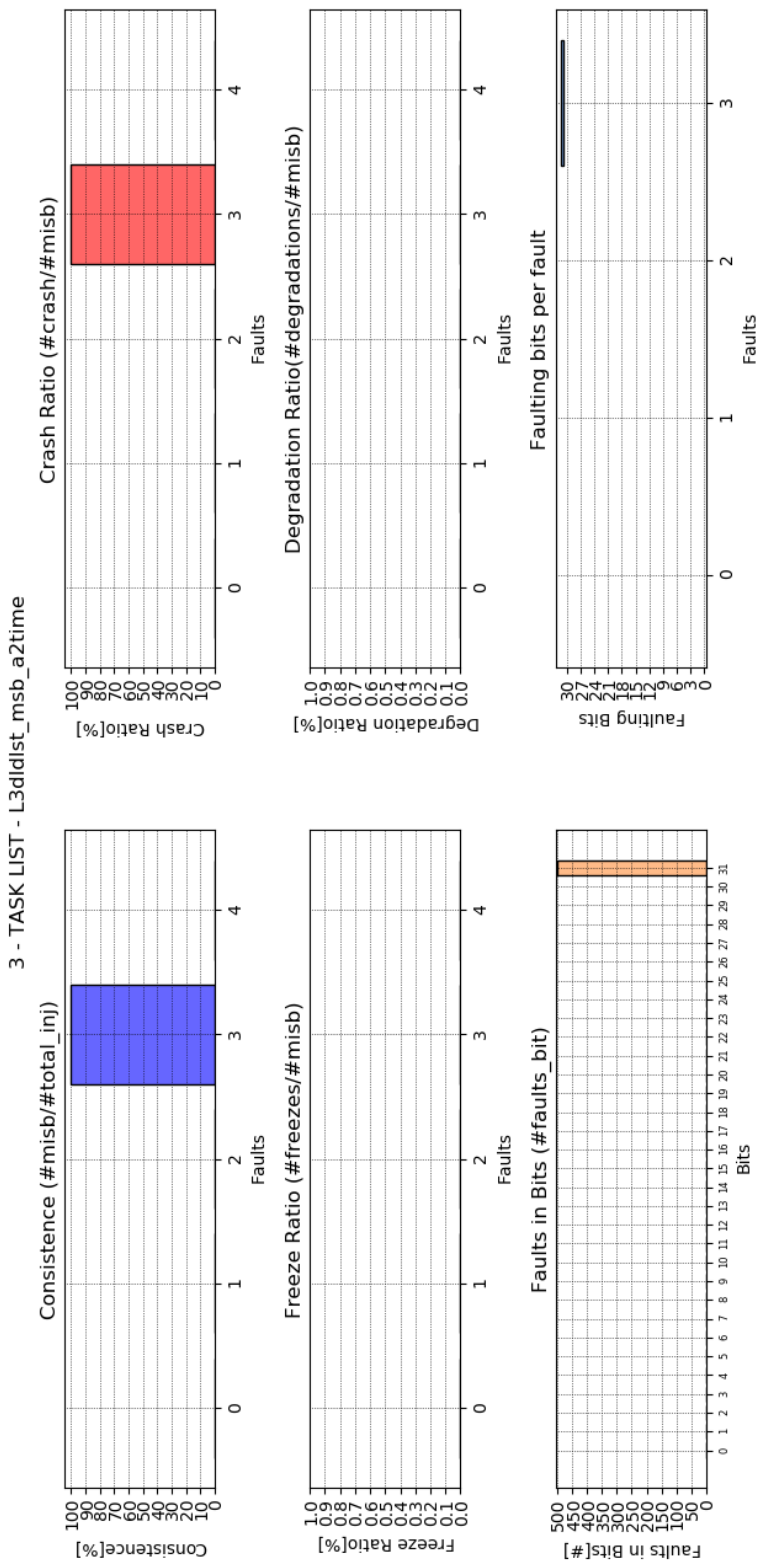

Figure 8.28: Results of injections in 1MSB, using a2time benchmark

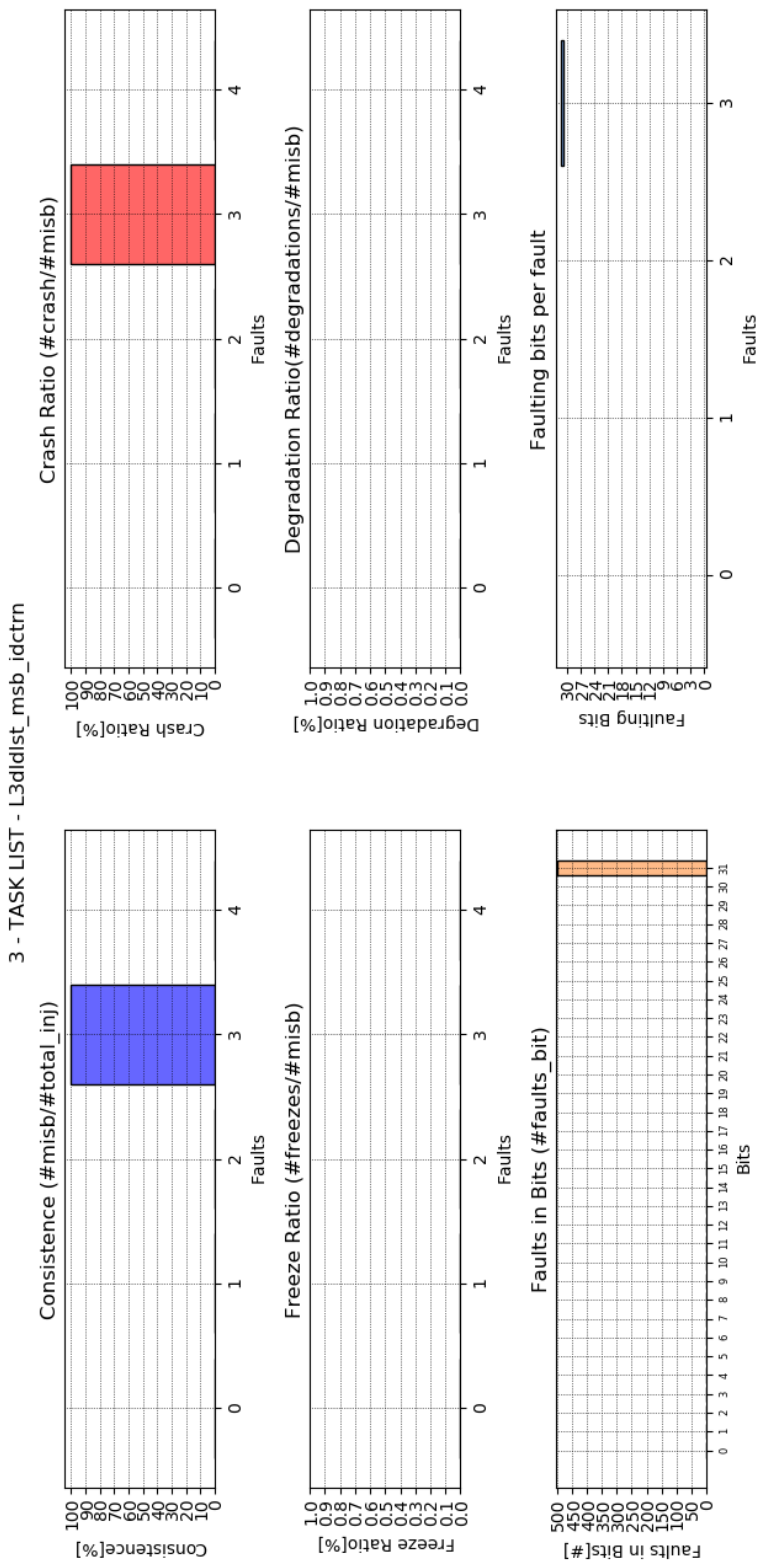

Figure 8.29: Results of injections in 1MSB, using idctrn benchmark

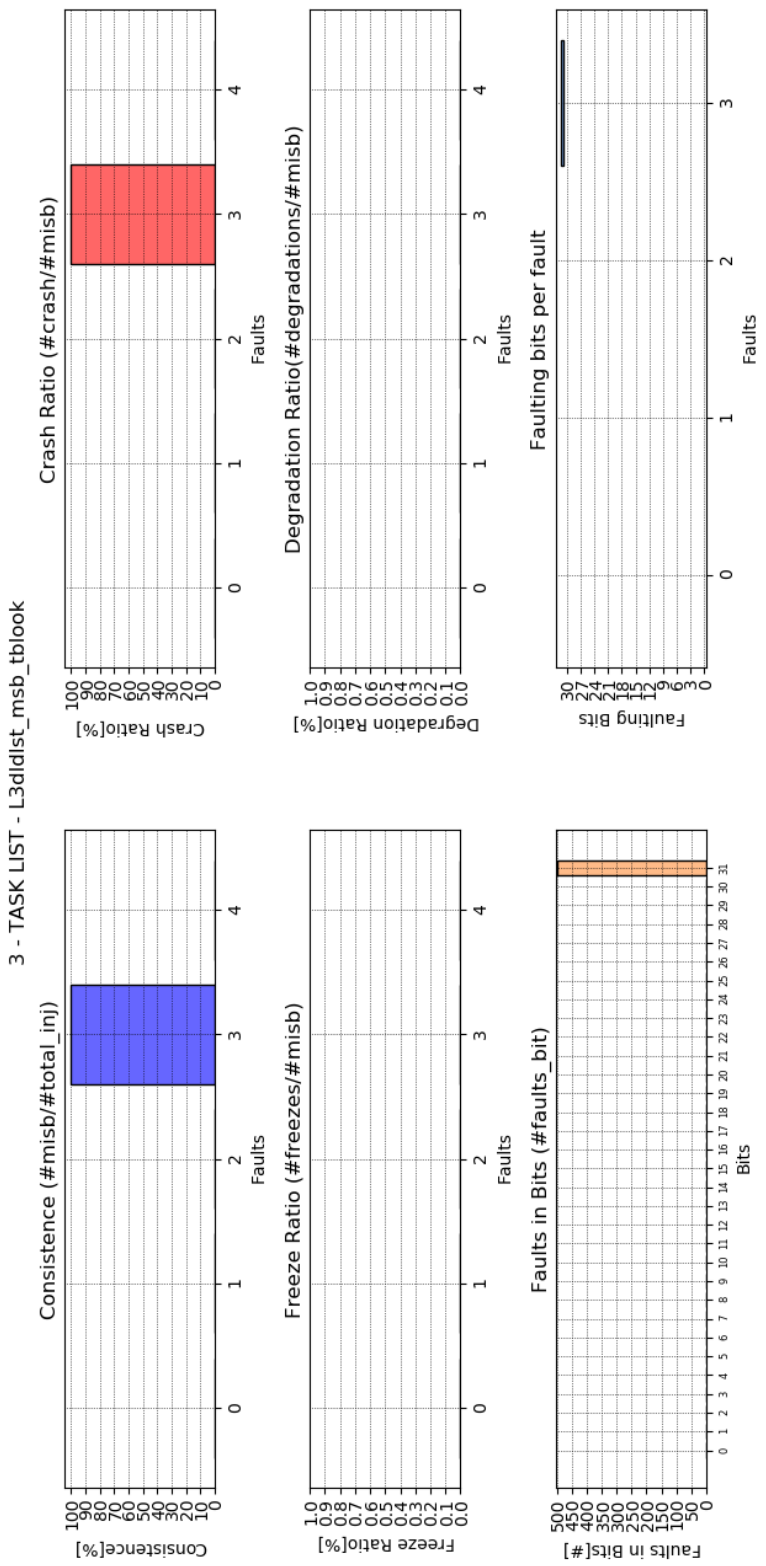

Figure 8.30: Results of injections in 1MSB, using tblook benchmark

8.7 List 4 - Mutex

Experiments done on mutexes are the ones which could lead to obtain the most interesting results: injections in RTOS variables could cause, in fact, a change in user data due to the fact that a multiple access could be erroneously done if the mutex does not work as expected. However, this kind of behavior was never experienced because the take/give operations were made around very small portions of code, not sufficiently long to allow in the meantime a context switch and an attempt of another task to take the same mutex.

8.7.1 Bits 0-7 injection results

Fault number	Fault name	Consistency
5	xTasksWaitingToSend.uxNumberOfItems	38%<C<100%
10	xTasksWaitingToReceive.uxNumberOfItems	38%<C<100%
15	uxMessagesWaiting	4%<C<15%
16	uxLength	4%<C<15%
17	uxItemSize	38%<C<100%

Table 8.13: Faults producing misbehaviors for experiments in list 4, 0-7 LSB

Results obtained in both LSBs and MSB targeted injections are similar. Faults 5, 10, 15, 16 and 17 are visible, corresponding to faults *xTasksWaitingToSend.uxNumberOfItems*, *xTasksWaitingToReceive.uxNumberOfItems*, *uxMessagesWaiting*, *uxLength* and *uxItemSize*.
5) Fault 5 (*xTasksWaitingToSend.uxNumberOfItems*) causes always freezes: xTasksWaitingToSend is an event list containing all those items belonging to tasks which are waiting to release the mutex. When the mutex is taken by a task, xQueueGenericReceive() function is called and that event list is checked: if it is not empty, then the mutex can be taken.
10) Fault 10 (*xTasksWaitingToReceive.uxNumberOfItems*) is the opposite of fault 5 as it is a value which represents the number of tasks waiting to take the mutex.
15) This fault (*uxMessagesWaiting*) is visible only for 1st LSB injections: the kernel just checks if this value is different from 0 and if so it performs the needed operations, regardless of the actual number of messages waiting in the mutex. As only first bit gives result, after the injection the value of this variable is always the opposite of the expected one, leading to 100% of degradations or crashes.
16) Fault *uxLength* is used by xQueueGenericSend() function, called when the mutex is given, and it is compared to uxMessagesWaiting: if the latter value is lower than uxLength, then the mutex is actually released: when injecting in this variable, it happens often that its value is flipped from 1 to 0, preventing other tasks to take the mutex (even if the releasing task successfully gave it). In idctrn benchmark this variable causes a crash because 8 tasks are trying to access the same mutex, leading to more complex and critical mechanisms; the other two benchmark applications use a mutex shared just among two tasks.
17) Fault 17 (*uxItemSize*) causes always freezes: when xQueueGenericSend() is called to give the mutex, there is always a check on the operation. The item to be added to the queue must be not NULL or uxItemSize must be equal to zero in order to perform correctly the operations; as in mutexes send operation is made adding a NULL element to

the queue, uxItemSize must be zero: when injection is performed, such variable becomes in any case different from 0, leading a configASSERT() to hang the system.

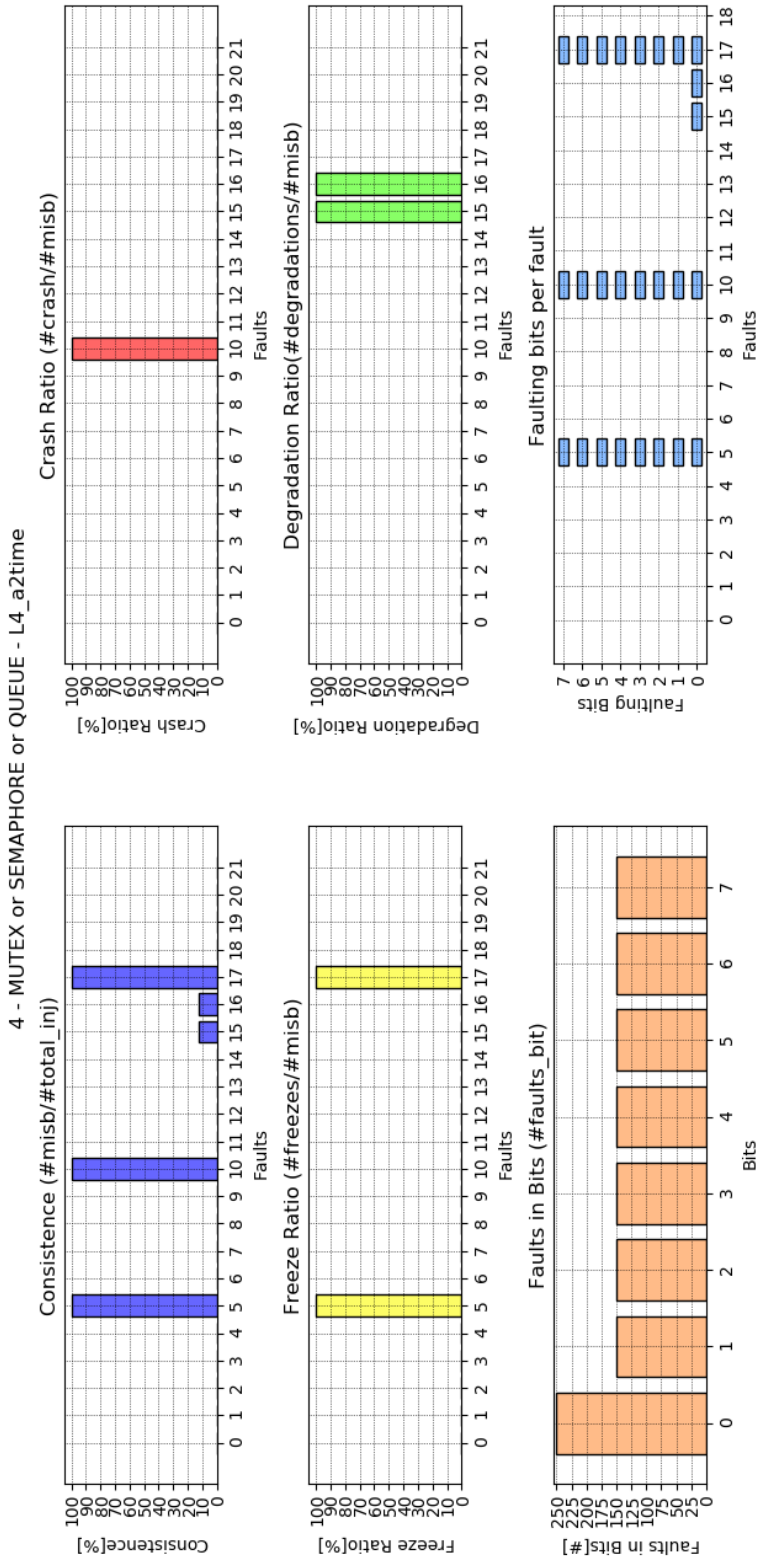

Figure 8.31: Results of injections in 8LSB, using a2time benchmark

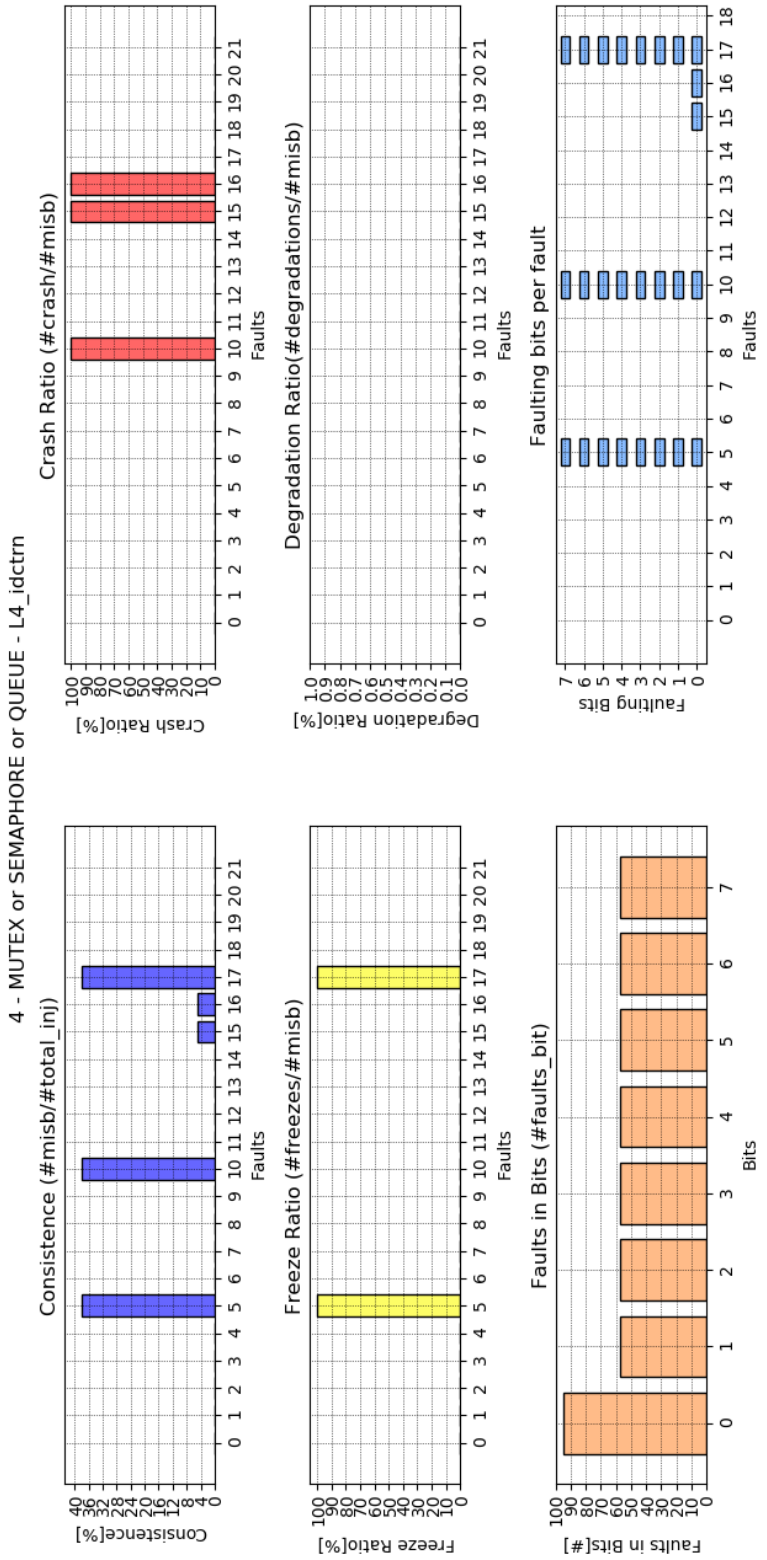

Figure 8.32: Results of injections in 8LSB, using idctrn benchmark

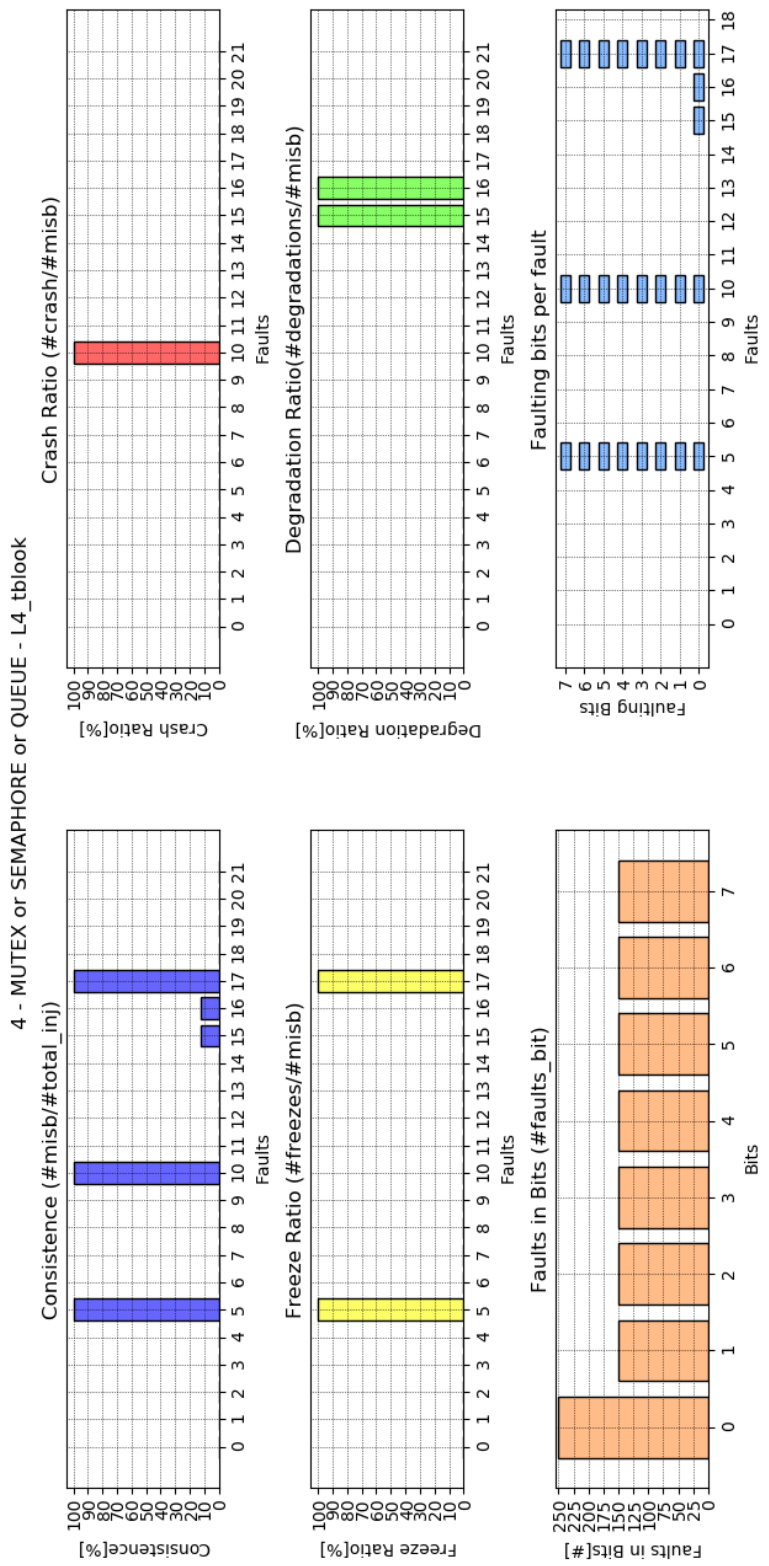

Figure 8.33: Results of injections in 8LSB, using tblook benchmark

8.7.2 MSB injection results

Fault number	Fault name	Consistency
1	pcTail	C~0%
5	xTasksWaitingToSend.uxNumberOfItems	36%<C<100%
10	xTasksWaitingToReceive.uxNumberOfItems	36%<C<100%
17	uxItemSize	36%<C<100%

Table 8.14: Faults producing misbehaviors for experiments in list 4, 1 MSB

Behavior of injections in MSB are more or less the same obtained for experiments in 8LSB, except for faults 15 and 16 which disappear.
15) This fault (*uxMessagesWaiting*) is not visible anymore because only 1st LSB injections cause a misbehavior.
16) Same behavior of fault 15.

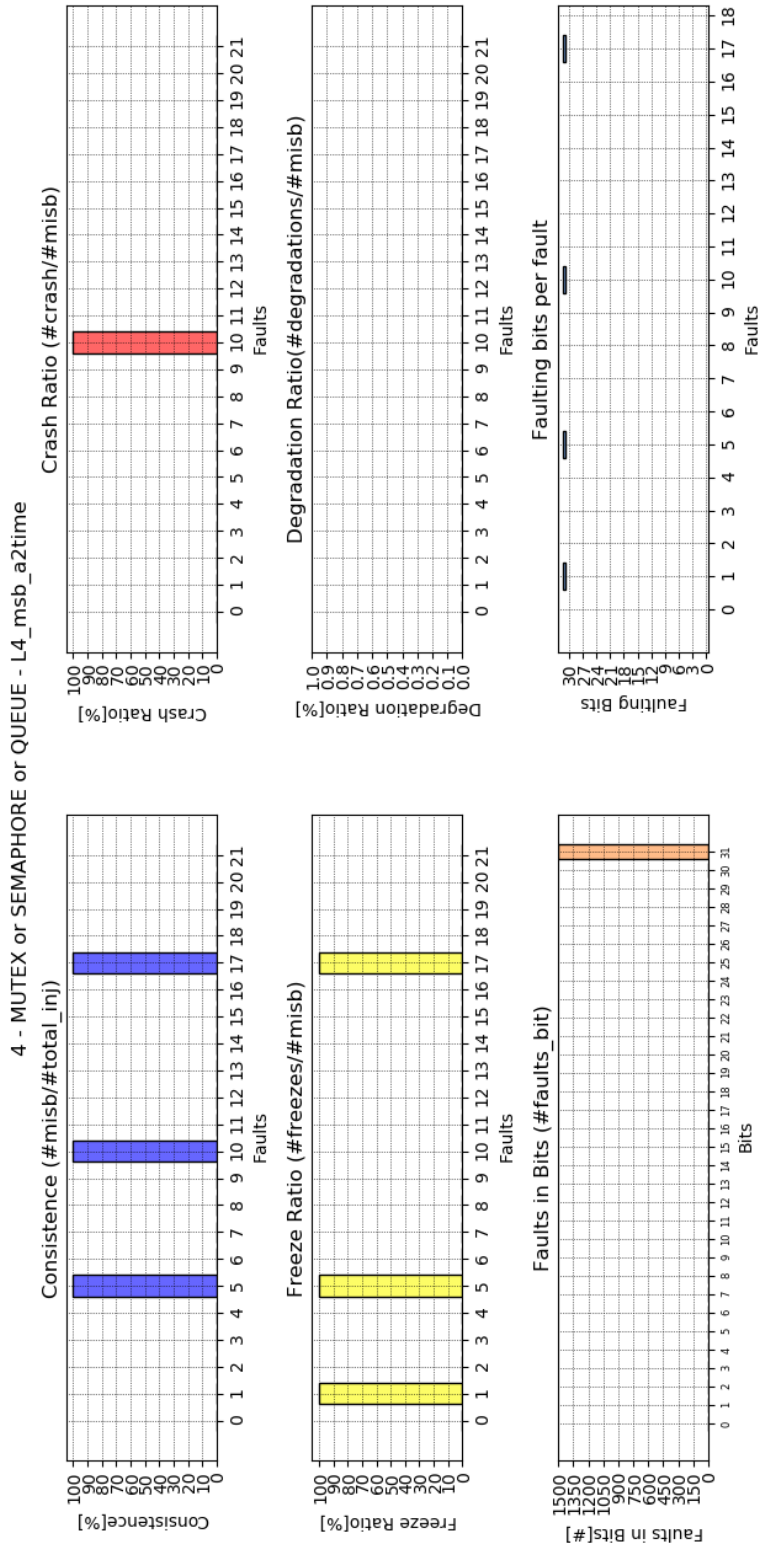

Figure 8.34: Results of injections in 1MSB, using a2time benchmark

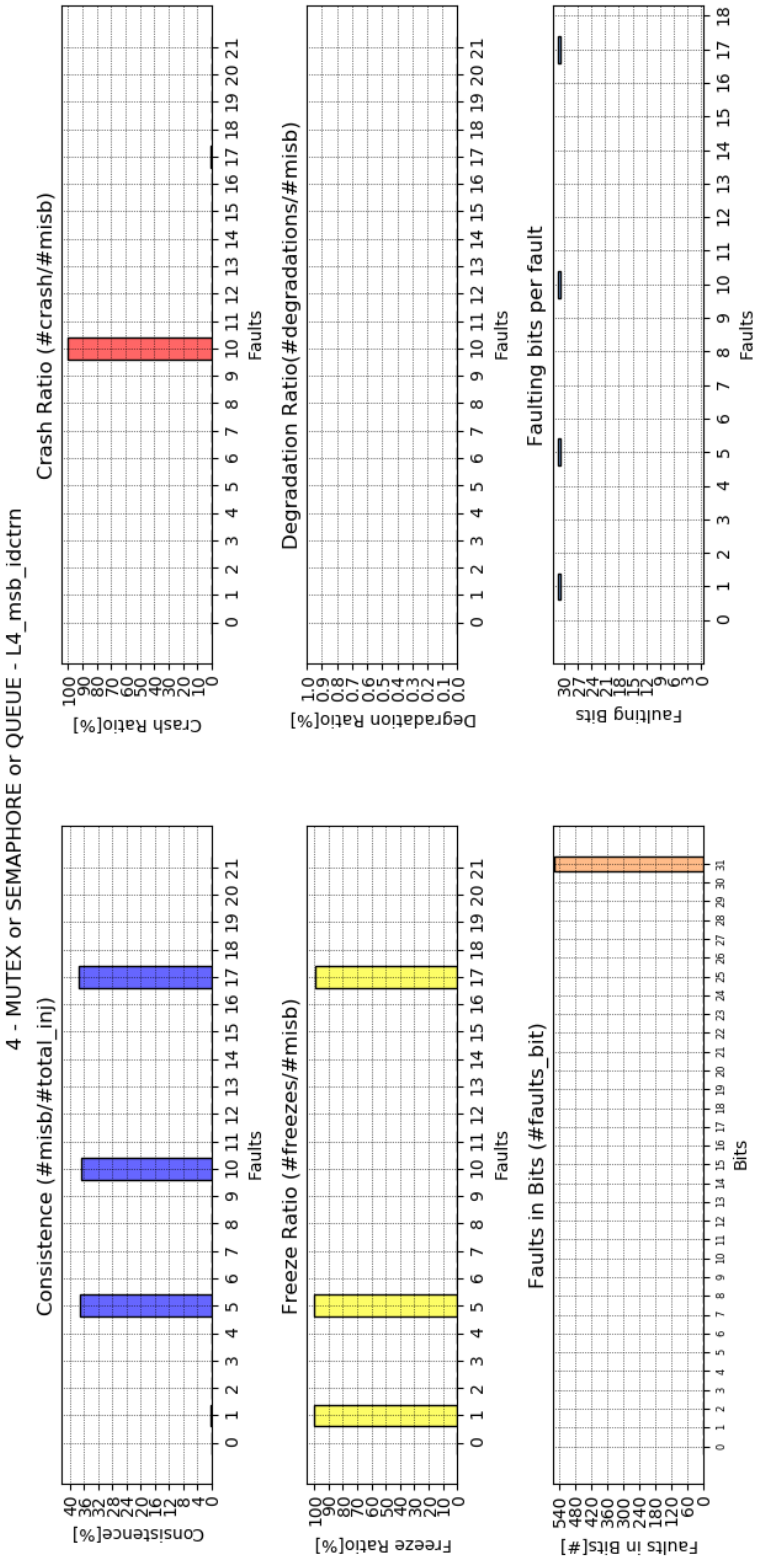

Figure 8.35: Results of injections in 1MSB, using idctrn benchmark

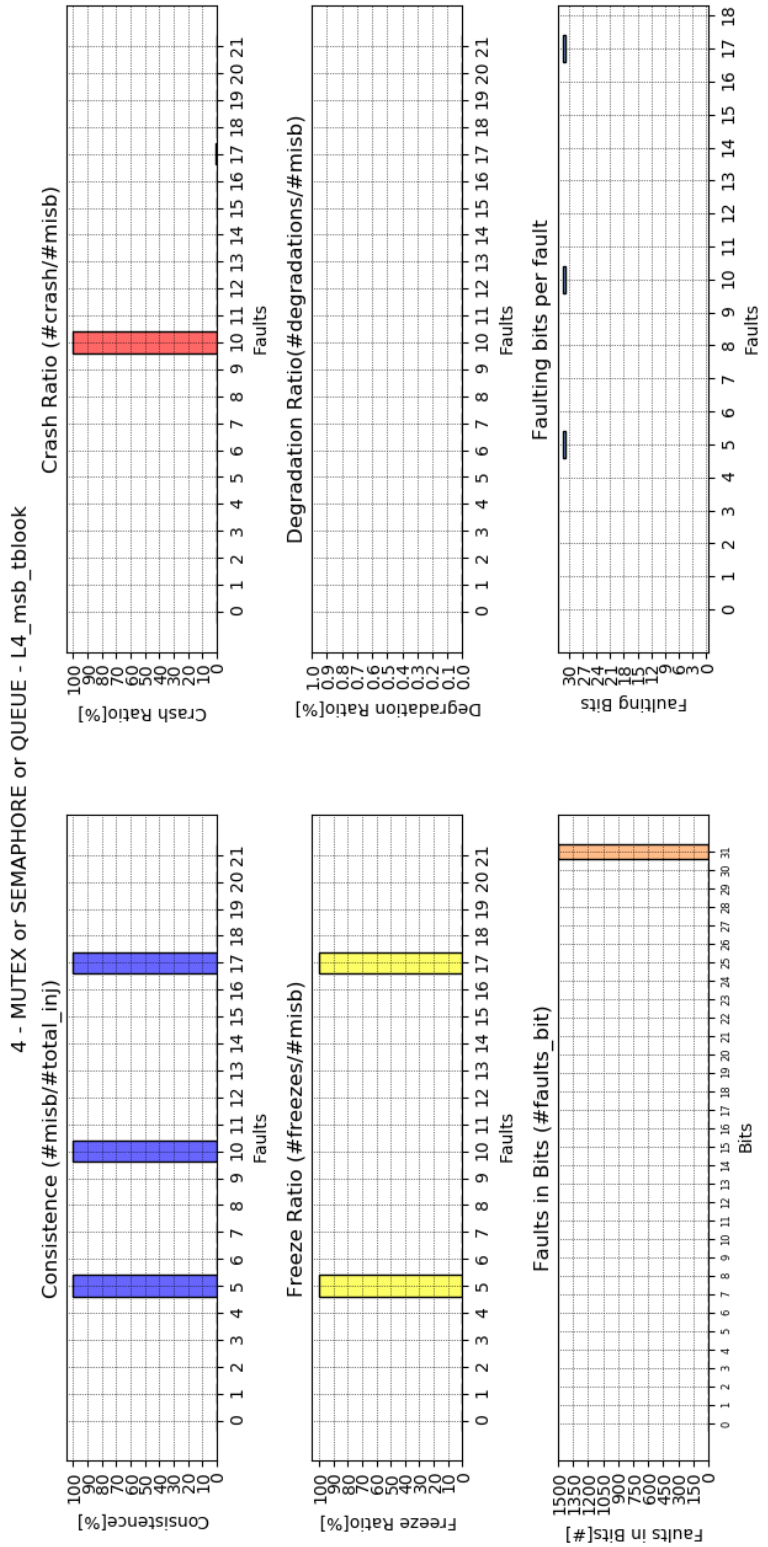

Figure 8.36: Results of injections in 1MSB, using tblook benchmark

8.8 Consistency dependence on tolerance

As the algorithm used to extract results performs analysis on the behavior of the system, there is a dependence between the results obtained and the value of the tolerance parameter passed to the parsing script: in particular, for those faults that cause freezes and degradations, such relation is highly visible as a different tolerance leads the parser to be more or less sensitive to differences between the golden run and the injection run; such dependency is clearly visible in a restricted subset of faults in list 1 (FreeRTOS global variables) while other lists give more homogeneous results for different tolerance values. Figure 8.37 shows how consistency increases for lower values of tolerance, as, even small variations (that are probably not necessarily due to the injection) are classified as misbehaviors. It is anyway interesting to notice that only faults related to the real-time scheduling of tasks are very dependent on tolerance: uxTopReadyPriority, for example, is a variable not used to manage timing-related operations but it is exploited just to know, at every system tick, which is the highest priority available in the OS, so for every value of tolerance it shows always the same results. Other variables like xTickCount and xNextTaskUnblockTime, one the contrary, are used exactly to manage time-related events, to choose when a task must be moved back from delayed to ready state: a different level of tolerance causes a different number of misbehaviors detected to be identified since injections in such places harms the system in a slight way; such misbehaviors can be seen only reducing tolerance, but not too much to not classify erroneously also normal differences between the two runs of the campaign.

Set of graphs in 8.38 instead shows how faulting bits are identified for different values of

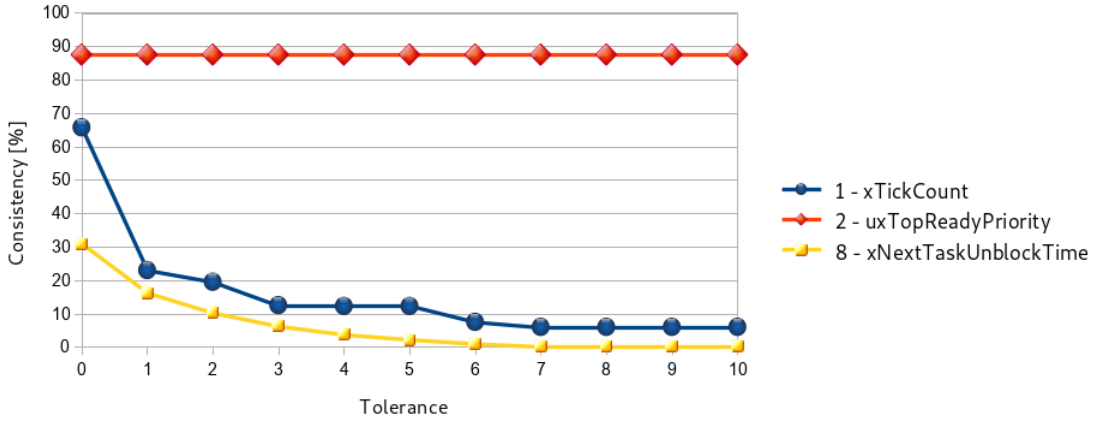

Figure 8.37: Tolerance-Consistency dependency for idctrn benchmark

tolerance (please, focus on faults 1 and 8): when such parameter is set to 0, all injections seem to have a failing outcome but this is clearly due to a wrong classification; to remove these false positive results, tolerance is increased, but this time the number positives found for time-related variables decreases with a dependency on bits: injections in higher bits lead to more visible behaviors which can be identified also with higher tolerance values but other misbehaviors which requires a higher sensitivity disappear.

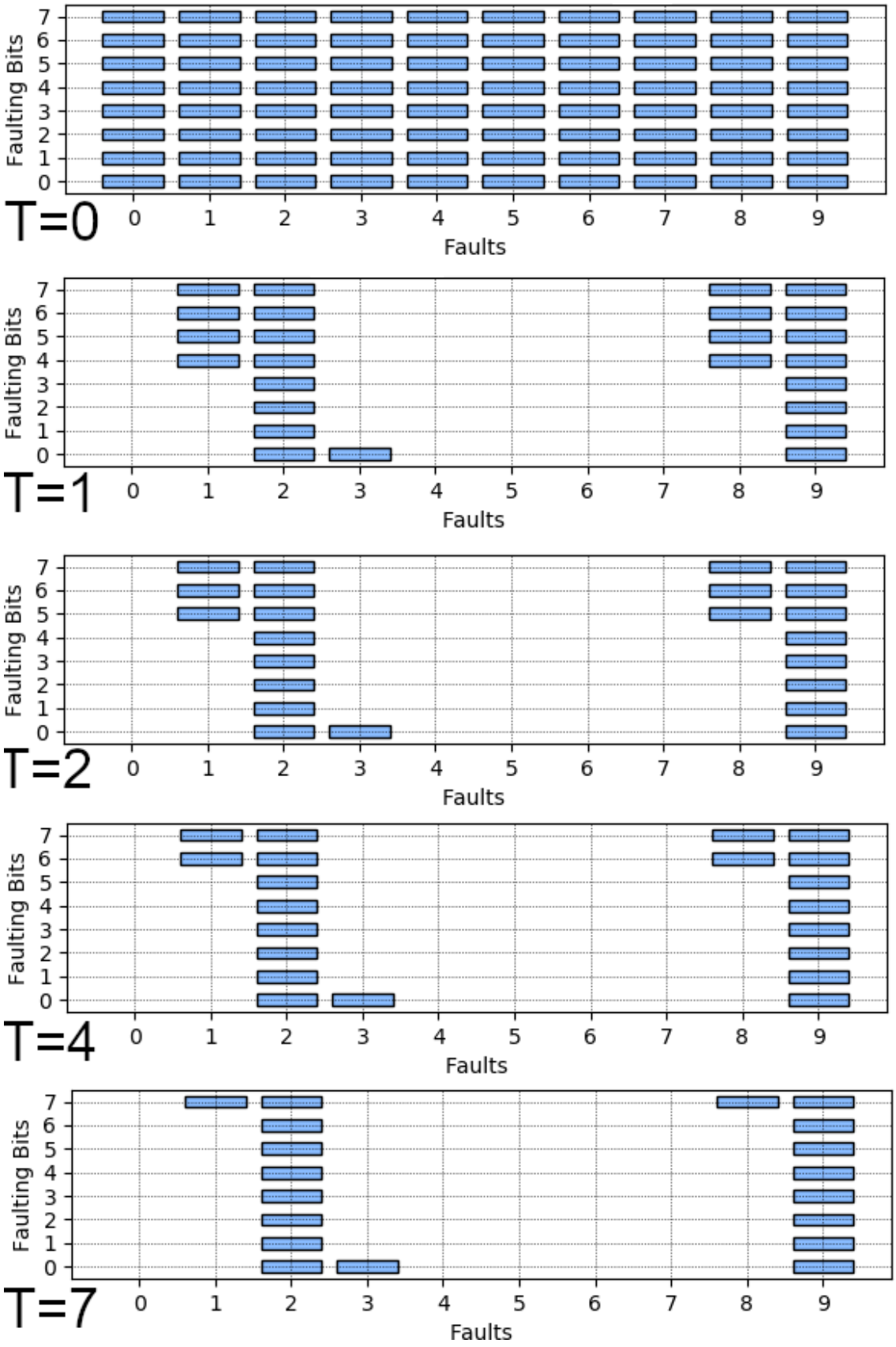

Figure 8.38: Identification of faulting bits dependence on tolerance, for tolerance values of 0, 1, 2, 4, 7 and for idctrn benchmark, fault list 1

Chapter 9

Conclusions

After long injection campaigns, some vulnerabilities have been found in FreeRTOS: many variables show always the same behavior, with some justifiable differences, for all the benchmarks used. In some cases such differences are due to the speed of the algorithm executed, in other cases instead they depend on the moment when injection was performed and finally other ones are due to the different way in which the various features of the RTOS are exploited.

9.1 Summary

Most critical vulnerabilities are pointers and numerical values stored in integer variables (both signed and unsigned) used to address elements of lists or vectors, especially when the bits injected are the first LSBs or the last MSBs: a modification of high order bits can lead to point to parts of the memory that could not even map in a valid position; changing the first LSBs instead leads to a memory access in a location which in many cases is not tolerated.
Other variables instead produce freezes; in this case the problem is due to a critical error that does not induce a crash and that is solved internally: when a critical datum is used by a kernel function, a configASSERT() function is usually called in order to perform a preliminary analysis on its integrity and eventually on its expected value, and it blocks the execution stalling in a infinite while loop if such datum is different from the expected one. This is already a partial solution to injections but freezing a system instead of allowing it to run is only a way to avoid further problems: a better solution is the use of a sort of data redundancy in the RTOS code and the implementation of a voter which performs a choice every time sensible variables are accessed.
Finally degradations are caused by changes in the scheduling process: this mainly happens when the kernel modifies its choices using values stored in injected variables and so showing a final behavior different from the expected one. In this case no problem is detected by the configASSERT() function and so system does not freeze.

In general, injections done in faults of the first fault list (global variables) cause freezes and degradations, whereas crashes occur only in injections in higher order bits.
Concerning experiments involving TCBs - second fault list - instead, system showed a slight worse tolerance to faults in the case of injections in ready task's TCB rather than current task's one: this is due to the fact that values flipped in current task's TCB, in some occasions, are overwritten or not read by the kernel after the injection and so in this

case they do not produce any misbehavior; a ready task's TCB, on the other hand, leads to more misbehaviors because it is inevitably used as soon as the task is selected and moved to running state: looking at consistency values in graphs, one must keep in mind that they are lower in the case of ready tasks TCB injections only because, sometimes, no ready task is available to perform the injection, reducing thus such value. Injections in MSB produced critical misbehaviors too, with a higher number of crashes with respect to experiments in LSBs in the same locations.
A similar reasoning can be applied to the third list: when injecting in the ready tasks list the number of misbehavior is very high and most of them are crashes; when experiments are done instead on the delayed task list, the number of misbehaviors, like the crashes, reduces and freezes and degradations become more relevant. This happens because the ready tasks list is continuously consulted by the kernel to take decisions while the delayed tasks list is accessed a reduced number of times and is used less intensively.
Finally injections in mutexes structure are not so destructive: crashes happens only in few faults while freezes are the recurrent misbehavior.

As final remark, exploiting the runtime task creation procedure, used by the multithread task *idctrn*, solicits ulteriorly the system, causing more misbehavior and highlighting in this case an even lower tolerance to faults.

Tables 9.1 and 9.2 contain a summary of the most sensitive variables of the system among the tested ones: marked entries are pointers.

Fault list	**Fault number and name**	**Consistency**
1	2 - uxTopReadyPriority	80%<C<90%
	9 - uxSchedulerSuspended	C=100%
2, Current task TCB	10* - xStateListItem.pxNext	C=100%
	12* - xStateListItem.pvOwner	C=100%
2, Ready task TCB	10* - xStateListItem.pxNext	35%<C<80%
	11* - xStateListItem.pxPrevious	35%<C<80%
	12* - xStateListItem.pvOwner	35%<C<80%
	13* - xStateListItem.pvContainer	35%<C<80%
	18* - xEventListItem.pvContainer	35%<C<80%
3, Ready task list	0 - uxNumberOfItems	C=100%
	1* - pxIndex	C=100%
3, Delayed task list	3* - xListEnd.pxNext	C=100%
4	5 - xTasksWaitingToSend.uxNumberOfItems	38%<C<100%
	10 - xTasksWaitingToReceive.uxNumberOfItems	38%<C<100%
	17 - uxItemSize	38%<C<100%

Table 9.1: Summary of most sensitive faults to LSB injections

9.2 RTOS hardening

FreeRTOS can be hardened in different ways. One of the best methods is the introduction of a certain level of redundancy, especially in those places where most critical problems

Fault list	Fault number and name	Consistency
1	2 - uxTopReadyPriority	C=100%
	4 - uxPendedTicks	C=100%
	9 - uxSchedulerSuspended	C=100%
2, Current task TCB	10* - xStateListItem.pxNext	C=100%
	12* - xStateListItem.pvOwner	C=100%
2, Ready task TCB	10* - xStateListItem.pxNext	32%<C<100%
	11* - xStateListItem.pxPrevious	32%<C<100%
	12* - xStateListItem.pvOwner	32%<C<100%
	13* - xStateListItem.pvContainer	32%<C<100%
	18* - xEventListItem.pvContainer	32%<C<100%
3, Ready task list	0 - uxNumberOfItems	C=100%
	1* - pxIndex	C=100%
3, Delayed task list	3* - xListEnd.pxNext	C=100%
4	5 - xTasksWaitingToSend.uxNumberOfItems	36%<C<100%
	10 - xTasksWaitingToReceive.uxNumberOfItems	36%<C<100%
	17 - uxItemSize	36%<C<100%

Table 9.2: Summary of most sensitive faults to MSB injections

occurred during experiments; this means that all the most sensitive data must be duplicated or triplicated and a voting system must be implemented, adding some computational overhead to all kernel procedures. Error correction mechanism would be preferable in the case in which an application with a high reliability must be designed; on the contrary, if the application has to show a high availability, a simple error detection system would be sufficient, forcing a software reset or isolating the affected subpart of the system in case a critical error is detected. It is important to notice that FreeRTOS already includes some macros which must be implemented by the programmer to perform checks on the integrity of some data in the system or in some parameter passed to kernel functions before using them. This is a good starting point, supported already by the presence, in some critical points of the code of the RTOS, of some configASSERT() macros, aiming at hanging the system in case of unexpected values.

9.3 Future improvements

This work will be improved by separating, in the FIEbrd section, the hardware-dependent code from the rest of the DUT-side system, in order to make simpler the porting to other platforms. Then, always in the DUT-side, an advanced tracing system can be developed: actually the written tracer already has an additional beta feature (which was not used during the experiments), but it needs to be improved by compressing logged data and by speeding up the transmission of such information to the host machine; this feature is already provided by third-part software like SEGGER RTT but such system requires too much memory, so an optimized ad-hoc solution would be better.
Another idea consists in the improvement of the algorithm used to identify and classify results: a more accurate set of misbehaviors can be created and outcomes can be analyzed with a deeper accuracy by exploiting other types of comparisons or by tracing system and tasks events in a more efficient way.

Appendices

Appendix A

Mutex take and give algorithm

A.1 Mutex take operation pseudocode

Listing A.1: Pseudocode of the take operation on mutexes

```
IF there is a message in the queue THEN
  - Set the holder of the mutex to the current TCB
  - Decrement the number of waiting messages
  IF the list of tasks that are waiting to release the mutex
  is not empty THEN
    - Remove that task from the xTaskWaitingToSend list
    IF that task preempts the current one (equal or higher
    priority) THEN
      - Force a switch by setting the PendSV interrupt bit to
       one in the interrupt control register
ELSE
  - Set a timeout
Suspend the scheduler
Lock the queue
Check for timeout
IF it did not expire THEN
  - Inherit the priority: if the mutex holder has a lower
  priority than the current task, the former will inherit the
   higher priority, it will be removed from the list of tasks
   with the old priority and it will be reinserted in the
  list of tasks with the new (higher) priority
  - Put the tasks in the list of the tasks waiting to receive
   the mutex
  - Unlock the queue
  - Resume the scheduler
ELSE
  - Unlock the queue
  - Resume the scheduler
```

A.2 Mutex give operation pseudocode

Listing A.2: Pseudocode of the give operation on mutexes

```
IF there is a message in the queue THEN
  - Set the holder of the mutex to NULL
  - Disinherit the priority (set again uxPriority to
  uxBasePriority)
```

```
  IF the list of tasks that are waiting to get the mutex is
  not empty THEN
    - Remove that task from the xTaskWaitingToReceive list
    IF that task preempts the current one (equal or higher
    priority) THEN
      - Force a switch by setting the PendSV interrupt bit to
       one in the interrupt control register
ELSE
  - Set a timeout
Suspend the scheduler
Lock the queue
Check for timeout
  Check only to know if overflow occurred or not in the
  meantime, because the Give operation is made with a delay
  time of 0 (so it is not needed to see if the timeout
  expired, as in Take).
Unlock the queue
Resume the scheduler
```

Appendix B

FIEmon.py detailed algorithm

The detailed pseudocode of the host-side script FIEmon.py is reported below.

Listing B.1: Detailed pseudocode of the host-side algorithm

```
// Initialization section
Initialize USART serial connection
  Set baud rate
  Set timeout of blocking operations
  Set parity
  Set byte size
  Open channel
Create a log file with desired name
Flush serial buffer
Initialize injection parameters
  Set cycle counter $FIE_INJ_CYCLE = 0;
  IF ( $cFIE_INJ_BITN >= 0 )
    Set bit number counter $FIE_INJ_BITN = 0
  ELSE
    Set bit number counter $FIE_INJ_BITN = max(#bit of the
    first fault)
  Set locus counter $FIE_INJ_LOCUS = 0
  Set period timer counter $FIE_INJ_TIME_T = 0
  Set prescaler timer counter $FIE_INJ_TIME_D = 0

// Main loop controlling the run
$end = 0;
WHILE ( $end == 0 ){

  // Wait for STARTITCHAR to start the synchronization
  $startchar = Receive char from serial (blocking)
  IF ( $startchar == $STARTITCHAR ) break

  // Send synchronization confirmation and back paramters
  Send INIT command through serial port
  Send injection parameters

  // Receive results of the injection and log them in the
  file
  $bufchar = Receive char from serial (blocking)
  $buf = ""
  WHILE ( $bufchar != $STOPITCHAR ){
    $buf = $buf + append $bufchar
  }
  Save $buf in the log file
```

```
// If SIJ mode activated exit here
IF ( $cFIE_ENABLED == $cFIE_SIJmode ){
  EXIT
}

// Update, checking one by one, the five variables
IF ( cycle counter < $cFIE_INJ_CYCLE ){
  cycle counter ++
}
IF ( $cFIE_INJ_BITN > 0 ){
  IF ( bit number counter < $cFIE_INJ_BITN ){
    cycle counter = 0
    bit number counter ++
  }
  ELIF ( locus counter < $cFIE_INJ_LOCUS ){
    cycle counter = 0
    bit number counter = 0
    locus counter ++
  }
  ELIF ( timer period counter < $cFIE_INJ_TIME_T ){
    cycle counter = 0
    bit number counter = 0
    locus counter = 0
    timer period counter ++
  }
  ELIF ( timer prescaler counter < $cFIE_INJ_TIME_D ){
    cycle counter = 0
    bit number counter = 0
    locus counter = 0
    timer period counter = 0
    timer prescaler counter ++
  }
  ELSE {
    $end = 1
  }
}
ELIF ( $cFIE_INJ_BITN < 0 ){
  IF ( bit number counter > 0 ){
    cycle counter = 0
    bit number counter --
  }
  ELIF ( locus counter < $cFIE_INJ_LOCUS ){
    cycle counter = 0
    locus counter ++
    bit number counter = max(#bit of the first fault)
  }
  ELIF ( timer period counter < $cFIE_INJ_TIME_T ){
    cycle counter = 0
    locus counter = 0
    bit number counter = max(#bit of the first fault)
    timer period counter ++
  }
  ELIF ( timer prescaler counter < $cFIE_INJ_TIME_D ){
    cycle counter = 0
    locus counter = 0
    bit number counter = max(#bit of the first fault)
    timer period counter = 0
    timer prescaler counter ++
  }
  ELSE {
    $end = 1
```

```
    }
  }
}
Close log file
Close serial connection
EXIT
```

Appendix C

FIEparser.py detailed algorithm

The detailed pseudocode of the host-side script FIEparser.py is reported below.

Listing C.1: Detailed pseudocode of the host-side algorithm

```
// Initialization section
Define the fault lists
Define lengths expressed in bits of all the faults
Set the tolerance passed as parameter or use the default one
Create void lists
  $l_vuln_loci = []
  $l_crash = []
  $l_freeze = []
  $l_degr = []
Initialize counters
  $_n_of_injections = 0
  $_n_of_misbehaviors = 0
  $_n_of_crashes = 0
  $_n_of_freezes = 0
  $_n_of_degradations = 0
Begin the parsing operations
  Open golden run file
  Open injection run file
  FOR each experiment line in both golden and injection run
  files{
    Divide the line in three parts
      $bm_params_inj = Find injection parameters
      $bm_res_inj = Find tasks log informations
      $sys_res_inj = Find system status
      $bm_params_golden = Find injection parameters
      $bm_res_golden = Find tasks log informations
      $sys_res_golden = Find system status
    $warning = 0
    FOR each task log in $bm_res_inj {
      Compare number of events
      IF (|var_inj| > var_golden+threshold OR |var_inj| <
      var_golden-threshold){
        $warning ++
      }
    }
    IF ( $sys_res_inj == crash ){
      Set the misbehavior as crash
      $_n_of_crashes ++
    }
    ELIF ( $warning == total number of tasks ){
      Set the misbehavior as freeze
```

```
      $_n_of_freezes ++
    }
    ELIF ( $warning < total number of tasks ){
      Set the misbehavior as degradation
      $_n_of_degradations ++
    }
  }
Plot results in graphs
```

Bibliography

[1] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. Ferrari: A tool for the validation of system dependability properties. *IEEE*, pages 336–344, 1992.

[2] João Carreira, Henrique Madeira, and João Gabriel Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998.

[3] Timothy K. Tsai and Ravishankar K. Iyer. Ftape: a fault injection tool to measure fault tolerance. *American Institute of Aeronautics and Astronautics*, 1994.

[4] Gregor Wicklein. *A Generic Fault Injection Framework for the Android OS.* Technische Universität Darmstadt, Germany - Darmstadt, July 2012.

[5] EunJin Jeong, Namgoo Lee, Jinhan Kim, Duseok Kang, and Soonhoi Ha. Fifa: A kernel-level fault injection framework for arm-based embedded linux system. *10th IEEE International Conference on Software Testing, Verification and Validation*, pages 23–34, 2017.

[6] Nejmeddine Alimi, Mohsen Machhout, Younes Lahbib, and Rached Tourki. An rtos-based fault injection simulator for embedded processors. *International Journal of Advanced Computer Science and Applications*, 8(5):300–306, 2017.

[7] Andréas Johansson. *Robustness Evaluation of Operating Systems.* Technische Universität Darmstadt, Germany - Darmstadt, January 2008.

[8] Claus Grupen. *Astroparticle physics.* Springer, Germany - Siegen, 2005.

[9] Luca Sterpone. *Electronics System Design Techniques for Safety Critical Applications.* Springer, Italy - Torino, 2008.

[10] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Günther H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, 52(9):1115–1133, September 2003.

[11] P. Rech, A. Bosio, P. Girard, S. Pravossoudovitch, A. Virazel, and L. Dilillo. A memory fault simulator for radiation-induced effects in srams. *IEEE Asian Test Symposium*, pages 100–105, 2010.

[12] Richard Barry. *Using the FreeRTOS real time kernel.* FreeRTOS, www.freertos.org, 2009.

[13] EEMBC. *Autobench - Software benchmark data book.* EEMBC, www.eembc.org, 2015.