Journal of Electronic Testing manuscript No.
(will be inserted by the editor)

Selective SWIFT-R

A flexible software-based technique for soft error mitigation in low-cost embedded systems

Felipe Restrepo-Calle · Antonio Martínez-Álvarez · Sergio Cuenca-Asensi · Antonio Jimeno-Morenilla

Received: April 2013 / Accepted: date

Abstract Commercial off-the-shelf microprocessors are the core of low-cost embedded systems due to their programmability and cost-effectiveness. Recent advances in electronic technologies have allowed remarkable improvements in their performance. However, they have also made microprocessors more susceptible to transient faults induced by radiation. These non-destructive events (*soft errors*), may cause a microprocessor to produce a wrong computation result or lose control of a system with catastrophic consequences. Therefore, *soft error* mitigation has become a compulsory requirement for an increasing number of applications, which operate from the space to the ground level. In this context, this paper uses the concept of *selective hardening*, which is aimed to design reduced-overhead and flexible mitigation techniques. Following this concept, a novel flexible version of the software-based fault recovery technique known as *SWIFT-R* is proposed. Our approach makes possible to select different registers subsets from the microprocessor register file to be protected on software. Thus, design space is enriched with a wide spectrum of new partially protected versions, which offer more flexibility to designers. This permits to find the best tradeoffs between performance, code size, and fault coverage. Three case studies have been developed to show the applicability and flexibility of the proposal.

Keywords Fault tolerance · Reliability · Embedded systems · Soft errors · Single Event Upset (*SEU*) · COTS electronic components

This work was funded by the Ministry of Science and Innovation in Spain with the project *'RENASER+: Integral Analysis of Digital Circuits and Systems for Aerospace Applications'* (`TEC2010-22095-C03-01`).

Authors are with the Computer Technology Department, University of Alicante, Carretera San Vicente del Raspeig s/n, 03690 Alicante, Spain. E-mail: {frestrepo, amartinez, sergio, jimeno}@dtic.ua.es.

1 Introduction

In recent decades, major technological advances in the development of microprocessors have occurred. Some of these developments include the dramatic increase in their performance, and the ever increasing integration density. These were made possible mainly thanks to the progressive miniaturization of electronic components. Nevertheless, this fact has also led to some adverse consequences. One of the most concerning is that, due to the reduction of the electronic components size to nanometric scales, voltage source levels and noise margins have also been reduced, which has caused electronic devices to become less reliable and, therefore, microprocessors to be more susceptible to several types of faults, especially those induced by radiation [1].

Radiation effects on electronic components can cause catastrophic consequences in mission-critical systems. Radiation-induced faults are originated by the impact of high-energy particles against the electronic components which may result, directly or indirectly, in the ionization of their internal silicon structures. These events can affect the components operation permanently (*permanent faults*) or temporary (*transient faults*). In this paper, we will focus on the latter. *Transient faults* do not result in permanent damage, but may affect the system behavior by altering temporarily signal transfers or stored values. These are also known as *soft errors*. Specifically, we will focus on the type of *transient faults* known as *Single Event Upset* (*SEU*), which is characterized by the logic state alteration of a single

memory element in the system [2]. *SEUs* have usually been considered as a concern for space application systems, because it is in outer space where these are more frequent. However, in recent decades, this problem has been extended to the electronic circuits that must operate in the atmosphere [3], and even at ground level [4].

Moreover, thanks to their programmability, performance and cost-effectiveness, Commercial Off-The-Shelf (*COTS*) electronic components offer important capabilities and benefits in the implementation of low-cost safety-critical and high availability systems, such as micro-satellites [5,6] or avionics safety systems [7]. Nevertheless, their high sensitivity to radiation-induced effects, particularly transient faults, limit their applicability in the near future. At this juncture, *soft error* mitigation has become a mandatory issue for an increasing number of application domains (avionics, automotive, defense, medicine, and communications) [1, 8].

To overcome the problems produced by *soft errors*, applying fine-grain redundant hardware has been the usual solution in qualified *RadHard* microprocessors. However, *COTS* components prevent the application of hardware-based soft error mitigation methods directly within the processor, therefore other approaches have to be adopted. Duplication or triplication of *COTS* components are the most usual coarse-grain hardware alternatives for these kind of embedded systems [9]. Despite the majority of these hardware-based approaches provide an effective solution to the *transient faults*, in general, these techniques have serious drawbacks to the system in terms of used resources, power consumption, die size, design time, and economic costs; limiting their use in low-cost and small systems.

In recent years, however, several proposals based on redundant software have been developed, adding both detection and fault correction capabilities to programs. As software-based techniques do not require any modifications in the hardware of the microprocessor, they are particularly suitable for *COTS* based systems ensuring an acceptable dependability level [10–12]. In fact, some of these approaches have already been used in mission critical systems, including *COTS* microprocessors, for satellites and space missions [7].

Although software-based approaches are more cost-effective than the hardware-based ones, they provoke a non-negligible overhead to the programs in terms of execution time and code size [13]. In many cases, this is the main difficulty for the software-based techniques feasibility. In order to reduce these overheads and to offer more flexibility to designers, recent works have proposed the *selective hardening based on software* [14–16]. This consists of protecting only specific parts of the program or the microprocessor architectural resources (reachable from the instruction set architecture - *ISA*) by means of redundant software. Protected parts can be chosen according to their vulnerability or their contribution to the overheads. In the first case, to prioritize the protection of the most vulnerable resources, and in the second case, to avoid causing a high impact to the system, such as a high overhead in memory or an unacceptable degradation of performance.

Based on this concept, we present a novel selective version of the software-based technique known as *SWIFT-R* [17], namely *selective SWIFT-R* (*S-SWIFT-R*). Our proposal allows applying the protection to different register subsets from the microprocessor register file looking for a reduction in the overheads but keeping a high fault coverage. The feasibility of the proposal is demonstrated by means of experimental results in three representative case studies.

As a result, this technique is suitable for low-cost dependable applications which use *COTS* microprocessors. Furthermore, the flexibility of our approach allows the designer applying the technique in an incremental way to explore the solutions space on the software side effectively. This not only facilitates to find a software version that best satisfies the dependability requirements, but also avoids the excessive overheads caused by usual software-based techniques.

The main contributions of this work can be summarized as follows. Firstly, it is proposed *S-SWIFT-R*: a new selective fault recovery software-based technique based on the well known *SWIFT-R*. The presented approach leverages the idea of selective hardening to enrich the (software-side) design space with a wide spectrum of new possibilities. The main improvement over SWIFT-R is determined by the increase of flexibility and the possibility to explore different solutions offering the best reliability/overhead compromise. This is a mandatory task for the low-cost dependable design of *COTS*-based applications. Secondly, it is presented a comprehensive set of experimental results which demonstrate the flexibility and applicability of *S-SWIFT-R* by representing several trade-offs between overheads and fault coverage.

This paper is organized as follows. Next section provides background information about the related work. Section 3 presents the proposed *soft error* mitigation technique: *S-SWIFT-R*. Section 4 reports and discusses the experimental results obtained in the proposal evaluation. Finally, Section 5 summarizes some concluding remarks and suggests the future works.

2 Related work

Hardware redundancy has traditionally been the most common approach to address reliability issues in the design of digital circuits. This includes a wide variety of solutions based on: *Error Detection and Correction Codes — EDACs* [18], gate-level logic redundancy [19, 20], and architectural level protection [21]. More recent techniques propose selective hardening of the system, adding protection only to the most vulnerable hardware parts [22]; or reducing the performance degradation by applying partial redundant threading [23,24].

In recent decades, thanks to the current proliferation of processor based systems and the need for dependable low-cost solutions, a large number of protection approaches based on the use of redundant software have emerged. The so-called *Software Implemented Hardware Fault Tolerance* (*SIHFT*) [25] techniques can be divided in two main categories according to the type of error they pretend to detect/correct: errors that may affect the control flow execution [26]; or errors that may affect the program data [27].

The first group is also known as *signature checking techniques*. Some examples of these include *Control-Flow Checking by Software Signatures* (*CFCSS*) [26], *Assertions for Control Flow Checking* (*ACFC*) [28], *Yet Another Control flow Checking Approach* (*YACCA*) [29], and *Control-flow Error Detection through Assertions* (*CEDA*) [30]. Moreover, those approaches included in the second group are mainly based on the *N-versions programming* approach [31], which can be applied at different granularity levels: program [11,32], procedure [33], and the most commonly used, instruction level [27, 34–36].

Most software-based approaches are aimed at detecting faults. Some of them apply redundancy to high-level source code (e.g., C language) by means of automatic transformation rules [37], whereas others use instruction redundancy at a low level (assembly code) in order to reduce the code overhead and performance degradation, and improve detection rates [26,27,36]. Only a few of these techniques have been extended to allow system recovery, but they incur, as a consequence, in higher overheads in terms of code size and execution time [17,38]. Overall, even though software-based techniques can be a suitable protection solution for low-cost *COTS*-based applications, they also could pose a design problem such as high overhead in memory, and disproportionate penalties in performance. This is especially true for those techniques that are addressed not only to fault detection but recovery tasks.

To reduce the implicit overheads of software-based techniques, a few works based on *selective hardening on software* have been proposed recently. They propose to transfer to the software world the concept of *selective hardening*, typical from the hardware world [22,39,40]. In [15,16], the authors propose the selective instruction replication to guarantee the application-level correctness in multimedia applications. This kind of applications can tolerate, in some cases, a execution which is not 100% numerically correct, and the program results can still appear to be correct from the user perspective [41]. In our case, mission-critical systems require the architecture-level correctness instead. In addition, the work presented in [14] uses the selective hardening on software focused on the detection of data-flow errors. Our proposal allows the system recovery as well.

The main advantage offered by selective software-based techniques is the flexibility. Designers are provided with a wide set of possibilities, being able to explore deeply the design space provided by the software techniques, taking into account factors such as code overhead, performance degradation, and reliability level. For instance, if applying a particular set of hardening routines results inconvenient according to the requirements of an application (e.g., if the maximum execution time is exceeded), the technique can be applied partially depending on the critical program resources or sections. In short, the designer is able to fine-tune a tailored fault mitigation strategy based on software.

Moreover, recent hybrid hardware/software fault mitigation approaches have shown promising results. These techniques combine software redundancy with additional hardware support [42–46]. In this context, the *S-SWIFT-R* technique here proposed can be used as a part of a more complex hybrid technique, or as a component of a cross-layer protection strategy. In fact, our previous works [47,48] present some preliminary results of this, when tailored hybrid approaches are used by combining partial protection on both hardware and software (*co-hardening*). Unlike our previous approaches that were aimed at the hardware/software co-design, the presented proposal in this work focuses on the presentation of a new selective software-only fault recovery technique suitable for low-cost *COTS*-based applications. In addition, a comprehensive experimentation is presented to support the proposal.

3 Selective SWIFT-R

Since memories are designed to reach the highest possible density, they are more sensitive to ionizing particles than other parts of the circuit. In addition, considering that they represent the largest parts of modern designs, memories are the first candidates to be hardened in a design. However, there already is a large

number of fault tolerance techniques, mainly based on *EDACs*, that may solve this problem properly [18]. In this work, therefore, we address the protection of the microprocessor register file due to both its criticality in microprocessor-based applications and the difficulties to protect it.

To do so, we focus on this issue by means of the technique proposed by Reis et al. known as *SWIFT-R* [17]. It is a software-only recovery technique based on low-level instruction transformation rules (assembly code), which is based on the well known *Triple Modular Redundancy* (*TMR*). *SWIFT-R* stands for *SoftWare Implemented Fault Tolerance - Recovery*. It intertwines three copies of the program and adds majority voting before critical instructions. In other words, it consists of the triplication of data and instructions, jointly with the insertion of verification points to check data consistency (by means of majority voters).

Fig. 1 presents an example of a basic program hardened using *SWIFT-R* (assembly code). Notice that register copies (`s0'`, `s0''`, ...) are stored in other available registers from the microprocessor register file, i.e., unused registers in the program. Furthermore, majority voters are recovery procedures that compare if at least two versions of a register have the same value, correcting the third copy (possibly corrupted).

#	*Non-hardened code*	*SWIFT-R code*
1	`main: LOAD s0, 00`	`main: LOAD s0, 00`
2		***Create*** `s0` ***copies***
3	`LOAD s1, 2A`	`LOAD s1, 2A`
4		***Create*** `s1` ***copies***
5	`ADD s0, s1`	`ADD s0, s1`
6		`ADD s0', s1'`
7		`ADD s0'', s1''`
8	`CALL incr`	`CALL incr`
9		***Majority voter for*** `s0`
10	`STORE s0, 00`	`STORE s0, 00`
11	`RETURN`	`RETURN`
12		
13	`incr: LOAD s2, 0F`	`incr: LOAD s2, 0F`
14		***Create*** `s2` ***copies***
15	`ADD s0, s2`	`ADD s0, s2`
16		`ADD s0', s2'`
17		`ADD s0'', s2''`
18	`RETURN`	`RETURN`

Fig. 1 Example hardened program using *SWIFT-R*

When applying software techniques, it is mandatory to take into account their needs in terms of microprocessor resources. This consideration may hinder the feasibility of the hardening in case the technique itself demands a lot of resources or if the microprocessor is very limited. Features that have to be considered include the number of available registers in the microprocessor register file to create redundant copies, and the amount of available space in the program memory for instructions replication. In case of *SWIFT-R*, two additional copies are required for each protected register. This is, a total of $2n$ additional registers are necessary to fully implement *SWIFT-R* (where n is the number of used registers in the non-hardened program). This fact makes that *SWIFT-R* may not result suitable in many cases for reduced-cost solutions. Furthermore, due to its fault recovery capabilities, *SWIFT-R* produces high overheads that, regarding the application, can easily surpass $3\times$ the original code size and execution time. To alleviate this problem, the original authors proposed to apply *SWIFT-R* to superscalar processors, in which the instruction level parallelism (*ILP*) can be exploited to execute redundant instructions and, in this way, the impact on performance can be diminished. Nevertheless, in case of low-cost solutions, where microprocessors usually have more resource limitations, another solution is required.

Thus, we propose several improvements to the original technique based on selective hardening in order to increase its flexibility and make it suitable for reduced-cost embedded systems. *S-SWIFT-R* is also applied by means of low-level instruction transformation rules but, in our approach, the strategy consists of applying software protection mechanisms only to some selectively chosen registers from the microprocessor register file. That is, it is possible to select several register subsets to be protected from all the registers, e.g., hardening only the most critical registers.

Taking advantage of the selectiveness on the application of *S-SWIFT-R*, designers can obtain different hardened versions of the same program exploring the design space in a fine-grained manner. The options were exactly two in the past: whether to use the non-hardened program or to use the hardened one; however, now there are many new possibilities between these two extremes, whose overheads are reduced but still may offer a high fault coverage. The number of versions (m) is equal to the number of possible combinations (without repetition) among the program used registers, i.e., $m = 2^n$, where, n represents the total number of used registers in the program (m includes the non-hardened version as well). Therefore, using S-SWIFT-R, design space is enriched with several new solutions, which offer more flexibility to designers, and facilitate to find the best trade-offs among reliability, performance, and code size.

In addition, this approach is useful as well in cases when is not possible to apply *SWIFT-R* completely. This can occur due to the limitations of the microprocessor (e.g., small number of registers available in the register file, reduced space in the program memory, ...) or the high demand for resource utilization in the pro-

gram (e.g., if the non-hardened code uses most registers available in the register file and, therefore, there are not enough available registers for the necessary redundant copies). In these cases, it is possible to prioritize the registers depending of their impact of overheads and/or reliability and protect only a subset of them.

To implement this novel approach successfully, we propose to use the concept of *Sphere of Replication* (*SoR*) in a flexible way. *SoR* was first proposed in [49]. The *SoR* defines the logic domain of redundant execution. This means that the architectural resources located within the *SoR* are considered to have redundant mechanisms; consequently, they are protected against faults. Hence, the *SoR* delimits the protection coverage of hardening techniques. Moving the borders of the *SoR*, it is possible to modify the protection level of different fault tolerance techniques by including or excluding various components inside the sphere.

One can include/exclude the memory subsystem, the microprocessor register file, or even select only a subset of critical registers from the microprocessor register file. For instance, in *EDDI* [27] the memory subsystem is considered to be located inside the *SoR*, so the instructions responsible to perform read/write operations over the memory do not cause that any data cross the *SoR* borders. In the same way, if the memory subsystem is considered to be outside of the *SoR* (as in *SWIFT* [36]), those instructions reading from memory or writing into memory are causing some data to cross through the sphere frontiers and must be handled in a special way. In our case, *S-SWIFT-R* allows to include/exclude selectively chosen subsets of registers form the register file.

To facilitate the proposal implementation we propose that the program instructions, whose execution imply a data flow crossing the borders of the *SoR*, have to be classified in a special way. In case only the microprocessor register file is located inside the *SoR*, when an instruction causes that some data enter inside the *SoR* (e.g., reading an input port, loading a value into a register or reading a value from memory), it is classified as *inSoR*. In contrast, when an instruction provokes data to go out from the *SoR* (e.g., writing on an output port, storing a value into the memory), it is classified as *outSoR*. Otherwise, instructions whose execution do not imply a data flow (e.g., an unconditional branch) are classified as *none*.

In the original *SWIFT-R*, the *SoR* is considered to hold the complete microprocessor register file. Thus, in Fig. 1, instructions 1, 3, and 13 are classified as *inSoR* and are followed by data replication instructions on the hardened code (lines: 2, 4, and 14). Instructions 5, 8, 11, 15, and 18 are classified as *none*. Instructions 5, and 15 perform an arithmetic operation, thus, they are replicated after the original instruction using the register copies (lines: 6, 7, 16, and 17). Finally, the only instruction that sends data outside of the *SoR* is the instruction number 10 (it sends data to the memory subsystem), and therefore, data should be verified before leaving the sphere by means of a majority voter (line number 9).

The application of *S-SWIFT-R* to a source code (assembly code) can be explained as follows:

1. Each program instruction is classified according to the direction of the data flow it provokes with regard to the *SoR*, whose elements should be previously defined. The architectural resources located within the *SoR* are considered to be protected against faults. Instructions are classified as:
   (a) *inSoR*: those instructions whose execution provokes a data flow entering to the *SoR*.
   (b) *outSoR*: those instructions whose functionality causes a data flow leaving the *SoR*.
   (c) *none*: those instructions whose execution do not imply a data flow (e.g., an unconditional branch) or those that provoke a data flow that does not cross the *SoR* borders.
2. Data triplication the first time that any data enter to the *SoR*. Therefore, for each instruction classified as *inSoR*, two additional copies will be created of the data entering to the sphere. These redundant copies have to be created by copying the register values, without repeating memory or input port accesses.
3. Triplication of instructions that perform any data operation (e.g., arithmetic, logic, shift, rotation instructions). Notice that redundant instructions should operate using register copies (replicated data).
4. Verify the correctness of the data involved in the instructions classified as *outSoR* before their execution. This verification is made by inserting majority voters and recovery procedures just before these instructions. This is necessary to avoid erroneous data leaving the sphere, because once the data have left the *SoR*, recovery will be not possible, and the corrupted data may cause a system error.
5. Special consideration should be given to instructions located before a conditional branch which alter the *ALU* flags (*zero*, *carry*, ...). Data involved in these instructions have to be checked as well (using again majority voters and recovery procedures before their execution). This verification is necessary because if a register value is corrupted, an operation using this register may produce an erroneous resultant flag, and consequently, this may provoke an incorrect branch somewhere in the program's control flow graph after the conditional branch execution.

6. Release redundant registers (copies) if they are not needed anymore in the rest of the program; otherwise, copies should be kept along the program execution. This condition implies a detailed analysis to the control flow graph.

This selective approach of *SWIFT-R* is made possible by the flexible implementation of the sphere of replication. Basically, the new proposal consists in moving out of the *SoR* the registers that are not required to be protected, while some other registers remain within the *SoR* and, consequently, code transformations are responsible for protecting only this subset of registers.

Observe that program instructions should be reclassified when elements within the *SoR* change (when some registers are considered within the *SoR* and others outside of it). Each instruction is classified according to the direction of the data flow it causes with regard to the new *SoR* components. Thus, instructions involving data stored in unprotected registers may cause some data to cross through the *SoR* frontiers. The data flow between two registers (in an instruction considered before as *none* because the data flow occurred inside the sphere) could have changed, in case one of the registers had been removed from the the *SoR*. Depending on the new data flow direction that instruction should be classified as *inSoR* or *outSoR*. For instance, in the instruction `ADD s0, s1`, where the `s0` register is considered to be located within the sphere, and the `s1` register is considered outside of it, while reading the data stored in the register `s1` to sum it to the data stored in `s0`, a data flow is produced from outside the sphere to the inside of it; therefore, the instruction `ADD` should be classified as *inSoR*. Similarly, in the case that `s0` was considered outside the sphere and `s1` inside of it, the produced data flow (when reading the value stored in `s1` to sum it to the data stored in `s0`) would be from inside of the sphere to outside of it and, thus, the `ADD` instruction should be classified as *outSoR*.

In order to illustrate the approach, Fig. 2 and Fig. 3 show an example with several versions of a basic program hardened using S-SWIFT-R in several register subsets. Notice that the fully hardened version obtained by *S-SWIFT-R*, i.e., the version with protection in all the program used registers ('`s0` and `s1` protected' version in Fig. 3), is the same than the one obtained by the original *SWIFT-R* approach.

Consideration should be given to the fact that if a fault affects data enclosed in the *SoR*, and then the corrupted data leave the *SoR* (as a result of the execution of an *outSoR* instruction), it may provoke an unrecoverable error because there will be no additional mechanisms outside of the *SoR* to detect the inconsistency.

#	*Non-hardened*	*Protected register:* `s0`
1	`LOAD s0, 00`	`LOAD s0, 00`
2		*Create* `s0` *copies*
3	`LOAD s1, 2A`	`LOAD s1, 2A`
4		
5		
6	`ADD s0, s1`	`ADD s0, s1`
7		`ADD s0', s1`
8		`ADD s0'', s1`
9		*Majority voter for* `s0`
10		
11	`STORE s0, (s1)`	`STORE s0, (s1)`

Fig. 2 Example hardened program using *S-SWIFT-R* ('non-hardened' and 'register `s0` protected' versions)

#	*Protected register:* `s1`	*Protected registers:* `s0,s1`
1	`LOAD s0, 00`	`LOAD s0, 00`
2		*Create* `s0` *copies*
3	`LOAD s1, 2A`	`LOAD s1, 2A`
4	*Create* `s1` *copies*	*Create* `s1` *copies*
5	*Majority voter for* `s1`	
6	`ADD s0, s1`	`ADD s0, s1`
7		`ADD s0', s1'`
8		`ADD s0'', s1''`
9		*Majority voter for* `s0`
10	*Majority voter for* `s1`	*Majority voter for* `s1`
11	`STORE s0, (s1)`	`STORE s0, (s1)`

Fig. 3 Example hardened program using *S-SWIFT-R* ('register `s1` protected' and 'registers `s0` and `s1` protected' versions)

Hence, it is necessary to verify the data correctness before leaving the sphere.

A particular case of this can be seen in Fig. 3 ('register `s1` protected' version) when a majority voter is inserted before the instruction `ADD s0, s1` (line 6). In this example, only the `s1` register is considered within the *SoR*. Therefore, when executing the instruction in line 6 (whose function is to do `s0` = `s0` + `s1`), there will be a data flow from `s1` to `s0`, or in other words, there will be a data flow from inside the SoR going outward; thus, a majority voter should be inserted to verify the correctness of the value stored in `s1`, before it leaves the sphere.

Moreover, it is worth mentioning that triplication of instructions imply the protection not only of the registers but also of all datapaths where instructions pass through. Replicas of instructions will pass all pipeline paths so they are all protected through not only specified register subset but all other components in the execution pipeline.

4 Evaluation

In order to evaluate the proposed technique, firstly, it was implemented using the *API* (Application Programming Interface) exposed by the *Software Hardening Environment* (*SHE*) (proposed in [47]). Secondly, the fault coverage of the approach was assessed using an *FPGA*

emulation-based fault injection tool called *FTUnshades* [50].

The experimental setup is described in the first part of this section. Afterwards, analysis with respect to the usage of registers in the non-hardened versions is discussed. Next, overhead results caused by the application of *S-SWIFT-R* are presented in the third part. Fault coverage results are remarked in the fourth part. Finally, the last part of this section presents experimental results for an incremental hardening strategy based on *S-SWIFT-R*.

4.1 Experimental setup

S-SWIFT-R has been implemented using the *Software Hardening Environment* (*SHE*). This a tool aimed to implement, automatically apply, and preliminarily evaluate software-based fault tolerance techniques. It comprises a flexible hardening multi-target compiler (source-to-source) and an instruction set simulator to assist the design decisions.

The hardened code generated by *SHE* was targeted to the *PicoBlaze* microprocessor [51]. *PicoBlaze* is a widely used *IP* (intellectual property core) with similar features to common 8-bit *COTS* microprocessors, and since it is a soft-core, it can be used within *FPGA*-based fault emulation tools in order to exhaustively assess the fault coverage in real conditions. Moreover, *PicoBlaze* has severe limitations in performance and resources. These facts make *PicoBlaze* particularly appropriate for this work. Firstly, software-based techniques cannot always be completely applied to the programs running in this microprocessor because of both their high code and performance overheads, and the microprocessor limitations; therefore, *selective hardening on software* may result more suitable in these cases. Secondly, it is necessary to inject a large number of faults to the system to obtain statistically representative reliability results, which can be carried out using a *FPGA*-based fault emulation tool.

The main features of the microprocessor are: 16 byte-wide general-purpose data registers (numbered from 0 to *F* in hexadecimal notation), 1K instructions (10 bits) of programmable on-chip program store, byte-wide Arithmetic Logic Unit (ALU) with CARRY and ZERO indicator flags, 64-byte internal scratchpad RAM, 256 input and 256 output ports, 31-location CALL/RETURN stack, and *KCPSM3* assembly syntax.

The benchmark software suite used in the experiments is made up of three representative programs used in embedded systems: proportional-integral-derivative controller (*PID*), finite impulse response filter (*FIR*), and advanced encryption standard (*AES*). A more complex benchmark (such as MiBench [52] or MediaBench [53]) could not be evaluated due to the mentioned resource limitations in *PicoBlaze*.

The test programs, for demonstration purposes, were rewritten using only 5 from 16 available registers in the microprocessor register file, so enough resources remain free to harden all registers. This first code transformation was performed manually. Then, the three programs were automatically hardened with *S-SWIFT-R* using *SHE*. Since each one of them used 5 registers, a total of 32 different software versions were considered for each program (including the non-hardened version and all the selectively hardened possibilities). This means that a total of 96 different program versions were evaluated. This *brute force* strategy is followed only for demonstration purposes to show the flexibility of *S-SWIFT-R*. However, as can be seen in the next section (Tables 1 and 2), the designer has enough information to preselect only a subset with some of the best candidates for further analyses (especially the reliability evaluation, which is the most time-consuming task).

We performed fault-injection experiments to evaluate the technique fault coverage. *FTUnshades* was used for this purpose. This is an *FPGA*-based fault emulation tool that permits to assess several dependability parameters on the real system implementation. Unlike other emulation or simulation tools, *FTUnshades* allows the fault injection without modifications in the original code, and without hardware instrumentation. This tool is composed of an *FPGA* emulation board and a suite of software tools for testing the emulated design and analyzing test results.

4.2 Registers usage analysis for the non-hardened software version

Table 1 shows the registers usage report for the test programs, which is provided automatically by *SHE*. It presents information about the accesses to each register during the program execution, and their *lifetime*. Accesses give information about the registers usage for write, read, and read/write operations. These values are expressed as the percentage of the total number of each operation type. Moreover, the register *lifetime* represents the time when necessary data for the correct program execution are present in the register [54]. Any fault occurring to the register during that time destroys data integrity. *Lifetime* is expressed as the percentage of the total program time.

Register *lifetime* is expressed as the sum of clock cycles of all the register *living intervals* during the program simulation. A *living interval* starts with a generic

write operation and ends with the last read operation which precedes the next write operation or the end of the program execution.

This information is an useful guide for the designer to pre-select which registers should be hardened and, in this manner, avoid to explore all the possible combinations in the design space. Number of accesses to each register is related to how the overheads will be affected in terms of code size and execution time, whereas *lifetime* is related to reliability.

Table 1 Registers usage report (non-hardened programs)

Test	Reg	Write[%]	Read[%]	Read/Write[%]	Lifetime[%]
	0	23.93	47.19	0.00	63.70
	1	17.09	5.55	18.96	66.38
PID	2	13.67	11.10	35.88	81.67
	3	13.67	11.10	23.48	81.67
	4	31.63	25.05	21.68	74.99
	0	39.28	44.85	0.00	34.81
	1	24.99	10.34	28.96	75.16
FIR	2	17.86	20.68	38.47	88.27
	3	17.86	20.67	31.67	88.27
	4	0.01	3.46	0.90	99.99
	0	48.57	25.19	50.82	80.57
	1	8.22	35.36	29.59	92.34
AES	2	9.78	19.39	10.80	62.49
	3	22.39	12.85	7.00	34.81
	4	11.04	7.21	1.78	35.56

When *S-SWIFT-R* is applied to some highly accessed registers, such as 4 in the *PID* program or 0 in the *AES* program, the overheads are expected to increase considerably because this protection implies to use a higher number of redundant instructions. Nonetheless, protecting these registers does not guarantee improved reliability, since the vulnerability of each register depends, in part, on its *lifetime* during the program execution, and this is not always correlated with the number of times that the register is accessed.

Lifetime has a high impact on reliability, since the higher the *lifetime* is, the longer the register is prone to soft errors. However, it should be considered as well that the vulnerability of each register depends not only of its *lifetime* but also of the criticality of the functions that the register is used for within the program source code. Therefore, more investigation is required as additional criticality criteria have to be taken into account in order to accurately select and prioritize the best candidates to be hardened.

For the scope of this work, however, we will focus only on the *lifetime* using a slight adjustment as it is important considering how the *living intervals* are distributed along the timeline. In cases of registers having the same/similar *lifetime*, criticality is lower for those presenting a larger number of *living intervals*. Since a new *living interval* is created every time there is a write operation to the register, during the write operation there is a fraction of the time in which the register has not yet taken the correct value, and any fault affecting it during that time will be overwritten when it finally takes the written value. According to this, we propose to adjust the *lifetime* by subtracting the number of *write operations* from the total *lifetime* of a register. Finally, the *adjusted lifetime* is then normalized with respect to the duration of the program.

The proposed strategy to prioritize the registers to be hardened is to establish a ranking of the most critical registers according to their normalized *adjusted lifetime*. Ranks are assigned to these values in descending order, which means that registers on top of the ranking will also be the most critical ones. Then, registers will be selected for hardening in the same order as its position in the rank indicates. Table 2 presents the results for the studied cases.

Table 2 Prioritization and selection of registers to be hardened

Test	Register	Adjusted lifetime[%]	Criticality rank
	0	62.49	5
	1	65.52	4
PID	2	80.98	$1-2$
	3	80.98	$1-2$
	4	73.40	3
	0	32.91	5
	1	73.95	4
FIR	2	87.41	$2-3$
	3	87.41	$2-3$
	4	99, 99	1
	0	74.29	2
	1	91.27	1
AES	2	61.23	3
	3	31.92	5
	4	34.13	4

Notice that if we had considered the *lifetime* (instead of the *adjusted lifetime*) to establish the criticality rankings, the rank order would have been the same. Nevertheless, this is not a general rule; there might be different cases in which the rank order may be altered.

Results showed in Table 2 indicate the order in which registers have to be selected for hardening. For instance, in the *AES* case the first register to be protected is the register number 1, followed by the register 0, then the number 2, and so on, according to the criticality ranking.

In Sections 4.3 and 4.4 (as mentioned before), we will evaluate not only a few hardened program versions, but all the versions offered by *S-SWIFT-R* to demonstrate the flexibility of the proposed technique. However, in order to show the usefulness of the proposed pre-selection strategy, Section 4.5 presents the

case when an incremental hardening approach is followed based on the criticality rankings.

4.3 Overheads

Fig. 4 presents the overhead results for all the software versions hardened selectively. Static code size overhead and execution time overhead are showed for all the possibilities of our test programs (*PID*, *FIR*, and *AES*). These results are normalized with a baseline built with the non-hardened version of each program.

A hardened software version obtained by applying *S-SWIFT-R* only to the register subset {0-2-4} means that only these registers are protected (from 16 possible general-purpose *PicoBlaze* registers). Hereafter, the names of the hardened versions correspond to the register group that is protected.

When highly accessed registers are protected, the overheads increase considerably (as expected). This can be observed clearly in the registers 0 and 1 of the *AES* program, which cause a high overhead when protected. For example, in the {0} version, the code size and execution time overheads are 2.54× and 2.61×, respectively.

Notice that overheads results increase incrementally when more registers are protected. In case of the *PID*, static code size overhead goes from 1.28× (in the {2} version) up to 2.65× in the fully protected version (i.e., the {0,1,2,3,4} version), whereas execution time overhead ranges between 1.20× (in the {0} version) and 2.75× (in the *SWIFT-R* version). Moreover, in the *FIR* case, code overhead varies from 1.01× (in the {4} version) to 2.67× in the fully protected version, and execution time overhead ranges from 1.01× (in the {4} version) to 2.53× (in the *SWIFT-R* version). Finally, in the *AES* case, code size overhead goes from 1.22× (in the {4} version) up to 3.30× (in the *SWIFT-R* version), whereas execution time overhead varies from 1.44× (in the {4} version) to 3.72× (in the *SWIFT-R* version).

The above mentioned fact means that, as expected, more resources (code lines and execution time) are required when more protection is implemented (more registers are protected). However, it is very important to notice two additional issues as well, which are related to the contribution to the overheads that each register makes when it is protected. Firstly, each register makes its contribution to code overhead and execution time overhead independently. For instance, the {4} version in the *PID* case presents a considerable code overhead (1.89×) while its execution time overhead is lower than that (1.48×). Secondly, each register makes its contribution to overheads in very different manners. There are versions in which the protection of some registers causes an almost negligible impact, such as in the {4} version of the *FIR* case (the code size and execution time overheads are both 1.01×), whereas at the same time, there are other versions in which the protection of only one register can provoke a high impact, like in the {2} version of the same test program (code overhead 1.61× and execution time overhead 1.56×).

In addition, in the case of *AES*, it is worth noting that due to the high overheads, it was necessary to artificially expand the microprocessor address space. In this way, it was possible to use an additional memory block to fit some of the hardened versions within the program memory properly.

Consideration should be given to the several intermediate protected versions that might result suitable for many applications domains. Although the version with the maximum expected fault coverage is the one with protection in all its registers (*SWIFT-R* version), there are many other versions with protection in groups of four, three, two, or one registers, whose overheads are lower than those caused by the complete protection and can offer enough fault coverage depending on the application requirements. These versions should be considered within the analysis as well.

4.4 Fault coverage

In the following tests, we focus on the type of *soft error* known as *Single Event Upset* (*SEU*). This is a radiation effect that is caused by the direct or indirect ionization provoked by the impact of an incident energetic particle against an electronic component. This effect provokes a change in the logic state of a single memory element (memory cell, flip-flop, latch). We will use the *bit-flip fault model* to represent this fault. In this model, only one bit-flip of a storage element occurs throughout the circuit operation. Since this fault model closely matches the real fault behavior, it is widely used by the fault tolerance community to model real faults [37].

In order to evaluate the fault coverage provided by *S-SWIFT-R*, a fault injection campaign was carried out for each system version using *FTUnshades* (using the real implementation of the different systems). Injected faults were classified according to their effect on the expected system behavior, similarly as it was first proposed by Mukherjee et al. [55]:

1. In case the system completes its execution, and obtains the expected output after that a fault is injected, the memory element (bit) affected by the fault and, consequently the fault itself, are classified as *unnecessary for Architecturally Correct Execution - unACE*.

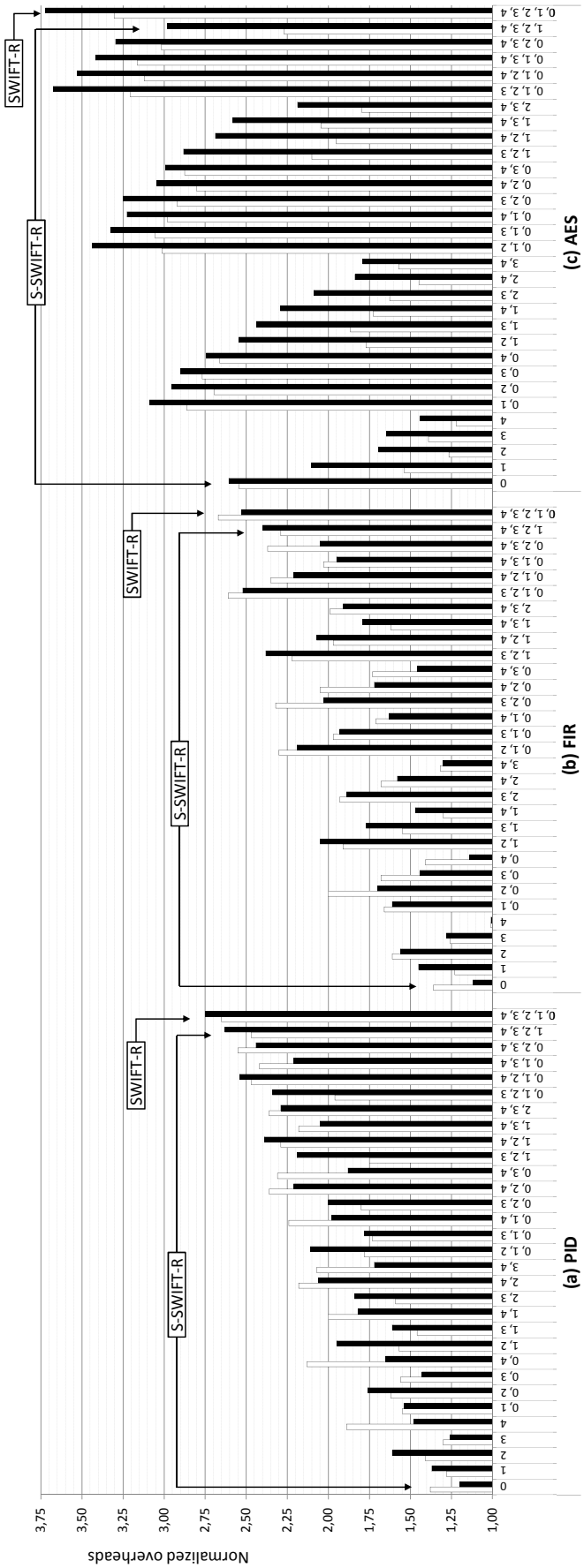

Fig. 4 Normalized code size and execution time overheads using *S-SWIFT-R* for: (a) *PID*, (b) *FIR*, and (c) *AES* selective hardened versions

2. If the fault has not been detected/corrected and provokes the program to complete its execution with an erroneous output, this fault is called *Silent Data Corruption - SDC*.
3. When the fault causes an abnormal program termination or an infinite execution loop, the fault is categorized as a *Hang*.

Note that *SDC* and *Hang* are both undesirable effects (categorized together as *ACE* faults).

Firstly, an initial experiment was carried out to calibrate the *FTUnshades*. This experiment consisted of an incremental fault injection campaign (increasing the number of injected faults on each iteration), performed until statistically representative results were obtained. Notice that the minimum number of representative injected faults depends on each application complexity jointly with the microprocessor architecture. The fault injection campaigns have been performed against the register file of the microprocessor. The non-hardened programs were chosen for the tests because they represent the worst case scenarios. Obtained results show that the 95% confidence interval is less than ±1.0% after 80000 fault injection tests. According to the *bit-flip fault model*, only one fault was emulated during each program execution.

Secondly, for each program version, the fault injection campaign consisted of injecting 80000 faults (*SEUs*), emulating only one single fault per program execution. Each fault was emulated by means of a single bit-flip in a randomly selected bit from the microprocessor register file (16-byte-wide registers) in a randomly selected clock cycle from all the workload duration.

Fig. 5 shows the fault classification percentages obtained for each system after all the fault injection experiments in the *FTUnshades*.

It is worth noting that the high fault coverage results (greater than 75% *unACE* in all cases) obtained for the non-hardened versions are due to the fact that the fault injection test was performed over the complete register file, even though the programs do not use all the sixteen available general-purpose registers. Therefore, a fault injected in an unused register bit is considered as *unACE* because it does not affect the expected program output. This way of testing has been carried out by others researchers as well [26,27,36,17,37], which allows to obtain homogeneous result sets comparable to each other.

One can observe the remarkable increase in the fault coverage that it is obtained using *S-SWIFT-R*. In case of the first test program, the *PID*, the fault coverage ranges between 76.06% *unACE* faults (non-hardened program) to 98.12% *unACE* faults (*SWIFT-R* version). Moreover, in the second case, for the *FIR* program, it goes from 79.19% *unACE* to 99.26% *unACE* faults. In the *AES* case, the values vary from 81.74% *unACE* to 98.77% *unACE* faults. These results represent the percentages of injected faults that do not provoke any undesirable behavior to the circuit operation.

In addition, there are several intermediate-protected versions that might be suitable for many applications depending on the requirements, especially for low-cost solutions. For example, in the *PID* program, when protection is applied only to the registers number 0, 2, and 4 ({0,2,4} version), a considerable fault coverage increase is produced (up to 96.67% *unACE* faults). Another example can be observed for the *FIR* program when the protection is applied to the registers 2 and 4 ({2,4} version). In this case, the increase is up to 94.01% *unACE* faults, which is remarkable, taking into account that only two registers are being hardened. A similar example can be noticed from the *AES*, when fault coverage in the {0,1,2} version is up to 97.87% *unACE* faults.

In the same manner that each register impacts the overheads independently when it is protected, each register contributes apart to the fault coverage improvements. This can be seen, for instance in the *PID* program, in which the fault coverage is 89.67% *unACE* when only the register 2 is protected, whereas protecting only the register 3, the percentage is down to 85.76% *unACE* (a 3.91% difference).

In many cases, the selective protected versions can be better candidates for systems where not only the fault coverage is important, but also the time execution. Protecting all registers, using a software technique, could result in the best fault coverage, but at the same time, it provokes the highest performance degradation. Hence, overheads and fault coverage results have to be studied jointly, representing several trade-offs among code size, performance, and fault coverage. This analysis facilitates to guide the design decisions to find the solutions having the best reliability/overhead compromise. For example, the {1,2,4} *FIR* version is an interesting choice, because it offers both, high fault coverage (95.66% *unACE* faults), and acceptable code size and execution time overheads (1.97× and 2.07×, respectively). In addition, a similar example can be observed in the {1,4} version of the *AES* case, when fault coverage of 95.49% *unACE* faults is reached, whereas the code and time overheads are 1.73× and 2.29×, respectively. It is worth noting that our technique provides detection and recovery of faults, therefore, acceptable trade-offs can be easily reached between overheads and fault coverage.

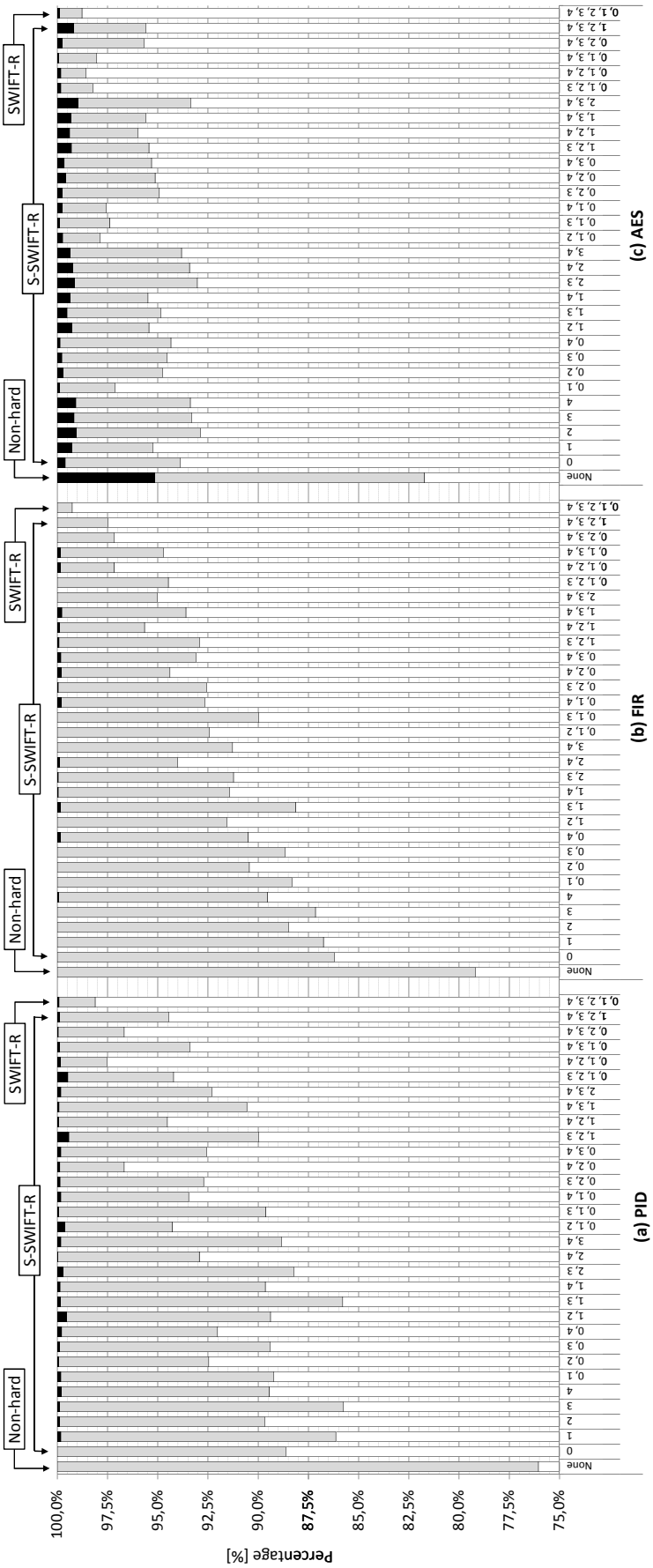

Fig. 5 Fault classification percentages for every selective hardened version of the test programs: (a) *PID*, (b) *FIR*, (c) *AES*

4.5 Incremental hardening strategy

As previously discussed in 4.2, designers can make the selection and prioritization of registers to be hardened according to their criticality. This approach facilitates the design and assessment of a software-based incremental protection strategy, avoiding a *brute force* strategy since it is a very time-consuming task.

Using the criticality rankings presented in Table 2, it is proposed to make an *a priori* selection based on the criticality rank of each register, which determines the order in which the register is hardened in an incremental protection approach. This is a straightforward strategy that allows evaluating only the most effective software versions in terms of fault coverage.

Fig. 6, on the one hand, illustrates the fault classification percentages only for the software versions indicated by the criticality rankings. On the other hand, it also presents, in a secondary axis, their code size and execution time overheads normalized to a baseline built with the non-hardened versions. This figure permits to see at a glance, the representation of several trade-offs between fault coverage and overheads for the each test program.

This approach constitutes a remarkable reduction in the design space, which not only facilitates to reach the best compromise between reliability and overheads, but also to assess effectively only the most interesting selective hardened versions.

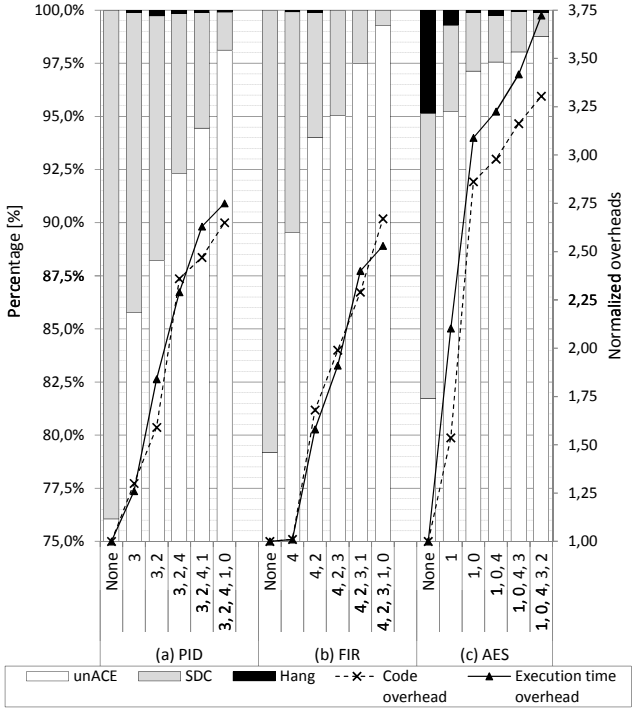

Fig. 6 Fault classification percentages and overheads for the incremental hardening of the test programs: (a) *PID*, (b) *FIR*, (c) *AES*

5 Conclusions and future works

The *selective hardening based on software* permits to enrich the software-side design space for the *soft errors* mitigation techniques. In this way, reliability engineers have more flexibility to find solutions having the best reliability/overhead compromise.

In this paper, a selective version of the software-based technique known as *SWIFT-R* has been presented and called *S-SWIFT-R*. It is possible to select different registers subsets to be protected from the microprocessor register file. Thanks to its flexibility, this technique is appropriated for low-cost dependable applications which use *COTS* microprocessors. Furthermore, it can be used for hardware/software hybrid mitigation approaches as well.

The *S-SWIFT-R* evaluation results prove that not only this technique facilitates to find the best trade-offs among code size, reliability, and performance, but also can be automated to be applied automatically to the programs.

Taking into account parameters like the number of times registers are accessed, and their *lifetime* and vulnerability, as a part of our future works, new relationships among them will be investigated to permit automatically prioritize the order in which registers should be protected. This will allow designers to predict the impact of each register in overheads and reliability when fault tolerance techniques are applied selectively; therefore, this will facilitate the decision making process.

Furthermore, new software-only or hardware/software soft error mitigation techniques will be proposed by exploiting the advantages that the *selective hardening on software* offers.

References

1. R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Trans. Device Mater. Reliab.*, 5(3):305–316, 2005.
2. T Karnik, P Hazucha, and J Patel. Characterization of soft errors caused by single event upsets in CMOS processes. *IEEE Trans. Depend. Secure Comput.*, 1(2):128–143, April-June 2004.
3. R Edwards, C Dyer, and E Normand. Technical standard for atmospheric radiation single event effects (SEE) on avionics electronics. In *IEEE Radiation Effects Data Workshop (REDW)*, pages 1–5. IEEE, 2004.
4. JL Barth, CS Dyer, and EG Stassinopoulos. Space, atmospheric, and terrestrial radiation environments. *IEEE Trans. Nucl. Sci.*, 50(3, Part 3):466–482, June 2003.

5. Leonardo M. Reyneri, Claudio Sansoè, Claudio Passerone, Stefano Speretta, Maurizio Tranchero, Marco Borri, and Dante Del Corso. *Aerospace Technologies Advancements*, chapter 9: Design Solutions for Modular Satellite Architectures. Intech, Olajnica 19/2, 32000 Vukovar, Croatia, January 2010.
6. Ian Vince McLoughlin and Timo Rolf Bretschneider. Reliability through redundant parallelism for micro-satellite computing. *ACM Trans. Embed. Comput. Syst.*, 9(3):26:1–26:25, March 2010.
7. M. Pignol. COTS-based Applications in Space Avionics. In *Proc. 13th Design, Automation and Test in Europe conf., DATE*, pages 1213–1219, March 2010. Dresden, Germany.
8. Michael Nicolaidis. *Soft Errors in Modern Electronic Systems*, volume 41 of *Frontiers in Electronic Testing*. Springer, 1st edition, 2011.
9. M. Pignol. How to cope with SEU/SET at system level? In *Proc. 11th IEEE Int. On-Line Testing Symp, IOLTS*, pages 315 – 318, july 2005.
10. B. Nicolescu, Y. Savaria, and R. Velazco. Software detection mechanisms providing full coverage against single bit-flip faults. *IEEE Trans. Nucl. Sci.*, 51(6):3510–3518, Dec 2004.
11. N. Oh, S. Mitra, and E. J McCluskey. ED4I: error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.*, 51(2):180–199, feb 2002.
12. Maurizio Rebaudengo, Matteo Sonza-Reorda, and Massimo Violante. *Soft Errors in Modern Electronic Systems*, volume 41 of *Frontiers in Electronic Testing*, chapter 9. Software-Level Soft Error Mitigation Techniques. Springer, 1st edition, 2011.
13. José Rodrigo Azambuja, Samuel Pagliarini, Lucas Rosa, and Fernanda Lima Kastensmidt. Exploring the Limitations of Software-based Techniques in SEE Fault Coverage. *J. Electron. Test.*, 27:541–550, August 2011.
14. E. Chielle, J.R. Azambuja, R.S. Barth, F. Almeida, and F. Lima Kastensmidt. Evaluating selective redundancy in data-flow software-based techniques. In *Proc. 13th European Conf. on Radiation and its Effects on Components and Systems RADECS*, September 2012.
15. J. Cong and K. Gururaj. Assuring application-level correctness against soft errors. In *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, pages 150–157, 2011.
16. Ayswarya Sundaram, Ameen Aakel, Derek Lockhart, Darshan Thaker, and Diana Franklin. Efficient fault tolerance in multi-media applications through selective instruction replication. In *Proc. 2008 workshop on Radiation effects and fault tolerance in nanometer technologies*, WREFT '08, pages 339–346, 2008.
17. G. A Reis, J. Chang, and D. I August. Automatic instruction-level software-only recovery. *IEEE Micro*, 27(1):36–47, jan-feb 2007.
18. M. Nicolaidis. Design for soft error mitigation. *IEEE Trans. Device Mat. Reliab.*, 5(3):405 – 418, Sept 2005.
19. Sheng Lin, Yong-Bin Kim, and F. Lombardi. A 11-Transistor Nanoscale CMOS Memory Cell for Hardening to Soft Errors. *IEEE Trans. VLSI Syst.*, 19(5):900–904, May 2011.
20. Naga Durga Prasad Avirneni and Arun K. Somani. Low overhead soft error mitigation techniques for high-performance and aggressive designs. *IEEE Trans. Comput.*, 61(4):488–501, April 2012.
21. MA Gomaa, C Scarbrough, TN Vjaykumar, and I Pomeranz. Transient-fault recovery for chip multiprocessors. *IEEE Micro*, 23(6):76–83, Nov-Dec 2003.
22. PK Samudrala, J Ramos, and S Katkoori. Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs. *IEEE Trans. Nucl. Sci.*, 51(5, Part 4):2957–2969, Oct 2004.
23. V. K. Reddy, S. Parthasarathy, and E. Rotenberg. Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. *ACM Sigplan Notices*, 41(11):83–94, Nov 2006.
24. Xavier Vera, Jaume Abella, Javier Carretero, and Antonio González. Selective replication: A lightweight technique for soft errors. *ACM Trans. Comput. Syst.*, 27(4):8:1–8:30, January 2010.
25. O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. *Software-Implemented Hardware Fault Tolerance*, volume XIV. Springer, 2006.
26. N Oh, PP Shirvani, and EJ McCluskey. Control-flow checking by software signatures. *IEEE Trans. Reliab.*, 51(1):111–122, Mar 2002.
27. N Oh, PP Shirvani, and EJ McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. Reliab.*, 51(1):63–75, Mar 2002.
28. R. Venkatasubramanian, J.P. Hayes, and B.T. Murray. Low-cost on-line fault detection using control flow assertions. In *9th IEEE On-Line Testing Symp. IOLTS*, pages 137–143, 2003.
29. O. Goloubeva, M. Rebaudengo, M.S. Reorda, and M. Violante. Improved software-based processor control-flow errors detection technique. In *Proc. Annual Reliability and Maintainability Symposium, 2005*, pages 583 – 589, 2005.
30. R. Vemu and J.A. Abraham. Budget-dependent control-flow error detection. In *14th IEEE Int. On-Line Testing Symp. IOLTS'08*, pages 73 –78, july 2008.
31. A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans. Software Eng.*, 11(12):1491–1501, dec 1985.
32. M. Jochim. Detecting processor hardware faults by means of automatically generated virtual duplex systems. In *Proc. Int. Conf. on Dependable Systems and Networks, DSN*, pages 399 – 408, 2002.
33. N. Oh and E.J. McCluskey. Error detection by selective procedure call duplication for low energy consumption. *IEEE Trans. Reliab.*, 51(4):392 – 402, dec 2002.
34. A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. A C/C++ source-to-source compiler for dependable applications. In *Proc. Int. Conf. on Dependable Systems and Networks, DSN*, pages 71 –78, 2000.
35. C. Bolchini. A software methodology for detecting hardware faults in VLIW data paths. *IEEE Trans. Reliab.*, 52(4):458 – 468, dec. 2003.
36. G. A Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I August. SWIFT: software implemented fault tolerance. *CGO 2005: Int Symposium on Code Generation and Optimization*, pages 243–254, 2005.
37. M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. *Proc. First IEEE Int. Workshop on Source Code Analysis and Manipulation*, pages 33–42, 2001.
38. M Rebaudengo, M. S. Reorda, and M Violante. A new approach to software-implemented fault tolerance. *J. Electron. Test.*, 20(4):433–437, Aug 2004.
39. O. Ruano, J.A. Maestro, and P. Reviriego. A Methodology for Automatic Insertion of Selective TMR in Digital Circuits Affected by SEUs. *IEEE Trans. Nucl. Sci.*, 56(4):2091–2102, August 2009.

40. B. Pratt, M. Caffrey, J.F. Carroll, P. Graham, K. Morgan, and M. Wirthlin. Fine-Grain SEU Mitigation for FPGAs Using Partial TMR. *IEEE Trans. Nucl. Sci.*, 55(4):2274–2280, August 2008.
41. Thomas Y. Yeh, Glenn Reinman, Sanjay J. Patel, and Petros Faloutsos. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. *ACM Trans. Graph.*, 29(1):5:1–5:11, December 2009.
42. J.R. Azambuja, A. Lapolli, L. Rosa, and F.L. Kastensmidt. Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique. *IEEE Trans. Nucl. Sci.*, 58(3):993 –1000, june 2011.
43. Roshan G. Ragel and Sri Parameswaran. A hybrid hardware–software technique to improve reliability in embedded processors. *ACM Trans. Embed. Comput. Syst.*, 10(3):36:1–36:16, May 2011.
44. Jongeun Lee and Aviral Shrivastava. A compiler-microarchitecture hybrid approach to soft error reduction for register files. *IEEE Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29:1018–1027, July 2010.
45. Jie Hu, Feihui Li, Vijay Degalahal, Mahmut Kandemir, N. Vijaykrishnan, and Mary J. Irwin. Compiler-assisted soft error detection under performance and energy constraints in embedded systems. *ACM Trans. Embed. Comput. Syst.*, 8:27:1–27:30, July 2009.
46. P. Bernardi, L. M. Bolzani Poehls, M. Grosso, and M. Sonza Reorda. A Hybrid Approach for Detection and Correction of Transient Faults in SoCs. *IEEE Trans. Dependable Secur. Comput.*, 7(4):439–445, Oct-Dec 2010.
47. Antonio Martínez-Álvarez, Sergio Cuenca-Asensi, Felipe Restrepo-Calle, Francisco R. Palomo, Hipolito Guzmán-Miranda, and Miguel A. Aguirre. Compiler-directed soft error mitigation for embedded systems. *IEEE Trans. Depend. Secure Comput.*, 9(2):159 –172, march-april 2012.
48. S. Cuenca-Asensi, A. Martínez-Álvarez, F. Restrepo-Calle, F.R. Palomo, H. Guzmán-Miranda, and M.A. Aguirre. A novel co-design approach for soft errors mitigation in embedded systems. *IEEE Trans. Nucl. Sci.*, 58(3):1059 –1065, june 2011.
49. SK Reinhardt and SS Mukherjee. Transient fault detection via simultaneous multithreading. In *27th Int. Symp. on Computer Architecture*, pages 25–36, 2000. Vancuver, Canada, Jun 12-14.
50. H. Guzmán-Miranda, M.A. Aguirre, and J. Tombs. Non-invasive fault classification, robustness and recovery time measurement in microprocessor-type architectures subjected to radiation-induced errors. *IEEE Trans. Instrum. Meas.*, 58(5):1514–1524, May 2009.
51. XILINX. *PicoBlaze 8-bit Embedded Microcontroller User Guide. UG129 (v1.1.2).* Xilinx Ltd., June 2008.
52. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE Int. Workshop of the Workload Characterization. WWC-4.*, pages 3–14, 2001.
53. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. 30th annual ACM/IEEE int. symp. Microarchitecture*, MICRO 30, pages 330–335, 1997.
54. Jongeun Lee and Aviral Shrivastava. Compiler-Managed Register File Protection for Energy-Efficient Soft Error Reduction. In *Proc. of the ASP-DAC 2009: 14th Asia and South Pacific Design Automation Conf.*, pages 618–623, Jan 2009.
55. SS Mukherjee, C Weaver, J Emer, SK Reinhardt, and T Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. 36th Int. Symp. on Microarchitecture*, pages 29–40, Dec 2003.