electronics

MDPI

Article

A Study on Selective Implementation Approaches for Soft Error Detection Using S-SWIFT-R

Mohaddaseh Nikseresht *, Jens Vankeirsbilck * and Jeroen Boydens *

Department of Computer Science, KU Leuven, Spoorwegstraat 12, 8200 Bruges, Belgium
* mohaddaseh.nikseresht@kuleuven.be (M.N.); jens.vankeirsbilck@kuleuven.be (J.V.); jeroen.boydens@kuleuven.be (J.B.)

Abstract: This article analyzes diverse criteria for effectively implementing selective hardening against soft errors through software-based strategies. The goal is to obtain maximum fault coverage with the least amount of overhead for each specific application. To achieve this objective, the analysis is conducted based on two important phases, pre-selection and selective hardening of registers. In the pre-selection phase, the impact of the two most used selection metrics has been examined: (1) selecting registers based on their memory interaction vs. (2) selecting registers depending on fault injection vulnerability. Toward the selective hardening phase, the impact of gradually increasing the number of registers to protect is examined. Experiments have been conducted on 8 academic case studies and 1 industrial case study. Faults have been injected into the case studies using our in-house fault injector. The results indicate that selecting registers based on the fault injected into the system has an overall 10% better performance in comparison to selecting registers based on the memory interaction in 6 out of 8 academic case studies and also the industrial case study. Additionally, there is a significant improvement in reliability when increasing the number of registers to protect at the expense of rising overhead. In this work, these comparisons and analyses are presented.

Keywords: fault tolerance; reliability; embedded systems; soft errors; single event upset (SEU); selective implementation

Citation: Nikseresht, M.; Vankeirsbilck, J.; Boydens, J. A Study on Selective Implementation Approaches for Soft Error Detection Using S-SWIFT-R. *Electronics* **2022**, *11*, 3380. https://doi.org/10.3390/electronics11203380

Academic Editors: Sanjay Misra, Robertas Damaševičius and Bharti Suri

Received: 23 August 2022
Accepted: 14 October 2022
Published: 19 October 2022

1. Introduction

An embedded system is a combination of computer hardware and software that is intended to perform a particular function inside a larger system [1]. A wide range of embedded systems can be found in different applications, from digital watches and cameras to medical systems, autonomous systems, and avionics [2]. Microprocessors are the heart of embedded systems. Over the last few decades, there have been significant scientific advancements in the creation of microprocessors. Some of these advancements include a huge boost in their performance, as well as an ever increasing integration density. Most of these improvements are thanks to advancements toward microtechnology; electronic components have become smaller, which has made it feasible to achieve these significant outcomes. However, as a result of these modifications, transistor sizes are diminishing, and voltage source levels and noise margins have also been reduced [3]. The result is that electronic devices became less reliable, and thus microprocessors became more susceptible to various types of faults, particularly those caused by radiation [4]. This matters since, in mission-critical systems, radiation impact on electronic components could have devastating repercussions.

These negative impacts are caused by high-energy particles influencing electronic components, which may result in the ionization of their internal silicon structures, either directly or indirectly. These incidents might have a long-term effect on the functioning of the component (permanent faults) or just have a short-term effect (transient faults). In contrast to permanent faults, transient faults may cause the system's behavior to change briefly

by interfering with signal transfers or storing data in memory. Among transient faults, Single Event Upset (SEU), which is defined as a change in the logic state of a single memory element, could cause the most damage to the system [5]. SEUs have traditionally been seen as a source of concern for space application systems, owing to the fact that they occur more often in outer space than on earth. However, in recent decades, the scope of this challenge has been broadened to include electrical circuits that must operate in the atmosphere, and even at the surface of the earth, consequently, their safety and dependability are seriously compromised [6].

From another aspect, the usage of commercial off-the-shelf (COTS) processors in mission-critical embedded systems, add a challenge to embedded systems' dependability as well. Due to their programmability, performance, and cost-effectiveness, COTS electronic components offer important capabilities and advantages in the development of low-cost safety-critical systems. However, COTS components were designed with general-purpose applications in mind, not mission-critical applications. Thereby, they have a high vulnerability to radiation-induced impacts, such as SEU, which make the system's reliability more difficult [7]. As a consequence of these two significant obstacles, more and more applications are realizing the need for hardening embedded systems against SEU's faults. Additionally, COTS components restrict the implementation of hardware-based mechanisms directly inside the CPUs, therefore, alternative software redundancy technique approaches must be adopted.

To address these obstacles, recently several publications have advocated the selective hardening of software-based systems in order to decrease software overheads while also providing more flexibility to designers. This involves protecting just certain parts of the application or the CPU's architectural resources while using duplicate software. These protected components may be chosen, among other factors, depending on their vulnerability or their contribution to overheads. It is essential to prioritize the protection of the most susceptible resources while avoiding significant system impacts, such as an increase in memory overhead or a performance loss.

In this paper, which is primarily focused on transient faults, **the goal is to examine different techniques for identifying desired (vulnerable) registers in a program.** To determine which registers to protect, two well-known methodologies are chosen to conduct research. One is focused on selecting registers based on their number of accesses to the memory [8]; the other is selecting registers based on their vulnerability to faults. We have compared the efficacy of these two strategies. Later, the desired registers are protected using a selective software-implemented fault tolerance recovery technique known as S-SWIFT-R. This technique is a well-known selective hardening technique [9]. This protection method has been used since it is a well-known method suitable for low-cost, dependable applications using commercial off-the-shelf microprocessors. However, the impact of increasing the number of registers to protect and its direct influence on the system's overhead has not been studied yet. **In this work, we analyzed the trade-off between the number of registers to protect, i.e., the increase in reliability, and the imposed overhead using S-SWIFT-R.**

This paper is structured as follows: the background is detailed in the next section. Section 3 describes the rationale and implementation of the pre-selection approaches for selecting the desired registers. Following the application of S-SWIFT-R, our selective register hardening technology, for hardening the desired registers. Section 4 describes our algorithm and methodology for applying S-SWIFT-R to software automatically. The experimental findings are described and debated in Section 5. Finally, Sections 6 and 7 conclude with suggestions for further studies and closing views.

2. Background

Traditionally, hardware redundancy has been the most frequent method for addressing reliability problems in the design of digital circuits. This encompasses an extensive array of solutions based on Error Detection and Correction Codes (EDACs) [10], gate-level logic

redundancy [11], and architectural level protection [3]. However, due to the expansion of processor-based systems and the need for reliable low-cost solutions, a wide variety of software techniques have evolved over time, and a new generation of approaches based on software redundant have arisen during the past few decades. In this regard, the first generation of software protection techniques known as Software Implemented Hardware Fault Tolerance (SIHFT) have been introduced [12]. SIHFT techniques address how to deal with faults affecting the hardware by acting on the software level. The SIHFT techniques try to identify faults toward instruction redundancy at a low level (assembly code). These techniques are categorized depending on the kind of fault that they attempt to detect/correct divided into two families:

- **Control flow execution faults** (also referred to as signature verification techniques) cause a jump in the execution order of the program, resulting in a system hang or crash [7]. The most common members of this family are the Control Flow Checking technique by Software Signatures (CFCSS) [8], and RASM [10]. In [10], the authors compared seven signature monitoring techniques and developed the new RASM technique based on their findings, which outperforms the studied techniques.
- **Program data faults** could corrupt data values in the program resulting in an incorrect intermediate value and consequently final output [8]. This family is mostly based on the N-versions programming methodology [11], which may be implemented at many granularity levels: program [10,11,13], procedure [12], and, most often, instruction [14–16]. The emphasis of this article is on program data faults. Several strategies have been used to prevent data flow faults. Some examples of these techniques: error detection by diverse data and duplicated instructions (ED4 I) [17], are error detection by duplicated instructions (EDDI) [17], and soft error detection using software redundancy (SEDSR) [18]. In CDFEDT [19], the author compares and reviews seven leading-edge approaches using various case examples and criteria.

The drawback of the majority of SIHFT approaches is that as detection rates grow, so do code size and execution time overhead. As a result, to fully protect the software a significant overhead is exposed to the system [16,18,20]. In conclusion, while SIHFT techniques may be a feasible protection solution for low-cost COTS-based systems, they provide a large number of design challenges, such as substantial memory costs or unreasonable performance penalties. Moreover, if these systems are also intended to enable the system's recovery, the code size and execution time overhead grow even further [17,18,21,22].

These limitations, the desire to overcome them, and the need for them in the industry gave rise to selective software protection approaches. This involves protecting just the sections of the code that are most vulnerable to the fault instead of the whole program to reduce the code size and execution time overhead. This concept is known as selective hardening. The main advantage offered by selective hardening-based techniques is flexibility. Designers have access to a range of options, enabling them to exhaustively explore the design space provided by software techniques, taking into account factors, such as code size overhead, performance degradation, and reliability level. For example, if a certain set of hardening methods is inconvenient according to the application's needs (e.g., if the maximum execution time is exceeded), the approach can be applied selectively based on the important program resources or parts. In this context, we have used the S-SWIFT-R selective hardening technique. The reason for choosing this technique is that the majority of selective techniques are used alongside hardware resilience techniques. S-SWIFT-R, on the other hand, is capable of selective software-only fault recovery strategies. As a result, we could study the effect of the selective hardening process in isolation. In the next section, the pre-selection approaches for picking registers and the steps implementing S-SWIFT-R, our selective hardening technique are described.

3. Materials and Methods

The following steps must be completed in order to examine the impact of different approaches to selecting desired registers, as well as the effects of increasing the number of

registers to hardening the program. In the first step, it is necessary to specify the approaches for picking the registers and design a practical implementation known as pre-selection techniques. The second phase explains the S-SWIFT-R technique and the mechanism for hardening the desired registers inside the program. Additionally, the abstract pictorial representation of methodology has been shown in Figure 1.

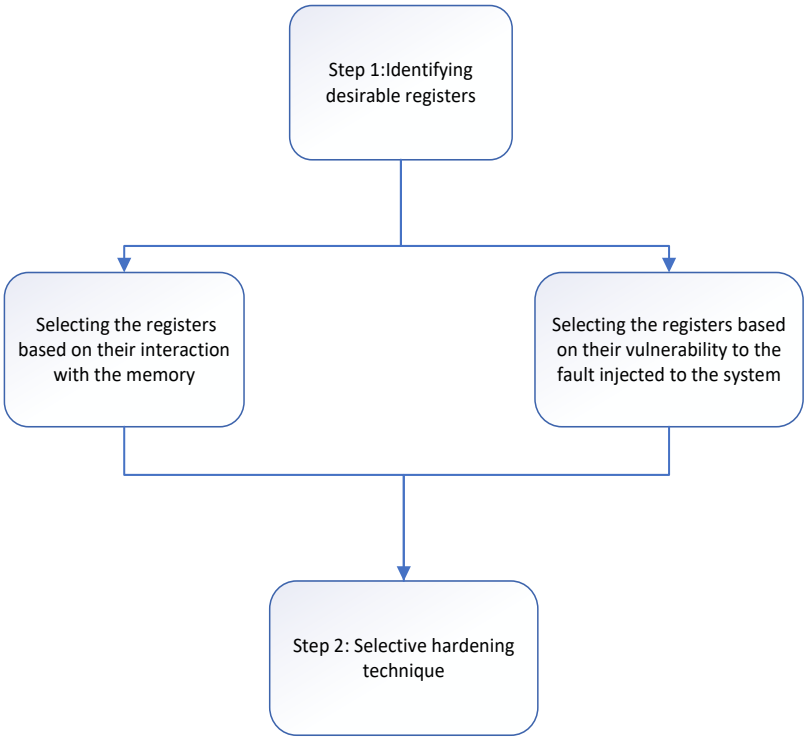

Figure 1. The abstract pictorial representation of the methodology.

3.1. Step 1: Identifying Desirable Registers

In the initial phase of implementation, the registers sensitive to transient faults must be specified. Pre-selecting is the important phase since it avoids exploring all the possible combinations in the design space and only hardening the registers that have the most impact on the reliability of the system. When S-SWIFT-R is applied to the desired registers, the overheads are predicted to grow significantly if the register is frequently accessed, since this protection entails the usage of a greater number of duplicated instructions. Protecting these registers does not ensure increased dependability since the vulnerability of each register is determined by the criticality of the functions for which the register is utilized, which is not necessarily associated with the number of times the register is visited. There are different metrics for the efficient implementation of selective hardening using software-based techniques. To be able to compare these techniques and their impact on each other we have chosen the two well-known ones among these techniques. One is selecting registers based on their interaction with the memory. Whenever we are talking about using the S-SWIFT-R technique based on memory interaction initialization throughout the paper would be indicated in the short term as S-SWIFT-R-M. In a similar way, whenever we are talking about using the S-SWIFT-R technique based on the fault injected initialization throughout the paper it would be indicated in the short term as S-SWIFT-R-F. When these techniques are applied to the program each register will be given a value. These values are given ranks in decreasing order, which indicates that the registers at the top of the list are likewise the most essential and first candidates to be hardened. To give an example, Table 1 presents the results of the DIJ case study in both pre-selecting strategies. The registers will then be chosen for hardening in the order indicated by their rank.

Table 1. Register usage report of DIJ case study.

S-SWIFT-R-M	Score	S-SWIFT-R-F	Score
R3	130,056	R0	2048
R2	64,874	R2	1824
R0	66	R1	621
R1	65	R13	512
R7	12	R4	490
R4	9	R10	312
R13	3	R3	256
R8	0	R8	218
R9	0	R5	192
R11	0	R7	148
R12	0	R5	123
R10	0	R4	107
R13	0	R6	101

3.1.1. Selecting the Registers Based on Their Interaction with the Memory

In the work that is proposed in SHARC [23], it is attempted to find the desirable register that, if hardened, will have the greatest influence on the application's overall dependability. The formula shown in Equation (1) is used to investigate the registers. Two elements are considered for this purpose, the first being the importance of the resources and the second being the vulnerability overhead (the overhead introduced into the system as a result of hardening that section). Both factors are formalized in the following definition by their respective terminology.

$$SHARC(R_x) = \frac{\Sigma_{i \neq x} ABC_i}{\Sigma_i ABC_i} \cdot W_{ABC} + \frac{CI_{Rhx}}{TI_{Rhx}} \quad (1)$$

According to Equation (1), the first component is weighted by W_{ABC} and represents the contribution of unprotected resources to global criticality. Assuming that the hardening of register x will have a minor impact on other resources, it is calculated as the total of the individual criticalities and normalized by the initial criticality. The importance of each individual resource may be summarized as follows: how likely it is that a resource-related issue may turn into an error during the code execution. By doing a dynamic study of the program, the W_{ABC} measure was utilized to estimate the program's criticality beforehand. The evaluation results of different case studies in embedded systems show that the registers that are in contact with the memory are more vulnerable to a fault in comparison to other registers. Based on those findings, we developed a framework that dynamically analyzes a given program and counts how many times each CPU register comes into contact with memory. In essence, this framework gradually steps through the program and analyzes the current instruction. In case this is a load or store instruction, i.e., an instruction interacting with memory, the used registers are discovered and their memory interaction counter is incremented. At the end of the analysis, a list is returned which contains the number of memory interactions per CPU register. For more information on this framework, the reader is referred to [24].

3.1.2. Selecting the Registers Based on Their Vulnerability

To obtain the registers that are more susceptible to faults, we have used the DFE fault injector tool from our previous work. During the execution of a program, this framework will inject hardware faults. Later, the program will be evaluated, and a register priority list will be constructed based on the number of faults inserted into each register that is able to corrupt the program output.

This process begins with an initialization step that initializes variables and generates all potential DFEs. The user-defined parameters disasmFile and range are used to generate

these potential DFEs. The disasmFile stores the path of the target program's disassembly file. This file is required to determine which program counter (PC) values are legitimate. The range parameter specifies the PC range for which the software will be evaluated. The option enables testing of a particular function (or functions) as opposed to the complete program. All conceivable DFEs developed fall within this range. This work is discussed thoroughly in [25].

3.2. Step 2: Selective Hardening Technique

After picking registers using these two approaches, the second step is to determine the technique to protect the selected registers. In this study, the fault tolerance technique must be able to cover just a subset of register elements. We adopt Felipe et al. S-SWIFT-R method [9]. It uses Triple Modular Redundancy and low-level instruction translation techniques. It combines two program copies and uses majority voting over susceptible instructions. It involves duplication of data and instructions and includes data consistency checks, in the form of a majority vote. Every protected register needs 2 shadow registers, a shadow register is a reserved register in which the only function is to keep track of all the changes made to the original register inside the program and to ensure that the original register's value is valid. The total number of registers will be 3 × n. Table 2 depicts the difference between Non-hardened and S-SWIFT-R. In code that is not hardened, no registers are protected. On the other hand, in the second and third cases, only one register is protected; hence, this is an accurate portrayal of S-SWIFT-R when just one register is protected. In each scenario, each register contains two shadow registers. In the next section, to implement S-SWIFT-R automatically, in contrast to implementing it manually, is explained.

Table 2. The difference between non-hardened and S-SWIFT-R.

		S-SWIFT-R	
#	**Non-Hardened**	**Pro.Register: S0**	**Pro.Register: S1**
1	LOAD S0, 00	LOAD S0, 00	LOAD S0, 00
2		create s0 copies	
3	LOAD S1, 2A	LOAD S1, 2A	LOAD S1, 2A
4			create S1 copies
5			majority voter for S1
6	ADD S0, S1	ADD S0, S1	ADD S0, S1
7		ADD S0′, S1	
8		ADD S0″, S1	
9		majority voter for S0	
10			majority voter for S1
11	STORE S0, (S1)	STORE S0, (S1)	STORE S0, (S1)

4. Automatic Implementation

Software protection techniques, such as S-SWIFT-R, have to be written in low-level code, such as assembly, which can be difficult and prone to mistakes. To address this issue, we have used our compiler extension in the form of a plugin that can automatically apply any supported technique to the target. This GNU Compiler Collection (GCC) compiler plugin, which is first suggested by Vankeirsbilck et al. [26], is modified so that the S-SWIFT-R technique is included. GCC converts high-level source code, such as C and C++, to low-level machine code in three steps: the front-end, the middle-end, and the back-end. The front-end component handles high-level language, the middle-end component handles intermediate language, and the back-end component handles system-specific code [27]. Please see our previous works for more information on the GCC plugin [28,29].

Figure 2 depicts the S-SWIFT-R compiler plugin implementation as a UML class diagram. The classes and methods necessary for S-SWIFT-R are shown in this figure. Upon the first execution of the plugin, it registers itself as a compilation pass. A pass is a series

of instructions that accomplishes a particular task during the compilation process. In the next step, for each function, the compiler calls the *gate* method of the plugin class. Based on the programmer's requirements, this method decides whether or not the supplied function should be protected by an error detection approach. The *execute* method is invoked if the gate method returns true. It will assess the plugin-specific compiler options, launch *DFED Creator*, and execute the *implTech* function from *DFED Creator*. This methodology is better known as the Factory Design pattern [30]. It is aware of the instruction sets the plugin supports and how the soft error detection techniques should be implemented. First, the employed instruction set is recognized. Then, the *DFED Creator* object instantiates the class of the given method (S-SWIFT-R in Figure 2). All classes using *DFED techniques* derive from the *abstract GeneralDFED* class and are implemented through its *implement()* method. This method calls the *insertError()*, *insertSetup()* and *implementDupComp()* methods. The *insertError()* method inserts a call to the *DFED error handler* at the conclusion of the evaluated function and puts a label at the insertion point. When an error is discovered, the plugin may subsequently utilize this label to branch to this call. *insertSetup()* and *ImplementDupComp()* are abstract functions implemented in the S-SWIFT-R classes described in the next section. This method guarantees that the algorithm's structure stays constant, while subclasses incorporate technique-specific phases.

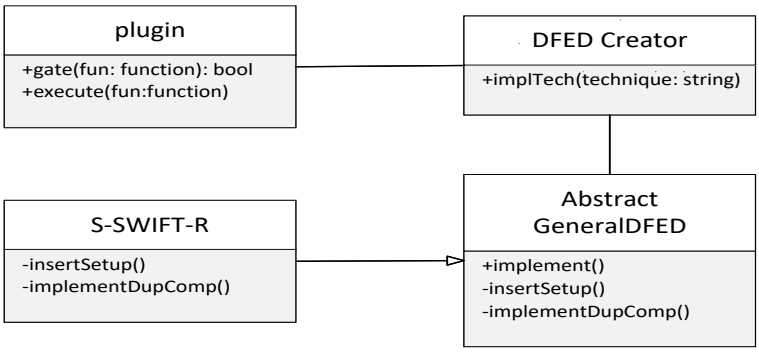

Figure 2. UML class diagram of the S-SWIFT-R compiler plugin.

The S-SWIFT-R GCC Plugin Implementation

As stated in the previous section, the *insertSetup()* method of S-SWIFT-R will be invoked first. The method will inject shadow register initialization instructions. Using a register map, shadow registers are mapped to the protected registers.

Then, the *implementDupComp* method is called. The implementation of this approach is shown in Algorithm 1. It begins by evaluating all of the instructions in the function and duplicating those that should be replicated (lines 3 and 4). This means duplicating the instruction twice, replacing the utilized registers with their equivalent shadow registers, and inserting the duplicate after the original instruction. Any instruction in S-SWIFT-R that affects the value of the intended register should be duplicated. As a result, compare instructions and control flow instructions, such as branch instructions, are not repeated. Store, push, and pop instructions are exceptions because repeated instructions may push or pop the erroneous value onto or off the stack. However, when pop instructions change the value of a register, the accompanying shadow registers must also be changed (lines 5 and 6). Furthermore, pop instructions will change the stack pointer. As a result, the shadow registers of this stack reference should be updated after each pop instruction. Push commands follow a similar logic (lines 7 and 8).

Now that the instructions have been replicated, the accuracy of the computations may be checked by comparing the original registers to the shadow registers at various synchronization points in the program code. Whenever the value of the original register is required during the program execution, the two shadow registers are compared. If they match, they must be accurate, thus they are copied back to the original register to

remedy any potential corruption. If they do not match, the value of the original register is transferred to both shadow registers (from lines 9 to 15). This will allow the S-SWIFT-R to recover from a single event upset. One last point to consider while implementing S-SWIFT-R is call handling. When a function call happens, program control is passed to a subroutine that may or may not be protected by the error detection approach. This implies that when the subroutine returns, the actual registers may have changed but the shadow registers may not have, leading the program to pass control to the DFE error handler improperly at the next synchronization point. After each function call, all shadow registers are re-initialized. Due to the fact that a function call is already a synchronization point, this has no significant impact on the error detection capacity (line 16). The complexity of the algorithm is $O(n)$. After explaining the methodology and automatic process of implementation, in the next section, the results have been discussed and explained.

Algorithm 1 implementDupComp function of S-SWIFT-R.

```
1:  for all instr ∈ function do
2:      if shouldDuplicate(instr) then
3:          duplicate(instr)
4:      else if isPopInstruction(instr) then
5:          updateShadowRegistersAfter(instr)
6:      else if isPushInstruction(instr) then
7:          updateShadowSpRegisterAfter(instr)
8:  unsafeAreas ← findUnsafeAreas()
9:  for all instr ∈ function do
10:     if shouldCompareBefore(instr) then
11:         if instr ∉ unsafeAreas then
12:             addCmpBneBefore(instr)
13:     else
14:         addCmpBneBefore(unsafeAreas[instr])
15: initShadowRegsAfterEachFunctionCall()
```

5. Experimental Validation

To evaluate the two pre-selection methods for picking the registers and analyze the impact of increasing the number of registers to protect in the selection phase, we employed fault injection to test the fault tolerance characteristics of techniques. This section starts by going through the experiment setup and the case studies that are utilized. Following that, the measured criteria are described. Finally, the experiment's results are presented and evaluated.

5.1. Setup for the Experiment

To find the registers toward their interaction with the memory, the mentioned framework was built using the Imperas simulator. The Imperas simulator is an instruction set simulator that enables the execution of target instructions at the speed of the host. The DFE fault injector was used to inject faults into the registers, and after evaluating the results, a priority list ranking the registers from most to least vulnerable was compiled. This list determines which registers will be hardened. After choosing registers using one of these two approaches, our GCC plugin will add the necessary code to protect the selected registers in accordance with the S-SWIFT-R standards. After protection codes were included, the case studies were executed on a simulated ARM Cortex-M3 CPU, a typical 32-bit processor used in several embedded systems. In the following the list of used academic case studies has been included. Additionally, Table 3 shows which registers are protected in each case study for both S-SWIFT-R-F and S-SWIFT-R-M approaches.

- **Bit Count (BC):** The bit count algorithm, commonly known as the hamming weight, counts the bits set, or 1's, in a given data word. This feature is utilized in the commu-

nication domain to compute a parity bit and in the cryptography, area to create keys, among other applications.

- **Bubble Sort (BS)/Quicksort (QS)**: was chosen because it is employed in a range of applications, such as setting priorities or aiding quicker data processing.
- **Matrix Multiplication (MM):** Matrix multiplication was selected because matrices and matrix multiplication is used in a broad variety of embedded fields, including image processing (e.g., CAT and MRI scans), robotics, and data compression.
- **CRC 32:** The primary purpose of the cyclic redundancy check method is to provide error detection information for data transfers. When utilizing CRC, extra bits are added to the sent data, allowing for an analysis of whether or not the information received is accurate.
- **Cubic Solver (CU):** In physics-related applications, such as the estimation of a vehicle's speed or the power density of wind turbines, the cubic function solver method is used.
- **Dijkstra's algorithm (DIJ):** Dijkstra's method determines the shortest route between two or more nodes, a function that is widely used in routing applications,
- **Fast Fourier Transform (FFT):** The fast Fourier transform is used in several applications. including file reduction, speech recognition, and vibro-acoustic analysis in automobiles.

Table 3. The selected registers to protect for each academic case study.

Case Study	S-SWIFT-R-M	S-SWIFT-R-F	Case Study	S-SWIFT-R-M	S-SWIFT-R-F
BC	R2 R2, R4 R2, R4, R13 R2, R4, R13, R6	R3 R3, R5 R3, R5, R4 R3, R5, R4, R5	QS	R3 R3, R7 R3, R7, R0 R3, R7, R0, R1	R2 R2, R13 R2, R13, R8 R2, R13, R8, R4
BS	R2 R2, R0 R2, R0, R3 R2, R0, R3, R4	R4 R4, R1 R4, R1, R2 R4, R1, R2, R7	MM	R2 R2, R13 R2, R13, R3 R2, R13, R3, R9	R3 R3, R4 R3, R4, R2 R3, R4, R2, R5
CRC32	R0 R0, R8 R0, R8, R2 R0, R8, R2, R13	R5 R5, R13 R5, R13, R4 R1, R13, R4, R5	CU	R3 R3, R13	R4 R4, R8
FFT	R2 R2, R1	R6 R6, R10	DIJ	R3 R3, R2	R0 R0, R2

5.2. Criteria

In order to compare each approach and its outcomes, we assessed the following criteria: code size overhead and execution time overhead.

$$\text{code size relative to unprot. code} = \frac{\text{Inst.count.protected}}{\text{Inst.count.unprotected}} \tag{2}$$

$$\text{exec. time relative to unprot. code} = \frac{\text{exec.time.protected}}{\text{exec.time.unprotected}} \tag{3}$$

To see the impact of fault coverage offered by S-SWIFT-R, Single Event Upset (SEU), which is a type of soft error is injected into the system. The injected faults were categorized based on their impact on the projected system behavior into three categories:

- **Hardware Detection (HD)**: Numerous processors currently have several internal fault handlers capable of detecting particular hardware errors, such as inappropriate bus utilization or memory access violations. This category shows the errors that were noticed by the fault handler.

- **Silent Data Corruption (SDC)**: These are the faults that were not recognized by the implemented approach and thus led the algorithm to provide an incorrect result.
- **No Effect (NE)**: This is the proportion of faults that were not recognized or have been recovered by S-SWIFT-R and therefore did not influence the algorithm's intended output.

5.3. Fault Coverage

To show the influence of hardening on the program, we assessed not just a few hardened program versions, but all of the versions of both pre-selection techniques. The overall results of the experiments are shown in Figures 3–10, showing the impact of hardening one, two, three, and four (full cover) registers. For a detailed look, Appendix A includes the results for each case study. Note that for three and four registers we have to cut out FFT, CU, and DIJ case studies. For these case studies, the compiler was unable to create the programs with the low amount of registers that remained available when reserving the shadow registers necessary to protect three and four registers.

To initiate the experiment, the non-hardened programs are selected as the baseline since they reflect the worst-case scenarios. Secondly, the subsequent fault injection for each case study in both hardened and non-hardened case studies consisted of injecting 80,000 errors. Each error was simulated by flipping a single bit from the microprocessor register file at a random location. Figures 3–10 illustrate the effects of fault coverage. It is important to note that even in non-hardened programs the NE category is already present, showing the inherent resilience to these errors of the case studies. Concerning the importance of the protection case studies, we are interested in the SDC impact, as it demonstrates the effect of the protective approach. To have a more detailed look at the results, first, the CFE results are discussed. As can be seen in Figures 3, 5, 7 and 9, neither the S-SWIFT-R-F nor the S-SWIFT-R-M implementation of S-SWIFT-R is suited to detect CFEs. Although an SDC reduction is certainly achieved for both variants, even when four registers are protected, the remaining SDC ratio is between 10% and 20%. Techniques, such as CFCSS and RASM, achieve a remaining SDC ratio of 5% or less, so clearly, neither implementation deals with CFEs adequately.

When comparing S-SWIFT-R-F and S-SWIFT-R-M, it is clear from the results that overall selecting the registers to protect based on their vulnerability, i.e., S-SWIFT-R-F proves to provide better protection against CFEs. This is detailed in Table 4, which shows the SDC ratios for the unprotected case studies, S-SWIFT-R-M and S-SWIFT-R-F when protecting one register. As shown, the difference between these two approaches can be 0%, e.g., BS case study, it can be small, e.g., the CU case study where the difference is only 2%, but can be as large as 11%, e.g., CRC32 case study.

Table 4. The SDC amount in CFE fault injection.

Case Study	Non-Hardening	S-SWIFT-R-M	S-SWIFT-R-F
BC	61%	29%	26%
BS	38%	25%	25%
CRC32	68%	47%	36%
MM	65%	45%	30%
QS	47%	29%	27%
CU	48%	31%	29%
FFT	59%	28%	19%
DIJ	52%	24%	16%

The results for the DFE fault injection, Figures 4, 6, 8 and 10, show a similar story when comparing both approaches. Again, the S-SWIFT-R-F variant can achieve lower SDC ratios, i.e., it deals with DFEs better, than the S-SWIFT-R-M approach. One side-note, however, is that the results also show that when selecting the registers to protect based on their memory interaction, the achieved reduction in SDC is more case study independent. This

is indicated by the smaller box plots. Additionally, the DFE results show that there is little difference, for either approach, between protecting only one register (Figure 4) or protecting two registers (Figure 6). A noticeable extra SDC reduction is achieved when protecting three registers, shown in Figure 8. When protecting four registers, both approaches perform similarly with a remaining SDC ratio of approximately 4% to 5%, as indicated in Figure 10. These last sets of results are not surprising since this boils down to almost protecting all registers on our Cortex-M3 target. To protect 4 registers, 8 shadow registers are needed, meaning 12 of the available 14 registers (including the link register and stack pointer) are now used and protected.

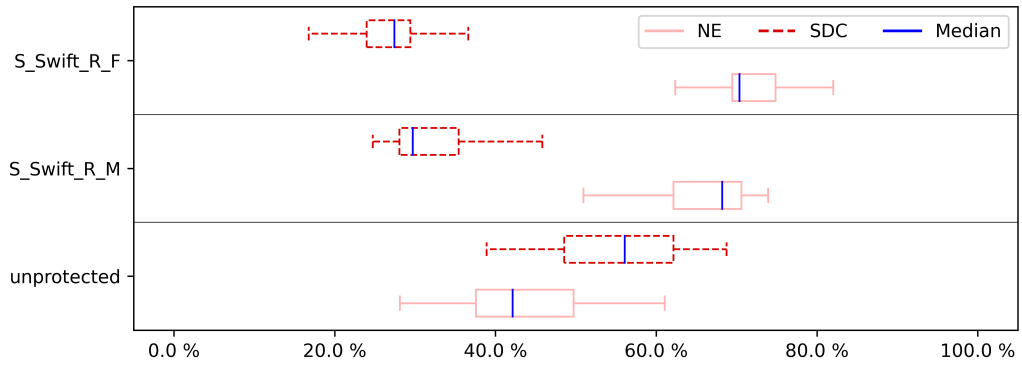

Figure 3. CFE fault injection results for 1 register.

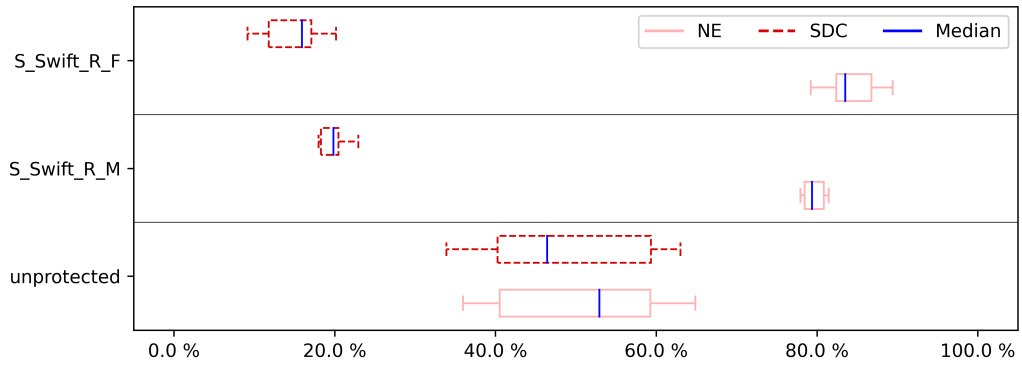

Figure 4. DFE fault injection results for 1 register.

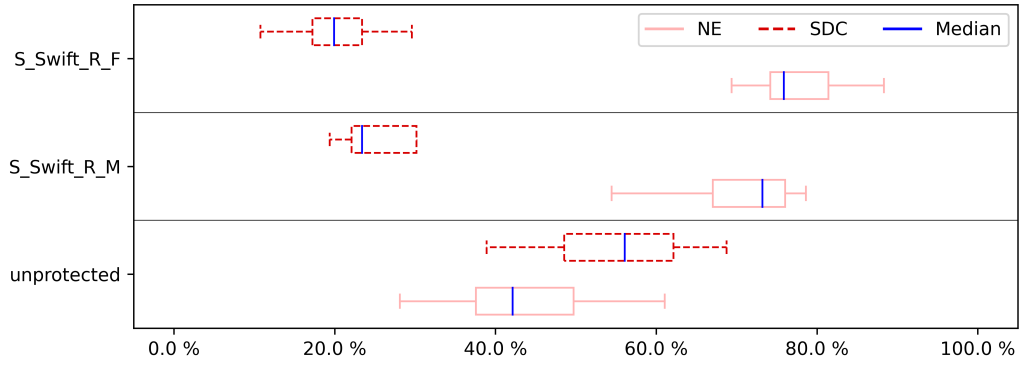

Figure 5. CFE fault injection results for 2 registers.

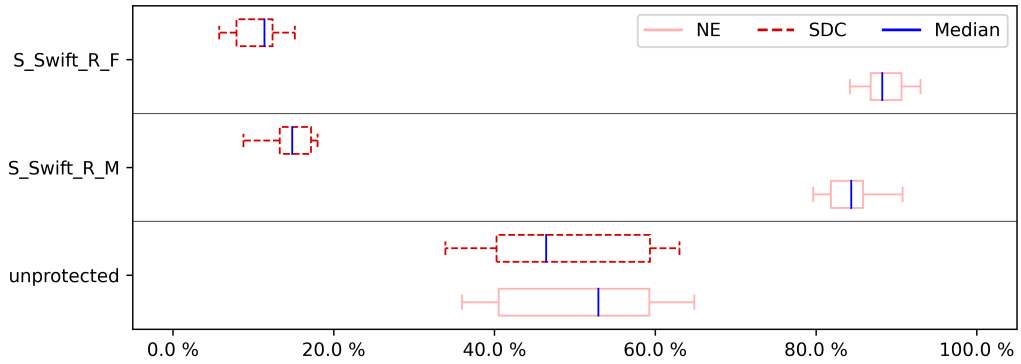

Figure 6. DFE fault injection results for 2 registers.

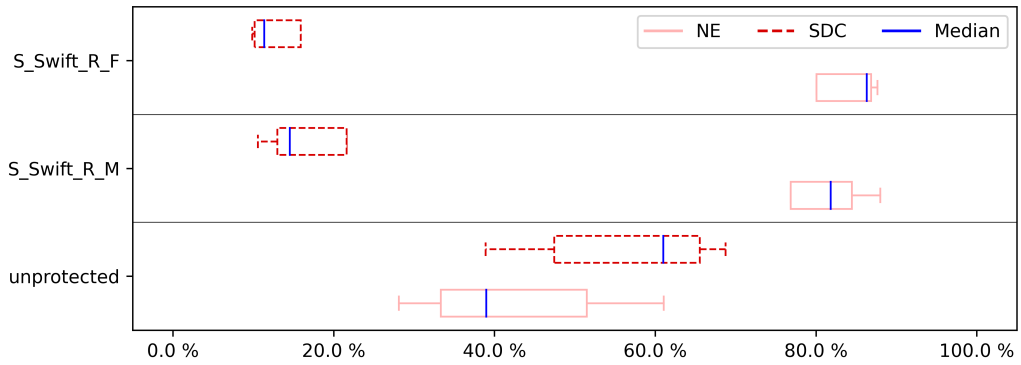

Figure 7. CFE fault injection results for 3 registers.

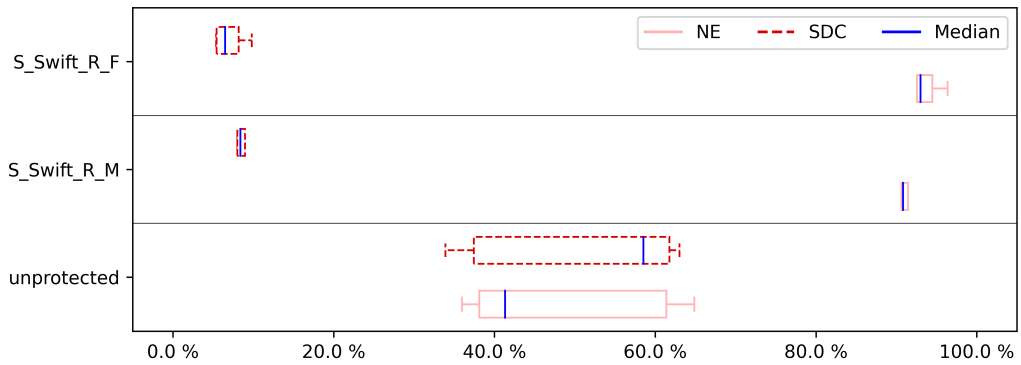

Figure 8. DFE fault injection results for 3 registers.

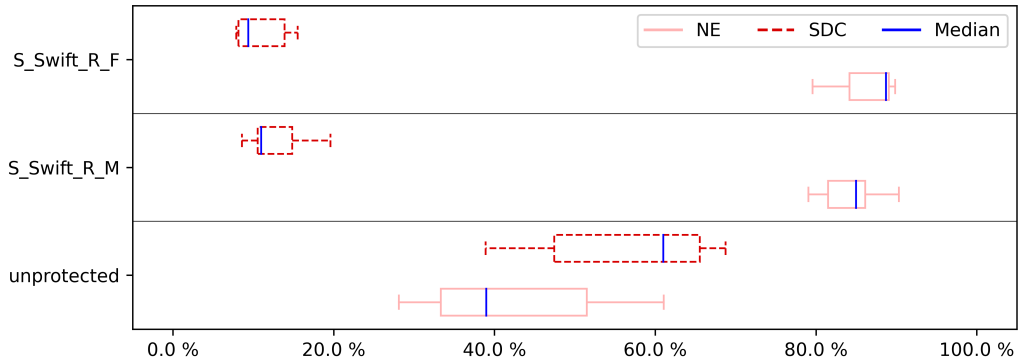

Figure 9. CFE fault injection results for 4 registers.

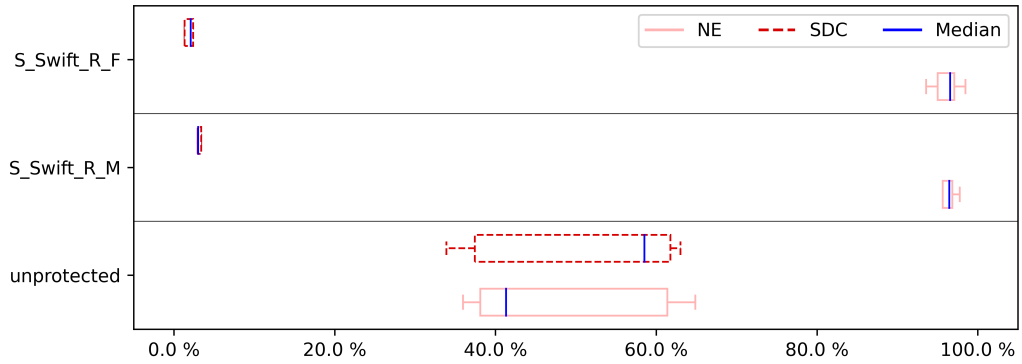

Figure 10. DFE fault injection results for 4 registers.

5.4. Overhead

Figures 11 and 12 present the overhead results for all the selectively hardened case studies. These results are normalized with a baseline built with the non-hardened version of each case study. The bar lines of green, blue, red, and black indicate the number of registers that have been hardened from 1 to 4, respectively. When highly accessed registers are protected, the overheads increase considerably (as expected). This can be observed clearly in BS, CRC, QS, and MM case studies when moving from 2 to 3 registers, which causes a high overhead when protected. For example, for these case studies, the code size and execution time overheads are 2.45×, 2.25×, 2.95×, and 2.75×, respectively.

Notice that the overhead results increase exponentially when more registers are protected. For example, in the BC case study, code size overhead goes from 1.1× (protecting one register) up to 1.8× in the fully protected version of all four registers, whereas execution time overhead ranges between 1.1× (protecting one register) and 2.2× (protecting all four registers). In the BS case study, code overhead varies from 1.4× (protecting one register) to 3.0× in the fully protected version, and execution time overhead ranges from 1.25× (protecting one register) to 3.30× (protecting all four registers), as the same for other case studies. If we only decide to protect one register in the BS case study, the overhead would be only impacted by 1.4× in comparison to all four registers that cost 3.0× extra overhead. It shows that selective hardening could be beneficial in a large number of applications in comparison to heavy full software protection techniques.

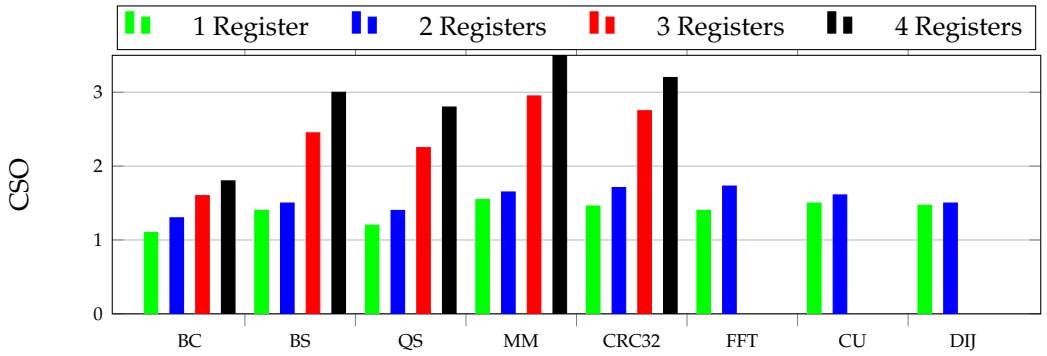

Figure 11. Measured code size for different case studies.

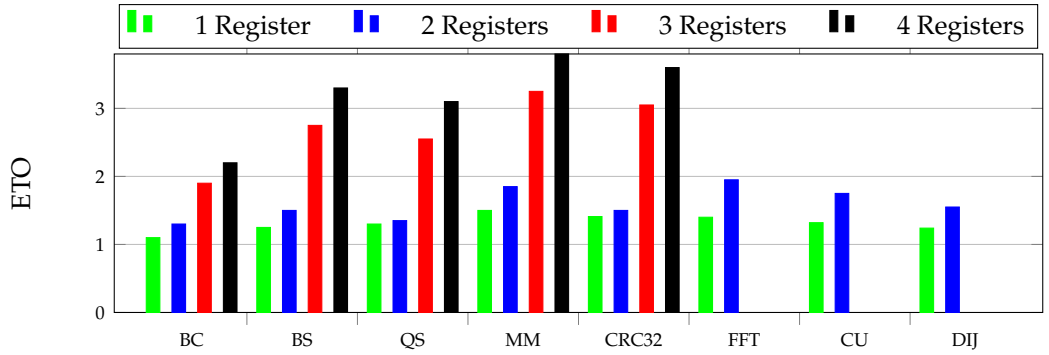

Figure 12. Measured execution time overhead for different case studies.

5.5. Industrial Case Study

In the last phase, to analyze the two approaches to select the desirable registers, they have been applied to an industrial case study in the form of our small scale factory [31]. The small-scale plant is divided into three stations: distribution, testing, and sorting. They form a closed process in which workpieces are pushed out of a stacked magazine and brought to the testing area, where only the best workpieces are transferred to the final station for color sorting. The small-scale factory may discriminate between six distinct product groups. It separates three categories of color: silver, red, and black, and for each color, there are right and improper workpieces. Each station, as their names imply, contributes to this process. A microcontroller powered by ARM Cortex-M3, the NXP LPC 1768, was chosen to operate each station and hence run the control software. Figures 13–16 show the results related to small scale factory for CFE, and DFE fault injection. For this case study, the variants in which 1 and 2 registers are protected are implemented and experimented. As with the CU, DIJ, and FFT case studies, this case study can also not be implemented when protecting 3 and 4 registers due to the aforementioned compiler error. Overall, these results tell a similar story as the results of the data processing case studies. Again, both for CFE and DFE detection, S-SWIFT-R-F outperforms its S-SWIFT-R-M counterpart in SDC reduction. However, the difference between the two register selection approaches is less clearly shown in the NE ratio, with both approaches achieving similar numbers. Especially in Figures 15 and 16, S-SWIFT-R-F even achieves the low remaining SDC ratio typically achieved by techniques specifically designed for CFE or DFE detection, while only protecting 2 registers. Similarly, the code size overhead and execution time overhead are illustrated in Figures 17 and 18, which indicates the enhanced reliability with an overall modest amount of overhead that is supposed to the system's runtime and memory. This clearly shows the strength of selective implementation and the achievement it could bring as a light software protection mechanism. In the following, the future works and plans have been explained.

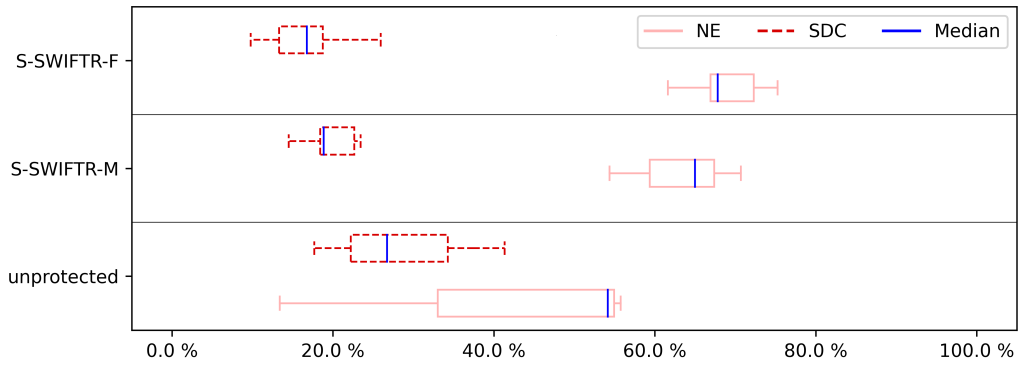

Figure 13. CFE small scale factory fault injection results for 1 register.

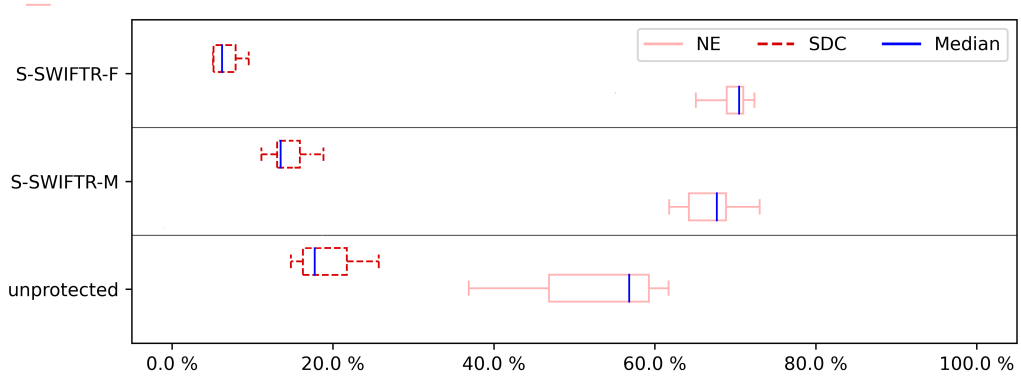

Figure 14. DFE small scale factory fault injection results for 1 register.

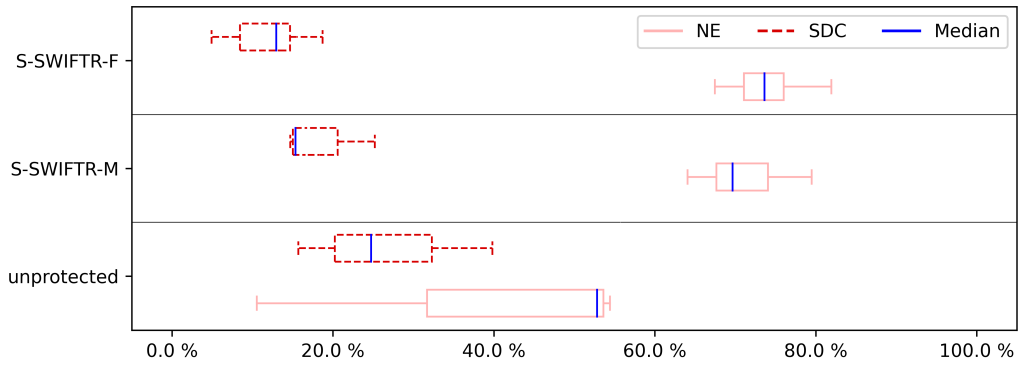

Figure 15. CFE small scale factory fault injection results for 2 registers.

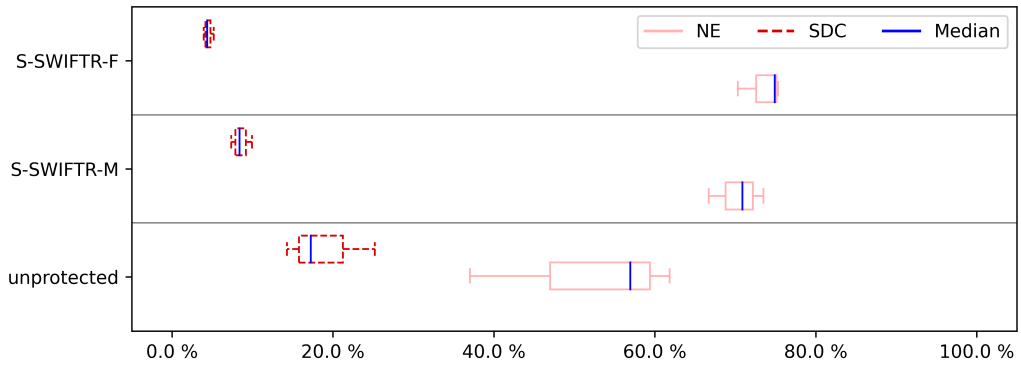

Figure 16. DFE small scale factory fault injection results for 2 registers.

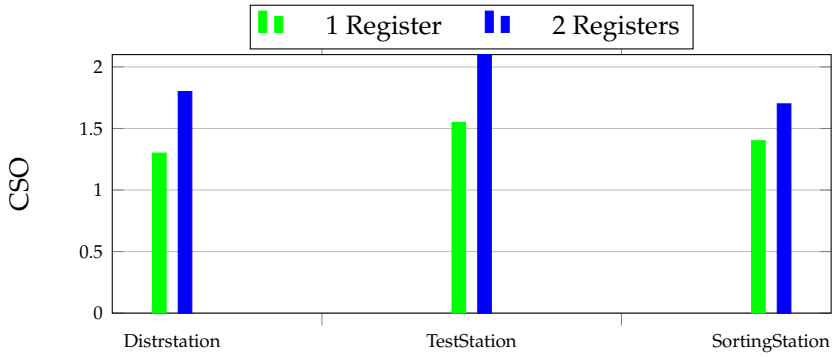

Figure 17. Measured code size overhead.

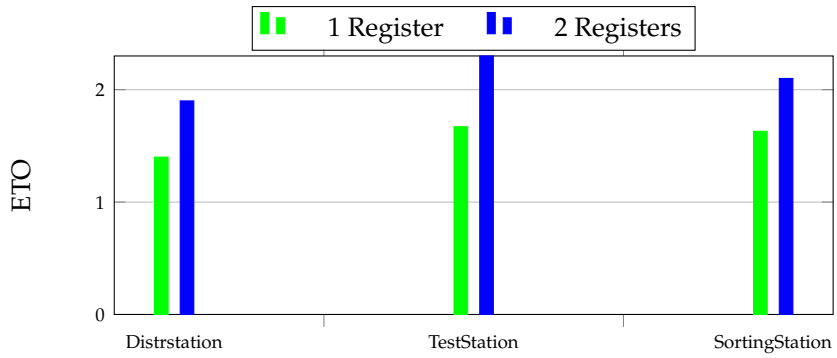

Figure 18. Measured execution time overhead.

6. Future Work

There are two outlines for future research: one examines the effect of developing S-SWIFT-R in a hybrid environment alongside hardware resilience techniques, and the other is investing in further selective techniques and compares the procedures to acquire the best outcomes. As the first outline, it is interesting to examine what the ideal hybrid combination of S-SWIFT-R and hardware protection methods could be. Therefore, the cost of using hardware techniques will be drastically reduced, as they will only be used on selective parts of the program, while the protection could reach its highest percentage. In the second phase, it is engaging to invest in additional selecting approaches and compare the corresponding processes in order to obtain the greatest results. The methods, such as S-SETA [32], S-RACFED [28], and S-DETECTOR [33], could be used in future studies to see the impact of CFE and DFE techniques on the concept of selective hardening and the impact of that on the variety of case studies.

7. Conclusions

This article examines different criteria for achieving selective hardening against soft faults using the S-SWIFT-R technique, which is a software hardening technique. The goal is to provide a selective strategy that preserves resources while providing maximum fault coverage with little overhead. The analysis is undertaken from two angles to achieve this goal: in the first case, the influence of two commonly used measures for picking registers was investigated as follows: (1) choosing registers based on memory interaction vs. (2) choosing registers based on susceptibility to the fault inserted into the system. The findings reveal that, although the differences between these two approaches are minor, picking registers depending on the fault introduced into the system may yield a superior overall result. In the second part, we investigated the effect of increasing the number of registers to protect S-SWIFT-R.

The improvement in overall dependability concerning system overhead has been investigated. A variety of academic case studies have been analyzed. The overall results show that increasing the number of registers to protect reliability will improve however, the code size overhead and execution time overhead will increase as well. By the impact shown in the result section, the designer could estimate the overhead cost that is in balance with increased reliability for its application. Additionally, alongside with two goals of the paper, we displayed that implementing a soft error detection system manually is a time-consuming, slow, and mistake-prone operation. As a result, we demonstrated our GCC compiler extension for S-SWIFT-R. This plugin interacts with RTL, a low-level intermediate representation that may be immediately converted into machine code or assembly code. After demonstrating and discussing the underlying workings of the compiler plugin, we proved that the low-level implementations of S-SWIFT-R function as intended by executing fault injection experiments on seven separate case studies. Using the compiler extension reduces the work and time required to implement the S-SWIFT-R approach greatly.

Author Contributions: Funding acquisition, J.B.; Investigation, M.N.; Methodology, M.N.; Resources, J.V. and J.B.; Software, M.N. and J.V.; Supervision, J.V. and J.B.; Writing—original draft, M.N.; Writing—review and editing, J.V. and J.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research has received funding from the European Union's EU Framework Programme for Research and Innovation Horizon 2020 under Grant Agreement No. 812.788 (MSCA ETN SAS). Project website: https://etn-sas.eu (accessed on 17 Augest 2022). This research is partially funded by the Research Fund KU Leuven.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

In this part, the results for all case studies in detail have been presented to give the reader a close look into the results and their impact on different case studies. The results presented on CFE and DFE fault injection in different case studies are represented in Figures A1–A8, and as it is shown when the number of registers to protect is increased toward A5 to A8 there are not enough shadow registers for CU, FFT, and DU case studies.

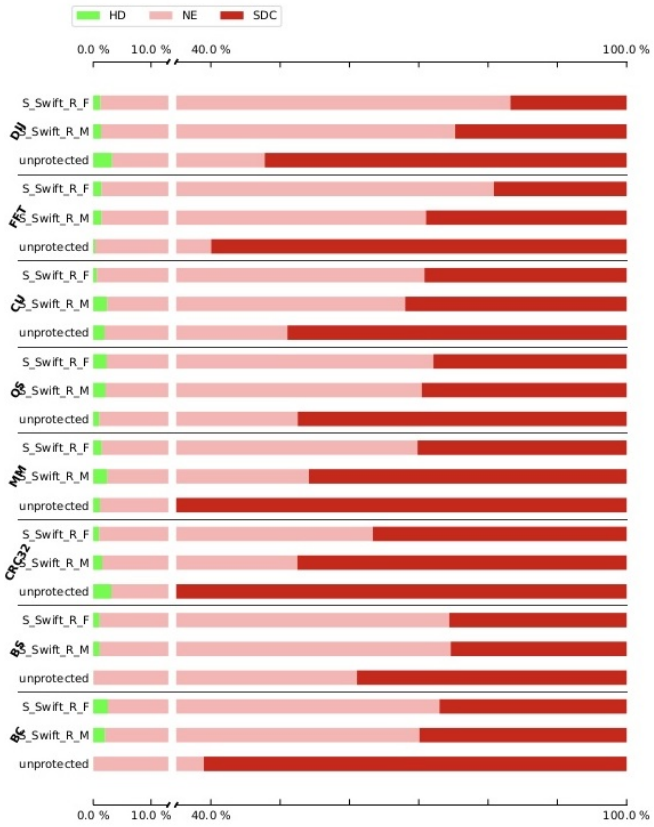

Figure A1. CFE fault injection results for 1 register.

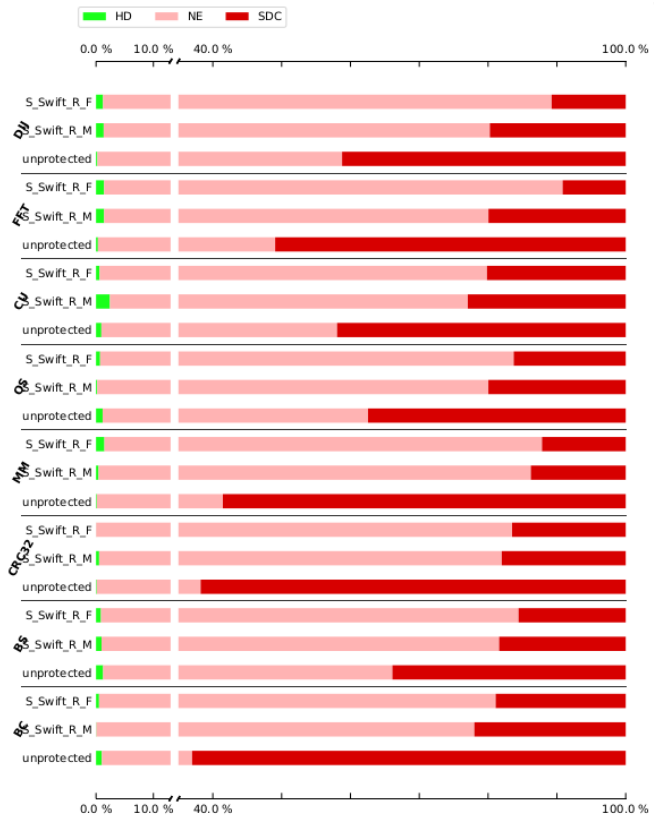

Figure A2. DFE fault injection results for 1 register.

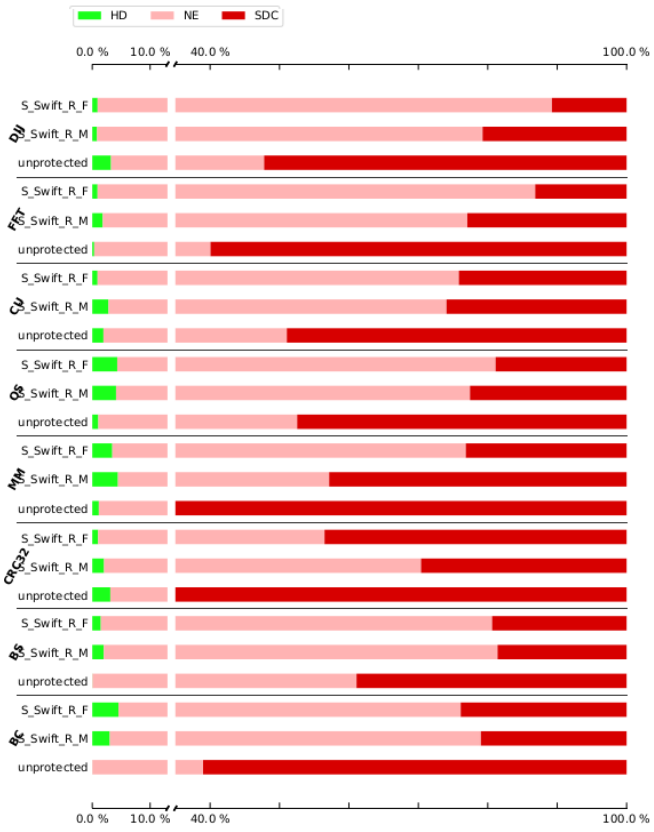

Figure A3. CFE fault injection results for 2 registers.

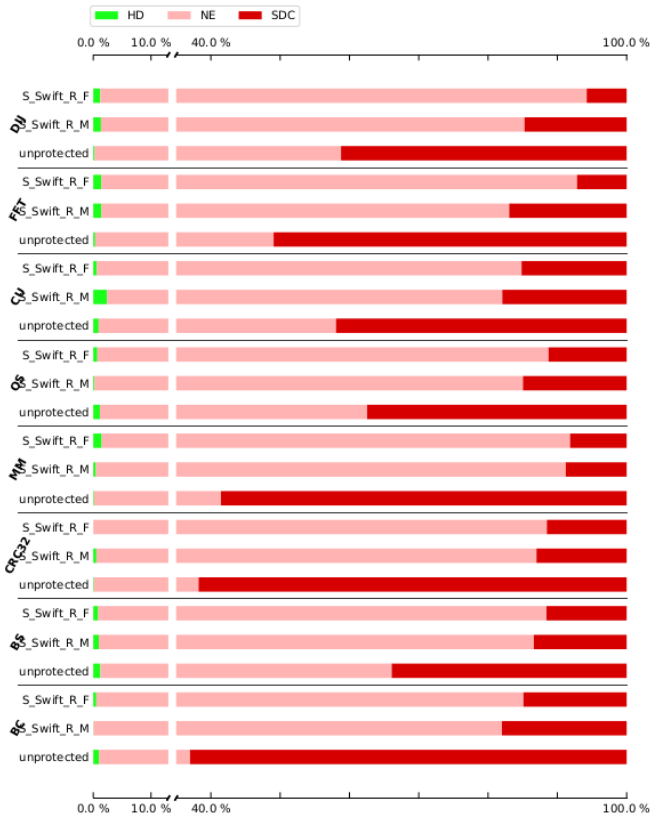

Figure A4. DFE fault injection results for 2 registers.

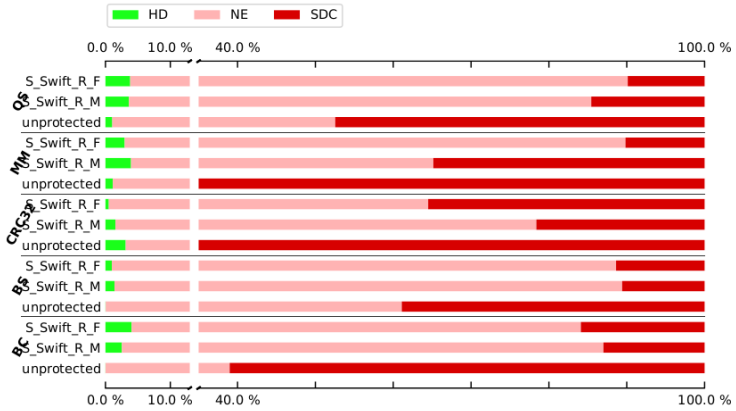

Figure A5. CFE fault injection results for 3 registers.

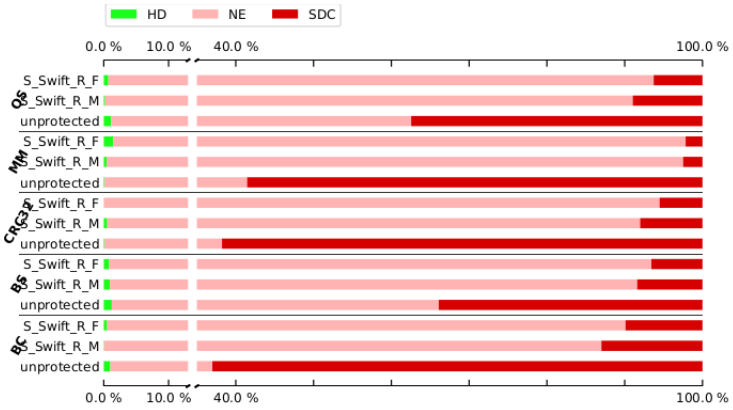

Figure A6. DFE fault injection results for 3 registers.

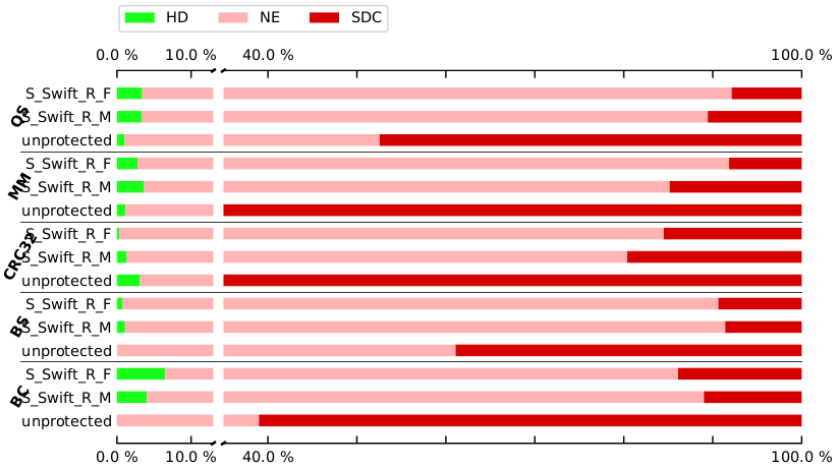

Figure A7. CFE fault injection results for 4 registers.

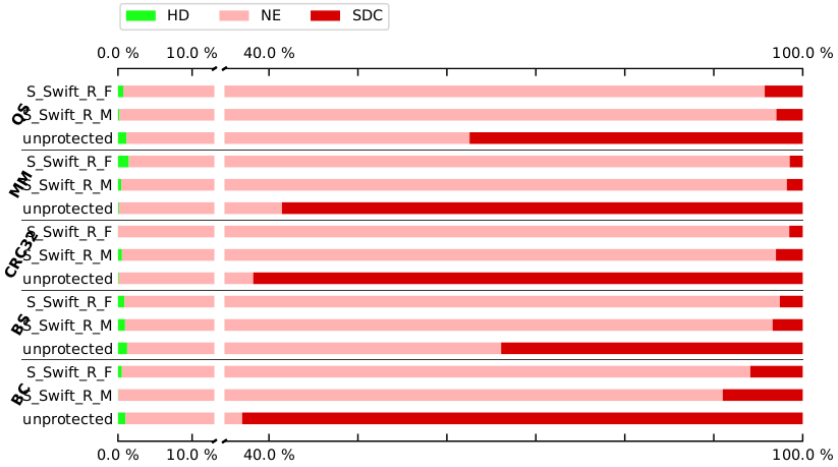

Figure A8. DFE fault injection results for 4 registers.

References

1. Blaškovic, K.; Candrlic, S.; Jakupovic, A. Systematic Review of Methodologies for the Development of Embedded Systems. *Int. J. Adv. Comput. Sci. Appl.* **2021**, *12*, 410–420 . [CrossRef]
2. Cintra, L.R.; Mollon, M.F.; Kaneko, E.H.; Montezuma, M.A.F.; Mendonça, M. Development of a Wireless Data Acquisition System for Application in Real-Time Closed-Loop Control Systems. *Int. J. Adv. Eng. Res. Sci.* **2018**, *5*, 223–231. [CrossRef]
3. Lombardi, M.; Milano, M.; Benini, L. Robust Scheduling of Task Graphs under Execution Time Uncertainty. *IEEE Trans. Comput.* **2013**, *62*, 98–111. [CrossRef]
4. Edwards, R.; Dyer, C.; Normand, E. Technical standard for atmospheric radiation single event effects, (SEE) on avionics electronics. In Proceedings of the 2004 IEEE Radiation Effects Data Workshop (IEEE Cat. No.04TH8774), Atlanta, GA, USA, 22 July 2004; pp. 1–5. [CrossRef]
5. Thati, V.B.; Vankeirsbilck, J.; Boydens, J.; Pissort, D. Selective Duplication and Selective Comparison for Data Flow Error Detection. In Proceedings of the 2019 4th International Conference on System Reliability and Safety (ICSRS), Rome, Italy, 20–22 November 2019; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2019; pp. 10–15. [CrossRef]
6. Reddy, V.K.; Rotenberg, E.; Parthasarathy, S. Understanding prediction-based partial redundant threading for low-overhead, high- coverage fault tolerance. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems—ASPLOS-XII, San Jose, CA, USA, 21–25 October 2006; p. 83. [CrossRef]
7. Geetha, S.; Kumar, K.K.; Rao, C.R.; Vijayan, M.; Trivedi, D.C. EMI shielding: Methods and materials—A review. *J. Appl. Polym. Sci.* **2009**, *112*, 2073–2086. [CrossRef]
8. Liu, Q.; Jung, C.; Lee, D.; Tiwari, D. Compiler-Directed Soft Error Detection and Recovery to Avoid DUE and SDC via Tail-DMR. *ACM Trans. Embed. Comput. Syst.* **2017**, *16*, 1–26. [CrossRef]
9. Restrepo-Calle, F.; Martínez-Álvarez, A.; Cuenca-Asensi, S.; Jimeno-Morenilla, A. Selective SWIFT-R: A Flexible Software-Based Technique for Soft Error Mitigation in Low-Cost Embedded Systems. *J. Electron. Test.* **2013**, *29*, 825–838. [CrossRef]
10. Nicolaidis, M. Design for soft error mitigation. *IEEE Trans. Dev. Mater. Reliab.* **2005**, *5*, 405–418. [CrossRef]

11. Lin, S.; Kim, Y.B.; Lombardi, F. A 11-Transistor Nanoscale CMOS Memory Cell for Hardening to Soft Errors. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2011**, *19*, 900–904. [CrossRef]
12. Martinez-Alvarez, A.; Cuenca-Asensi, S.; Restrepo-Calle, F.; Pinto, F.R.P.; Guzman-Miranda, H.; Aguirre, M.A. Compiler-Directed Soft Error Mitigation for Embedded Systems. *IEEE Trans. Dependable Secur. Comput.* **2012**, *9*, 159–172. [CrossRef]
13. Restrepo-Calle, F.; Cuenca-Asensi, S.; Martínez-Álvarez, A.; Chielle, E.; Kastensmidt, F. Efficient metric for register file criticality in processor-based systems. In Proceedings of the 2014 15th Latin American Test Workshop-LATW, Fortaleza, Brazil, 12–15 March 2014. [CrossRef]
14. Vankeirsbilck, J.; Penneman, N.; Hallez, H.; Boydens, J. Random Additive Control Flow Error Detection. In Proceedings of the International Conference on Computer Safety, Reliability, and Security, Västerås, Sweden, 19–21 September 2018; Springer: Cham, Switzerland, 2018; Volume 11093, pp. 220–234. [CrossRef]
15. Oh, N.; Mitra, S.; McCluskey, E. ED/sup 4/I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.* **2002**, *51*, 180–199. [CrossRef]
16. Guzman-Miranda, H.; Aguirre, M.; Tombs, J. Noninvasive Fault Classification, Robustness and Recovery Time Measurement in Microprocessor-Type Architectures Subjected to Radiation-Induced Errors. *IEEE Trans. Instrum. Meas.* **2009**, *58*, 1514–1524. [CrossRef]
17. Kumar, P.; Garg, R. Checkpointing Based Fault Tolerance in Mobile Distributed Systems. *Int. J. Res. Rev. Comput. Sci.* **2010**, *1*, 83–93.
18. Khudia, D.S.; Mahlke, S. Low cost control flow protection using abstract control signatures. *ACM Sigplan Not.* **2013**, *48*, 3–12. [CrossRef]
19. Zhu, D.; Aydin, H. Reliability effects of process and thread redundancy on chip multiprocessors. In Proceedings of the 36th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Philadelphia, PA, USA, 25–28 June 2006.
20. Rashid, F.; Saluja, K.; Ramanathan, P. Fault tolerance through re-execution in multiscalar architecture. In Proceedings of the International Conference on Dependable Systems and Networks, DSN 2000, New York, NY, USA, 25–28 June 2000; pp. 482–491. [CrossRef]
21. Thati, V.B.; Vankeirsbilck, J.; Penneman, N.; Pissoort, D.; Boydens, J. An Improved Data Error Detection Technique for Dependable Embedded Software. In Proceedings of the 2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC), Taipei, Taiwan, 4–7 December 2018; pp. 213–220. [CrossRef]
22. Didehban, M.; Shrivastava, A. nZDC: A Compiler technique for near Zero Silent data Corruption Moslem. In Proceedings of the 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), New York, NY, USA, 5–9 June 2016; pp. 1–6. [CrossRef]
23. Isaza-González, J.; Restrepo-Calle, F.; Martínez-Álvarez, A.; Cuenca-Asensi, S. SHARC: An efficient metric for selective protection of software against soft errors. *Microelectron. Reliab.* **2018**, *88–90*, 903–908. [CrossRef]
24. Arasteh, B.; Bouyer, A.; Pirahesh, S. An efficient vulnerability-driven method for hardening a program against soft-error using genetic algorithm. *Comput. Electr. Eng.* **2015**, *48*, 25–43. [CrossRef]
25. Nikseresht, M.; Vankeirsbilck, J.; Pissort, D.; Boydens, J. A Selective Soft Error Protection Method for COTS Processor-based Systems. In Proceedings of the 2021 International Scientific Conference Electronics (ET), Sozopol, Bulgaria, 15–17 September 2021.
26. Vankeirsbilck, J.; Hallez, H.; Boydens, J. Automatic Implementation of Control Flow Error Detection Techniques. In Proceedings of the 2019 3rd International Symposium on Computer Science and Intelligent Control, Amsterdam, The Netherlands, 25–27 September 2019; pp. 1–8. [CrossRef]
27. Stallman, R. *Using GCC: The GNU Compiler Collection Reference Manual*; Free Software Foundation: Boston, MA, USA, 2003.
28. Vankeirsbilck, J.; Van Waes, J.; Hallez, H.; Boydens, J. Impact of selectively implementing control flow error detection techniques. *Internet Things* **2020**, *12*, 100260. [CrossRef]
29. De Blaere, B.; Verstappe, E.; Vankeirsbilck, J.; Boydens, J. A Compiler Extension to Protect Embedded Systems Against Data Flow Errors. In Proceedings of the 2021 XXX International Scientific Conference Electronics (ET), Sozopol, Bulgaria, 15–17 September 2021; pp. 1–6. [CrossRef]
30. Gamma, E. (Ed.) *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Professional Computing Series; Addison-Wesley: Reading, MA, USA, 1995.
31. Vankeirsbilck, J. Advancing Control Flow Error Detection Techniques for Embedded Software using Automated Implementation and Fault Injection. Ph.D. Thesis, KU Leuven, Leuven, Belgium, 2020.
32. Chielle, E.; Rodrigues, G.S.; Kastensmidt, F.L.; Cuenca-Asensi, S.; Tambara, L.A.; Rech, P.; Quinn, H. S-SETA: Selective Software-Only Error-Detection Technique Using Assertions. *IEEE Trans. Nucl. Sci.* **2015**, *62*, 3088–3095. [CrossRef]
33. Nikseresht, M.; De Blaere, B.; Vankeirsbilck, J.; Pissoort, D.; Boydens, J. Impact of Selective Implementation on Soft Error Detection Through Low-level Re-execution. In Proceedings of the 2021 IEEE International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, International Conference on Cloud and Big Data Computing, International Conference on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), Calgary, AB, Canada, 25–28 October 2021; pp. 112–117. [CrossRef]