Real-time Embedded System Fault Injector Framework for Micro-architectural State Based Reliability Assessment

Journal of Electronic Testing (2025) 41:193–208
https://doi.org/10.1007/s10836-025-06170-w

RESEARCH

Real-time Embedded System Fault Injector Framework for Micro-architectural State Based Reliability Assessment

Enrico Magliano[1] · **Alessandro Savino**[1] · **Stefano Di Carlo**[1]

Received: 17 September 2024 / Accepted: 4 April 2025 / Published online: 17 May 2025

Abstract

The increasing complexity of Safety-Critical Real-Time Embedded Systems (SACRES) presents significant challenges regarding reliability, security, and trustworthiness. Key concerns include the system's vulnerability to instantaneous voltage spikes, electromagnetic interference, neutron strikes, and temperatures out of range, which can induce bit-flipping and consequentially temporary corruption of stored memory data and soft errors. These errors may result in system faults that could push the system into dangerous states. In high-stakes fields like automotive, aerospace, and avionics, such failures can have serious, real-world consequences, potentially endangering lives. This paper introduces an innovative, fully configurable fault injection tool designed to monitor and analyze the micro-architectural state of the system. This tool allows a tailored injection campaign, including both CPU registers and RAM, with a flexible fault model able to inject single and multi-bit-flipping in the application and Operating System (OS) space. Tracking the architectural events using the microprocessor's Performance Monitoring Unit (PMU) and debugging interface. A key feature is its ability to ensure the repeatability of fault injections, which focus on bit-flipping in memory systems. The results of these fault injections allow for a detailed analysis of how soft errors affect system performance, output integrity, and timing predictability, all of which are critical in SACRES.

Keywords Embedded System · Soft Error · Fault Injector · Architectural Event · Reliability

1 Introduction

The advancement of embedded systems has led to enhanced performance to meet the growing demands of increasingly complex applications. However, this progress has also amplified system complexity, introducing multi-core architectures with numerous Central Processing Units (CPUs) and intricate memory hierarchies featuring multiple cache levels. Consequently, modern chips have become more susceptible to soft errors, especially when coupled with layered software stacks, as faults can propagate and potentially lead to severe consequences during application execution [1]. Environmental factors interacting with electronic circuits can trigger bit-flips and subsequent data corruption, resulting in soft errors [2]. Numerous studies have focused on analyzing and mitigating this issue to ensure the reliability and trustworthiness of Safety-Critical Real-Time Embedded Systems (SACRES). Within this context, fault injection is a crucial tool, enabling either the simulation of soft error occurrences or the direct injection of faults into the system [3–5]. Various fault injection methodologies and tools exist. Hardware-based implementations, such as those discussed in [6–8], provide high accuracy but generally exhibit limited controllability. Conversely, software-implemented fault injectors like those presented in [9–12], typically introduce additional time overhead but offer enhanced controllability and observability.

This paper extends the fault injection framework designed for SACRES, presented in [13]. Real-time embedded systems exhibit unique characteristics, as software workloads must be designed to satisfy stringent timing constraints and deterministic execution requirements; these peculiarities can be

Responsible Editor: L. M. Bolzani Pöhls

✉ Enrico Magliano
enrico.magliano@polito.it

Alessandro Savino
alessandro.savino@polito.it

Stefano Di Carlo
stefano.dicarlo@polito.it

[1] Department of Control and Computer Engineering(DAUIN), Politecnico di Torino, Corso Duca degli Abruzzi, 24, 10129 Turin, Piedmont, Italy

leveraged to implement effective fault injection experiments. The existing framework has been expanded to support the injection of single and multiple bit-flips into actual embedded hardware boards by utilizing the debug unit of modern CPU, thus ensuring comprehensive controllability with minimal timing overhead.

Although the fault injector remains portable across various embedded platforms, it has been specifically tailored and validated on Xilinx Zynq™ boards running the FreeRTOS [14] embedded Operating System (OS), a widely adopted configuration in several embedded applications. Historically, reliability analyses in embedded systems have predominantly focused on the application layer [15, 16], with limited studies highlighting the importance of the OS in such analyses [17]. Initial explorations into faults within the OS layer were conducted in [18], while recent research efforts [19–21] have addressed specific OSs and extensively targeted various OS data structures. The proposed fault injector has been designed to be highly general, capable of injecting any desired number of faults into CPU registers and RAM across both the OS and application spaces, covering the entire computation timeline from OS bootstrapping to task completion. These features ensure complete controllability throughout the injection process.

The paper is structured as follows: Section 2 provides a brief overview of real-time embedded systems and fault injection. Section 3 outlines the methodologies for implementing the fault injection. Section 4 details and discusses the results of utilizing our fault injection. Finally, section 5 provides a comprehensive summary of this study.

2 Background

Real-Time Operating Systems (RTOSs) are specialized operating systems designed to manage hardware resources, tasks, and processes in environments where timing constraints are critical. Unlike general-purpose operating systems, an RTOS ensures critical tasks execute within strict timing limits, providing predictable and deterministic behavior. Such predictability is essential in systems with strict timing requirements, like embedded systems used in automotive, aerospace, industrial automation, and healthcare applications. Notably, the scheduler in an RTOS is typically more predictable than those in general-purpose systems [22]. This predictability stems from real-time schedulers that guarantee task execution within precise timing constraints. Unlike general-purpose schedulers, who prioritize fairness and efficiency, real-time schedulers apply deterministic policies, ensuring the timely execution of critical tasks. This deterministic behavior helps reduce execution variability, which can be exploited during fault injection experiments, as discussed in later sections. Moreover, real-time tasks are often implemented as periodic tasks, repeatedly executing the same operations across multiple periods, with the strict constraint of completing execution within each period.

Regarding the memory hierarchy, a typical embedded system managed by an RTOS commonly employs static memory allocation for predefined tasks. Embedded RTOSs usually lack memory protection mechanisms, making them susceptible to corruption. Fault injection can introduce faults during real-time operations, affecting both the OS bootstrapping phase and task execution. Furthermore, memory technologies utilized in embedded systems frequently do not include Error Correcting Codes (ECCs), further increasing vulnerability.

FreeRTOS is one of the most widely used open-source RTOS solutions, popular in embedded systems due to its portability, simplicity, and ease of integration [14]. It is designed for microcontrollers and small processors with limited resources, supports various architectures, and is configurable to meet specific application needs. When assessing RTOSs reliability, profiling the execution flow is a valuable tool for identifying potential issues indicating soft errors or missed timing constraints [23, 24]. For this purpose, modern CPUs often incorporates dedicated hardware known as a Performance Monitoring Unit (PMU), which tracks architectural events such as branch instructions, memory accesses, and cache hits or misses. This tracking is performed through specialized hardware counters stored in dedicated registers, i.e., Hardware Performance Counters (HPCs), each capable of monitoring a specific event throughout program execution. However, as each counter monitors only one event at a time, multiple runs may be required to collect data on several events.

Hardware faults are usually classified based on the different effects on the behavior of a computing system [21]:

1. *Benign*: The system continues to execute normally, with the faulty execution outputs matching the golden execution. Such faults typically have no observable impact on system behavior, though they might degrade performance.
2. *Silent Data Corruption (SDC)*: The system completes execution, but the outputs differ from the golden execution. Although the system might continue operating without visible signs of failure, internal data is silently corrupted. SDCs are particularly insidious as they lead to incorrect operations without activating error-handling mechanisms, posing critical risks in safety-critical systems.
3. *Other (crash, hang, and reboot)*: This category encompasses catastrophic failures such as system crashes or hangs caused by severe issues like memory corruption or pointer mismanagement. Crashes occur when the system attempts an operation, leading to exceptions or kernel

panics, such as invalid memory access. Hangs result from deadlocks or infinite loops halting execution. Such faults generally require a system reboot or intervention for recovery. For instance, pointer corruption can cause unintended memory accesses, leading to undefined behavior like executing invalid instructions or overwriting crucial data structures [25].

Fault injection is a valuable technique for assessing system reliability under faults [26]. It involves controlled experiments to observe system behavior in the presence of faults. Fault injection methods can be categorized based on their mechanisms:

(i) *Physical fault injection* exposes the system implementation directly to external faults, such as radiation-induced faults [16].
(ii) *Hardware-based fault injection* exploits existing hardware interfaces, like the debug interface, to modify system behavior by accessing CPU registers [27].
(iii) *Software-based fault injection* injects faults directly into memory locations by instrumenting the software layer [28, 29].
(iv) *Model-based fault injection* manipulates system models (e.g., Hardware Description Language (HDL) or micro-architectural descriptions) during simulation or emulation [30, 31].

This paper focuses on hardware-based fault injection, particularly methods exploiting the debug interface to control and perform the injection. Typically, the approach involves a *target* board running the application and OS, and a *host* controlling the experiment. The host uses the debug port to halt the target, modify accessible resources to emulate faults, and resume execution [12]. While many existing studies utilize debuggers [32–34], recent works [31, 35, 36] have focused on expanding fault models or parallelizing injections for quicker exploration. However, these studies often lack comprehensive data-gathering support for analyzing results, classifying outcomes, and tracing events leading to system failures.

3 Fault Injector

As mentioned previously, in real-time systems, computational correctness is not the only critical factor; execution predictability is equally vital. Consequently, combining fault injection experiments on real hardware with extensive profiling of the CPU's micro-architectural state offers a promising approach for detailed reliability analysis. Beyond reliability assessments, profiling hardware events in fault conditions has shown potential for developing effective machine learning-based fault detection methods [23, 37]. However, due to limitations in the number of events that current PMUs can monitor simultaneously, it is essential to repeat injection experiments multiple times under controlled conditions, each time profiling a different set of events.

It is important to highlight that using a debug unit for fault injection is not a novel aspect (see [38]). The novel contribution of this paper is the coupling of fault injection with HPCs profiling, which requires methods to make injections reproducible to increase the number of collectible events. This type of analysis has previously been conducted in [23, 37], extending FIMSIM [39], a fault injection framework based on the gem5 micro-architectural simulator. Although FIMSIM is a powerful fault injector and gem5 enables detailed monitoring of the microarchitectural state, it suffers from performance limitations because gem5 must emulate the entire CPU microarchitecture. In contrast, by running on real hardware, the proposed approach can fully exploit the computing capabilities of the target platform. Moreover, the HPCs are not modeled for all target architecture, leaving the analysis of logs as the only resource to extract profile-alike data [37].

This section outlines the architecture of the proposed fault injector and describes its main design characteristics.

3.1 High-level Architecture

Figure 1 shows the high-level architecture of the fault injection framework.

Modern microprocessors include a dedicated debug unit, enabling precise control over software execution. This unit allows applications to be paused, facilitates reading and modification of CPU registers and memory, and subsequently permits the resuming of normal operation. By setting hardware breakpoints, execution can be halted precisely at the assembly instruction level, achieving accurate temporal control. The CPU's debug unit, externally controlled through the IEEE 1149.1 JTAG protocol [40], introduces minimal timing overhead when injecting faults into embedded applications.

3.1.1 Fault Model

A fault injection campaign always initiates by defining a list of target faults. This paper extends the Single Bit Upset (SBU) fault model implemented in [13] to accommodate Multi Bit Upsets (MBUs). An MBU is represented by a tuple $(vector\_of\_target\_structures, target\_instruction)$. The $vector\_of\_target\_structures$, specifies a list of fault locations, such as CPU registers and memory areas containing code or data. The $target\_instruction$ specifies the exact moment of fault injection, utilizing breakpoints to halt execution. Although straightforward, this method might limit time granularity in loop-based code, as it typically targets only the first iteration. Alternative approaches, such as using timers on the host computer, can address this limitation. Each

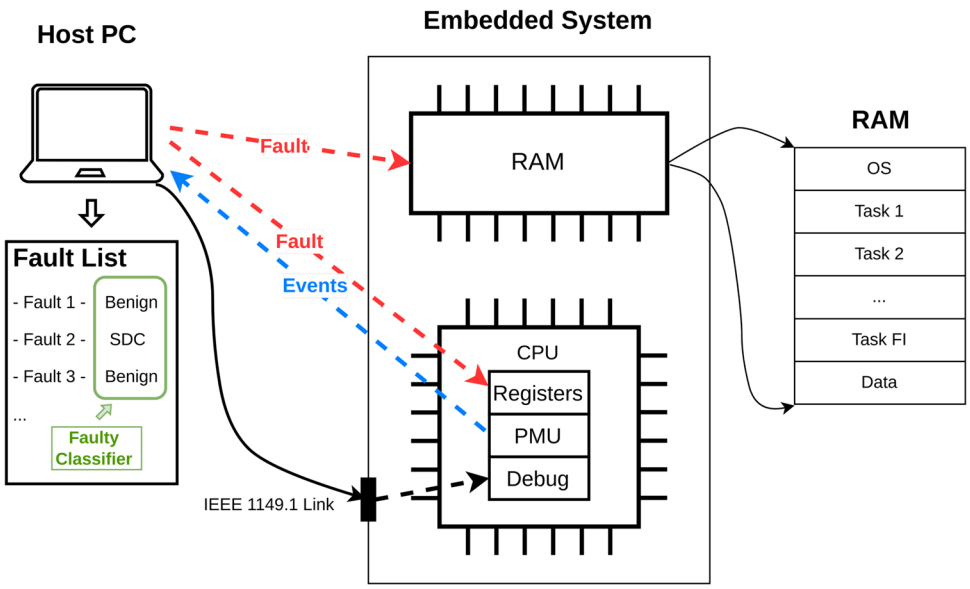


Fig. 1 High-level architecture of the fault injection framework. A host machine interacts with the target embedded system through an IEEE 1149 link to orchestrate the injection experiments

tuple provides complete control, representing a unique combination of fault injection locations and timing. An SBU, a special case where $vector\_of\_target\_structure$ contains only one element, can be expressed as the simpler tuple $(target\_structure, target\_instruction)$.

After each fault injection experiment, outcomes are classified by comparing the fault-free *golden execution* with the faulty execution. Fault classifications follow the definitions outlined in Section 2, categorizing outcomes as *benign*, *sdc*, or *crash/hang*.

The MBU fault model remains consistent across both injection locations (CPU registers and memory) and is fully configurable regarding the number of bit-flips and injection targets. For register injections, target registers can include general-purpose registers, floating-point units, and special registers (e.g., Program Counter (PC), link register). Memory injections allow the specification of multiple address ranges as potential targets.

Additionally, fault injection supports the profiling of hardware events during injection experiments through instrumentation with a dedicated injection task. The debug-based injection approach ensures reproducibility of faulty executions, an essential feature due to the limited number of HPCs in most CPU architectures, despite the capability to track numerous architectural events. Typically, only a subset of these events can be monitored during a single execution. Repeatable fault injection experiments enable monitoring of different events across multiple runs, allowing comprehensive profiling using the available HPCs in the CPU's PMU.

Another important feature for fault outcome classification involves identifying the specific assembly instructions at injection breakpoint locations. Given their addresses, retrieving these instructions from the Executable and Linkable Format (ELF) binary file facilitates statistical analysis and correlation studies between fault locations, instruction types, and registers involved. This approach is crucial for identifying the most sensitive instructions and registers, where corruption is most likely to result in SDC outcomes.

3.2 Implementation Details

Without loss of generality, this section provides additional implementation details specific to Xilinx Zynq™ boards running the FreeRTOS embedded OS.

The fault injection architecture comprises two distinct executable modules: one running on the embedded system (the *target*) and the other (the *host*) running on a separate host machine (see Fig. 1). The host module, implemented in Python, leverages the JTAG protocol to manage the debugging process of the target application through specific architectural commands, enabling fine-grained control over the target application's execution. Specifically, the host utilizes the Xilinx Software Command- Line Tool (XSCT) [41], an interactive and scriptable command-line interface from the Xilinx SDK. Built upon the Tools Command Language (Tcl), XSCT supports various tasks such as hardware configuration, creation of board support packages, application project management, and boot image flashing. The Python `Pexpect` module automates the interaction with XSCT, spawning and controlling the XSCT console and managing expected output patterns.

The target includes the entire embedded software suite (i.e., the workload) executing on the hardware board, encompassing the OS, application tasks, instrumentation code, PMU reading mechanisms, and input randomization for the golden run. All these components are compiled into a single ELF binary file, subsequently flashed onto the target hardware.

The proposed fault injector specifically targets embedded systems running flat operating systems such as FreeRTOS, which typically offer limited memory isolation. This limited isolation means faults can propagate from individual tasks to the operating system itself, making the OS a potential fault injection target. To prevent fault accumulation and ensure consistency, the system is rebooted between injections, thus guaranteeing each fault injection occurs in a clean state.

Figure 2 illustrates the host and target interaction. The host initiates the injection process by configuring the target board, and generating a fault list by randomly selecting tuples of locations and timings, as detailed in Section 3.1. In the fault injection target, the location specifies a CPU register or memory address and a particular bit position, while the timing refers to a memory address in the code space used to set a breakpoint. After the execution of the workload on the target, a final breakpoint is established, which is essential for reading the HPCs from the PMU. This approach ensures that the real-time OS (e.g., FreeRTOS) does not continue executing the `IDLE` task, facilitating the completion of the workload execution. For each location and timing tuple, the workload execution on the board occurs in two distinct phases: the *golden execution* and the *faulty execution*.

1. **Golden Execution**: This phase involves generating random inputs, saving this input, performing the golden run, and storing the golden output.
2. **Faulty Execution**: This phase includes reading the saved random inputs, configuring the PMU to monitor the selected architectural events, executing the faulty run, reading the golden output, comparing it with the faulty output, classifying the execution outcome as either SDC or Benign, and communicating this classification back to the host.

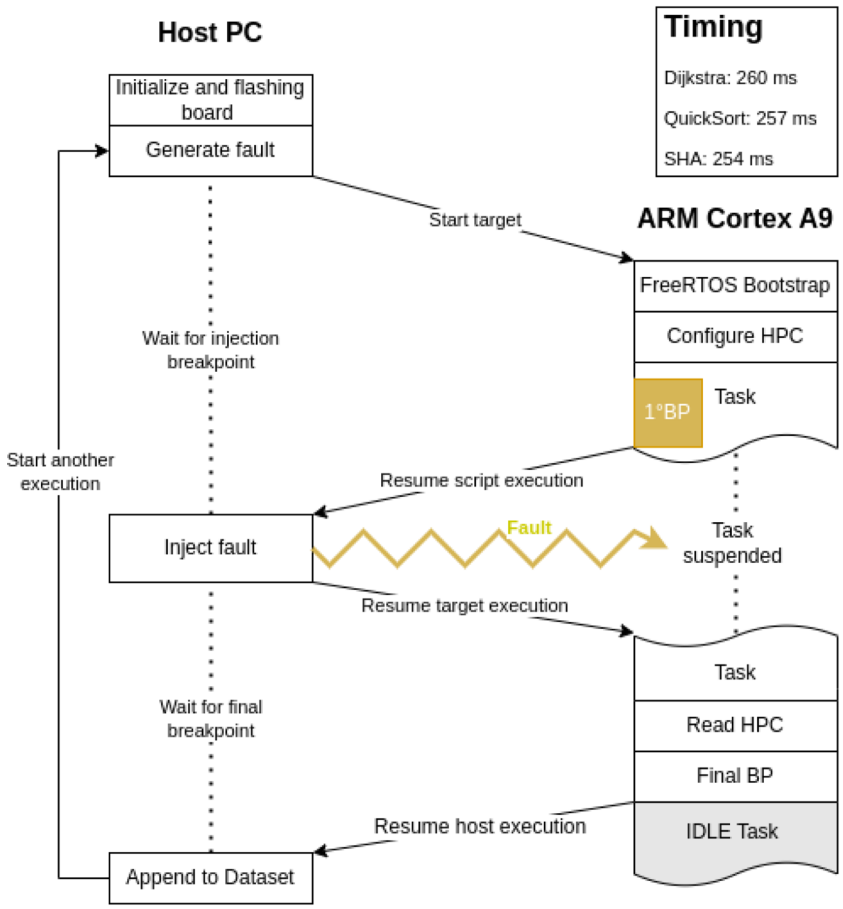


Fig. 2 The software execution flow involves interactions between the host, which runs the fault injection script, and the target embedded system, executing the real-time OS and application task; this interaction is synchronized through a timing reference, measured in milliseconds, corresponding to one iteration loop of each benchmark

During the **Golden Execution**, the host only needs to instruct the target to run the golden execution of the workload, boot the board, and await the final breakpoint. In contrast, during the **Faulty Execution**, the host enters an injection loop based on the number of architectural events tracked. At each iteration, the host instructs the target to execute the faulty run, specifies the architectural events to monitor, sets the injection breakpoint, boots the board, and waits until the initial execution phase-bootstrapping, HPC configuration, and task initiation-is complete.

When the injection breakpoint is reached, the host regains control, injects the faults, removes the breakpoint, and resumes execution. Without further breakpoints, the application tasks proceed toward the final breakpoint, signaling execution completion to the host. The host includes a timeout mechanism to manage cases where execution completion is not reached. To classify the execution outcome, the host compares the faulty output with the expected golden output. If the outputs match, the execution is classified as Benign; otherwise, it is classified as SDC. If a timeout occurs, the execution is classified as a crash/hang (Other). The host repeats this process for each fault as fault injections target various locations and timings.

The timeout period accounts for the combined time required for OS startup and benchmark execution, including additional overhead to distinguish between benign cases with minor timing deviations and genuine crashes/hangs clearly. The average startup time, measured across multiple executions, is approximately 0.119 seconds in our experimental setup. This distinction is crucial, as faulty executions may result in correct outputs with timing deviations, as illustrated in Fig. 8b.

The target software must be instrumented to support fault injection. This includes fault injection operations and event tracking via Tcl scripts. Listing 1 shows the pseudo-code for the main instrumentation of the RTOS. It includes a common main routine for the OS, defines a specific task for fault injection (line 2), and starts the scheduler (line 3). Since the fault injection process exploits the debugger, application tasks remain unchanged, with only necessary operations wrapped around them to ensure proper interaction with the PMU: initializing the PMU before target tasks (line 7) and reading HPCs after task completion (line 9). PMU configuration uses specific assembly instructions, such as `MCR <register> <value>` in ARM architecture. Reading from the PMU at the end of the benchmark is done using the `MRC <register> <value>` instruction instead.

3.2.1 Random Input Generalization

Generating random inputs directly on the target board is essential for enhancing fault injection performance and avoiding the overhead of transferring workload inputs from the host computer at each injection run. Randomization broadens fault injection coverage, increasing the likelihood of uncovering rare fault propagation scenarios. The input generation strategy is tailored to each benchmark: with workloads such as `quicksort` requiring integer arrays, while `SHA` operating on strings. To ensure randomness, inputs are sampled from a uniform distribution using the `rand` function, with the seed derived from the clock cycle count or Cycle Count Register (CCR) value. However, consistency is required when replicating the same fault injection to collect multiple architectural event measurements-each instance must receive the same input. Therefore, the seed used for input generation must be preserved. A key challenge is that FreeRTOS reboots at each fault injection run, reinitializing main memory and preventing the retention of critical data. To address this, all necessary parameters, including the input seed, are stored in a dedicated memory region separated from the main memory. This ensures both reproducibility and minimal interference with the system under analysis. This approach leverages the features of Xilinx Zynq™ systems, where the System on chip (SoC) comprises a Processing System (PS)-the ARM® Cortex® A9 running the software components, such as FreeRTOS and target benchmarks-and a Programmable Logic (PL) part containing reconfigurable hardware. The PL includes blocks of 36KB of Static RAM (SRAM), referred to as Block RAM (BRAM), which can be memory-mapped to an address space separated from the main memory. A single BRAM block can store all the necessary data between consecutive fault injection runs. This also keeps the information local to the PL, making it accessible to a local task and avoiding reloading fault injection data at every run, thus speeding up execution.

Pseudo-code 1 illustrates the implemented task management strategies. At the start-up of the system, the `faultInjectorTask` is created (line 2), and the control is given to the scheduler to start the normal execution of user tasks (line 3). The `faultInjectorTask` manages the execution of the target workload and communicates with the *host* to properly configure the experiment. In this phase, the BRAM serves as a shared memory space for exchanging data between the *host* and *target*. The *host* specifies which architectural events to monitor by listing them in the shared BRAM space. The *target* reads these data and determines the type of run based on the provided information (line 7): if no architectural events are specified, a Golden run is executed. In this case, a random input is generated (line 10) and stored in BRAM (line 11). If the *host* specifies a valid list of HPCs to monitor, the `faultInjectorTask` reads the input from BRAM (line 13), ensuring that the golden and faulty executions use the same stimulus, and configures the PMU to monitor the target HPCs (line 14). The task is then executed in the configured mode (golden or faulty). For golden executions, the output of the task is stored in BRAM

as a reference (line 18); otherwise, the PMU is interrogated to get the HPC values (line 20), the reference golden output is read from BRAM (line 21) and compared to the faulty execution to determine the type of fault (line 22).

Listing 1 *Task Creation Pseudo Code over the target embedded system*

```
1  int OS_main ( ){
2      taskCreate( faultInjectorTask );
3      taskStartScheduler();
4      for( ;; ); // endless loop for the OS
5  }
6  static void faultInjectorTask( ){
7      golden, events = commandFromHost()
8      input;
9      if (golden){
10         input = generateInput();
11         saveInput(input);
12     }else{
13         input = readInput();
14         confPMU(events);
15     }
16     output = startTasks(input);
17     if(golden){
18         saveOutput(output);
19     }else{
20         readPMU();
21         golden_output = readOutput();
22         result = compare(output, golden_output);
23     }
24
25     // taskDelete( NULL );
26 } // <- final breakpoint point here
```

3.2.2 Faults Reproducibility and Architectural Event Tracking

Ultimately, fault injection leveraging the debugger guarantees the reproducibility of each injection. In this context, it is crucial to distinguish between the actual fault injection process and the profiling activity based on performance counters. The fault injection process requires executing the entire workload for each injected fault. The debugger interface interrupts the workload execution at a specific instruction to inject the fault. The proposed fault injector enhances the injection process by integrating a profiling capability. However, due to hardware limitations, profiling requires collecting more architectural events than the available performance counters can track. Consequently, the same fault must be injected multiple times under identical environmental conditions to collect more events. The process involves selecting a list of architectural events and, depending on the number of these events and the available HPCs, fault injection repeats the injection with the same parameters but with different events to be monitored. This process introduces a performance overhead, as it increases the number of executions required to complete the fault injection campaign, as defined by:

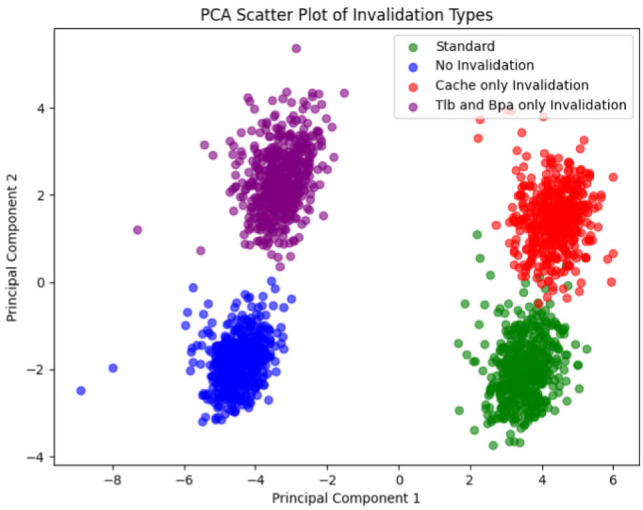
$$\#\,\text{repetitions} = \#\,\text{of events}/\#\,\text{of HPC} \tag{1}$$

In this context, rebooting the OS ensures the consistent re-injection of faults in the same architectural state (e.g., Cache, Branch Prediction Array (BPA), Translation Lookaside Buffer (TLB)). Nonetheless, during profiling, it would be valuable to analyze how the initial architectural state influences the profiling results. It is important to highlight that the fault injection process triggers a reboot but not a full hardware reset. Reproducibility is maintained during the boot sequence, as key architectural structures such as the cache, TLB, and branch predictor are flushed. For this purpose, an *ad hoc* experiment was performed, modifying the bootstrap of FreeRTOS to be able to inject faults and collect architectural events in four different scenarios:

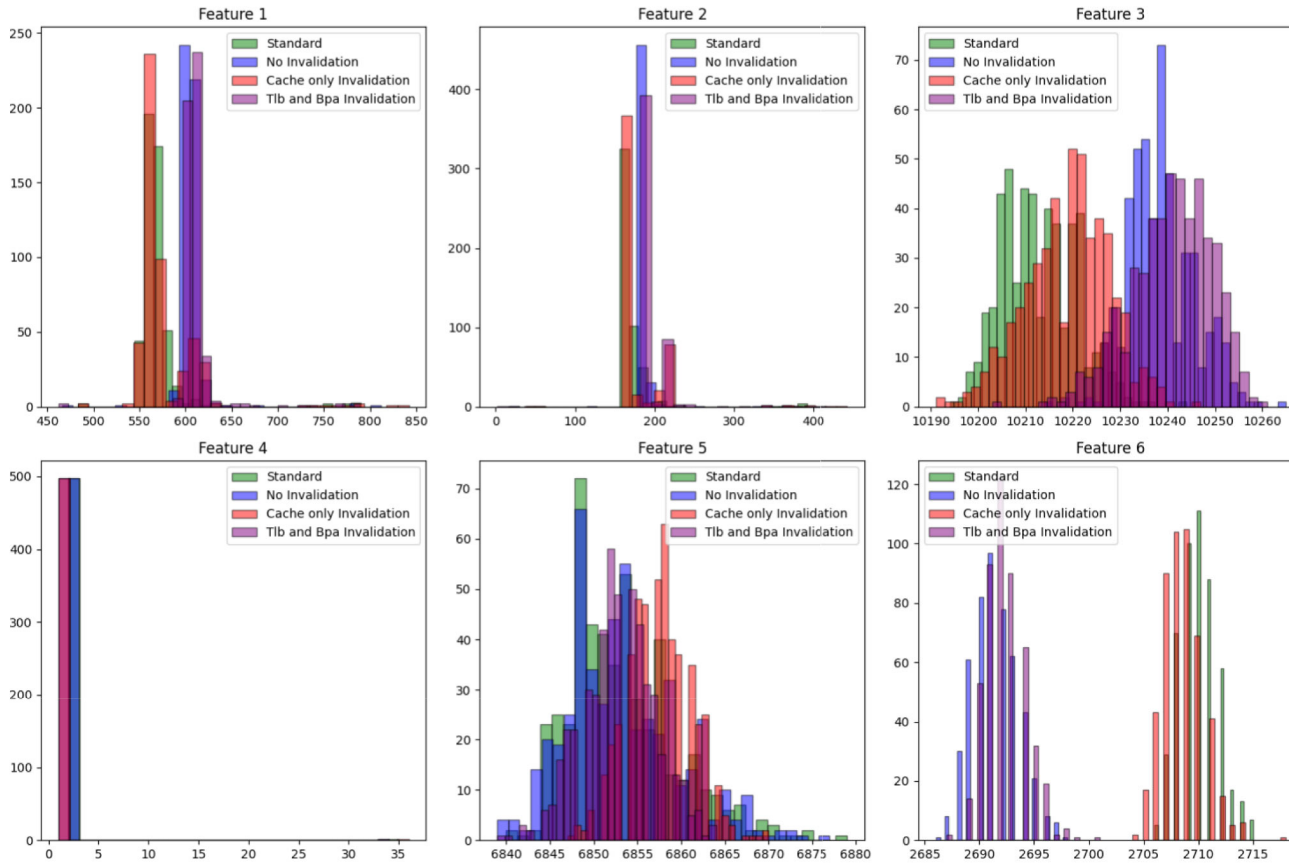
- Workload execution with a full reboot.
- Workload execution with a reboot, without invalidating the Cache, TLB, and the BPA.
- Workload execution with a reboot, invalidating the cache only.
- Workload execution with a reboot, invalidating only the TLB and the BPA.

This additional feature, introduced in the fault injector, allows us to highlight the effect of these different conditions on the application profiles. Figure 3a reports the results of this experiment in a PCA scatterplot, from which we can observe that different startup configurations lead to distinct profiles, with notable differences between the scenarios. Additionally, Fig. 3b allows us to appreciate the misalignment of some architectural events, particularly for features 3 and 6, while in other features, such as 2 and 5, the results remain consistent.

This analysis reveals that the initial hardware state configuration significantly impacts execution behavior. The availability of these data represents an important tool for understanding how these initial configurations affect the execution of the workload, thus collecting profiles that better reflect a realistic execution sequence representative of diverse hardware states.



(a) Scatter plot of PCA-2 of all the architectural events



(b) Histogram plot of a subset of architectural events

Fig. 3 Two plots: PCA-2 and Histogram of architectural events

Another important aspect that must be taken into consideration for reproducibility is that when multiple tasks are running, different execution profiles can introduce variability in the results. For the purpose of this paper, single-task experiments are performed to demonstrate the benefits of combining performance counter profiling with fault injection, thereby avoiding the aforementioned issue. Nevertheless, it is important to note that even when multiple tasks are scheduled, events can still be profiled per-task by intercepting context switches between tasks. This, of course, does not eliminate profile variability caused by different architectural states due to context switching and variations in execution order due to task interactions. However, as already mentioned, it is important to remark that in real-time embedded systems, schedulers are more predictable than those used in general-purpose systems and employ deterministic policies to ensure the timely execution of critical tasks [22].

4 Results

This section showcases the capabilities of the proposed fault injector on a selected experimental setup.

The target architecture is a Dual-core ARM Cortex A9 processor on the Xilinx PYNQ Z2 board. The PYNQ Z2, an open-source project from Xilinx [42], is based on the Xilinx Zynq™ SoC. The Arm Cortex A9, a 32-bit dual-core processor widely adopted platform, is based on `ARMv7` specifications, supports the Thumb and Thumb-2 instruction sets, and incorporates coherent cache management. It works at 650 MHz, and its features provide a cost-effective and performance-efficient representation of real-world environments. Moreover, the board features a USB port for configuration through JTAG.

The processor incorporates a configurable PMU designed for easy customization to monitor a diverse set of 168 architectural events. In the case of the ARM Cortex-A9, its PMU provides six HPCs, each associated with event-type registers that specify the tracked event, along with additional configuration registers. Access to these registers is facilitated through the internal CP15 interface, as elaborated in the ARM Architectural Reference Manual [43].

The application setup includes the FreeRTOS [14] operating system running a set of benchmarking tasks taken from the MiBench [44] suite. As benchmarks, this work uses a subset of the embedded benchmarks:

1. **SHA**: randomly create a string with a fixed length.
2. **Dijkstra**: Given a fixed graph, randomly generate the starting and the ending nodes.
3. **QuickSort**: randomly generate a vector of a string with fixed length.
4. **Rijndeal**: randomly generate a string used as a key for cryptography algorithm.
5. **Basicmath**: randomly generate an integer for the square root computation and a double for the degree to radian conversion.
6. **Stringsearch**: randomly select a string from a set of substrings to search inside the text.

They are selected due to the varying characteristics in te rms of time and complexity: (i) computationally intense tasks (`SHA` and `Dijkstra`, `Rijndeal and Basicmath`), and (ii) memory-intense tasks (`QSort` and `Stringsearch`). This diversity allows for a more detailed analysis of the effect of the fault injection when the execution time is affected. In the experiments, the faults occur in the task address space running the target benchmark.

We initially executed the benchmarks and profiled the variations in HPC during each run to speed up the injection process. Based on this preliminary analysis, we restricted the tracked architectural events to only those that exhibited changes across different executions. This considerably reduced the required runs since, as mentioned before, multiple runs with varying configurations of the PMU are needed to monitor larger architectural events.

4.1 Multi Task Workload

An OS configuration with a single task limits investigation capabilities and may not reflect realistic scenarios. To enhance the fault injection evaluation, we have implemented a `Multi Task` workload, which runs a subset of considered benchmarks (`QSort`, `SHA`, `Dijkstra`, `Rijndeal`, `Basicmath`, `Stringsearch`) concurrently, based on a configurable task schedule.

Using this generalized `Multi Task` workload, we leveraged FreeRTOS's scheduler functionality to perform context switching among various tasks. This setup enabled fault injection directly into OS code and memory, allowing us to analyze the impact on benchmark outcomes and timing.

Practically, multiple tasks are executed concurrently. An initial function launches the tasks according to the user-defined parameter *number_of_task*. Each task receives a `golden` parameter, influencing how results are processed. Task outputs are evaluated upon completion if the system is not performing the golden run. If *all* task results match the *Golden Output*, the execution is classified as benign; otherwise, it is classified as SDC. If at least one task fails to return, the execution is classified as a crash/hang.

4.2 Fault Injection Campaign

A total of 23 fault injection campaigns were conducted (see Figs. 4, 5, and 6) with the 6 target benchmarks discussed previously. Each campaign differs for the injection locations (memory, CPU registers, and PC), the injection time (during the execution of the task or the execution of the OS), and the fault type (SBU or SBU). In particular, among the 23 performed campaigns, 6 campaigns targeted the *CPU* registers with SBUs and 10, 000 faults, and 6 campaigns targeted the *memory* injecting 5,000 SBUs per campaign. Further, 3 campaigns specifically targeted SBUs in the PC. To demonstrate the capability of injecting MBUs, 6 campaigns targeting the *CPU registers*, injecting 5,000 MBUs with 3 different bit flips have been carried out. Eventually, 2 campaigns were dedicated to studying the effects of the faults in the OS, with the multitask benchmark presented in Section 4.1: a first one injecting in the *memory* and a second one considering *CPU registers* only. In both cases, 5,000 faults were injected. The fault injector is fully configurable, allowing users to select the number of faults and the injection locations to conduct relevant campaigns. However, in this paper, we focus solely on demonstrating the features and effectiveness of the fault injector.

In our preliminary experiments, we inject sufficient faults in each campaign to ensure a low error margin. The number of samples was determined based on the statistical fault injection approach introduced in the seminal paper [45].

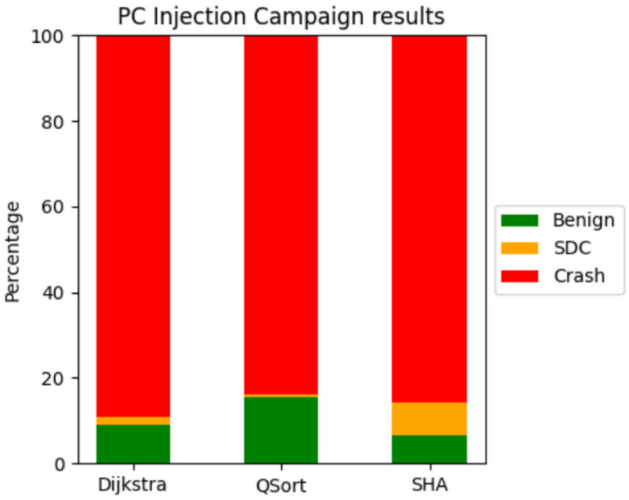


Fig. 5 Breakdown of all injection campaigns. For each benchmark in the PC

Specifically, for CPU register injections, we computed the number of injections required to achieve an error margin of $e = 1\%$. Conversely, for memory injections, where the fault space is significantly larger, we set the number of injections to achieve an error margin $e = 2\%$. These configurations

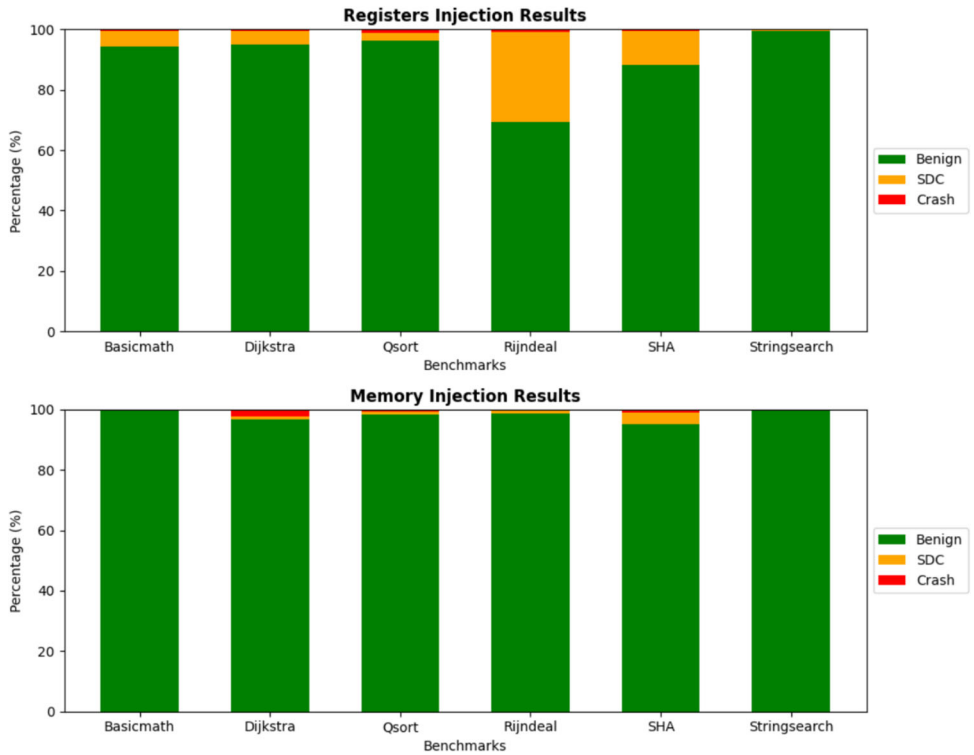


Fig. 4 Breakdown of all injection campaigns. For each benchmark, classification percentages are reported for all potential targets: Registers, PC, and Memory

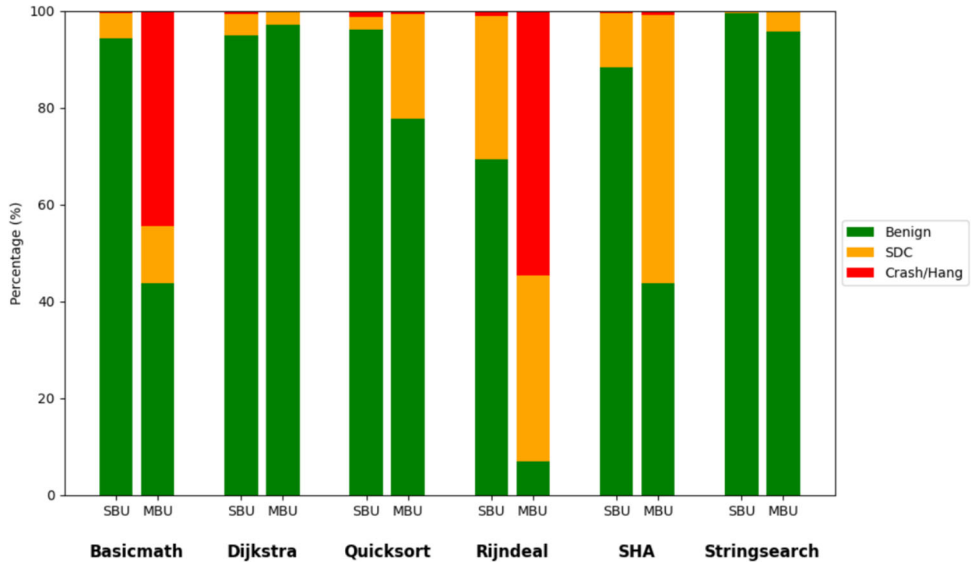


Fig. 6 Breakdown of all injection campaigns. For each benchmark in the registers, compare *Single-bit* with *Multi-bit*

were computed using a confidence score of $t = 2.58$ (corresponding to a 99% confidence level).

Table 1 reports the average injection time for a single fault injection among all the benchmarks. These data show similar results between memory and CPU registers campaigns. Instead, the average time needed to complete a faulty execution in PC campaigns is very high compared to the others. This is due to the high percentage of crashes/hangs of these campaigns, as identifying them requires the host to leverage a timeout exception, which takes 30 seconds before occurring.

The experiments targeting the memory used all addresses according to the ELF header, which reports all sections without distinguishing between OS and application sections. Regarding CPU registers, the solely standard registers were corrupted, excluding critical registers like the PC, which is specifically targeted in specific fault injection campaigns. During all experimental campaigns, the capability of profiling HPCs was used for further analysis. Eventually, every injected fault was labeled as Benign, SDC, or Other to characterize the fault effect.

The experimental campaign is divided into two parts. The first part focuses on faults occurring within the task address space executing the selected benchmark. The second part (section 4.2.3) focuses on faults in the operating system.

Table 1 Average time of single fault execution separated by the location

Location	Injection Time (ms)
Memory	257
Registers	261
PC	30280

4.2.1 SBU Injection Campaigns

Figure 4 illustrates injection outcomes based on fault locations and benchmarks. In CPU register campaigns, the predominant outcome is Benign for all benchmarks (94.3%, 95%, 96.2%, 69.4%, 88.4%, and 99.5% for `Basicmath`, `Dijkstra`, `QSort`, `Rijndeal`, `SHA`, and `Stringsearch` , respectively). SDC is the second most common result, with higher occurrences in benchmarks like `Rijndeal` (29.6%) and `SHA` (11.2%) while being much lower in benchmarks such as `Dijkstra`, `QSort`, and `Stringsearch` (4.4%, 2.6%, and 0.3%, respectively). The higher impact of faults on the `Rijndeal` algorithm can indicate that the encryption algorithm strongly depends on intermediate values and that a slight modification in one of these can cause an avalanche effect, which tends to modify the output more if compared to the other algorithms. Crashes/hangs are rare, occurring at low percentages across all benchmarks (0.4% for `Basicmath`, 1.2% for `QSort`, 1% for `Rijndeal`, and even lower for `Stringsearch` at 0.2%).

The prevalence of benign outcomes aligns with expectations. At the same time, the slightly elevated SDC rates for `Rijndeal` and `SHA` may be attributed to the complexity of cryptographic operations, which involve numerous register manipulations. In contrast, the low occurrence of crashes in most benchmarks can be explained by excluding critical registers, such as the PC and Stack Pointer, from the injection campaigns, focusing solely on general-purpose registers.

Memory injection campaigns reflect a similar pattern, with the majority of outcomes being Benign (99.7%, 96.9%, 98.5%, 98.7%, 95%, and 99.9% for `Basicmath`, `Dijkstra`, `QSort`, `Rijndeal`, `SHA`, and `Stringsearch`,

respectively). SDCs occur at notably lower percentages compared to register injections, particularly in benchmarks like `Basicmath` (0.1%), `Dijkstra` (0.7%), `QSort` (0.9%), and `Stringsearch` (0%). However, `SHA` continues to exhibit a relatively higher SDC rate at 3.9%, reflecting its complexity. Crashes/hangs remain uncommon, with most benchmarks showing crash rates below 1%, except for `Dijkstra` (2.4%) and `SHA` (1.1%).

The low occurrence of SDCs and crashes in memory campaigns can be attributed to the larger addressable memory space, which spreads the probability of an injection affecting critical parts of the application, unlike the limited number of CPU registers.

The PC is a register in a computer's CPU that holds the address of the next instruction to be executed in the program's code. It plays a crucial role in controlling the flow of a program by ensuring that instructions are fetched and executed in the correct sequence.

As largely expected, injection campaigns targeting the PC see crashes/hangs dominate (89.1%, 83.9%, and 85.7% for `Dijkstra`, `QSort`, and `SHA`). Bit-flipping in PC causes significant flow modification. Corruption of the less significant byte leads to more heterogeneous outcomes, with SDC (1.9%, 0.6%, and 7.7%) and Benign (9%, 15.5%, and 6.6%). Specifically, corruption of the two least significant bits results in a benign outcome [46]. When corruption occurs in the more significant bits, the resulting address change can be large, causing the program to jump to a distant, often unintended, part of the code. This can lead to major disruptions, such as a crash or a system hang, as the program may start executing nonsensical instructions or access memory areas that lead to faults. On the other hand, when the corruption affects the less significant bits, the change in the address is minor. Since the LSBs correspond to small changes in the PC 's address, flipping them might result in slight deviations from the intended instruction, depending on the instruction avoided. This might cause the program to jump to a nearby instruction, which could still result in an unintended outcome but is less likely to cause severe disruptions.

Based on the SBU injection campaign results, a comparative analysis of the system components reveals their varying susceptibility to faults. The **CPU registers** exhibit high vulnerability to SDC due to their computational complexity and reliance on intermediate values. Crashes and hangs are minimal, likely due to the exclusion of critical registers like the PC and Stack Pointer. Faults in **memory** mostly result in benign outcomes, with even fewer SDCs compared to register injections. This lower susceptibility to critical failures can be attributed to the larger addressable space, which distributes the probability of affecting essential data or instructions. The **PC** is the most susceptible part of the system. Since the PC determines the execution flow, even minor corruptions can significantly alter program behavior, often leading to the execution of unintended instructions or jumping to invalid memory regions. Among these components, the PC is the most vulnerable, as most faults result in severe failures. CPU registers follow, with moderate SDC occurrences in specific benchmarks, while memory remains the least affected, showing resilience due to its distributed nature.

4.2.2 Multi-bit Injection

Figure 6 compares the results of SBUs and MBUs injection campaigns for the six benchmarks (`Basicmath`, `Dijkstra`, `QSort`, `Rijndeal`, `SHA`, `Stringsearch`). The fault injection rate in the MBU experiment is fixed to 3 faulty bits per experiment; the position is completely randomized across all *CPU's registers*. It is worth noting that the overhead of injecting MBU compared to SBUs is negligible. Most of the fault injection overhead arises from the need to set a breakpoint, which halts program execution and allows the fault injector to corrupt specific data. The additional overhead of corrupting multiple bits stems from performing multiple write operations. However, given the low cardinality of multiple faults in a real setup, this overhead remains negligible compared to the overhead introduced by the breakpoint.

The results show the effect of MBUs is far more catastrophic compared to SBUs. In all benchmarks, the multi-bit injections significantly increased both SDC and Crash/Hang rates, highlighting the destructive potential of multi-bit faults, which cause more severe system failures.

4.2.3 Operating System Injection

Fault injection within the OS is conducted using the *multitask* benchmark described in Section 4.1, where multiple tasks are scheduled and executed concurrently. In this scenario, OS execution encompasses various activities, including task memory management, scheduling, time management, and context switching, increasing the likelihood of selecting an OS instruction as the target for the injection breakpoint.

Regarding injection locations, both *CPU registers* and *memory* are targeted. Specifically, injections target all registers utilized by the OS and memory sections storing OS variables, particularly sections `.rodata`, `.data`, and `.bss`.

Injection results indicate that benign behavior dominates both injection locations, accounting for 99.8% in *memory* and 96.9% in *register* injections. Few SDCs were observed

in either scenario, suggesting the system effectively maintained data integrity for scheduled tasks. However, there is a significant difference regarding crash occurrences. The *memory* injections resulted in only 0.2% crashes, whereas *register* injections showed a notably higher crash rate of 3.1%. This indicates that registers are more susceptible to faults that lead to system failures than memory (Fig. 7).

The disparity in crash occurrences can be attributed to the characteristics of the targeted injection locations. The extensive memory space includes initialized and uninitialized global variables and read-only variables. Transient faults injected into uninitialized variables are frequently inconsequential, as these variables are typically overwritten during subsequent execution. Not all variables are actively utilized, as the benchmarks do not fully exercise all available operating system functions. This broader and more diverse memory footprint reduces the likelihood of faults impacting critical structures directly. Conversely, register operations are localized and frequently accessed, making them more vulnerable to faults that can result in system crashes.

4.2.4 Feature Analysis

Previous studies (e.g., [23, 37]) have demonstrated that performance counters, combined with machine learning models, can effectively distinguish between benign and corrupted applications. This insight motivated our work to integrate fault injection and profiling capabilities on an actual hardware board. In particular, previous works highlight that periodic data collection in complex applications enhances detection accuracy. However, it is crucial to consider the unique characteristics of real-time embedded systems [22]. Real-time tasks are typically implemented as periodic tasks, executing the same operations repetitively across multiple periods, with the strict constraint of completing execution within each period. Given this structure, a single execution within a period is generally short enough to allow information collection at the end of each periodic execution. This is a good starting point for performing fault analysis. Our experiments were conducted under conditions that mimic this scenario. That said, in cases where tasks have longer execution times, collecting HPCs with higher granularity is not a significant challenge and would require only minimal modifications to the fault injector.

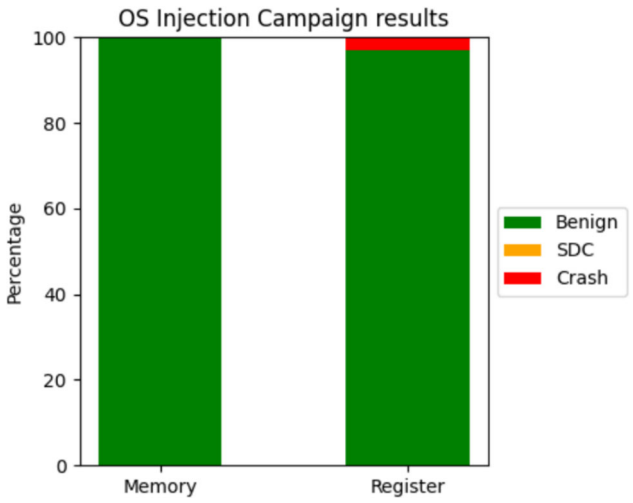
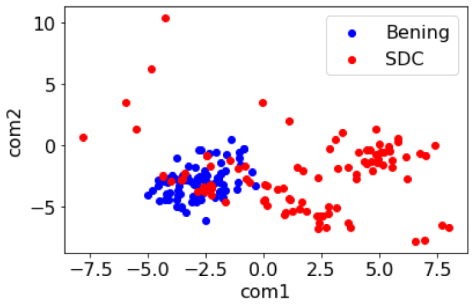


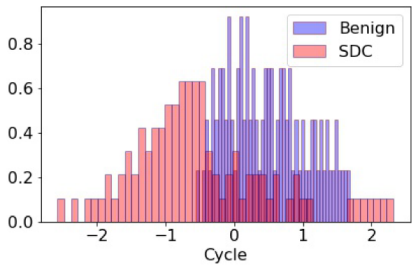
Fig. 7 Breakdown of all injection campaigns done on the OS. Separated by injection location, memory, or CPU register

Figure 8a highlights the profiling capabilities of the proposed fault injector. It reports a scatter plot of the first two principal components of the `QSort` Principal Component Analysis (PCA), including all tracked features with outcomes labeled as Benign or SDC. The PCA is computed using architectural events as features, revealing the distribution of these two categories. The collected data were preprocessed via z-normalization, which normalizes every value in a dataset such that the mean of all values is zero and the standard deviation is 1. Then, Gaussianization is performed to apply transformation so that the data distribution of the transformed data is as Gaussian as possible. Generally, SDCs exhibit greater dispersion with a higher prevalence of outliers than the Benign category. This behavior is expected because bit flipping leading to an SDC likely introduces a nondeterministic behavior, causing a significant alteration of the architectural events. Looking at the Benign class, data are more concentrated, but variations still exist between executions. This observation suggests that soft errors can disrupt temporal constraints even if they do not alter the outcome. Further investigations are required to delve into this analysis. Nonetheless, the developed fault injection is valuable for such studies.

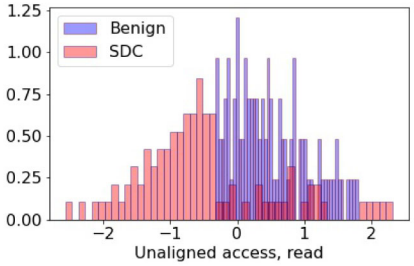
In particular, Fig. 8b and 8c show the distribution of the Cycles (z-normalized and Gaussianized). The histogram plot highlights the differences in the number of cycles between the Benign and the SDCs, as SDCs present a higher deviation, which means a higher time variability during the execution. However, benign distribution also exhibits variability, suggesting faults may introduce time deviations despite the correct final output, breaking the constraint of timing predictability demanded by real-time systems. Generally, benign executions exhibit a certain degree of variability due to several factors, including architectural states and scheduling differences caused by task interactions. However, this variability tends to be minimized in real-time embedded systems, as real-time schedulers prioritize predictability over performance [22]. Despite these inherent variations, our experiments indicate that faults may introduce significant perturbations in the distribution of specific profiled features in several scenarios. This observation is critical for evaluating whether these features can be leveraged for fault detection mechanisms. Among the possible perturbations, delays in



(a) *Scatter plot of the PCA-2 of all the features (architectural events).*



(b) *Histogram plot of the number of cycles distributions.*



(c) *Histogram plot of the number of unaligned access reads.*

Fig. 8 Data visualization of QuickSort CPU registers campaigns dataset for the outcomes Benign and SDC. These figures highlight the different distributions of the two outcomes

execution time are particularly critical in real-time systems, as they may lead to deadline violations, compromising system reliability.

5 Conclusion

This paper extends the fault injector environment from [13], enabling bit-flip injections into embedded hardware via modern CPU debug units. Our configurable framework performs fault injections in CPU registers and memory, using various benchmarks and collecting micro-architectural statistics.

Key advancements include an on-board automated random input generator using FPGA memory to store state information across runs. We also expanded classification features beyond traditional events, allowing for more precise tracking of instructions and registers during fault injections. We also implemented multi-bit fault injection for a broader range of error scenarios, complemented by new benchmarks to evaluate individual processes and the operating system, particularly its scheduler.

Tests on a Xilinx PYNQ Z2 board running FreeRTOS showed that single-bit faults generally caused benign outcomes, though PC faults often led to crashes and hangs. Multi-bit faults were more disruptive, significantly increasing SDC and crash rates. OS faults were mainly benign, with register faults causing more crashes than memory faults. PCA further revealed that SDCs caused greater execution cycle variability than benign outcomes, highlighting our framework's effectiveness in simulating and analyzing fault impacts.

Acknowledgements We thank Dr. Alessio Carpegna for his insightful discussions and technical advice during the initial phases of this project.

Author Contributions Enrico Magliano, Alessandro Savino, and Stefano Di Carlo conceived the presented idea. Enrico Magliano implemented the tool, led the experimental phase, and wrote the paper. Alessandro Savino and Stefano Di Carlo supervised the implementation and experimental results and revised the manuscript.

Funding Open access funding provided by Politecnico di Torino within the CRUI-CARE Agreement. This study was conducted within the "COLTRANE-V" project, funded by the Ministero dell'Università e della Ricerca, under the PRIN 2022 program (D.D.104 - 02/02/2022, GA n.2022HWM3T9). The manuscript reflects only the views and opinions of the authors, and the Ministry cannot be held responsible for them.

Data Availability To support ongoing research in this field, we have made the code related to our experiments available as open-source: https://github.com/smilies-polito/marvin. All data generated and analyzed during this study are included in the manuscript. Additional

datasets or materials used in the research can be made available upon reasonable request from the corresponding author.

Declarations

Competing Interests The authors declare no competing interest.

References

1. Vallero A, Savino A, Chatzidimitriou A, Kaliorakis M, Kooli M, Riera M, Anglada M, Di Natale G, Bosio A, Canal R et al (2018) Syra: Early system reliability analysis for cross-layer soft errors resilience in memory arrays of microprocessor systems. IEEE Trans Comput 68(5):765–783
2. Kritikakou A, Sentieys O, Hubert G, Helen Y, Coulon J-F, Deroux-Dauphin P (2022) Flodam: Cross-layer reliability analysis flow for complex hardware designs. In: Proceedings of the 2022 conference & exhibition on design, automation & test in Europe. DATE '22, pp. 819–824. European Design and Automation Association, Leuven, BEL
3. Kooli M, Natale GD (2014) A survey on simulation-based fault injection tools for complex systems, pp 1–6
4. Portolan M, Savino A, Leveugle R, Di Carlo S, Bosio A, Di Natale G (2019) Alternatives to fault injections for early safety/security evaluations. In: 2019 IEEE European Test Symposium (ETS). IEEE, pp 1–10
5. Benso A, DiCarlo S (2011) The art of fault injection. J Control Eng Appl Inf 13(4):9–18
6. Liang H, Sun H, Sun J, Huang Z, Xu X, Yi M, Ouyang Y, Lu Y, Yan A (2017) Fpga-based soft error sensitivity analysis method for microprocessor 39:245–249
7. Di Carlo S, Gambardella G, Prinetto P, Reichenbach F, Lokstad T, Rafiq G (2014) On enhancing fault injection's capabilities and performances for safety critical systems. In: 2014 17th Euromicro conference on digital system design. IEEE, pp 583–590
8. Di Carlo S, Prinetto P, Rolfo D, Trotta P (2014) A fault injection methodology and infrastructure for fast single event upsets emulation on xilinx sram-based fpgas. In: 2014 IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT), pp 159–164. https://doi.org/10.1109/DFT.2014.6962073
9. Carreira J, Madeira H, Silva J (1998) Xception: A technique for the experimental evaluation of dependability in modern computers. Softw Eng, IEEE Trans on 24:125–136
10. Kanawati GA, Kanawati NA, Abraham JA (1995) Ferrari: A flexible software-based fault and error injection system. IEEE Trans Comput 44(2):248–260
11. Benso A, Bosio A, Di Carlo S, Mariani R (2007) A functional verification based fault injection environment. In: 22nd IEEE international symposium on defect and fault-tolerance in VLSI systems (DFT 2007), pp 114–122 (2007). https://doi.org/10.1109/DFT.2007.31
12. Benso A, Di Carlo S, Di Natale G, Prinetto P, Solcia I, Tagliaferri L (2003) Faust: fault-injection script-based tool. In: 9th IEEE on-line testing symposium, 2003. IOLTS 2003, p 160. https://doi.org/10.1109/OLT.2003.1214386
13. Magliano E, Carpegna A, Savino A, Di Carlo S (2024) A micro architectural events aware real-time embedded system fault injector. In: 2024 IEEE 25th Latin American Test Symposium (LATS), pp 1–6. https://doi.org/10.1109/LATS62223.2024.10534595
14. Barry R (2023) FreeRTOS Real Time Kernel. Real Time Engineers Ltd., Real Time Engineers Ltd. https://www.freertos.org/
15. Wang N, Mahesri A, Patel S (2007) Examining ACE analysis reliability estimates using fault-injection 35:460–469
16. Bodmann P, Papadimitriou G, Rech Junior RL, Gizopoulos D, Rech P (2021) Soft error effects on arm microprocessors: Early estimations vs. chip measurements. IEEE Trans Comput 1–1
17. Mamone D, Bosio A, Savino A, Hamdioui S, Rebaudengo M (2020) On the analysis of real-time operating system reliability in embedded systems. In: 2020 IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT), pp 1–6
18. Barton JH, Czeck EW, Segall ZZ, Siewiorek DP (1990) Fault injection experiments using fiat. IEEE Trans Comput 39(4):575–582
19. Silva D, Stangherlin K, Bolzani L, Vargas F (2011) A hardware-based approach for fault detection in rtos-based embedded systems, p 209
20. Bosio A, Rebaudengo M, Savino A (2022) Reliability assessment of freertos in embedded systems. In: 2022 52nd Annual IEEE/IFIP international conference on dependable systems and networks - supplemental volume (DSN-S), pp 28–30
21. Casseau E, Dobiáš P, Sinnen O, Rodrigues GS, Kastensmidt F, Savino A, Di Carlo S, Rebaudengo M, Bosio A (2021) Special session: Operating systems under test: an overview of the significance of the operating system in the resiliency of the computing continuum. In: 2021 IEEE 39th VLSI Test Symposium (VTS), pp 1–10
22. Buttazzo GC (2011) Hard real-time computing systems: predictable scheduling algorithms and applications. Springer, ???. https://doi.org/10.1007/978-1-4614-0676-1 . https://doi.org/10.1007/978-1-4614-0676-1
23. Kasap D, Carpegna A, Savino A, Di Carlo S (2023) Micro-architectural features as soft-error markers in embedded safety-critical systems: preliminary study. In: 2023 IEEE European Test Symposium (ETS), pp 1–5
24. Bosio A, Di Carlo S, Rebaudengo M, Savino A (2022) Toward the hardening of real-time operating systems. In: 2022 IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT), pp 1–6. https://doi.org/10.1109/DFT56152.2022.9962356
25. Mukherjee S (2011) Architecture design for soft errors. Morgan Kaufmann
26. Di Natale G, Gizopoulos D, Di Carlo S, Bosio A, Canal R (2020) (eds): Cross-layer reliability of computing systems. INST OF ENGIN AND TECH, Place of publication not identified . OCLC: 1191709900. http://public.eblib.com/choice/PublicFullRecord.aspx?p=6341977 Accessed 2020-11-24
27. Mosdorf M, Sosnowski J (2011) Fault injection in embedded systems using gnu debugger. Pomiary Automatyka Kontrola 57:825–827
28. Lu Q, Farahani M, Wei J, Thomas A, Pattabiraman K (2015) Llfi: An intermediate code-level fault injection tool for hardware faults. In: 2015 IEEE international conference on software quality, reliability and security, pp 11–16. https://doi.org/10.1109/QRS.2015.13

29. Kooli M, Di Natale G, Bosio A (2016) Cache-aware reliability evaluation through llvm-based analysis and fault injection. In: 2016 IEEE 22nd international symposium on on-line testing and robust system design (IOLTS), pp 19–22. https://doi.org/10.1109/IOLTS.2016.7604663
30. Chatzidimitriou A, Bodmann P, Papadimitriou G, Gizopoulos D, Rech P (2019) Demystifying soft error assessment strategies on arm cpus: Microarchitectural fault injection vs. neutron beam experiments. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp 26–38. https://doi.org/10.1109/DSN.2019.00018
31. Gava J, Bandeira V, Rosa F, Garibotti R, Reis R, Ost L (2022) Sofia: An automated framework for early soft error assessment, identification, and mitigation. J Syst Arch 131:102710. https://doi.org/10.1016/j.sysarc.2022.102710
32. Kanawati GA, Kanawati NA, Abraham JA (1995) Ferrari: a flexible software-based fault and error injection system. IEEE Trans Comput 44(2):248–260. https://doi.org/10.1109/12.364536
33. Carreira J, Madeira H, Silva JG (1998) Xception: a technique for the experimental evaluation of dependability in modern computers. IEEE Trans Software Eng 24(2):125–136. https://doi.org/10.1109/32.666826
34. Ebrahimi M, Mohammadi A, Ejlali A, Miremadi SG (2014) A fast, flexible, and easy-to-develop fpga-based fault injection technique. Microelectron Reliab 54(5):1000–1008
35. Angione F, Bernardi P, Di Gruttola Giardino N, Appello D, Bertani C, Tancorre V (2023) A guided debugger-based fault injection methodology for assessing functional test programs. In: 2023 IEEE 41st VLSI Test Symposium (VTS), pp 1–7. https://doi.org/10.1109/VTS56346.2023.10140099
36. Hanneman A, Gava J, Bandeira V, Garibotti R, Reis R, Ost L (2023) Debate-fi: A debugger-based fault injector infrastructure for iot soft error reliability assessment. In: 2023 IEEE 9th World Forum on Internet of Things (WF-IoT), pp 1–6. https://doi.org/10.1109/WF-IoT58464.2023.10539573
37. Dutto S, Savino A, Di Carlo S (2021) Exploring deep learning for in-field fault detection in microprocessors. In: 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1456–1459. IEEE
38. Pattabiraman K, Li KG (2022) Fault Injection at the Instruction Set Architecture (ISA) Level, pp 239–260. Chap. Chapter 9. https://doi.org/10.1049/PBCS057E_ch9 . https://digital-library.theiet.org/doi/abs/10.1049/PBCS057E_ch9
39. Yalcin G, Unsal OS, Cristal A, Valero M (2011) Fimsim: A fault injection infrastructure for microarchitectural simulators. In: 2011 IEEE 29th International Conference on Computer Design (ICCD), pp 431–432 .https://doi.org/10.1109/ICCD.2011.6081435
40. Robinson GD (1994) Why 1149.1 (jtag) really works. In: Proceedings of ELECTRO '94, pp. 749–754
41. Xilinx Software Command-Line Tool (XSCT). https://www.xilinx.com/products/design-tools/software-zone/sdsoc/xsct.html. Accessed: November 25, 2023
42. Xilinx PYNQ-Z2. https://www.xilinx.com/products/boards-and-kits/pynq-z2.html. Accessed: November 25, 2023
43. Limited A (2011) ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition. ARM Limited, Cambridge, UK. ARM Limited. Document ID: DDI 0406C.b
44. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB (2001) Mibench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), pp 3–14
45. Leveugle R, Calvez A, Maistri P, Vanhauwaert P (2009) Statistical fault injection: Quantified error and confidence. In: 2009 Design, automation & test in europe conference & exhibition, pp 502–506. https://doi.org/10.1109/DATE.2009.5090716
46. Mukherjee SS, Emer J, Reinhardt SK (2005) The soft error problem: An architectural perspective, pp 243–247

Enrico Magliano is a Ph.D. Student in Artificial Intelligence at the Department of Control and Computer Engineering of Politecnico di Torino. His research focuses on Artificial Intelligence for Trustworthiness of Computing Systems. He is a Computer Engineer with a specialization in Artificial Intelligence

Alessandro Savino is an Associate Professor in the Department of Control and Computer Engineering at Politecnico di Torino (Italy). He holds a Ph.D. (2009) and an M.S. equivalent (2005) in Computer Engineering and Information Technology from the Politecnico di Torino in Italy. Dr. Savino's research contributions include Approximate Computing, Reliability Analysis, Safety-Critical Systems, Software-Based Self-Test, and Image Analysis. He has been part of the program and organizing committee of several IEEE and INSTICC conferences and has served as a reviewer of IEEE conferences and journals. His research interests include Operating Systems, Imaging algorithms, Machine Learning, Evolutionary Algorithms, Graphical User Interface experience, and Audio manipulation.

Stefano Di Carlo received an M.Sc. degree in computer engineering and a Ph.D. in information technologies from Politecnico di Torino, Italy, where he is a Full Professor. His research interests include DFT, BIST, and dependability. He has coordinated several national and European research projects. Di Carlo has published over 200 papers in peer-reviewed IEEE and ACM journals and conferences. He regularly serves on the Organizing and Program Committees of major IEEE and ACM conferences. He is a Golden Core member of the IEEE Computer Society and a senior member of the IEEE. He currently serves on the editorial board of JETTA.