Microprocessors and Microsystems 96 (2023) 104723

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Evaluation of fault injection tools for reliability estimation of microprocessor-based embedded systems

Alexander Aponte-Moreno [a], José Isaza-González [b], Alejandro Serrano-Cases [c], Antonio Martínez-Álvarez [c,*], Sergio Cuenca-Asensi [c], Felipe Restrepo-Calle [a,*]

[a] Department of Systems and Industrial Engineering, Universidad Nacional de Colombia, Bogotá, Colombia
[b] Centro Internacional de Desarrollo Tecnológico y Software Libre. Universidad Tecnológica de Panamá, Panamá City, Panama
[c] Department of Computer Technology. University of Alicante, Carretera San Vicente del Raspeig s/n, Alicante, 03690, Spain

ARTICLE INFO

Keywords:
Fault injection tools
Soft errors
Microprocessors
Radiation effects
Architectural simulation

ABSTRACT

Statistical fault injection is widely used to estimate the reliability of mission-critical microprocessor-based systems when exposed to radiation and to evaluate the performance of fault mitigation strategies. However, further research is needed to gain a better understanding of the accuracy of the results and the feasibility of their application under realistic radiation conditions. In this article, an understanding of scenarios in which Instruction Set Architecture simulators or emulators may be relied upon for realistic statistical fault injection campaigns is advanced. An analysis is presented of the results from two simulation-based fault injection tools versus a set of fault emulation results on a real processor. The conclusions of the analysis assist the selection of the most efficient tool and method for testing many different software-based fault mitigation techniques within reasonable time periods and at affordable costs throughout an irradiation campaign. In particular, it was established that a partially ordered set of relations could be defined on the basis of statistical fault injection in relation to the effects of different versions of an application and a given simulator that remained unaltered during the irradiation experiments. The tests were conducted with a Texas Instruments MSP430 microcontroller to perform both fault injection campaigns and irradiation experiments using neutrons at the Los Alamos Neutron Science Center (LANSCE) Weapons Neutron Research Facility at Los Alamos, USA.

1. Introduction

The technological trend towards the miniaturization of electronic components has not only led to unprecedented microprocessor performance levels, but also to side effects such as increased susceptibility to natural-radiation induced faults. *Single Event Effects* (SEEs) are faults that either cosmic rays and high energy particles present in space or secondary particles generated through atmospheric interactions can provoke in the circuits of computing systems [1]. These faults have been of concern in mission-critical applications working in harsh environments under radiation such as aerospace and nuclear applications. However, current use of nanometric technologies has extended the impact of SEEs to application domains operating at atmospheric and ground levels, such as telecommunications, transportation, and medical applications.

A particular type of SEE, the *Single Event Upset* (SEU) also known as the *soft error* [2] results from the interaction of a particle with the semiconductor substrate, generating an undesired transition in the state of a transistor. Eventually, the visible effect is the change in the data stored in a memory cell or in a *flip-flop* [3]. In microprocessor-based systems, those soft errors are expressed as faulty program execution results or system hang issues. Despite the fact that these faults may be temporary and inflict no physical damage on the devices, they are currently considered as one of the main challenges to be solved in modern electronics [4]. Thus, the tolerance of the circuits (i.e.,: their functional capability to operate in the presence of these sorts of faults) is an important research topic and a required feature of any system used in mission-critical applications [3].

One of the most widespread methods for assessing the reliability of the systems at the early design stages is statistical fault injection. It consists of the deliberate introduction of faults during the functional operation of the system, so as to observe their effects, and to estimate the error rate and different reliability metrics [5]. Fault injection approaches are usually classified according to their physical or logical nature [6]. Physical approaches are based on the use of external sources

* Corresponding authors.
E-mail addresses: jaapontem@unal.edu.co (A. Aponte-Moreno), jose.isaza@utp.ac.pa (J. Isaza-González), aserrano@dtic.ua.es (A. Serrano-Cases), amartinez@dtic.ua.es (A. Martínez-Álvarez), sergio@dtic.ua.es (S. Cuenca-Asensi), ferestrepoca@unal.edu.co (F. Restrepo-Calle).

https://doi.org/10.1016/j.micpro.2022.104723
Received 30 November 2021; Received in revised form 10 October 2022; Accepted 8 November 2022
Available online 12 November 2022

to induce faults within the system that is tested, such as either particle accelerators or pulsed laser [7]. Although this option is preferred for provoking realistic radiation-induced faults, it is highly costly and depends on special facilities that are within the reach of very few researchers. Furthermore, the experiments cannot be conducted until the final system is available. In consequence, logical fault injection has therefore gained attention, as it yields early reliability assessment, even during the design phases, at lower costs. It includes 2 main strategies: emulation-based fault injection, in which the real system is either emulated, usually with an FPGA, or the faults are emulated on real devices [8]; and simulation-based fault injection, where a model of the system to be tested is used, and faults are produced during its simulation [9,10]. These tools offer different levels of accuracy (bit accurate, cycle accurate) and observability depending on the processor model that is employed: ISA (Instruction Set Architecture) model, RTL (Register Transfer Level) model, etc. Moreover, some fault injection tools are based on hybrid approaches, combining both hardware and software components [11].

Simulators are typically used both to estimate the reliability of mission-critical systems at an early stage of development and to evaluate the effectiveness of different fault mitigation strategies. However, when using either simulation or emulation, faults are always injected in a limited set of resources. In fact, the injection of faults is only possible in resources that are exposed to the user, typically the Instruction Set Architecture (ISA), which is the set of instructions, registers, flags, and addressing modes that define a microprocessor and its internal accessible structures. Simplified error models, such as single bit flip, are usually adopted, due to time constraints. As a consequence, non-ISA resources, such as the pipelining registers are always inaccessible. Fault injection results must be validated or tuned with radiation tests at all times to ensure precise results [12].

Ascertaining the accuracy of the simulated results with respect to the emulated and the real experiments is therefore very necessary. ISA simulators are focused on the performance and the functionality of the applications running on the processor without taking into account any other consideration (e.g., unexpected behaviors, collateral effects of errors, etc...). However, simulation usually leads to inaccurate estimations of system reliability when simulators are used in statistical fault injection campaigns. In this respect, a quantitative evaluation of soft-error injection techniques, which proved that ISA level fault-injection can be non-accurate, was presented in [13].

In this work, we set out to highlight the importance of understanding the limitations of the simulators, in order to produce consistent estimations of the actual reliability levels. As case studies, we have analyzed two simulators and propose to use emulation on real devices to understand and eventually to surpass their limitations. Our approach was assessed through comparisons of the fault injection estimations with the radiation test results.

Thus, the results of three fault-injection tools are tested in this work to analyze their applicability to realistic radiation scenarios: the first two (i.e., MiFIT [14] and Naken-FIM [15]) are ISA-level simulators, whilst the third one, FIM, is an emulation-based facility. The comparative assessment, conducted with a Texas Instruments MSP430 microcontroller, performed both fault injection simulated and emulated campaigns and irradiation experiments using real radiation at the Los Alamos Neutron Science Center (LANSCE) Weapons Neutron Research Facility at Los Alamos, USA. The main contribution of this article is twofold: 1. to obtain the limits of the tools when obtaining real estimations of the error rates by adjusting the setup. 2. to propose guidelines to, under certain conditions, extrapolate the effects of simulation/emulation results to those of radiation.

Preliminary results on this topic were published in [16] and the present article extends our previous work in several ways: firstly, a more in-depth description of the background and the state-of-the-art is presented; secondly, extensive experiments have led to new insights related to specific scenarios when the faults directly affect critical components; thirdly, the study has been extended with irradiation campaigns, during which the accuracy of simulated results was analyzed for assessing the effectiveness of software mitigation techniques.

The rest of the article is organized as follows. In Section 2, relevant literature is reviewed. The fault injection tools selected for the experimental evaluation are then introduced in Section 3, and the experimental setup is described. A comprehensive set of experimental results is then presented in Section 5. Subsequently, the overall results and some specific fault occurrence scenarios are discussed in Section 6. The comparative analysis of the various irradiation experiments is related in Section 7. The article is concluded in Section 8 and some future lines of research are also outlined.

2. Related works

Fault injection is a commonly used experimental technique to assess the dependability of microprocessor-based systems. In particular, simulation-based fault injection makes use of a software program to model both the target system and the faults. The injection of faults can be performed by modifying either the state of the hardware components (e.g., flip-flops), or the state of the architectural resources (e.g., register file), or the state of the software structures (e.g., variables). In any event, it presents several advantages regarding physical and emulated methods. First, the controlability of simulation models is greater and their fault propagation patterns may be traced through the hardware and software layers. Second, they demand less effort in terms of development time and economic costs. Finally, no special resources nor facilities are required, which increases their availability and flexibility during the experiments. However, simulation-based fault injection is not always possible, because of the unavailability of models and/or the lack of information on the micro-architectural details required for accurate simulations. Moreover, in comparison with emulation-based techniques, simulators usually require extra computing time when developing extensive fault injection campaigns.

Over past decades, simulation-based fault injection has proliferated, targeting different levels of abstraction and fault models and relying on multiple toolkits. According to [17], they can be divided into generalists and specialists. Generalist tools provide a common framework either to support different simulator back-ends or to facilitate portability between simulators and real hardware. Within this category, GOOFI [18] was conceived as a generic architecture that facilitates the adoption of new system targets and new fault injection techniques. A second version was introduced in [19], extending the capabilities to support real hardware through Nexus-compliant test ports. Using the same debug port, Fidalgo et al. [20] proposed a tool for real-time fault injection through built-in debug circuitry included in real processors. David and Campbell [21] adopted a similar approach, although in their case they used the GDB debugger interface. A more ambitious effort was the FAIL* framework [22], which provided an abstraction layer for different simulators, virtualization tools, and hardware back-ends (ARM, x86). The fault injection campaigns were implemented with a C++ API that offers access to both the target back-end meta-information, and the current state. In contrast, the specialists tools are tightly coupled to a single target or simulator. Examples include the extensions of QEMU [23], which is a virtual machine capable of running complete operating systems, such as those described in [24,25]. Another example is GeFIN, a fault injection framework built on top of the Gem5 micro-architectural simulator for ARM processors [26].

In addition, fault emulation on real devices is gaining attention as a complementary method to simulation-based approaches. In some cases, such as in high-performance computing, it represents an effective and reliable method of speeding up testing in multi-threaded applications [27,28]. In other cases, when either low-level simulation models are unavailable (e.g., Commercial off-the-shelf (COTS) processors) or micro-architectural details are not disclosed, due to industrial secrecy (e.g., GPUs), then fault emulation is the only way to arrive at realistic

results. Following this approach, several works have proposed the use of on-chip debugging infrastructures to produce faults on real devices in real time [29,30]. Other proposals use the compilation process to instrument the code transparently and to produce faults on processors [28] and NVIDIA GPU devices [31].

Different approaches have been adopted for the evaluation of fault injection methods. In [32], the authors compared physical methods (heavy-ion radiation, pin level injection, and electromagnetic interference) with software-based techniques on a Motorola 68 070 processor. They concluded that the single bit-flip model is capable of generating a similar set of errors as the physical techniques, except for those caused in the data segment. Chatzidimitriou et al. [12] analyzed micro-architectural fault injection versus neutron beam experiments. They used the Gem5 cycle-accurate simulator for extensive fault injection campaigns, observing that, in general, the number of failures over time underestimates the beam results. Recent works, such as [33], compared the effect of faults injected at different abstraction levels: IR level (Intermediate Representation level) versus assembly code level. This study confirmed that both were of similar accuracy for silent data corruption faults, although the IR-level was less accurate with respect to crashes when aggressive compiler optimizations were applied.

3. Fault injection tools

Three fault-injection tools were tested in this work: The first two (MiFIT and Naken-FIM) were ISA-level simulators, while the third, FIM, was an emulation-based facility. MiFIT [14] is a modular, open source fault injection tool for microprocessors which is supported by a standard interface that is usually available in most modern simulators and real devices. Naken-FIM is a specialized tool derived from an extension of Naken [15], an open source MSP processor simulator. Their results are compared to those obtained on a real platform where runtime faults are emulated using the FIM framework [34] using the real device as the target. An MSP430 processor was selected to perform the fault campaigns. This processor is often used within the scientific community to test radiation effects on processor-based systems for early analysis of reliability and fault tolerance properties [35,36]. The selected tools and their main features are described below.

3.1. Naken-FIM

Naken is an open source instruction-accurate simulator for a number of ISAs including MSP430, ARM64, MIPS, and RISC-V [15]. We modified the original simulator to enhance several analysis features (Naken-FIM): Firstly, selective fault injection capabilities. More precisely, we can select any particular memory section from an executable file (e.g., `.data`, `.bss`, `.stack`, `.text`, etc.) as a fault target; any set of memory regions or single data (e.g., a subset of `.data`); and any set of registers within the register file, or any global variable. Secondly, enriched simulation/debugging trace capabilities can be used to collect information on accurate read/write accesses, fault labeling, etc... By gathering and post-processing this information, we can focus on the effects of a fault campaign over a certain resource and study its impact on program fault tolerance levels. A facet inherited from the original tool also makes it possible to simulate either an Executable and Linkable Format (ELF) file or an Intel HEXadecimal object file format (HEX) file. This characteristic is actually quite interesting, because using a compiler-agnostic output format, such as Intel HEX, makes it possible to interoperate with several compilers and hardening tools.

3.2. MiFIT

The Microprocessors Fault Injection Tool (MiFIT) is a modular fault injection tool [14]. Its source code is publicly available at: https://github.com/UNPLaS/MiFIT.

MiFIT simulates faults in specific and general-purpose registers; currently, this tool does not support the simulation of faults in the microprocessor's main memory. At the configuration stage, the MiFIT module offers a selection whereby the injection campaign can be performed on a single register or on the complete register file. At this stage, users can also set the number of injections to be performed and the place where the result is stored (register file or RAM). Moreover, in this case, we used the debugging and programming tool *mspdebug* in the injection interface to simulate faults in the MSP430 microcontroller. The injection interface used the mspdebug 'step' command to control the injection time, instead of controlling with a hardware time-triggered interrupt, which bypasses the need to instrument the source code. The first step of the injection process was a golden execution to store the expected result of the program under evaluation. Then, the injections decided upon at the configuration stage were performed, injecting a single fault in each execution. The results of the fault injection campaigns were classified, according to their effects on the system, and they were stored in CSV (Comma Separated Value) files for later analysis.

3.3. FIM

The Fault Injection Manager (FIM) is a highly portable fault injection tool for different (ISA) processor architectures and emulation/simulation platforms [34], suitable for this research where a real device was needed to undertake a fault campaign. It uses the built-in hardware debugging facilities, such as *On-Chip Debugging* (OCD), and the GNU Debugger (GDB) to access internal processor resources (such as memory sections, register file, etc.) and for monitoring the execution process. This setup provides support for many processor implementations (*softcore* and *hardcore*) and *Commercial-off-the-shelf* (COTS) devices.

FIM also uses a lightweight Interrupt Service Routine added to the original code and driven by a built-in hardware timer to speed up the fault injection on real COTS processors. During the initialization phase, a clock cycle is randomly selected and the timer is configured. A GDB debugging session is then initiated and the program that is launched in test execution mode continues its execution routine until interrupted by the timer. Then GDB takes control to inject a fault on a random resource and to resume the execution. Once the program finishes, FIM reads the result and classifies the faults according to the effect on the system.

3.4. Tools comparison

Table 1 presents a summary of the main characteristics of the fault injection tools. As comparative criteria, we considered the following: type of fault injection performed (emulation/simulation), fault model, ability to inject faults into different processor resources, speed of the fault injection campaign, availability of the source code in public repositories, and possibility of adapting the tool to different architectures.

It is worth mentioning that Naken-FIM and MiFIT present two important limitations: (1) they are incapable of simulating several hardware functionalities present in real devices, such as hardware multiplier accelerators, timers and communications interfaces (e.g., UART); (2) memory protection is not implemented, which means that faults affecting the program counter in such a way that an invalid memory address that is accessed may prompt a different behavior compared to the real MSP430. In contrast, this limitation is not applicable to FIM that works on the real device, although its campaigns are much lengthier over time than that of the simulation tools, due to the direct (not simulated) interaction with the real device. There is, therefore, an inherent trade-off when comparing Naken-FIM and MiFIT versus FIM between execution accuracy and its duration. Nevertheless, we can select pure ISA programs without access to interrupts or accelerators/co-processors

Table 1
Comparative table of selected fault injection tools.

Tool	Fault implementation	Fault model	Fault targets	Campaign speed	Public availability	Adaptable to other architectures
MiFIT	Simulation	Bitflip	Reg.File	Fast (≈minutes)	Yes	Yes
Naken-FIM	Simulation	Bitflip	Reg.File/Mem.	Very fast (≈minutes)	On demand	Yes
FIM	Emulation on real processor	Bitflip	Reg.File/Mem.	Slow (≈hours)	No	Yes

to minimize this tradeoff in favor of simulation strategies. Regarding the campaign duration, we selected the built-in gdb simulator on FIM, because it is a de-facto standard, however it is difficult to modify, because of the lack of documentation. Consequently, the fault injection process has to be performed using the gdb standard interface, which is a slow procedure. In the case of Naken-FIM the fault injection infrastructures are integrated in the simulator, leading to faster campaigns. Furthermore, any required modifications to mimic the real processor behavior can be accomplished more easily. Another remarkable difference is that, unlike the simulation options, FIM instruments the code that is to be executed, which can be problematic when the program is too small compared to the ISR.

4. Experimental setup

4.1. Device under test

The two main tools, MiFIT and Naken, used in this study were chosen to simulate the Texas Instruments MSP430 microcontroller (the MSP430G2553 device) whereas the FIM tool was adapted to emulate faults on the Launchpad prototyping board.

The core of the TI-MSP430 is a 16-bit RISC processor included in the Texas Instruments low-power microcontroller family. It includes a register file with 16 registers (R0–R15). The first 4 are special-purpose registers: R0 is the Program Counter (PC); R1 is the Stack Pointer (SP); R2 is the Status Register (SR); and the R3 register is used for constant generation. The Status Register (R2) stores the content of arithmetic flags (carry, overflow, negative, and zero), and some control bits such as System Clock Generator 1 (SCG1), System Clock Generator 0 (SCG0), Oscillator (OffOSCOFF), and CPU Off (CPUOFF) that are used to control the operational mode of the CPU. The General Interrupt Enable (GIE) bit is used to enable or to disable maskable interrupts. The remaining registers, R4 to R15, are general purpose registers.

Fig. 1 presents the memory map of the different memory sections of the TI-MSP430. The memory section distribution is important to gain an understanding of the fault injection campaigns, as it has different memory sections as its targets, such as: *.data*, *.stack* and *.text*.

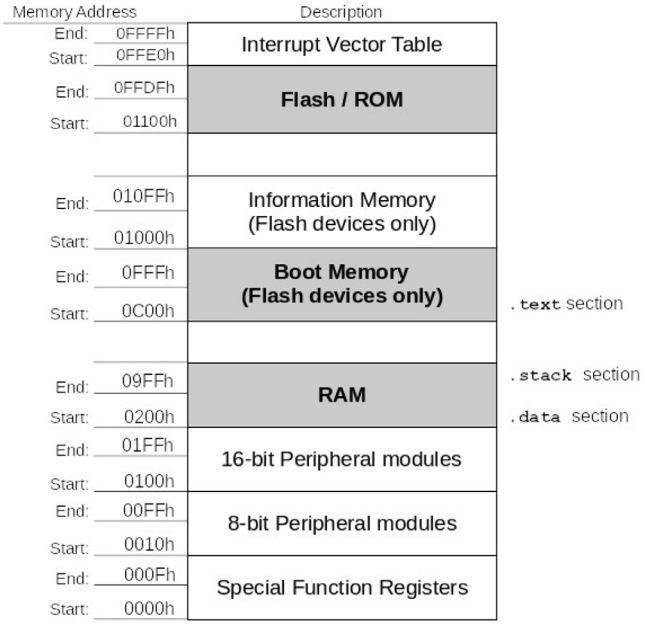


Fig. 1. TI-MSP430G2553 memory map [37].

4.2. Benchmarks and campaign configurations

The benchmark suite used in the experiments included applications of different levels of complexity and different computation models (data intensive vs. control intensive). These are: Euler algorithm (Euler), an iterative code for calculating the Euler's number, the recursive Quick Sort algorithm (Qsort), and the data intensive Matrix Multiplication (MxM). A Cyclic Redundancy Check (CRC) was added to the last 2 test programs to reduce the output data and to facilitate the evaluation of the results.

The programs were compiled using the msp430-gcc compiler version 6.4.0.32. The same binary files were tested on each tool during fault injection campaigns that used the bit-flip fault model and that injected a single fault *per* execution in a randomly selected bit and clock cycle (instruction in case of the simulators). The targets of the faults were: the registers of the register file and three memory sections — RAM data (*.data*), stack memory (*.stack*), and program memory (*.text*). A total of 1000 faults were injected in each register and in each memory section, numbering 16,000 faults in the register file and 3000 in the memory for each program. Memory campaigns were performed using only Naken and FIM, as memory injection was not supported in MiFIT at the time the experiments were performed. In all cases, the error margin was 1% with a confidence level of 99%, according to [38].

In addition, we performed a detailed analysis of the special processor registers that were affected, e.g., PC, SP, SR (including the ALU flags), and their effects. The specialized function of these registers makes them an important source of errors and it is therefore necessary to understand the possible consequences of these faults, and at the same time, the differences that can be expected between fault simulation and fault emulation in these specific cases.

Depending on the final result of the program execution the faults were classified as 'unACE', 'SDC' or 'Hang', according to [39]. If the system completed its execution and its results were in the expected output, the injected fault was classified as unnecessary for Architecturally Correct Execution (unACE). Faults which were not detected/corrected and that provoked the program to complete its execution with an erroneous output were labeled as Silent Data Corruption (SDC). Finally, faults that caused an infinite execution loop or an abnormal program termination were classified as Hang. SDC and Hang were categorized together as Architecturally Correct Execution (ACE) faults.

5. Fault injection results

5.1. Register file reliability analysis

One fault injection campaign was performed for each of the three test programs and each of the three tools, i.e., a total of nine campaigns. Fig. 2 presents the average percentages of the fault classification for all registers in the microprocessor register file. One can see a group of 3 bars for each program. Fault classification is shown in stacked bars with three categories: unACE, Hangs, and SDC.

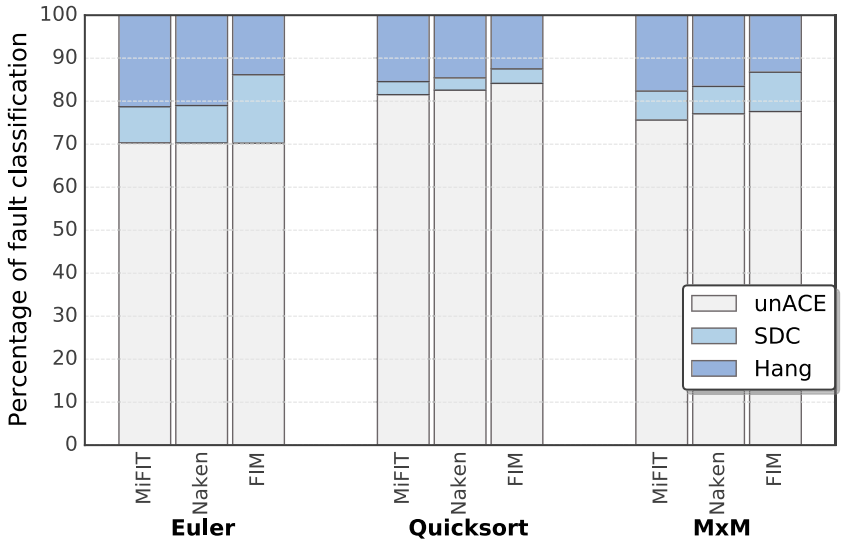


Fig. 2. Average percentages of fault classification in the register file by fault injection tool (MiFIT, Naken and FIM) and test program.

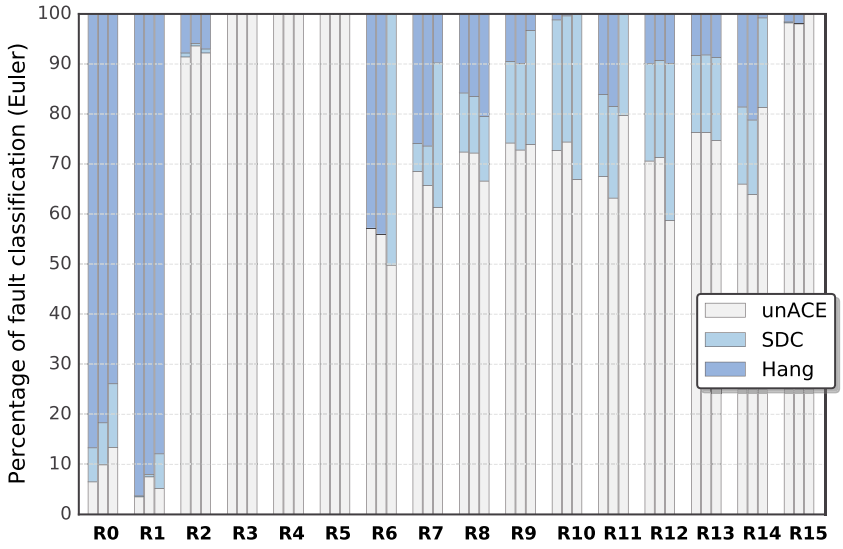


Fig. 3. Percentages of fault classification in the register file by fault injection tool (MiFIT, Naken, and FIM) for Euler.

As can be seen, the results of the different tools are generally similar, especially when comparing the results obtained with the 2 simulation-based tools. In all cases, the differences in the unACE percentages between the tools are less than 2.8%. It should be noted, however, that the distribution of undesirable effects (SDC and Hangs) shows a variation that is slightly higher when comparing the results from simulation-based tools (MiFIT and Naken) to those from the emulation tool (FIM). For example, this difference reaches 7.5% in Euler and is an aspect that will be discussed in greater detail in Section 6.

Detailed results can be seen in Fig. 3. This figure, instead of showing the aggregated result for the entire register file, depicts the results of the fault injection for each register individually for Euler. Within the figure, 16 groups of stacked bars can be seen, which correspond to the 16 registers of the processor register file. For each group, the first, second, and third stacked bars show the results with MiFIT, Naken, and FIM, respectively. Similar results were found for Quicksort and MxM, however, these results are not shown for the sake of brevity.

Fig. 3 shows differences in results, particularly between the results of the simulation tools and the emulation tool. It is important to clarify that the faults only affect registers involved in program instructions. Note that faults injected into unused registers (e.g., R3, R4 and R5 in Euler) have no consequence on the program result and are therefore considered as unACE. In contrast, critical registers, such as R0 (program counter) and R1 (stack pointer), presented the highest error rates.

Moreover, there were some differences between the injection results obtained from the three tools. The registers where some differences might be noted can be identified from this detailed representation, comparing the results of each register independently. For instance, the R6 register in Euler was one instance where the registers presented inconsistencies between simulation and emulation results. In this example, injecting a fault in this register in some specific instructions of the program workload caused the program to require 1 additional instruction to complete its execution: i.e., the assembly code fragment presented in Fig. 4. If a fault is injected in the R6 register, the simulation-based tools will abandon program execution within the expected time, which is classified as a Hang. In contrast, the same fault in the emulation tool is considered an SDC which is due to the differences between the characteristics of the real processor used by the emulation tool and those of the models used in the simulations. While the real processor considers the entire underlying microarchitecture, the simulated models are at the ISA level.

This fact shows a limitation of the specific ISA-level simulation models for the MSP430 used by the simulation tools. Although this

```
0xc26a: TST R6
0xc26c: JZ 0xc270
0xc26e: MOV R7, R10
0xc270: MOV R10, R12
```

Fig. 4. Assembly code extract from the Euler program.

could be improved by an enhanced implementation of the simulation model (e.g., considering microarchitectural features of the processor), it is a general drawback of simulation models that should be considered when analyzing results.

Considering the aforementioned analysis, a set of new fault injection campaigns were performed with an extra-time limit for 10 instructions to the simulators, before the classification of the faults as either SDC or Hangs. The test programs therefore have a soft time limit to finish their executions. Fig. 5 presents the average fault injection results by tool and test program when an extra-time limit was given to the simulators for the completion of the execution of the program before the fault classification. These results were more consistent between the simulators and the emulation tool than the results of the previous campaigns shown in Fig. 2 (without any extra time for the simulators).

We have compared the three means for the unACE and SDC results for each experiment (Euler, Quicksort and MxM) and each tool (MiFIT, Naken, and FIM) using an ANOVA statistical test. We have calculated the next p-values in the ANOVA test for the pair (unACE, SDC): Euler (0.915, 0.553), Quicksort (0.813, 0.962) and MxM (0.908, 0.987), respectively. They are all bigger than the significance value 0.05. Therefore, the null hypothesis is not rejected and we can state that there are no significant differences between the results obtained by MiFIT, Naken and FIM.

Furthermore, Fig. 6 illustrates, for the case of Euler, the fault classification in each individual register per test program in greater detail where an extra-time limit for the simulation tools was added before the classification.

The results showed that adding extra time significantly reduced the differences between simulation and emulation, e.g., in the aforementioned case of the R6 register in Euler, similar results were recorded from each of the three tools. However, some differences remain, especially in the results of the PC (R0) and SP (R1), and a few general-purpose registers, such as R11 in MxM and Qsort, which will be discussed in detail in Section 6.

5.2. Memory reliability analysis

Fig. 7 shows the percentage fault classifications of the faults injected in the campaigns on the memory sections (*.data*, *.stack*, and *.text*) using Naken and FIM. Please, note that fault simulation in memory sections is currently unavailable in MiFIT.

Highly similar results for the *.data* sections in the Euler and Matrix Multiplication test programs (100% unACE faults) were recorded, whereas the QuickSort test program showed a slight discrepancy below 7%. In the case of the *.stack* and *.text* memory sections, the results showed discrepancies in the undesirable effects (SDC and Hang faults) obtained with both injectors on the test programs. Moreover, the results of injection into the *.stack* for Euler and Matrix Multiplication showed differences of 31% and 37%, respectively; while the results of the QuickSort test program showed a discrepancy of less than 12% for these faults. With respect to the *.text* section, the results showed the following discrepancies: 28% for the Euler program, 30% for the Matrix Multiplication, and 22% for the QuickSort program. These differences in the results are discussed in greater detail in the next section.

6. Discussion

6.1. Comparing results from simulation-based tools

Fig. 8 shows the comparison of injection results obtained with both the Naken and the MiFIT tools. The percentage difference of the fault classification for each simulator is shown through bars for the Euler, Qsort, and MxM test programs. The fault effects were classified as unACE (lighter colored bars) and Hang (darker bars).

The horizontal axis shows the microprocessor registers, while the vertical axis represents the percentage difference in fault classification. Positive values are used for cases where the MiFIT fault classification is higher than Naken. Negative values indicate the opposite.

As can be seen from all three figures, the results obtained with both simulators are consistent. In all cases, the difference in fault classification was less than $\pm 7.3\%$. The highest differences were observed in the special purpose registers, specifically in registers R0 and R1. As in the case of Quicksort, where the most significant difference corresponds to UnACE for the R0 register, with 7.3%. For general-purpose registers, the observable differences were even less and were all below $\pm 5.1\%$.

Although the results of the previous comparison were calculated on the basis of the classifications presented in Fig. 3, these differences remained similar and were consistent with the results shown in Fig. 6. The differences between simulation-based tools can therefore be said to have remained constant, whether or not an extra-time limit was considered in the fault simulation campaigns before fault classification.

6.2. Simulation vs. emulation fault injection results

As mentioned in Section 5.1, the fault classification results obtained with the three injection tools presented slightly larger discrepancies for some cases, as shown in Fig. 6, due to the additional time that the simulation tools needed before fault classification. Fig. 9 presents the differences between the results of the simulators (MiFIT and Naken), versus the emulation results obtained using FIM. Figs. 9(a) to 9(e) illustrate the differences between MiFIT and FIM. In the same way, the other 3 sub-figures, Figs. 9(b) to 9(f), represent the comparison between Naken and FIM in terms of their results. The percentage differences with positive values indicated that the classification of the simulation tool was greater than the FIM, and *vice versa*.

Some registers recorded considerable fault classification differences under either simulation or emulation. These discrepancies amounted to as much as 20.0% for unACE faults for register 11 in qsort, as shown in sub Fig. 9(d); up to 19.9% for Hang faults shown in the same subfigure and register; and up to 11.3% for SDC faults in register 12 in Euler, as shown in sub Fig. 9(b).

After having analyzed the assembly code of the test programs and having performed simulation tests, it became clear that these discrepancies were due to classification errors between unACE and Hangs, and between unACE and SDC. The errors resulted from variations in the design of the fault injection campaigns. The simulation-based campaigns included all program subroutines, including initialization subroutines, as part of the fault injection target. Nevertheless, the initialization subroutines were given no consideration in the emulator-based campaigns. The fault injection in some registers used in these initialization subroutines, such as R11 and R14, therefore yielded different results. This sort of register was employed before the main routine to move data on the RAM for the test programs. By injecting a fault in these registers, data can actually be written to sections intended for the stack, which can cause an erroneous return of the procedure. These faults have been experimentally validated, using exactly the same fault injection campaign for all tools, without injecting faults during initialization subroutines. The results showed that the differences were considerably reduced between the results obtained in the simulation and emulation campaigns. In the worst case, the largest difference found was 10%, which related to register R14 in M×M.

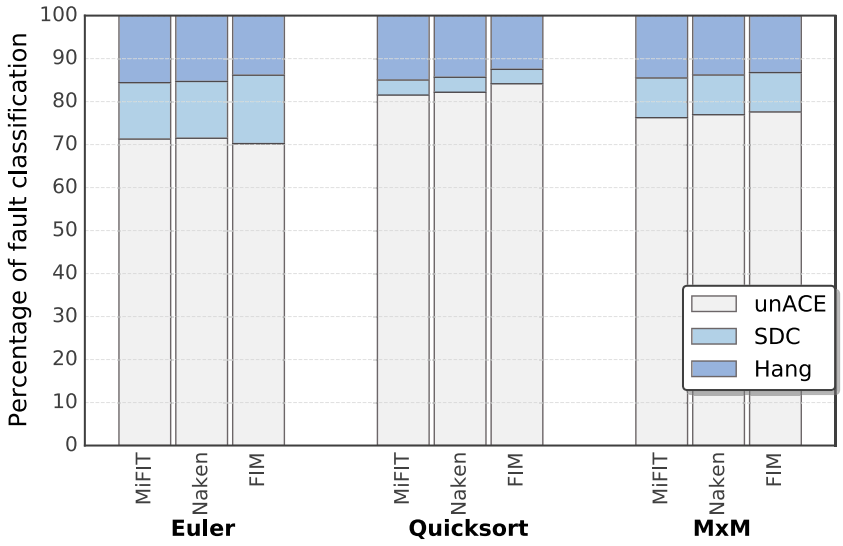


Fig. 5. Average fault injection results by fault injection tool (MiFIT, Naken, and FIM) and test, with an extra-time limit for the simulators.

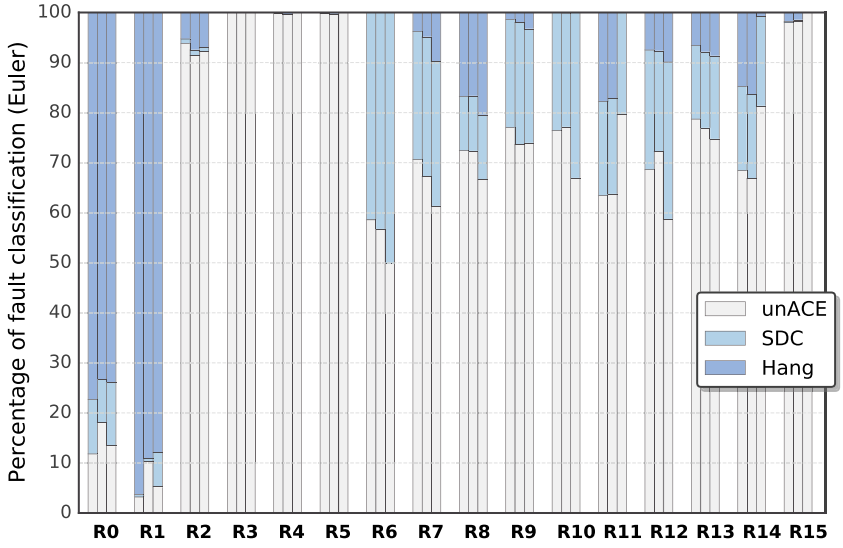


Fig. 6. Fault classification percentages in the registry file by fault injection tool (MiFIT, Naken, and FIM) for Euler, with an extra-time limit for the simulators.

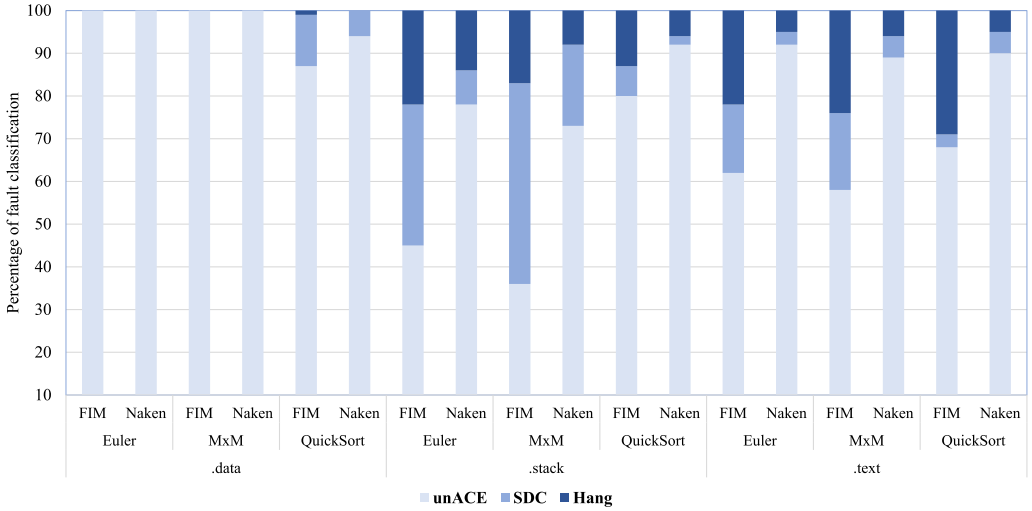
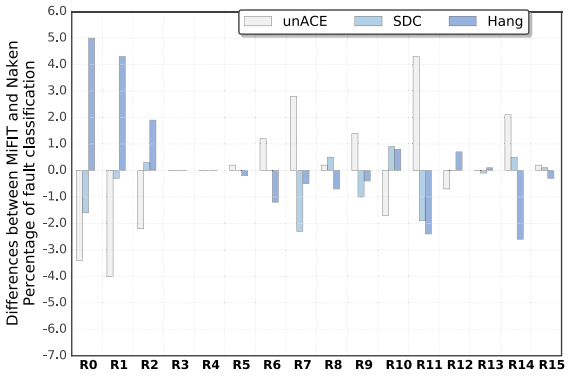
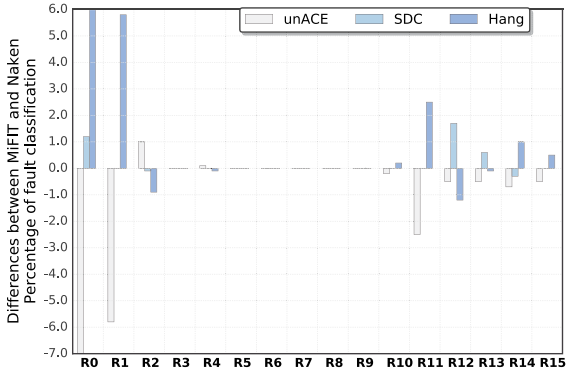


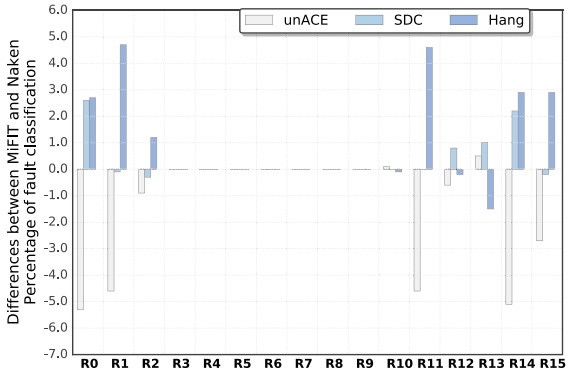
Fig. 7. Fault classification percentages for memory section and test program.



(a) Euler



(b) Qsort



(c) MxM

Fig. 8. Percentage differences between fault injection results from simulation tools: MiFIT and Naken. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

In addition, it is worth mentioning that the majority of the remaining differences between the results, both from the simulation-based fault injection campaigns and from the emulation-based campaigns, were due to limitations of the simulation models for the MSP430 used by the simulation-based tools. In general, the ISA-level simulation models should be used for a preliminary assessment of the reliability of the system during development. Simulation models are expected to take into account the microarchitecture of the processor and its clock cycle accuracy, to improve the precision of the results.

6.3. Memory fault injection

The results of the fault injection campaigns on the different memory sections showed that the effects of the faults varied according to the section of memory under consideration for the injection (as can be seen in Section 5.2). In the case of the data section (*.data*), it contains the explicitly initialized global and static variables of the programs. The size of the values in the program source code determine the size of this section, which remains unchanged at run time. The source code has read–write permissions, so the variable values located in this segment can be changed at run time. In this section, the effects of the faults show a similar behavior in both the Naken simulator and the FIM injector applied to the real processor.

The memory section of the *.stack* is located in a higher group of addresses (as shown in Fig. 1), and grows or shrinks in relation to the stack segment. It contains local function variables and related accounting data. During the call to a function, a stack frame is created, containing the arguments of the local variables of the function and the return value (each function has a stack frame). The experimental results showed discrepancies between the undesirable effects (SDC and Hang faults) of both injectors during the test programs, due to the limitations of the simulator when trying to model the behavior of the real processor.

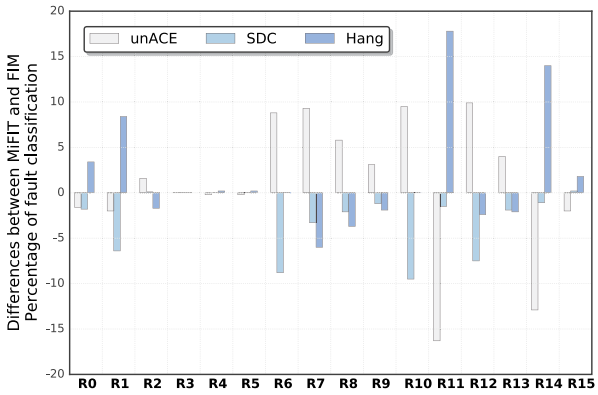
As mentioned above, the *.stack* section contains the program stack, which is used to store the return address when a function is called and the parameters that are passed to the function. In the same way, it is also used to store the intermediate results of the program. In this context, faults injected with FIM, rather than with Naken, can cause the real processor to behave in different ways. For example, in a case where the fault sends the Program Counter (PC) to a memory address in the non-implemented memory section, it can prompt an exception causing a reset, an interruption, or an infinite loop of program execution (see Section 6.4 Specific scenarios/cases). As Naken does not have this exception implemented, it will decode and execute trash code until it reaches the initial memory address (*_start*), causing the system to restart the program execution without faults (unACE fault). Therefore, the behavior of the simulator will differ from the real processor depending on the trash code in the *Flash* section of the processor. It is also expected that there will be a deviation in the results.

The *.text* section, also known as a code segment, contains the machine instructions of the program. In other words, this memory section stores the program code. It is a read-only section that prevents a program from being accidentally modified. The effect of a fault on this section can affect the control flow of the program execution by changing the memory content, for example, in a jump or call to a runtime function. In the same way as the previous case, the simulator will not have implemented some characteristics of the real processor, so a variation in the results obtained between both injectors may also be expected.

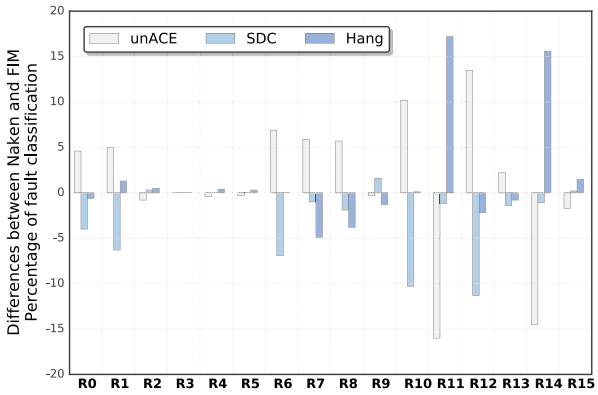
6.4. Specific scenarios/cases

The effects of faults affecting non-general purpose registers used by each test program are studied in this sub-section (Fig. 6). As stated in Section 4.1, we always have a number of specialized registers whose function makes them a major source of errors (control-flow errors, data corruptions, etc.). In addition to the function-specific registers, some general-purpose registers can also be used as pointers to memory locations where an intermediate or final result is stored. Therefore, the effect of a fault on this kind of registers has to be taken into consideration jointly.

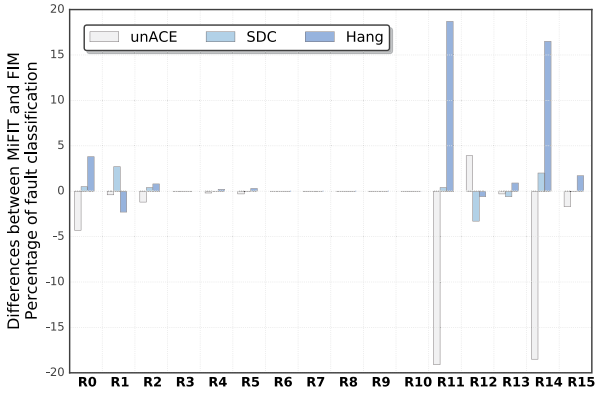
For example, a fault may cause the PC (R0) to decrease its value and, consequently, to induce the re-execution of some instructions. In this case, the best disruptive effect is the obvious increase in execution time. In fact, in the case where it does not exceed the timing requirements of the program, the fault is labeled as unACE. In this case, both Naken and MiFIT should generate a fault classification identical to that of FIM. Similarly, if the fault increases the PC value, both simulator outputs will match the FIM output. In this case, some instructions are bypassed which will make SDC or Hang labeling more likely. Moreover, the PC



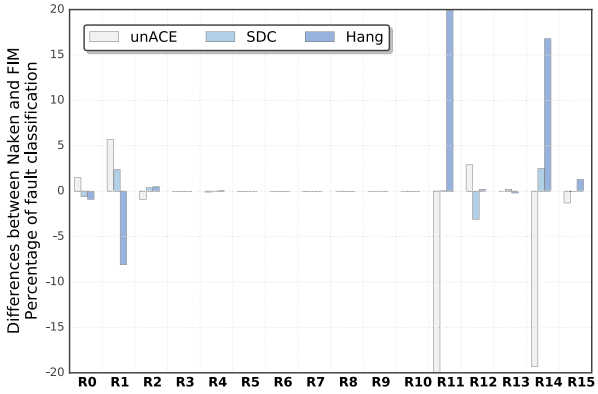
(a) MiFIT vs. FIM - Euler



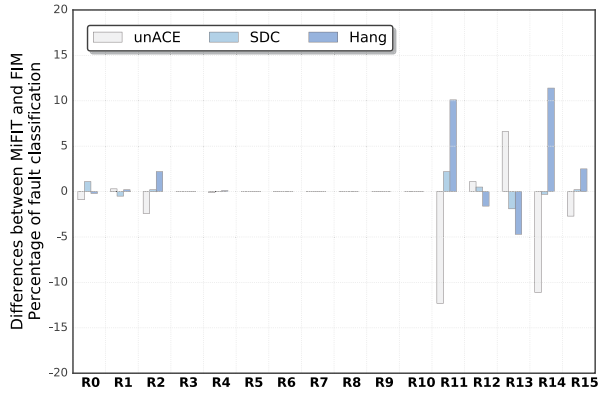
(b) Naken vs. FIM - Euler



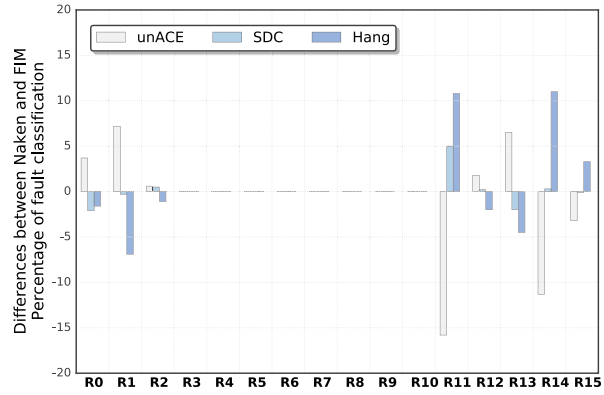
(c) MiFIT vs. FIM - Qsort



(d) Naken vs. FIM - Qsort



(e) MiFIT vs. FIM - MxM



(f) Naken vs. FIM - MxM

Fig. 9. Percentage difference between fault injection campaign results based on simulation and emulation (with an extra-time limit).

could also be modified in such a way that the processor has to fetch an instruction from whatever memory location from the addressable map (see Fig. 1 for reference to the memory map), which will result in different effect types depending on the affected resource:

- *ROM/flash section*: re-execution of the program from a random address or trash code execution until the initial memory address (*_start*), from which the program is loaded again, is reached. The program will end without errors, but depending on the timing requirements, the fault will be labeled as either Hang or unACE. In this case, output from simulators and actual devices my differ, depending on the trash code found in this section.
- *Peripheral module sections*, *special registers* and *memory not implemented*: in case the processor will attempt to fetch an instruction in any of these sections, an exception of *reset* will be triggered, and will therefore cause the program to be restarted without faults. For example, in the case of MSP430, if an instruction is fetched from the peripheral configuration registers, the device will trigger a *reset* exception, which may cause Hang or unACE faults. The simulators under study will implement no exceptions, so code will execute normally until *_start* is reached, as previously stated.
- *RAM section*: this memory section contains data (data sections, the stack and the heap) with no code instructions. If the PC points here, the instruction that is fetched can result in an illegal instruction (and consequently, an exception), which is more likely, or a legal decodable trash instruction.
- *Interrupt vector table section*: in the case of MSP430, an exception of *SIGTRAP_reset_vector_()* arises. This routine points the PC to the initial address of the program and stops its execution. In

consequence, it restarts the execution of the program. Hence, in many cases this event causes Hang faults. In turn, in the case of MiFIT and Naken, this exception is not incorporated, so the behavior will differ from the FIM-related behavior.

In addition, in case the fault points the PC to an address located after the code section, a trash code will be executed up to the *Interrupt vector table*, where the exception *SIGTRAP_reset_vector_()* is triggered, provoking either Hang or unACE faults. As both, MiFIT and Naken implement no exceptions, programs will continue running trash code until the end of the memory, which will cause Hang faults.

Moreover, a fault affecting the Stack Pointer (SP or R1), is very likely to affect stacked/automatic variables, which will generally lead to SDC faults. It can also affect the reading of the return address after the execution of a function call; in this case, the fault can make the PC return either to a wrong instruction (SDC) or to any of the previous possibilities, invariably yielding *Hang* faults.

Regarding the R2 register (or State Register), the effect of faults shows high percentages of unACE faults in all test programs (Figs. 3 and 6). Faults on this register can produce two main effects:

1. If faults affect the control bits of the clocks, they can deactivate the operation mode of the program (low power mode). However, only the CPUOFF bit produces this event, so its effect should slightly affect the result. In the case of the FIM injector, this effect may result in Hang faults in small proportions. Similarly, in the case of Naken and MiFIT, this effect produces no major consequences, because the simulators have yet to implement these microarchitectural features.
2. If the faults affect the *flags* (C, Z, N and V) of the ALU (Arithmetic Logic Unit), it may cause incorrect jumps. An effect that may therefore lead to SDC and Hang faults.

Finally, a fault affecting the remaining general-purpose registers can produce several events. For example, a bit-flip in a register used immediately before a conditional jump causes the program to execute a different number of instructions prior to termination. For example, in the source code from Fig. 4, a bit-flit on R6 will affect the program in the aforementioned way. If the number of instructions needed to complete the program increases, and no extra-time limit has been specified within the injection tool, the fault will be labeled as Hang. On the other hand, if the number of instructions is reduced, the injection may cause an SDC type of fault.

Having completed a detailed analysis of how faults introduced in different computational resources can affect the variability of simulation and emulation results, we have all the information and variables to assert when it is plausible to rely on Instruction Set Architecture simulators for estimating the reliability of a system. With no loss of generality, we can divide embedded systems software into 3 categories: pure data-flow, control-oriented, and hybrid systems. Pure data-flow systems always perform the same operations on a set of input data. The operations on the data are purely logical/arithmetic, and can be defined with the original ISA, i.e., no peripherals, co-processor units or accelerators of any kind are used. For example: CRC32, FIR filters, matrix multiplication, etc. Moreover, control-oriented systems use conditions to select between different tasks that need to be executed. These conditions can be calculated from either the input data (option A), or they can come from another external source to the microprocessor (option B) (peripheral, interrupt, etc.). Interactive systems and ISR-based software are good examples of such systems. Finally, hybrid systems integrate both possibilities in the same software. When assessing reliability, pure data-flow systems will yield the same results under both simulation and emulation, as argued before. Therefore, simulators represent the best option in terms of accuracy and speed. Option A control-oriented software is indistinguishable from pure data-flow when assessing its reliability using either emulation or simulation, so simulators will again be the best option here. Option B control-oriented and hybrid software will produce different output on emulation or simulation fault campaigns, as previously mentioned. Some software products permit the simulation (which must be hard-coded) of external events that have to be accessed (e.g., MSP430 matrix multiplier accelerator, timer events, etc.). In these cases, we must minimally instrument the code, to improve the reliability evaluation and to minimize the differences between emulation and simulation. In the most extreme cases (e.g., ISR-based software), emulation should be recommended to obtain reliable results. Finally, for designers of fault mitigation algorithms, whose algorithms are often highly parameterizable, and therefore subject to much variability in their effects, simulation is recommended as a key tool for fine-tuning their proposals.

7. Simulation & emulation *vs.* real irradiation campaigns on mitigation techniques

While simulation or emulation campaigns produce valuable results to show the effects of faults over different software/hardware structures (e.g., code and data sections, registry files, addressable memory, etc.), these effects must be validated in real irradiation campaigns. In this context, we define simulation/emulation results as valid, if the fault weights for each fault type (namely SDC or unACE) keep the rank order consistency in the corresponding irradiation experiment. In that way, if a mitigation technique reduces the total SDC-labeled faults of a program in either simulation or emulation, this effect should be also projected in the irradiation results. In consequence, this partial order relation can be leveraged to speed-up the search of the most appropriate mitigation technique for a given software: what goes wrong in simulation/emulation will also go wrong with real irradiation, and *vice-versa*.

We selected the well-known BubbleSort algorithm (a sorting algorithm that swaps contiguous elements of a vector until the vector is shortened) to demonstrate the above. Although the BubbleSort algorithm is indeed both basic and simple, we use it in fault injection campaigns, because we can delimit its effects on the executing device: fine control of resources used (registers, memory), easy fault localization, scalability, duration control, etc. Simple but well-known benchmarks such as BubbleSort, MxM, CRC, Qsort, AES etc. are commonly used in the fault tolerance community to provide evidence of the goodness of their hardening techniques [35,36,40]. In addition, in our case, BubbleSort provides computation of relatively low complexity and of a relatively long duration. Thus, it is easier to trace and to explain its fault effects. Moreover, testing more complex applications introduces no additional benefits, however, it makes it extremely difficult or unfeasible to run their much slower associated campaigns. The BubbleSort algorithm was subjected to the following three techniques:

- **T1-MOOGA**: a *genetic algorithm* controlled by a *multi-objective optimization algorithm* that is a non-intrusive technique for automatically testing thousands of compiler flags/parameters [41]. The final goal is to find the best set of compilation alternatives to improve features such as code size, computing time, and fault coverage altogether. We selected a possible solution for optimizing the above-mentioned features.
- **T2-SHE**: a source-to-source compiler (SHE) that is used as a method to inject redundant code (assembly) using the S-SWIFT-R technique [42]. The compiler has the capability of producing selective register hardening, in order to reduce the time overheads of replica computation. In this case, the whole registry file was protected.
- **T3-HData**: a high-level technique that uses a template-based C++ hardening library, named HData [43], to add Triple Modular Redundancy (TMR) to the variables manually as they are selected.

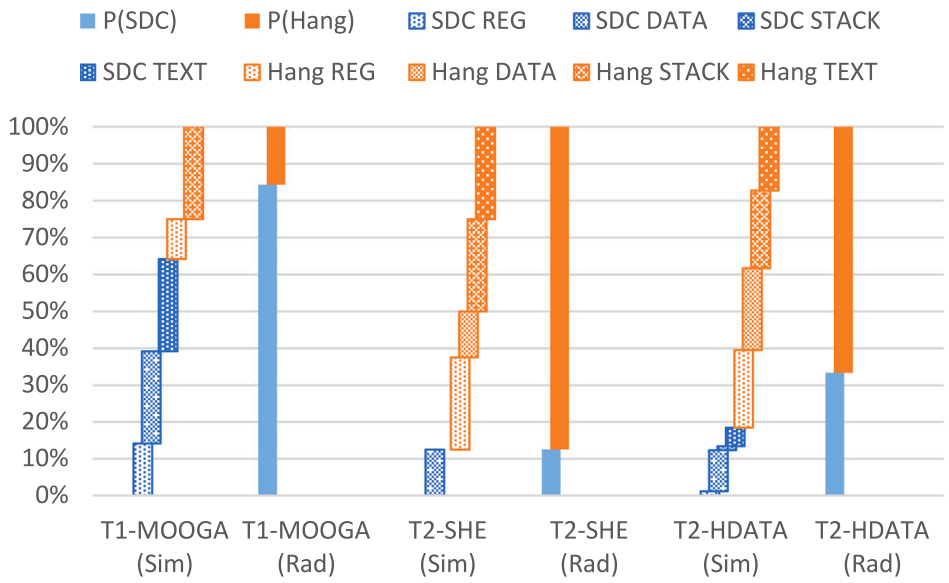


Fig. 10. Observed Architecturally Correct Execution (ACE) radiation events (P_{SDC} and P_{Hang}) vs. simulation events. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

An MSP430F5529 Device-under-Test (DUT) was used that was also part of the MSP430 family. It shares the same architecture of the previous devices studied by simulation/emulation. It is built in 130 nm, works at 25 MHz, and is equipped with 128 KiB and 8 KiB of flash and RAM memory, respectively.

The Naken tool was used to perform the corresponding fault simulation campaigns. Finally, an irradiation campaign using neutrons was undertaken to compare the results.

The neutron campaign was performed at the Los Alamos Neutron Science Center (LANSCE). The experiments were carried out at the Weapons Neutron Research Facility (WNR), using Target 4 Flight Path 30L (ICE I). The LANSCE dosimetry data yielded a constant neutron flux of $1.7 \cdot 10^5$ n/(s cm^2), above 10 MeV. Taking into account the times to complete each run, the total fluence was calculated with an accuracy of 10%. With this configuration, the shape of the neutron spectrum was very similar to the one produced in the atmosphere by cosmic rays.

Simulation and radiation results for the three aforementioned techniques are shown in Fig. 10. Given the fact that it is not possible to trace each fault event in radiation experiments (only those that produce soft errors can be reliably traced), the figure only represents normalized ACE events, in each case. Events labeled as SDC and as Hang are represented in blue and orange, respectively. While in radiation a soft error can be only labeled as SDC or Hang, because the microprocessor resource that was hit cannot be ascertained, we have more control over simulation errors. In effect, in the simulation experiments we distinguish between faults affecting the register file (Reg), the initialized program data (Data), the program stack (Stack), and the program code itself (Text).

In this case, we observed that, as expected, the ratio between the total amount of SDC and Hang effects under simulation in each case matched the corresponding radiation results. For example, the first technique (MOOGA), summarized 65% of total SDC events, and consequently 35% of Hang events, so it follows that SDC events were of greater frequency and importance than Hang in simulation. The irradiation results for this case, >80% SDC and <20%, support this conclusion. The second technique (SHE) even yielded exact matches for the SDC and Hang values. Conversely, Hang events implied the most important simulation and irradiation-related effects. The last technique was also validated by radiation. In this last case, Hang events also represented the most common soft error with a difference of less than 15% between simulation and irradiation.

We were able to conclude that the order relation maintained its behavior both in simulation and in radiation experiments; which validates our assumption. Moreover, Naken, MiFIT, and FIM can be used to conduct valuable and reliable simulations to investigate the effects of fault campaigns over critical applications. The main benefits range from speeding up the search for new mitigation algorithms with highly selective targeting (register file, memory, etc.), to having a method that will quickly obtain candidates for irradiation.

8. Conclusions

In this work, we have presented experimental insights relating to the accuracy of fault injection tools, to estimate the reliability of microprocessor-based systems. In the first place, two architectural level simulation-based tools have been assessed and compared to the emulation of faults in the real device. The results of the simulation campaigns were consistent across both tools. In all cases, the differences in the fault classification, in each individual register from the microprocessor register file, were less than ±7.3%. When comparing these overall results for the complete register file (as an average), the difference was reduced to only ±3.3%. Moreover, simulation tools offer a downward estimate, about 10%, of the reliability when compared to more realistic results obtained on the real device by fault emulation. Several specific scenarios were discussed, to better understand the reasons for those discrepancies, highlighting the relevance of the low-level architectural features. We can conclude that simulation-based fault injection campaigns are comparable and consistent with fault emulation campaigns using the real processor, but considering several restrictions. These restrictions are mainly due to the various features of the micro-architectures that are not supported by ISA simulators and that are important when studying the behavior of processors in the presence of faults, e.g., hardware interruptions, built-in peripherals (timers, watchdogs, GPIO), and several runtime exceptions (memory not implemented, unauthorized accesses, among others). A very valuable secondary outcome of the experimental results for the fault tolerance community has to do with the investigation of new fault mitigation techniques. In fact, since fault coverage has similar effects in both simulation and emulation (same trend in device resources involved), we can choose any of the three proposed tools to pre-evaluate different versions of a given hardening technique on the same device and thus improve its effectiveness. In fact, we would recommend Naken-FIM or MiFIT to perform a design space exploration of a parameterized

hardening technique, due to their inherent campaign speed compared to FIM. We have completed a detailed analysis of fault injections affecting different computational resources and their effects on the variability of simulation and emulation results. Some recommendations have also been provided on when it is thought appropriate to use either simulation or fault emulation campaigns to assess microprocessor reliability under certain conditions.

In second place, irradiation experiments with neutrons were performed on real devices running different versions of one benchmark hardened with three mitigation techniques: high-level code replication, low-level and non-intrusive (without including redundancy). We observed that reasonable consistency was preserved between simulation and radiation results when evaluating the effectiveness of the mitigation techniques. This conclusion is important, as fault injection offers valuable information to reduce both the time and the cost of searching for the most suitable protection-related configuration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgment

This work has been funded by the Spanish Ministry of Science and Innovation, Spain under the project PID2019-106455GB-C22.

References

[1] T. Karnik, P. Hazucha, J. Patel, Characterization of soft errors caused by single event upsets in CMOS processes, IEEE Trans. Dependable Secure Comput. 1 (2) (2004) 128–143, http://dx.doi.org/10.1109/TDSC.2004.14.

[2] Q. Huang, J. Jiang, An overview of radiation effects on electronic devices under severe accident conditions in NPPs, rad-hardened design techniques and simulation tools, Prog. Nucl. Energy 114 (November 2018) (2019) 105–120, http://dx.doi.org/10.1016/j.pnucene.2019.02.008.

[3] M. Nicolaidis (Ed.), Soft Error in Modern Electronic System, Vol. 41, in: Frontiers in Electronic Testing, Springer US, 2011.

[4] ECSS, Techniques for Radiation Effects Mitigation in ASICs and FPGAs Handbook (1 September 2016) | European Cooperation for Space Standardization, ESA Requirements and Standards Division, 2016.

[5] R. Natella, D. Cotroneo, H.S. Madeira, Assessing dependability with software fault injection, ACM Comput. Surv. 48 (3) (2016) 1–55, http://dx.doi.org/10.1145/2841425.

[6] M. Kooli, G. Di Natale, A survey on simulation-based fault injection tools for complex systems, in: 9th IEEE Int Conf on Design and Tech of Integrated Systems in Nanoscale Era DTIS, 2014, pp. 1–6, http://dx.doi.org/10.1109/DTIS.2014.6850649.

[7] H. Quinn, D. Black, W. Robinson, S. Buchner, Fault simulation and emulation tools to augment radiation-hardness assurance testing, IEEE Trans. Nuclear Sci. 60 (3) (2013) 2119–2142, http://dx.doi.org/10.1109/TNS.2013.2259503.

[8] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, C. Lopez-Ongil, Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection, IEEE Trans. Comput. 61 (3) (2012) 313–322.

[9] S. Wang, G. Duan, On the characterization and optimization of system-level vulnerability for instruction caches in embedded processors, Microprocess. Microsyst. 39 (8) (2015) 686–692, http://dx.doi.org/10.1016/j.micpro.2015.09.011, URL https://www.sciencedirect.com/science/article/pii/S0141933115001532.

[10] Z. Mohseni, P. Reviriego, Reliability characterization and activity analysis of lowRISC internal modules against single event upsets using fault injection and RTL simulation, Microprocess. Microsyst. 71 (2019) 102871, http://dx.doi.org/10.1016/j.micpro.2019.102871, URL https://www.sciencedirect.com/science/article/pii/S0141933118305477.

[11] H. Ziade, R. Ayoubi, R. Velazco, A survey on fault injection techniques, Int. Arab. J. Inf. Technol. 1 (2) (2004) 171–186, https://hal.archives-ouvertes.fr/hal-00105562.

[12] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, P. Rech, Demystifying soft error assessment strategies on ARM CPUs: Microarchitectural fault injection vs. Neutron beam experiments, in: Proc. 49th IEEE/IFIP Int Conf on Dependable Systems and Networks, DSN 2019, IEEE, 2019, pp. 26–38, http://dx.doi.org/10.1109/DSN.2019.00018.

[13] H. Cho, S. Mirkhani, C.-Y. Cher, J.A. Abraham, S. Mitra, Quantitative evaluation of soft error injection techniques for robust system design, in: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), 2013, pp. 1–10.

[14] A. Aponte-Moreno, F. Restrepo-Calle, C. Pedraza, MiFIT: A fault injection tool to validate the reliability of microprocessors, in: 2019 IEEE 20th Latin-American Test Symp LATS, IEEE, 2019, pp. 8–12.

[15] M. Kohn, http://www.mikekohn.net/micro/naken_asm.php, 2018.

[16] A. Aponte-Moreno, J. Isaza-González, A. Serrano-Cases, A. Martínez-Álvarez, S. Cuenca-Asensi, F. Restrepo-Calle, An experimental comparison of fault injection tools for microprocessor-based systems, in: 2020 IEEE Latin-American Test Symposium (LATS), 2020, pp. 1–6.

[17] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, O. Spinczyk, Fail *: Towards a versatile fault-injection experiment framework, in: ARCS Workshops, ARCS 2012, 2012, pp. 1–5.

[18] J. Aidemark, J. Vinter, P. Folkesson, J. Karlsson, GOOFI: generic object-oriented fault injection tool, in: 2001 Int Conference on Dependable Systems and Networks, 2001, pp. 83–88, http://dx.doi.org/10.1109/DSN.2001.941394.

[19] D. Skarin, R. Barbosa, J. Karlsson, GOOFI-2: A tool for experimental dependability assessment, in: 2010 IEEE/IFIP Int Conf on Dependable Systems Networks (DSN), 2010, pp. 557–562, http://dx.doi.org/10.1109/DSN.2010.5544265.

[20] A. Fidalgo, M. Gericota, G. Alves, J. Ferreira, Using NEXUS compliant debuggers for real time fault injection on microprocessors, in: Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design, SBCCI '06, Association for Computing Machinery, New York, NY, USA, 2006, pp. 214–219, http://dx.doi.org/10.1145/1150343.1150397.

[21] F.M. David, R.H. Campbell, Building a self-healing operating system, in: Third IEEE Int Symposium on Dependable, Autonomic and Secure Computing (DASC 2007), 2007, pp. 3–10, http://dx.doi.org/10.1109/DASC.2007.22.

[22] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, O. Spinczyk, FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance, in: 2015 11th European Dependable Computing Conference (EDCC), 2015, pp. 245–255, http://dx.doi.org/10.1109/EDCC.2015.28.

[23] F. Bellard, QEMU, a fast and portable dynamic translator, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, USENIX Association, USA, 2005, p. 41.

[24] F. de Aguiar Geissler, F.L. Kastensmidt, J.E.P. Souza, Soft error injection methodology based on QEMU software platform, in: 2014 15th Latin American Test Workshop, 2014, pp. 1–5, http://dx.doi.org/10.1109/LATW.2014.6841910.

[25] D. Ferraretto, G. Pravadelli, Efficient fault injection in QEMU, in: 2015 16th Latin-American Test Symposium (LATS), 2015, pp. 1–6, http://dx.doi.org/10.1109/LATW.2015.7102401.

[26] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, D. Gizopoulos, Differential fault injection on microarchitectural simulators, in: 2015 IEEE International Symposium on Workload Characterization, 2015, pp. 172–182, http://dx.doi.org/10.1109/IISWC.2015.28.

[27] G. Georgakoudis, I. Laguna, H. Vandierendonck, D.S. Nikolopoulos, M. Schulz, SAFIRE: Scalable and accurate fault injection for parallel multithreaded applications, in: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 890–899, http://dx.doi.org/10.1109/IPDPS.2019.00097.

[28] J. Wei, A. Thomas, G. Li, K. Pattabiraman, Quantifying the accuracy of high-level fault injection techniques for hardware faults, in: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2014, pp. 375–382, http://dx.doi.org/10.1109/DSN.2014.2.

[29] A.V. Fidalgo, M.G. Gericota, G.R. Alves, J.M. Ferreira, Real-time fault injection using enhanced on-chip debug infrastructures, Microprocess. Microsyst. 35 (4) (2011) 441–452, http://dx.doi.org/10.1016/j.micpro.2010.10.002, URL http://www.sciencedirect.com/science/article/pii/S0141933110000724.

[30] M. Portela-Garcia, M. Grosso, M. Gallardo-Campos, M. Sonza Reorda, L. Entrena, M. Garcia-Valderas, C. Lopez-Ongil, On the use of embedded debug features for permanent and transient fault resilience in microprocessors, Microprocess. Microsyst. 36 (5) (2012) 334–343, http://dx.doi.org/10.1016/j.micpro.2012.02.013, URL http://www.sciencedirect.com/science/article/pii/S0141933112000300.

[31] S.K.S. Hari, T. Tsai, M. Stephenson, S.W. Keckler, J. Emer, SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation, in: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2017, pp. 249–258, http://dx.doi.org/10.1109/ISPASS.2017.7975296.

[32] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, G.H. Leber, Comparison of physical and software-implemented fault injection techniques, IEEE Trans. Comput. 52 (9) (2003) 1115–1133.

[33] L. Palazzi, G. Li, B. Fang, K. Pattabiraman, A tale of two injectors: End-to-end comparison of IR-level and assembly-level fault injection, in: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), 2019, pp. 151–162, http://dx.doi.org/10.1109/ISSRE.2019.00024.

[34] J. Isaza-Gonzalez, A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, A. Martinez-Alvarez, Dependability evaluation of COTS microprocessors via on-chip debugging facilities, in: LATS 2016 - 17th IEEE Latin-American Test Symposium, 2016, pp. 27–32, http://dx.doi.org/10.1109/LATW.2016.7483335.

[35] H. Quinn, T. Fairbanks, J.L. Tripp, G. Duran, B. Lopez, Single-event effects in low-cost, low-power microprocessors, in: 2014 IEEE Radiation Effects Data Workshop (REDW), IEEE, 2014, pp. 1–9, http://dx.doi.org/10.1109/redw.2014.7004596.

[36] M. Bohman, B. James, M.J. Wirthlin, H. Quinn, J. Goeders, Microcontroller compiler-assisted software fault tolerance, IEEE Trans. Nuclear Sci. 66 (1) (2019) 223–232, http://dx.doi.org/10.1109/tns.2018.2886094.

[37] Texas Instruments, MSP430 ultra-low-power MCUs, 2018, URL http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html.

[38] R. Leveugle, A. Calvez, P. Maistri, P. Vanhauwaert, Statistical fault injection: Quantified error and confidence, in: 2009 Design, Automation & Test in Europe Conf & Exhibition, IEEE, 2009, pp. 502–506, http://dx.doi.org/10.1109/DATE.2009.5090716.

[39] S.S. Mukherjee, C.T. Weaver, J. Emer, S.K. Reinhardt, T. Austin, Measuring architectural vulnerability factors, IEEE Micro. 23 (6) (2003) 70–75, http://dx.doi.org/10.1109/MM.2003.1261389.

[40] H. Quinn, W.H. Robinson, P. Rech, M. Aguirre, A. Barnard, M. Desogus, L. Entrena, M. Garcia-Valderas, S.M. Guertin, D. Kaeli, F.L. Kastensmidt, B.T. Kiddie, A. Sanchez-Clemente, M.S. Reorda, L. Sterpone, M. Wirthlin, Using benchmarks for radiation testing of microprocessors and FPGAs, IEEE Trans. Nucl. Sci. 62 (6) (2015) 2547–2554, http://dx.doi.org/10.1109/TNS.2015.2498313.

[41] A. Serrano-Cases, Y. Morilla, P. Martin-Holgado, S. Cuenca-Asensi, A. Martinez-Alvarez, Nonintrusive automatic compiler-guided reliability improvement of embedded applications under proton irradiation, IEEE Trans. Nuclear Sci. 66 (7) (2019) 1500–1509, http://dx.doi.org/10.1109/tns.2019.2912323.

[42] A. Martínez-Álvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F.R. Palomo, H. Guzmán-Miranda, M.A. Aguirre, Compiler-directed soft error mitigation for embedded systems, IEEE Trans. Dependable Secure Comput. 9 (2) (2012) 159–172, http://dx.doi.org/10.1109/TDSC.2011.54.

[43] L.M. Reyneri, A. Serrano-Cases, Y. Morilla, S. Cuenca-Asensi, A. Martínez-Álvarez, A compact model to evaluate the effects of high level C++ code hardening in radiation environments, Electronics 8 (6–653) (2019) 1–13, http://dx.doi.org/10.3390/electronics8060653.

Alexander Aponte-Moreno received his Electronic Engineer degree from Universidad Santo Tomás, Bogotá Colombia in 2005. In 2011 he received his M.Sc. degree at the University of Los Andes, Colombia. He is currently pursuing his Ph.D. in Systems Engineering at the Universidad Nacional de Colombia. Since 2014, he has been with the faculty of Telecommunications Engineering at Santo Tomás University as a professor. His research interests include fault tolerance, embedded systems, and approximate computing.

José Isaza-González received a degree in Computer Engineering in 2006 at the Universidad Tecnológica de Panamá (Panamá) and I obtained the Ph.D. in Computer Science in 2018 at University of Alicante (Spain). My main research topics are Internet of Things, Fault Tolerance, Source Code Analysis, Reliability Testing, Embedded Systems and their applications to Aerospace and the Earth's atmosphere systems. Especially interested in the study of the effects of radiation on integrated circuits. Currently employed at Centro Internacional de Desarrollo Tecnológico y Software Libre - AIP as a researcher.

Alejandro Serrano-Cases received the PhD degree from the University of Alicante in 2020. From 2016-2017 he was working at the Department of Computer Technology (University of Alicante, Spain) as a research engineer. He received the B.S. degree in Computer Engineering in 2015 from University of Alicante, Spain. He is currently working as research engineer at BSC (Barcelona Supercomputing Center).

Antonio Martínez-Álvarez was born in Granada, Spain in 1976. He received the M.S. and Ph.D. degree in Electronics Engineering from the University of Granada, Spain, in 2002 and 2006, respectively. From 2002–2006 he worked in the Department of Computer Architecture and Technology at the University of Granada, Spain. He is currently an Associate Professor with the Department of Computer Technology, University of Alicante, Spain. His main research interests deal with methods and tools for dependable design of digital integrated circuits and FPGAs, embedded systems, soft error mitigation in embedded systems, radiation effects, and fault tolerance. He is also interested in neuroengineering and neuroprosthesis devices.

Sergio A. Cuenca-Asensi is a Full Professor in the Computer Technology Department at University of Alicante, Spain. He received the B.S. degree in electronic physics in 1990 from University of Granada, Spain. He received a Ph. D. in Computer Engineering from the University Miguel Hernández of Elche, Spain in 2002. His current research interests are reconfigurable computing, hardware/software co-design, and soft error mitigation in embedded systems.

Felipe Restrepo-Calle received the Ph.D. degree cum laude from the University of Alicante (Spain) in 2011. He worked as a postdoctoral research fellow at the University of Seville (Spain) in 2012 and 2013. Since August 2014, he has been with the Department of Systems and Industrial Engineering at the Universidad Nacional de Colombia (Bogotá) where he is an Associate professor. His fields of research interests include programming languages, and methods and tools for dependable design in embedded systems.