Laser and Radiation Testing of Compiler-based Protection for Multi-Bit Upsets

Davide Baroffio*, Tomas Antonio López*, Federico Reghenzani, William Fornaciari
Department of Electronics, Information and Bioengineering
Politecnico di Milano, Milan, Italy
name[middlename].surname@polimi.it

***Abstract*—Software-Implemented Hardware Fault Tolerance (SIHFT) is advantageous in critical systems where hardware solutions cannot be used due to competing non-functional constraints. Recent works have focused on developing compiler-based protection mechanisms, relying on debuggers and other software mechanisms to introduce single event upsets. In this work, we test a compiler-based technique using physical radiation testing methods, including laser fault injection and alpha-particle exposure. During this evaluation, we identified previously unknown issues that required further development, including a novel memory allocation strategy for improved reliability. Furthermore, we integrated this fault detection solution with a hard real-time recovery mechanism that exploits mixed-criticality scheduling to demonstrate the overall system recovery capabilities. The results show the effectiveness of the proposed approach in detecting faults even under real-world radiation conditions, representing an important step toward the maturity of SIHFT techniques.**

***Index Terms*—SIHFT, compilers, fault tolerance, radiation testing**

I. Introduction

The resilience of a computing system to random hardware errors is one of the most concerning aspects of safety- and mission-critical systems development. Such errors are caused by external and internal unpredictable events, such as cosmic radiation or the decay of radioactive impurities in the chip package. Within the taxonomy of hardware faults, Single-Event Effects (SEEs) are the most studied, and are further divided into several categories. The one of interest for this study is that of transient faults, i.e., non-destructive erroneous outputs from a latch or memory cell, where the component reverts to its intended behaviour after the error manifested. SEEs can be further divided into Single-Event Upsets (SEUs), when the event causes a single flip in the hardware logic[1], and Multi-Bit/Cell Upsets (MBU/MCU), when multiple bits or memory cells are upset by the physical event [1].

Recent studies in dependable system design address the problem of implementing fault tolerance mechanisms against hardware faults through software methodologies. These approaches, which have already been studied for decades, fall under the umbrella term Software-Implemented Hardware Fault Tolerance (SIHFT) [2]. The goal of SIHFT is to reduce the need of hardware protections, which are usually expensive in terms of monetary costs, performance, area, and power. SIHFT also eases the adoption of uncertified Commercial-Off-The-Shelf (COTS) hardware to implement critical systems. Compared to traditional radiation-hardened components, COTS devices have many advantages in terms of non-functional requirements, including cost, weight, energy consumption, and thermal dissipation. As a result, the industry, especially space agencies [3], started considering the adoption of SIHFT on COTS components due to the pressing need to reduce costs and time-to-market while boosting production volumes.

A. State of the Art

The current established practice for implementing SIHFT techniques is mostly based on the manual modification of programs – see, for instance, the handbook used by the European Space Agency ECSS-E-HB-20-40A [4]. More recent academic works proposed compiler-based SIHFT reliability solutions as a way of achieving software-based radiation hardening via automatic code transformation, thereby simplifying the development of critical software. Examples of state-of-the-art compiler-based tools are SWIFT [5] (2005), nZSDC [6] (2016), COAST [7] (2019), COMPAS [8] (2022), and ASPIS [9] (2024). ASPIS is the most recent SIHFT compiler that tackles the shortcomings of previous tools [9], and is the only one still actively developed at the time of writing. Among the different tools, only COAST has been experimentally validated in a real irradiated environment via neutron beam testing, while all others have been evaluated only via software simulation. Other code transformation tools have been tested in the presence of real ionizing radiation [10] [11], yet they do not provide a comparable automated compiler-based solution.

While software-based fault injection simulations can provide an initial estimate of system reliability, they present many well-known limitations. Specifically, they cannot inject faults into locations inaccessible to software, can introduce spurious overheads that can break real-time properties, and cannot accurately model real fault patterns [12].

B. Contributions

To our knowledge, only one work in the literature presents a limited hardware fault injection campaign on a SIHFT solution. This prompted our research. We selected the most

This work has been supported by the National Resilience and Recovery Plan (PNRR) through the National Center for HPC, Big Data, and Quantum Computing.

* These authors contributed equally.

[1]Also known as Single-Bit Upsets (SBUs).

recent compiler-based solution, ASPIS, as the target of our radiation experiment, performed at the Materials & Electrical Components Laboratory, part of the ESA ESTEC facility at Noordwijk (NL). In particular, this paper provides the following contributions:

- We studied previously unknown side effects caused by the memory allocation strategy when considering multi-bit upsets.
- We implemented a real-time fault recovery strategy on FreeRTOS by exploiting a mixed-criticality re-execution model triggered by ASPIS.
- We used laser and radiation sources to: assess the ability of the ASPIS compiler to detect faults, compare the reliability achieved by two different memory allocation strategies, highlight the correlation of the results between laser and radiation testing.

To the best of our knowledge, none of these contributions are available in the literature, and this is the first paper that performs laser and ^{252}Cf testing for SIHFT.

The article is structured as follows. Section II provides the necessary technical and physics background. Section III describes the implemented memory allocation and fault recovery strategies. Section IV explains the methodology used for the experimental campaign, the results of which are reported and discussed in Section V. Section VI draws the conclusions and describes possible future works.

II. Background

A safety-critical system must guarantee that the required functionalities are performed correctly, even in the presence of faults. According to standard terminology:

- a *fault* is a hardware component malfunction (e.g., the content of an SRAM cell is altered by a cosmic ray);
- an *error* is a discrepancy between the actual and the expected state of the program (e.g., the alteration of the value of the variable stored in the affected SRAM cell);
- a *failure* is the inability of the system to perform its required functionalities (e.g., the changed variable leads to an incorrect system output).

A fault may or may not cause an error, and an error may or may not cause a failure. Traditionally, a pessimistic model in which all faults lead to failure is applied, encouraging the development of hardware solutions to mitigate component susceptibility to physical phenomena. SIHFT, however, requires fine-grained analyses of the failure propagation model to perform an accurate reliability assessment. Consequently, system-level fault injection became even more important for predicting how the system will behave in irradiated environments and deriving combined HW and SW reliability metrics.

A. ASPIS

ASPIS is an LLVM[2]-based framework that implements passes providing both *data protection* and *control-flow graph protection*. The former protects against SEUs affecting storage and logic, while the latter protects against SEUs affecting the execution flow of a program. Results obtained via software fault injection show that a system hardened by ASPIS is able to detect up to 99.988% of the single-bit flips injected [9].

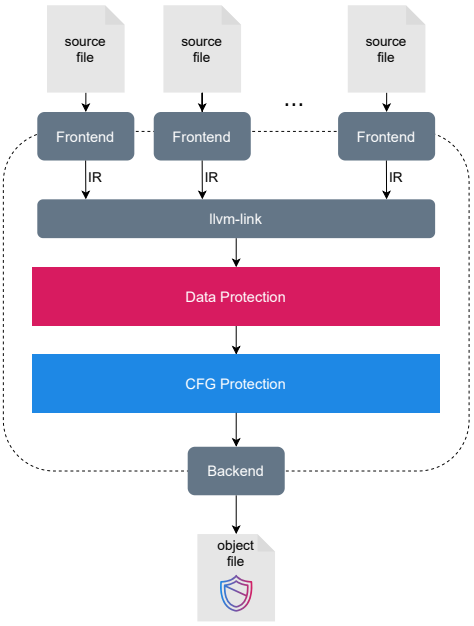


Fig. 1. The ASPIS compilation pipeline.

The ASPIS compilation flow, depicted in Figure 1, begins with the LLVM front-end, which transforms the original source code into the LLVM Intermediate Representation (IR). This approach makes the compilation flow completely agnostic to the source language and the underlying architecture, as long as they are both supported by LLVM. All the IR files are then linked by `llvm-link` and passed through ASPIS data protection and CFG protection passes.

The *data protection* mechanism of ASPIS implements a modified version of Error Detection by Duplicated Instructions (EDDI) algorithm [13] at the IR level. EDDI interleaves the program data and code, alternating original and duplicate data and instructions. Periodic checks verify the integrity between the two copies of the program state. In ASPIS, these checks are placed before control-flow instructions, memory-write instructions, and call instructions. The version of EDDI ported to ASPIS iterates on all program instructions, applying different transformations depending on the kind of IR instruction encountered. On top of that, ASPIS provides a set of optimizations that can be used by the developers to reduce the number of consistency checks inserted by EDDI and lower their overhead. Among the various ASPIS optimizations, sEDDI is an optimized version of EDDI in which consistency checks are added before control-flow and call instructions, but not memory-write instructions.

The *control-flow graph protection*, on the other hand, is implemented in ASPIS as two alternative signature-based Control-Flow Checking techniques from the state-of-the-art: Control-Flow Checking via Software Signatures (CFCSS) [14] and Random Additive Signature Monitoring (RASM) [15].

[2] `https://llvm.org/`

For further information on ASPIS, including implementational details, fault coverage, and timing overhead, please refer to the original paper [9].

B. Fault injection and modeling

The problem of testing software against radiation-induced faults can be tackled from different perspectives depending on the experiment objectives. The most common approach to software reliability testing is *fault injection*, in which faults are artificially injected into the system to trigger errors and study their consequences on the program execution. Fault injection can be performed at different abstraction levels, including software-based fault injection, in simulation (circuit, gate, RTL, microarchitectural), and hardware-based fault injection. The latter represents the most effective way of performing soft error analyses, as the physical phenomena affecting the system in its target operating environment can be reproduced with a considerable degree of statistical accuracy. For this study, we tested ASPIS with two real-world stress methodologies. Specifically, we tested via laser fault injection to perform thorough experiments in a controlled setup, and via exposure to californium (^{252}Cf) isotopes to test ASPIS against α radiation.

Laser testing. Although it has been known since the late 1980s [16], laser-based fault injection gained more interest only recently. It is cheaper and presents fewer safety hazards than radiation testing with particle accelerators, allowing developers to trigger faults at specific locations in a silicon die. Its energy can be easily regulated, leading to a more controlled experimental setup [17].

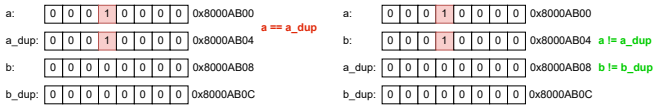
Radiation testing. Exposure to radioactive isotopes is a widely known fault injection technique used for testing the resilience of IC to various categories of high-energy particles. Specifically, ^{252}Cf experiments are traditionally used to check the resistance of devices to alpha radiation since they compose 97% of ^{252}Cf emissions. Alpha emissions are part of the decay products of radioactive impurities (such as ^{232}Th and ^{238}U) present in the packaging materials of COTS devices. ^{252}Cf also emits neutrons. As SIHFT enables the adoption of COTS, which package is less controlled, it is crucial to test the resiliency of these components against this kind of radiation.

III. Memory allocation strategies and fault recovery

The following paragraphs describe the impact of memory allocation strategies on the reliability metrics and how we extended the FreeRTOS operating system for handling recovery via task re-execution.

A. Memory allocation, its impact on reliability, and the implemented solution

All the compiler-based tools mentioned in Section I-A have been designed to protect the code under the assumption that only SEUs occur. However, our initial experiments with both laser and radiation testing revealed that the majority of SEEs were MCUs (as later detailed in Section IV). Due to the physical arrangement of the SRAM memory cells, a single particle could hit the same bit position of two address-consecutive words. This SRAM physical layout was also observed in other microcontrollers [18].



(a) *AlternatedMemMap* (b) *SequentialMemMap*

Fig. 2. Difference between an MCU (bits flipped are highlighted in red) affecting two contiguous cells with *AlternatedMemMap* and *SequentialMemMap*.

1) The identified problem: In this MCU/MBU scenario, which was not considered in previous studies, the memory allocation strategy plays a crucial role in preventing the corruption of both original and duplicate data at the same bit position. In the ASPIS case, the default allocation strategy interleaved original and duplicate data. We call this strategy *AlternatedMemMap*. This allocation can potentially lead to no mismatches in the EDDI consistency checks because a single particle can flip the same bits of two consecutive words, preventing the error detection.

2) The proposed solution: We, therefore, implemented a different policy (*SequentialMemMap*) that allocates all original variables before all duplicate variables, maximizing the distance between the two copies of the data, so that they are less likely to be contiguous and to suffer from the problems of *AlternatedMemMap*. Figure 2 provides an example of these two memory allocation policies. When the original variable `a` and its duplicate `a_dup` are interleaved, an MCU hitting two contiguous cells is not detectable, while a mismatch will be detected in the case of *SequentialMemMap* because it affects two different variables (`a` and `b`) but not their copies (`a_dup` and `b_dup`).

B. Implementing SequentialMemMap in ASPIS

SequentialMemMap, although apparently straightforward, presents some challenges that arise when implemented at the compiler IR-level.

1) Global variables and dynamic memory: In the simplest case of global variables, ASPIS reorders them in the compilation unit, such that the set of original global variables is placed before the set of duplicate global variables. To increase the separation, ASPIS also provides some API to change the linker sectors where to put the duplicate global variables so that users can place original and copy arbitrarily far from each other. The case of dynamic memory allocation (e.g. `malloc` in C), instead, cannot be tackled directly within ASPIS. How the dynamic memory is allocated depends on the implementation of the dynamic memory allocation API and cannot be controlled by the compiler. However, since dynamic memory allocation typically consists of library function calls, ASPIS can be instructed to protect such calls by duplicating the `malloc` operations.

2) Stack management: Local variables are defined by `alloca` instructions in LLVM IR. In our *SequentialMemMap* implementation, we reordered `alloca` instructions with the

same rationale of global variables, so that original and duplicate local variables are as far as possible from each other. However, reordering variables at the IR-level changes arbitrarily the register allocation process, since it affects register liveness. In general, this unpredictably affects register spilling, leading the compiler back-end to spill data into the stack. In fact, it is possible that, if variable reordering increases too much the register pressure, data may still be allocated contiguously. To demonstrate that, let us consider the following simple C expression: `(a * b)`. After the EDDI transformation, the resulting assembly would resemble[3]:

```
ldr r0, [a]       ; Load 'a' into r0
ldr r1, [b]       ; Load 'b' into r1
mul r2, r0, r1    ; Multiply r0 and r1 in r2
ldr r0, [a_dup]   ; Load 'a_dup' into r0
ldr r1, [b_dup]   ; Load 'b_dup' into r1
mul r3, r0, r1    ; Multiply r0 and r1 in r3
```

Consider the case in which, after the application of EDDI with *SequentialMemMap*, a live variable that was previously in memory is now in the registers, and, for instance, `r3` is not available anymore due to the higher register pressure. The previous assembly would become:

```
ldr r0, [a]
ldr r1, [b]
mul r2, r0, r1
str r2, [tmp]     ; Store result (register spill)
ldr r0, [a_dup]
ldr r1, [b_dup]
mul r2, r0, r1    ; Multiply r0 and r1 in r2
```

At this point, if `(a*b)` has to be used to compute another expression `c + (a * b)`, we need once again all three registers `r0`, `r1`, and `r2`, forcing the compiler to add `str r2, [tmp_dup]` at the end of the previous snippet. At this point, we have that `tmp` and `tmp_dup` are contiguous, which makes them subsceptible to SDCs in our fault model. By this example, we showed that critical data may still be allocated contiguously. Similar reasonings can be applied to function arguments, for which reordering may cause part of the argument list to spill onto the stack rather than into registers.

Even considering these limitations, Section V-A provides experimental evidence that *SequentialMemMap* provides comparable or better results than *AlternatedMemMap* depending on the specific program.

C. Implementing fault recovery in FreeRTOS

One of the aspects of SIHFT techniques that has not been explored in previous works regards the ability of the resulting overall system to recover from a detected error. The ASPIS toolchain delegates developers to write the code that attempts a recovery in a dedicated function called upon error detection. In this paper, we adopted FreeRTOS as a use case of a real workload for the development of the recovery functionality via task re-execution.

[3]Written in ARM-like syntax, where for simplicity we assume that variable names are registers containing the addresses of the variables.

1) Implementing Mixed-Criticality in FreeRTOS: Task re-execution often presents issues for real-time workload because it introduces unexpected and large overhead when a fault occurs, easily compromising the real-time properties. For this reason, we modified the FreeRTOS scheduler to introduce a mixed-criticality scheduling policy for fault recovery with the schema proposed by Reghenzani et al. [19]. The tasks are divided in multiple levels of criticalities. At each time instant, the system is in a *system mode*. In the traditional mixed-criticality setting, each task has multiple WCET estimations. The system switches mode if a WCET is overrun, increasing the criticality level and dropping all lower criticality tasks.

We implemented such mechanism in FreeRTOS, considering only two criticality levels. In particular, we extended the Task Control Block (TCB) data structure to include its criticality level, period, and WCET. We created a specific list for mixed-criticality tasks (*MC-Tasks*). Once the scheduler is initialized, the schedulability condition of EDF-VD given the taskset provided is verified [20]. If the taskset is schedulable, the real-time system begins executing using EDF by considering the so-called *virtual deadlines* [21]. Due to this change, MC-Tasks cannot use the usual FreeRTOS APIs for delay (`vTaskDelay` and `vTaskDelayUntil`), but a new function that we implemented for the purpose of notifying task termination to the scheduler: `vTaskMCTerminated`. Timing correctness is verified at each FreeRTOS tick by our scheduler that checks whether a deadline has been overrun.

2) Combining ASPIS detection with Mixed-Criticality: If a fault is detected by ASPIS, the recovery procedure increases the *system mode* criticality level. If the detection happens while executing a high-criticality task, the task is restarted so that the correct output is produced before the deadline is met. Otherwise, the faulty task is simply dropped. Timing correcness is guaranteed by setting the pessimistic WCET in such a way that it accounts also for task re-execution. Since ASPIS is only able to detect a symptom of a corruption – which could have happened in any variable involved in the computation from the last consistency check – the entire task that got corrupted is considered unsafe. Therefore, the task is restarted instantiating a new TCB and the same inputs are fed to the new instance to redo the same computation. An example of a mixed-criticality re-execution mechanism having two high-criticality and two low-criticality tasks is provided in Figure 3. Upon detection, t_1 is restarted, the system criticality level is increased, and low criticality tasks (t_3 and t_4) are dropped.

IV. Evaluation methodology

Evaluating ASPIS against via laser and californium testing, compared to software fault injection, provides results closer to the expected reliablity of the system in the target radiation environment. The DUT is an STM32 microcontroller (STM32F427VIT6TR), which we integrated as part of a custom HW/SW stack we designed specifically for radiation experiments. Before beginning the actual experiments, we etched the plastic package to reveal the die, so that the silicon

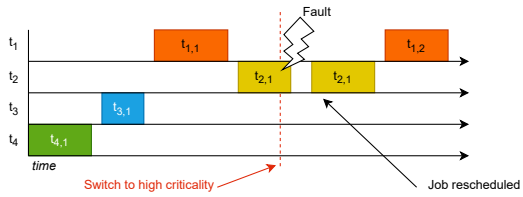


Fig. 3. Example of mixed-criticality re-execution after a fault in a high-criticality task given two high-criticality tasks (t_1, t_2) and two low-criticality tasks (t_3, t_4).

could be directly targeted by both our experimental sources. The details regarding the sensitivity of the DUT are provided in Section IV-B and Section IV-C.

A. Software configuration

ASPIS was configured to protect the program data with sEDDI and the CFG with RASM. This setup is one of the most promising ASPIS configurations [9]. The testing of other configurations is left as future work.

The experimental campaigns were conducted by checking the outputs of the programs when the DUT was subject to radiation. The outputs were verified against golden run outputs gathered offline without radiation. We tested the following bare-metal tasks: SHA from the MiBench benchmark suite [22], MatMult (MM) from the Mälardalen benchmark suite [23], and an aerospace-specific benchmark called Latnav (lateral navigation) for realistic aerospace workload simulation. As an additional benchmark, we ran an instance of FreeRTOS modified with our mixed-criticality scheduler. FreeRTOS was running SHA, MatMult, and Latnav, plus an additional aerospace-related benchmark, called ACAS (Airborne Collision Avoidance System), as concurrent real-time tasks. In this setup, we had Latnav and ACAS as high-criticality tasks and SHA and MatMult as low-criticality tasks. The output of a single FreeRTOS run is the checksum of the outputs of the tasks at the end of a hyperperiod[4]. To clarify the terminology, we call, in the case of bare-metal tasks, *job* the single execution of the program and, in the FreeRTOS case, the *job* is a single hyperperiod of the real-time scheduler.

The possible results of a job execution are:

- *No Effect*. The output of the job matches the one of the golden run, i.e., the job is executed correctly.
- *Timeout*. The DUT became unresponsive, e.g., due to a fault causing the execution to be stuck in an infinite loop.
- *Silent Data Corruption (SDC)*. The job output does not match the golden run and no detection occurred.
- *Hard fault*. The error was detected by the hardware (e.g., illegal memory access).
- *Detected*. ASPIS was able to detect the error.

From the dependability standpoint, the most dangerous result is the SDC because it is the only undetectable case that produces an incorrect output. All the other cases can be detected and the system can either recover or fall back to a safe state.

[4]The checksums of golden and test runs coincide if and only if the timing and logical correctness of the execution is guaranteed, in line with the requirements of hard real-time systems.

B. Laser-based fault injection

Laser testing was performed using the PULSYS Rad by PULSCAN deployed at the ESA ESTEC facility for SEE testing and reliability evaluation. The laser was configured to pulse at a constant rate while moving its head at a constant speed, repeatedly scanning the same area in a random pattern. This method aims at replicating the random distribution of ionizing particles across the entire target area. We scanned with the laser only the allocated sections of the SRAM, which was overestimated to target at least all the memory used by the task. When shooting on the SRAM with $300pJ$, we obtained from 1 to 8 multi-cell flips per shot, with an average of 4.5 flips and a standard deviation of 2.45. We tested all four benchmarks (SHA, MM, Latnav, and FreeRTOS) with laser-based fault injection.

C. Radiation-based fault injection

Radiation testing was performed at the ESA CASE (Californium Assessment of Single-event Effects) facility. CASE injects SEE in semiconductors using a ^{252}Cf source[5] in a vacuum bell to maximize the effect of the radiation. We discovered by performing "burst" exposures of a few milliseconds of the DUT to ^{252}Cf that the particles were causing multi-cell upsets. Exposing the DUT to the source for several minutes, revealed an average device fault rate of over 12 flips per second. We tested two benchmarks (MM and FreeRTOS) with the ESA CASE facility.

V. Experimental campaign

This section provides the results for both laser and ^{252}Cf testing. For each experimental environment, we provide the reliability metrics as: 1) outcome ratios with respect to the number of executed jobs; and 2) outcome ratios with respect to the number of jobs that experienced some errors (i.e. when the fault was *effective*). In the following discussion, *Unprot.* refers to the unprotected version of the benchmark, *Alt.* refers to the original version of ASPIS protected using *AlternatedMemMap*, while *Seq.* refers to the novel version of ASPIS protected using *SequentialMemMap*.

A. Laser experiments results

The results of the laser fault-injection campaign with respect to the number of jobs executed are provided in Table I. For completeness, the table also reports the average number of shots per job, because the laser shooting frequency was adapted in each configuration depending on the job execution time. The table reports the percentages of each result class and their margin of error ε using a simplified[6] formula from the literature [24, Eq. (2)] considering the number of samples (#jobs) with a confidence of 99%.

[5]Providing spontaneous fission particle with average Linear Energy Transfer of the fission product (LET) of $43\,\mathrm{MeV}\cdot\mathrm{mg}^{-1}\cdot\mathrm{cm}^{-2}$

[6]Assuming $p = 0.5$ and $N \to \infty$. These assumptions obtain a pessimistic estimate of the margin of error.

TABLE I

LASER TESTING OUTCOME RATIOS WITH RESPECT TO THE NUMBER OF EXECUTED JOBS ON 4 BENCHMARKS: SHA, MM, LATNAV, AND FREERTOS. EACH BENCHMARK WAS TESTED UNPROTECTED, WITH ASPIS USING ALTERNATEDMEMMAP, AND WITH ASPIS USING SEQUENTIALMEMMAP.

		SDC (%)	Detected (%)	Hard fault (%)	Timeout (%)	No Effect (%)	#jobs	#shot/#job	ε ($\pm$)
SHA	Unprot.	2.305	-	0.123	0.477	97.080	13840	0.65	1.095
	Alt.	0.196	4.460	0.208	0.190	94.945	112628	0.85	0.384
	Seq.	0.038	3.784	0.162	0.156	95.859	122424	0.89	0.368
MM	Unprot.	4.418	-	0.080	0.200	95.396	27910	0.66	0.771
	Alt.	0.025	16.707	0.108	0.141	82.977	12019	1.17	1.175
	Seq.	0.034	18.389	0.119	0.124	81.330	23449	1.16	0.841
Latnav	Unprot.	0.683	-	0.097	0.252	98.967	533658	0.37	0.176
	Alt.	0.084	1.319	0.173	0.198	98.226	36277	0.88	0.676
	Seq.	0.109	1.246	0.254	0.239	98.151	149209	0.92	0.333
FreeRTOS	Unprot.	0.635	-	0.948	0.627	97.790	66976	1.43	0.498
	Alt.	0.060	2.440	2.808	0.480	94.212	160435	2.48	0.322
	Seq. w/ Recovery	0.016	2.098	1.174	0.260	96.453	48486	1.48	0.585

The results in Table I show that ASPIS improves the reliability of the system by drastically lowering the number of SDCs that affect the system by up to two orders of magnitude. The data also highlights a difference in the susceptibility of the various programs in terms of error rate (i.e. the rate of effective faults). This difference is due to various factors, including the memory footprint of the code and the time in which variables are alive during the execution. In general, the transformation of ASPIS introduces time and space overheads that reduce the percentage of *No Effect* results. However, the most significant issue affecting the dependability of this system is the occurrence of SDCs and not the absence of *No Effect*. If the system were unaware of the corruption, it would continue to use the faulty data, potentially leading to severe consequences. ASPIS plays a crucial role in this regard since the system will be aware of the failure and can take appropriate measures to prevent further damage. On this note, the FreeRTOS recovery configuration prevents system downtime in more than 40% of ASPIS-detected errors. In all the other cases, the system was aware that a fault occurred but simply could not perform recovery; more efficient recovery mechanisms for FreeRTOS are left as future works.

1) Failure rate per effective fault: Figure 4 provides the outcome ratios with respect to the number of *effective* faults. ASPIS transforms almost all SDCs of the MM benchmark into detections and reduces the number of undetected SDC with respect to the total effective faults to 0.47% in the case of FreeRTOS with recovery. Compared with the unprotected system (28.72% SDCs), ASPIS with our recovery solution improves the SDC protection by a factor of more than $61\times$.

2) Alternated vs. Sequential MemMap.: One of the goals of this testing campaign was comparing the AlternatedMemMap and SequentialMemMap memory allocation strategies. The results show how SequentialMemMap lowers the SDC rate in the cases of SHA and FreeRTOS (Fig. 4, upper left and lower right plots). MM and Latnav, instead, manifest different behaviours (Fig. 4, upper right and lower left plots). In the case of MM, SequentialMemMap does not provide any meaningful advantage since the data of the program consisted of the three big matrices (two for input and one for output). Being each matrix a chunk of contiguous data, the original data and its copy were sufficiently apart from each other also in the AlternatedMemMap scenario. Latnav, on the other hand,

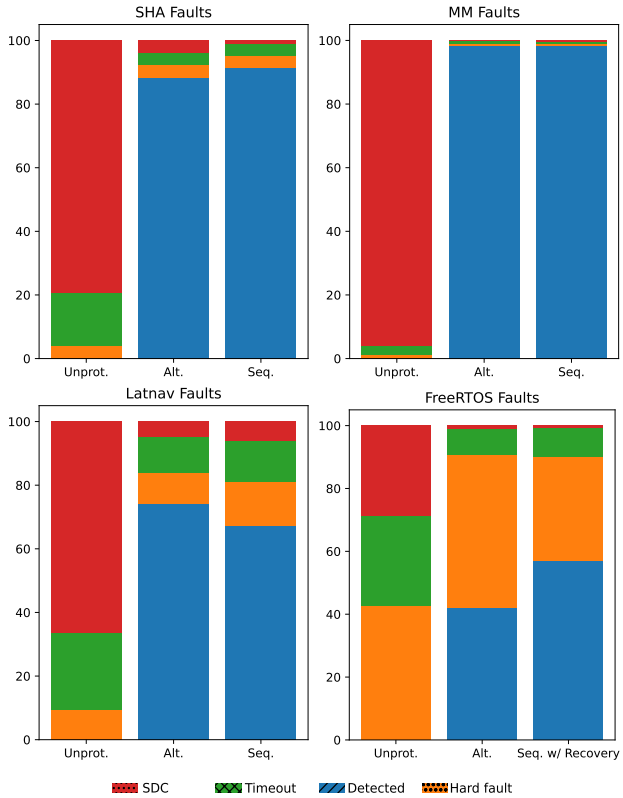
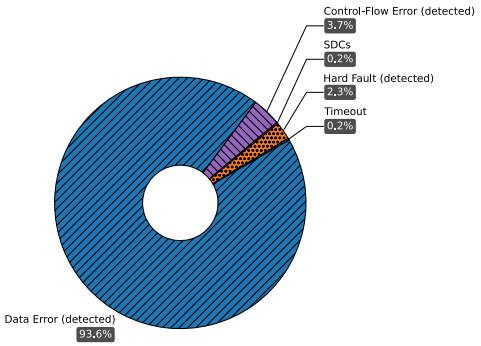


Fig. 4. Percentage of different outcomes occurring during laser experiment with respect to the number of effective faults.

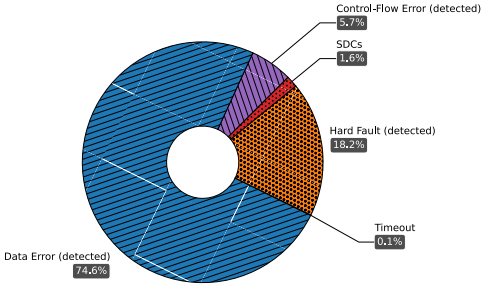
is not a memory-intensive benchmark which makes it very subsceptible to the problem of register spilling described in Section III-B2. Remarkably, by analyzing the emitted Latnav binary, we discovered that the code emitted with SequentialMemMap relies on memory 10% more often than the code emitted with AlternateMemMap (computed as the amount of memory-related instructions executed, like loads and stores). We argue that this is a symptom of the fact that applying SequentialMemMap on Latnav increases register pressure. Empirical evidence (Table I) supports the hypotesis of critical data spilling onto memory when using SequentialMemMap, as it can be seen from the higher SDC rate. Future experimental campaigns will further investigate this aspect.

From these laser experiments, we conclude that:

- SequentialMemMap is effective especially if the program



(a) MM californium results.



(b) FreeRTOS californium results.

Fig. 5. Californium percentage of outcomes with respect to the total amount of effective faults on MM and FreeRTOS.

is composed of small- and medium-sized data structures;
- SequentialMemMap and AlternatedMemMap can be considered equivalent if the program data is composed only of large data structures since original and duplicated data are already sufficiently separated by the struct size;
- The reliability difference of SequentialMemMap and AlternatedMemMap is minimal when the number of data structure is very small because the register spilling strategy of the backend compiler has a heavier impact.

B. Californium experiments results

The results of the exposure of the DUT to ^{252}Cf are provided in Table II. These results show the percentages of outcomes considering the number of jobs executed while the device was exposed to the radioactive source. Figure 5 displays the outcome ratios of ASPIS with respect to the number of effective faults. In these experiments, MM and FreeRTOS were tested and ASPIS was using SequentialMemMap.

The results highlight the efficacy of ASPIS in detecting the errors that are caused by a real radiation source: the system with MM suffers from 0.08% SDCs (0.210% of the effective), while FreeRTOS only 0.11% (1.551% of the effective).

C. Correlating laser experiments with radiation experiments

The mapping between real-world radiation results and software- and laser-based fault injection is one of the open challenges when performing system-level reliability assessments.

In our case, the proportion of undetected SDCs observed during ^{252}Cf testing (Table II) is larger than the one obtained during laser testing (Table I). However, when comparing laser experiments with real-world radiation experiments, we account for the fact that the radiation also hits the microcontroller core and other components, not only its memory. The core can suffer from a much wider set of faults since they can happen during different stages in different functional units. This difference compared to the laser led to a higher SDC rate with ^{252}Cf testing. Moreover, in our experiments, we had a higher bit-flip rate in the ^{252}Cf setting, due to the physical setup. For instance, in the case of FreeRTOS, the hyperperiod is 1.5 seconds. Since with ^{252}Cf we had an average of 12 memory bit-flips per second, we derive that the system was suffering an average of 18 SEUs per run compared to an average of around 7 SEUs per run in the laser case.

Finally, with ^{252}Cf we have slightly fewer timeouts compared to the laser experiments. We hypothesise this is due to the particle flux, that prevented our watchdog from expiring before a subsequent particle caused a hard fault. This hypothesis matches the slight increase in hard faults in the ^{252}Cf results compared to laser results.

We finally report a numerical analysis that aims at correlating the two experimental evaluation campaigns. For every given benchmark t and testing environment $E \in \mathrm{L}, \mathrm{A}$, where L stands for "laser" and A stands for "alpha", we define φ_{E_t} as the probability of causing an SDC per single bit flip. Let us call δ_{E_t} the probability of a job belonging to task t becoming silently corrupted under environment E. This quantity can be obtained directly from Tables I and II. Let D_{E_t} be the average *dose* of bit flips received by a job of task t under environment E. In the case of ^{252}Cf, we know that, on average, the device experiences 12 bit flips per second. By knowing the execution time of MM and FreeRTOS (350ms and 1.5s, respectively), we obtain $D_{A_{\text{MM}}} = 12 \cdot 0.35 = 4.2 \frac{\text{flips}}{\text{job}}$ and $D_{A_{\text{FreeRTOS}}} = 12 \cdot 1.5 = 18 \frac{\text{flips}}{\text{job}}$. For laser experiments, we have to account for the fact that a single shot is known to cause multiple-bit flips. Let γ be the average number of bit flips per shot. Let S_t be the number of shots delivered to a job of task t, as reported by Table I. Thus, $D_{L_t} = S_t \cdot \gamma$, and, by using the empirical value of $\gamma = 4.5 \frac{\text{flips}}{\text{shot}}$ from Section IV-B, we have $D_{L_{\text{MM}}} = 5.22 \frac{\text{flips}}{\text{job}}$ and $D_{L_{\text{FreeRTOS}}} = 6.66 \frac{\text{flips}}{\text{job}}$. Lastly, we can calculate φ_{E_t} as $\varphi_{E_t} = \frac{\delta_{E_t}}{D_{E_t}}$, which leads us to $\varphi_{L_{\text{MM}}} = \frac{0.034}{5.22} \approx 0.0065 \frac{\text{SDCs}}{\text{flip}}$ and $\varphi_{A_{\text{MM}}} = \frac{0.08}{4.2} = 0.019 \frac{\text{SDCs}}{\text{flip}}$ for MM, and $\varphi_{L_{\text{FreeRTOS}}} = \frac{0.016}{6.66} \approx 0.0024 \frac{\text{SDCs}}{\text{flip}}$ and $\varphi_{A_{\text{FreeRTOS}}} = \frac{0.11}{18} \approx 0.0061 \frac{\text{SDCs}}{\text{flip}}$ for FreeRTOS.

D. Comparison with previous compiler-based solutions

Among the other compiler-based solutions in the state of the art, only COAST was tested with realistic fault injection. As a result of the neutron beam experiments [7], COAST with Triple-Modular Redundancy (TMR) achieved a Mean-Work-To-Failure (MWTF) [25] improvement of $7.1\times$ on the MM benchmark, where MWTF = (the number of correct executions excluding detections) / (number of incorrect executions including SDCs, Hard faults, and Timeouts). Despite being different fault injection scenarios, we compare with this result for the sake of completeness. In our experiments, we

TABLE II
OUTCOME RATIOS OF CALIFORNIUM TESTING ON MM AND FREERTOS.

	SDC (%)	Detected (%)	Hard Fault (%)	Timeout (%)	No Effect (%)	#jobs	ε ($\pm$)
MM	0.08	35.08	0.82	0.09	63.93	21110	0.89
FreeRTOS	0.11	5.42	1.24	0.01	93.23	45677	0.60

TABLE III
MEAN-WORK-TO-FAILURE FOR THE MM BENCHMARK.

Experiment	MWTF	Increase
Laser Unprot.	20.31	1.00×
Laser Alt.	302.84	14.91×
Laser Seq.	293.61	14.46×

obtained the data presented in Table III. ASPIS achieved an MWTF increase of up to 14.91× on MM, almost doubling the reliability factor of the COAST state-of-the-art solution. This comparison must be further validated by running neutron beam experiments on ASPIS as well.

VI. CONCLUSIONS AND FUTURE WORKS

This paper describes the experimental testing of the ASPIS compilation toolchain, providing also valuable insights in the memory allocation strategy against MCUs. Multiple configurations of ASPIS were tested via laser-based fault injection and exposure to a ^{252}Cf source at the ESTEC facility of ESA in The Netherlands. To the best of our knowledge, this is the first experimental campaign testing compiler-based SIHFT under the MCU fault model and via laser/radiation fault injection. From the analysis of the results of the laser campaign, we observed that the system hardened with ASPIS suffered only 0.016% SDCs per execution, while the unprotected version suffered 0.635%. Concerning the ^{252}Cf, the system hardened with ASPIS suffered only 0.11% SDCs in over 45k runs. Undetected SDC on the total effective faults is 0.47% when running a full-fledged FreeRTOS system with recovery, which improves protection by over 61×. The gathered results validate the effectiveness of ASPIS in detecting and recovering from MCUs, which increases the overall credibility of SIHFT solutions in the field of dependable computing.

Future research directions will include a formal timing analysis of the recovery strategy from a real-time standpoint and further validation campaigns via beam experiments (with protons, neutrons and heavy ions) and further development of the protection algorithms.

REFERENCES

[1] *Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices*, JEDEC Std. JESD89B, sep 2021.

[2] O. Goloubeva, M. Rebaudengo, M. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Springer US, 2006.

[3] M. Nikulainen and F. Tonicello, "Utilization of COTS in ESA missions," 2021.

[4] E. C. for Space Standardization, *Engineering techniques for radiation effects mitigation in ASICs and FPGAs handbook*, European Space Agency Handbook ECSS-E-HB-20-40A.

[5] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "Swift: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[6] M. Didehban and A. Shrivastava, "nzdc: A compiler technique for near zero silent data corruption," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[7] M. Bohman, B. James, M. J. Wirthlin, H. Quinn, and J. Goeders, "Microcontroller compiler-assisted software fault tolerance," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 223–232, 2019.

[8] U. Sharif, D. Mueller-Gritschneder, and U. Schlichtmann, "Compas: Compiler-assisted software-implemented hardware fault tolerance for risc-v," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 2022, pp. 1–4.

[9] D. Baroffio, F. Reghenzani, and W. Fornaciari, "Enhanced compiler technology for software-based hardware fault detection," *ACM Trans. Des. Autom. Electron. Syst.*, apr 2024.

[10] H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran, "Robust duplication with comparison methods in microcontrollers," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 338–345, 2017.

[11] E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, J. Tonfat, E. Macchione, F. Aguirre, N. Added, N. Medina, V. Aguiar, M. A. G. Silveira, L. Ost, R. Reis, S. Cuenca-Asensi, and F. L. Kastensmidt, "Reliability on arm processors against soft errors through sihft techniques," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2208–2216, 2016.

[12] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[13] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[14] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[15] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive signature monitoring for control flow error detection," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1178–1192, 2017.

[16] A. K. Richter and I. Arimura, "Simulation of heavy charged particle tracks using focused laser beams," *IEEE Transactions on Nuclear Science*, vol. 34, no. 6, pp. 1234–1239, 1987.

[17] F. Reghenzani, D. Baroffio, T. A. Lopez, W. Fornaciari, T. Borel, and F. Krimmel, "Laser fault injection methodology for software reliability testing," *IEEE Reliability Magazine*, vol. 2, no. 1, pp. 52–63, 2025.

[18] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2002*. Berlin, Heidelberg: Springer, 2003, pp. 2–12.

[19] F. Reghenzani and W. Fornaciari, "Mixed-criticality with integer multiple wcets and dropping relations: New scheduling challenges," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASPDAC)*. ACM, 2023, p. 320–325.

[20] F. Reghenzani, Z. Guo, L. Santinelli, and W. Fornaciari, "A mixed-criticality approach to fault tolerance: Integrating schedulability and failure requirements," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2022, pp. 27–39.

[21] D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi, "EDF-VD scheduling of mixed-criticality systems with degraded quality guarantees," in *2016 IEEE Real-Time Systems Symposium*, 2016.

[22] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization*, 2001, pp. 3–14.

[23] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks: Past, Present And Future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, vol. 15, 2010, pp. 136–146.

[24] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 502–506.

[25] G. Reis, J. Chang, N. Vachharajani, S. Mukherjee, R. Rangan, and D. August, "Design and evaluation of hybrid fault-detection systems," in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005, pp. 148–159.