Reliability on ARM Processors Against Soft Errors Through SIHFT Techniques

Eduardo Chielle, Felipe Rosa, Gennaro S. Rodrigues, Lucas A. Tambara, Jorge Tonfat, Eduardo Macchione, Fernando Aguirre, Nemitala Added, Nilberto Medina, Vitor Aguiar, Marcilei A. G. Silveira, Luciano Ost, Ricardo Reis, Sergio Cuenca-Asensi, and Fernanda L. Kastensmidt

***Abstract*—ARM processors are leaders in embedded systems, delivering high-performance computing, power efficiency, and reduced cost. For this reason, there is a relevant interest for its use in the aerospace industry. However, the use of sub-micron technologies has increased the sensitivity to radiation-induced transient faults. Thus, the mitigation of soft errors has become a major concern. Software-Implemented Hardware Fault Tolerance (SIHFT) techniques are a low-cost way to protect processors against soft errors. On the other hand, they cause high overheads in the execution time and memory, which consequently increase the energy consumption. In this work, we implement a set of software techniques based on different redundancy and checking rules. Furthermore, a low-overhead technique to protect the program execution flow is included. Tests are performed using the ARM Cortex-A9 processor. Simulated fault injection campaigns and radiation test with heavy ions have been performed. Results evaluate the trade-offs among fault detection, execution time, and memory footprint. They show significant improvements of the overheads when compared to previously reported techniques.**

***Index Terms*—Aerospace applications, error detection, fault coverage, fault tolerance, mitigation techniques, processors, reliability, soft errors, software techniques.**

I. Introduction

SOFT errors affect processors by modifying values stored in memory elements (such as registers and data memory) [1]. Such faults may lead the processor to incorrectly execute an application or even to enter into a loop and never finish the execution. These faults can also modify some computed data values, generating errors in the data results. The mitigation of soft errors in aerospace applications is a must. In this context, hardware-based or software-based fault tolerance techniques can be used.

Hardware-based fault tolerance techniques rely on replicating or adding hardware modules. Although the high reliability they provide, they present a significant increase in area and power consumption, high design and manufacture costs, and they are not applicable to commercial off-the-shelf (COTS) processors [2], [3]. Software-based fault tolerance techniques, also referred in the literature as Software-Implemented Hardware Fault Tolerance (SIHFT) techniques [4], are methods to protect processor-based systems against soft errors by adding instruction redundancy and comparison to detect or correct errors, without having to modify the underlying hardware, which permits their use in COTS processors. SIHFT techniques provide high flexibility and low development time and cost. Although software redundancy brings reliability to the system, it requires extra processing time and increases the energy consumption since more instructions are executed [5], [6]. Furthermore, a reliable program uses more memory addresses due to the software redundancy [7], [8].

This work implements a set of data-flow techniques based on duplicating and checking rules that aim at reducing overheads. Furthermore, a new control-flow technique, with lower overheads when compared to a state-of-the-art technique [9], is introduced. The control-flow technique was designed to be used together with a data-flow technique to increase the overall system reliability. The SIHFT techniques are applied to the assembly code executed by processors. In this work, we focus on ARM Cortex-A9 processor [10] because of its growing employment in different domains where reliability is a must. Finding the best data-flow/control-flow combination is not an easy task and can be unaffordable if radiation experiments are involved. Therefore, to address that issue, we propose to compare the different solutions regarding fault coverage and overheads using a processor simulator. Then, only the most reliable configuration will be assessed in radiation test.

This paper is organized as follows. Section II briefly discusses state-of-the-art SIHFT techniques. Section III presents our techniques and the key points of their implementation. Section IV is devoted to analyzing, in terms of fault injection, the reliability of different combinations of techniques. Section V describes the radiation tests performed on the real hardware running a hardened and an unhardened version of a selected benchmark. Finally, in Section VI, we draw some conclusions and discuss future work.

Manuscript received October 08, 2015; revised December 09, 2015, January 19, 2016, and January 26, 2016; accepted January 29, 2016. This work was supported in part by CNPq and CAPES, Brazilian agencies.

E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, J. Tonfat, F. L. Kastensmidt, and R. Reis are with the Instituto de Informática, PGMICRO, UFRGS, 91509-900 Porto Alegre, Brazil (e-mail: echielle@inf.ufrgs.br; frdarosa@inf.ufrgs.br; gsrodrigues@inf.ufrgs.br; latambara@inf.ufrgs.br; jltseclen@inf.ufrgs.br; fglima@inf.ufrgs.br; reis@inf.ufrgs.br).

S. Cuenca-Asensi is with the Computer Technology Department, University of Alicante, 03080 Alicante, Spain (e-mail: sergio@dtic.ua.es).

E. Macchione, F. Aguirre, N. Added, N. Medina, and V. Aguiar are with the Instituto de Física, USP, 91501-970 São Paulo, Brazil (e-mail: macchion@if.usp.br; faguirre@if.usp.br; nemitala@if.usp.br; medina@if.usp.br; vpaguiar@if.usp.br).

M. A. G. Silveira is with the Centro Universitário da FEI, São Bernardo do Campo, 09850-901 Sao Bernardo do Campo, Brazil (e-mail: marcilei@fei.edu.br).

L. Ost is with the Department of Engineering, University of Leicester, Leicester LE1 7RH, U.K. (e-mail: luciano.ost@leicester.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNS.2016.2525735

TABLE I
TYPES OF SOFTWARE-BASED FAULT TOLERANCE TECHNIQUES

Data-flow techniques	Control-flow techniques
• Protect the data-flow ▪ data stored in registers ▪ data stored in memory	• Ensure the integrity of the execution flow
• Registers are replicated	• Divide the code into basic blocks
• Checkers are inserted	• A unique signature is assigned to each basic block
• Every operation performed on a register must be performed on its replica	• Checking instructions are inserted in the program code

II. FAULT TOLERANCE IN SOFTWARE

SIHFT techniques are techniques implemented in software to protect processor against soft errors that may affect the program flow or the data stored in registers or memory. The techniques that aim to protect the data-flow are called data-flow techniques, and the ones to protect the control-flow are the control-flow techniques. There are also techniques that combine features of both data-flow and control-flow techniques to protect the data-flow and the control-flow [11]. They consist of code transformation rules and can be understood as a data-flow and a control-flow technique applied together because some rules focus on protecting the data-flow and others, the control-flow. Table I summarizes the two types SIHFT techniques.

A. Data-Flow Techniques

Data-flow techniques are designed to protect the data stored in registers or memory. These techniques replicate the registers, assigning copies to the original ones. When the aim is error detection, registers are duplicated, and when correction is included, registers are triplicated. Checkers (voters, if correction) are inserted in the code to compare the registers with their copies. The points where checkers are inserted depend on the technique. Since error detection presents lower overheads than correction due to duplication instead of triplication, this work focus on that. Some data-flow techniques present in the literature are EDDI [12] and Variables [13].

B. Control-Flow Techniques

Control-flow techniques are designed to protect the program flow, i.e., to protect against incorrect jumps. Such techniques divide the code into basic blocks. A basic block (BB) is a branch-free sequence of instructions, i.e., a portion of code that is always executed in sequence. There only can be a branch instruction at the end of the basic block. Furthermore, there are no branches to the basic block, except to the first instruction. For each basic block, a signature is assigned. The signature is attributed to a global register at the beginning of the basic block. Checkers are inserted in the code to verify if the signature register contains the expected value. If it does not, it means the program flow was not correctly followed, and an error is reported. Some control-flow techniques present in the literature are CFCSS [14], YACCA [15], and CEDA [9]. Table II shows the execution time and fault coverage of these techniques. As one can see, CEDA is the one with the best trade-off between fault coverage and performance, and that is why it was used as the baseline control-flow technique of this work.

TABLE II
STATE-OF-THE-ART CONTROL-FLOW TECHNIQUES [9]

BENCHMARKS	CFCSS		YACCA		CEDA	
	UF	ET	UF	ET	UF	ET
PARSER	4.6%	1.14X	1.0%	1.34X	1.1%	1.14X
GZIP	3.4%	1.58X	0.7%	1.84X	0.6%	1.58X
AMMP	4.7%	1.04X	0.3%	1.79X	0.2%	1.03X
TWOLF	2.8%	1.08X	0.6%	1.40X	0.6%	1.10X
EQUAKE	2.8%	1.19X	0.5%	1.34X	0.5%	1.18X

UF: UNDETECTED FAULTS
ET: EXECUTION TIME

TABLE III
RULES FOR DATA-FLOW TECHNIQUES [16]

	Global Rules (valid for all techniques)
G1	each register used in the program has a spare register assigned as replica
	Duplication Rules (performing the same operation on the register's replica)
D1	all instructions
D2	all instructions, except stores
	Checking Rules (compare the value of a register with its replica)
C1	before each read on the register (except load/store and branch/jump instructions)
C2	after each write on the register
C3	the register that contains the address before loads
C4	the register that contains the datum before stores
C5	the register that contains the address before stores
C6	before branches or jumps

III. IMPLEMENTED TECHNIQUES

In this work, we aim at protecting both data-flow and control-flow of a running application. For this reason, we combined data-flow and control-flow techniques. They are presented as follows.

A. VAR Data-Flow Techniques

In reference [16], a set of seventeen data-flow techniques, called VAR, that aim at reducing overheads in performance, memory, and energy consumption, were presented and validated by fault injection for the miniMIPS processor [17]. They consist of three types of different rules: global, duplication, and checking rules, as one can see in Table III. The global rule states that every register used by the program must have a spare register assigned as replica. The global rule is applied by all techniques. Duplication rules regard how the instructions are duplicated. They are only applicable to instructions that perform write operations on registers or memory. Therefore,

TABLE IV
RULES FOR IMPLEMENTED VAR DATA-FLOW TECHNIQUES

Technique	Duplication Rule	Checking Rules
VAR3	D1	C3, C4, C5, C6
VAR3+	D2	C3, C4, C5, C6
VAR3++	D2	C3, C4, C5
VAR4	D1	C4, C5, C6
VAR4+	D2	C4, C5, C6
VAR4++	D2	C4, C5
VAR5	D1	C4, C6
VAR5+	D2	C4, C6
VAR5++	D2	C4

branch instructions are not considered in this case. There are two types of duplication rules: D1 and D2. Each technique can only use one duplication rule. D1 duplicates all instructions, including stores, which allow the use of unprotected memories because the original value and its replica can be stored in different positions in the memory. D2 duplicates all instructions, except stores. The last one is adequate when the memory is hardened because the data in memory do not need to be duplicated. Thus, the overhead caused by duplicating the code and the number of memory accesses are reduced. Checking rules indicate when a register and its replica are compared. Thus, it is possible to verify if an error has occurred (when the register and its replica present different values). Techniques can have more than one checking rule. Theoretically, the more checkers are included in one technique, the more reliability is achieved. Of the seventeen data-flow techniques, we selected the nine with the best results. They are listed in Table IV.

Table V shows a portion of code hardened by the nine implemented data-flow techniques for the ARM Cortex-A9 processor. The original code is formatted as normal text, the instructions inserted by the duplication rules are italicized, and the checkers are bold. As one can notice, there are differences among the codes generated by the techniques due to the different rules each technique implements. VAR3 was proposed by [13], and it is the technique with the highest overhead among the implemented ones. VAR4 is equivalent do EDDI.

In this work, VAR data-flow techniques are merged with a control-flow technique to protect both data-flow and control-flow. The goal is to achieve a high fault coverage with low overheads. Among the data-flow techniques, we are looking for the one with the lowest overhead that still provides very high fault coverage when combined with the control-flow technique for the ARM Cortex-A9 processor.

B. SETA Control-Flow Technique

In this paper, we introduce a technique called SETA (*Software-only Error-detection Technique using Assertions*) to detect control-flow errors in processors with no modification or addition of extra hardware. The penalties in performance and memory caused by SETA are lower than other control-flow techniques. SETA is based on HETA [18] and CEDA. These techniques use runtime signatures to detect errors affecting the control-flow of a running application. HETA uses an extra signature, which increases the overheads. Also, it makes use of a watchdog to help in the detection, which requires extra power. And, as the author stated, the watchdog needs access to the memory buses. Some processors that use on-chip embedded cache memories may not be accessible by the watchdog, which makes impossible to implement this technique in the target ARM processor. Furthermore, both CEDA and HETA are concerned about the error detection rate they achieve, but not about the overheads they cause. Aiming at providing similar error detection rate as CEDA with lower overheads, SETA is proposed. The technique uses signatures calculated a priori and processed during runtime. The program code is divided into basic blocks (BB), which are branch-free sequences of instructions with no branches into the basic block, except for a possible branch to the first instruction, and no branches out of the basic block, except for the last instruction. Signatures are assigned to the basic blocks.

Two Basic Block Types (BBT) are defined: A and X. A basic block is of type A if it has multiple predecessors, and at least one of its predecessors has multiple successors. And it is of type X if it is not of type A. Then, the basic blocks are grouped into networks. Basic blocks sharing a common predecessor belong to the same network. An example is shown in Fig. 1.

Every basic block has two different signatures: a *Node Ingress Signature* (NIS), for when entering the basic block, and a *Node Exit Signature* (NES), for when exiting the basic block. The NIS represents the current basic blocks, and the NES is used to identify the successor network and the valid successor basic blocks.

The signatures are divided in two parts: an upper half and a lower half, as shown in Table VI. The upper half identifies the network, and the lower half identifies the basic block. The NIS' upper half identifies the network that the basic block belongs to. The lower half has a random number assigned if the BB is of type X. If the BB is of type A, the lower half is calculated by the AND operation of the lower halves of all predecessor BBs' NES. The NES' upper half identifies the successor network, and the lower half has a random number. Table VII summarizes it. If a BB of type X has multiple predecessors, all its predecessors must have the same NES. The size of these "halves" is, actually, variable per application in order to maximize the basic block identifier (lower half) and, thus, avoid aliasing. The upper half receives the minimum number of bits it needs to represent all the networks, i.e., the first integer greater or equal to $log_2(N + 1)$, where N is the total number of networks. Let us define it as $ceil(log_2(N + 1))$. The remaining bits are used by the lower half. The networks are sequentially identified, from 0 to $\mathrm{N} - 1$. The identifier N is reserved for what we call *ghost network*. It is used as successor network of the basic blocks that have no successors. Thus, it invalidates any transition (caused by a fault) from such BBs to another BB.

At runtime, a signature register S is updated according to the conditions presented in Table VIII to keep track of the program execution. The operation to update S can be an XOR or an

TABLE V
EXAMPLE OF VAR DATA-FLOW TECHNIQUES FOR THE ARM CORTEX-A9

#	Unhardened code	VAR3	VAR3+	VAR3++	VAR4
1		**cmp r4, r9**	**cmp r4, r9**	**cmp r4, r9**	
2		**bne error**	**bne error**	**bne error**	
3	ldr r1, [r3, r4]	ldr r1, [r4, #0]	ldr r1, [r4, #0]	ldr r1, [r4, #0]	ldr r1, [r4, #0]
4		*ldr r6, [r9,#1000]*	*ldr r6, [r9,#0]*	*ldr r6, [r9,#0]*	*ldr r6, [r9,#1000]*
5		**cmp r4, r9**	**cmp r4, r9**	**cmp r4, r9**	**cmp r4, r9**
6		**bne error**	**bne error**	**bne error**	**bne error**
7	ldr r2, [r0, r4]	ldr r2, [r4, #4]	ldr r2, [r4, #4]	ldr r2, [r4, #4]	ldr r2, [r4, #4]
8		*ldr r7, [r9,#1004]*	*ldr r7, [r9,#4]*	*ldr r7, [r9,#4]*	*ldr r7, [r9,#1004]*
9	add r1, r2, r1	add r1, r2, r1	add r1, r2, r1	add r1, r2, r1	add r1, r2, r1
10		*add r6, r7, r6*	*add r6, r7, r6*	*add r6, r7, r6*	*add r6, r7, r6*
11		**cmp r1, r6**	**cmp r1, r6**	**cmp r1, r6**	**cmp r1, r6**
12		**bne error**	**bne error**	**bne error**	**bne error**
13		**cmp r4, r9**	**cmp r4, r9**	**cmp r4, r9**	**cmp r4, r9**
14		**bne error**	**bne error**	**bne error**	**bne error**
15	str r1, [r0, r4]	str r1, [r4, #4]	str r1, [r4, #4]	str r1, [r4, #4]	str r1, [r4, #4]
16		*str r6, [r9, #1004]*			*str r6, [r9, #1004]*
17		**cmp r1, r6**	**cmp r1, r6**		**cmp r1, r6**
18		**bne error**	**bne error**		**bne error**
19		**cmp r5, r10**	**cmp r5, r10**		**cmp r5, r10**
20		**bne error**	**bne error**		**bne error**
21	cmp r1, r5	cmp r1, r5	cmp r1, r5	cmp r1, r5	cmp r1, r5
22	ble loop	ble loop	ble loop	ble loop	ble loop

#	VAR4+	VAR4++	VAR5	VAR5+	VAR5++
1					
2					
3	ldr r1, [r4, #0]	ldr r1, [r4, #0]	ldr r1, [r4, #0]	ldr r1, [r4, #0]	ldr r1, [r4, #0]
4	*ldr r6, [r9,#0]*	*ldr r6, [r9,#0]*	*ldr r6, [r9,#1000]*	*ldr r6, [r9,#0]*	*ldr r6, [r9,#0]*
5	**cmp r4, r9**	**cmp r4, r9**	**cmp r4, r9**	**cmp r4, r9**	**cmp r4, r9**
6	**bne error**	**bne error**	**bne error**	**bne error**	**bne error**
7	ldr r2, [r4, #4]	ldr r2, [r4, #4]	ldr r2, [r4, #4]	ldr r2, [r4, #4]	ldr r2, [r4, #4]
8	*ldr r7, [r9,#4]*	*ldr r7, [r9,#4]*	*ldr r7, [r9,#1004]*	*ldr r7, [r9,#4]*	*ldr r7, [r9,#4]*
9	add r1, r2, r1	add r1, r2, r1	add r1, r2, r1	add r1, r2, r1	add r1, r2, r1
10	*add r6, r7, r6*	*add r6, r7, r6*	*add r6, r7, r6*	*add r6, r7, r6*	*add r6, r7, r6*
11	**cmp r1, r6**	**cmp r1, r6**	**cmp r1, r6**	**cmp r1, r6**	**cmp r1, r6**
12	**bne error**	**bne error**	**bne error**	**bne error**	**bne error**
13	**cmp r4, r9**	**cmp r4, r9**			
14	**bne error**	**bne error**			
15	str r1, [r4, #4]	str r1, [r4, #4]	str r1, [r4, #4]	str r1, [r4, #4]	str r1, [r4, #4]
16			*str r6, [r9, #1004]*		
17	**cmp r1, r6**		**cmp r1, r6**	**cmp r1, r6**	
18	**bne error**		**bne error**	**bne error**	
19	**cmp r5, r10**		**cmp r5, r10**	**cmp r5, r10**	
20	**bne error**		**bne error**	**bne error**	
21	cmp r1, r5	cmp r1, r5	cmp r1, r5	cmp r1, r5	cmp r1, r5
22	ble loop	ble loop	ble loop	ble loop	ble loop

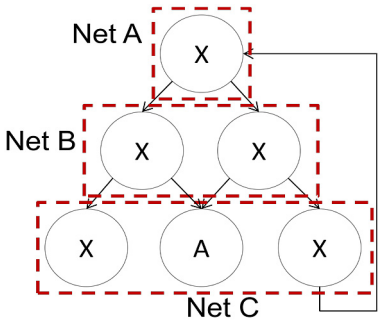

Fig. 1. Representation of a program flow. Basic blocks (circles) classified as of type A or X, and grouped into networks (dashed rectangles). The arrows indicate the valid directions that a basic block can take.

TABLE VI
SIGNATURE DIVISION

Upper half	Lower half
01000100	01001011110101001011110 1

↔ variable per application

TABLE VII
ROLE OF EACH HALF IN THE SIGNATURES

Signature	Upper half	Lower half
NIS	Identifies the BB's network	BB's signature 1
NES	Identifies the successor network	BB's signature 2

TABLE VIII
SIGNATURE UPDATE

BB Type	NIS	NES
A	AND	XOR
X	XOR	XOR

AND. It is performed with S and an invariant value, as shown below.

$$S \leftarrow S\ OP\ invariant, \text{where OP can be AND or XOR} \quad (1)$$

The invariant is a constant that will make S have the expected signature during a correct execution. Its calculation is described as follows. From NIS to NES (inside a BB), the NES invariant to update S depends only on the BB's signatures. The invariant is the result of an XOR operation of the BB's NIS and NES. From NES to NIS (BBs transition), the NIS invariant relies on the predecessor BBs' NES, and on the NIS and type of the current BB. If the BB is of type X, there are two possible ways to calculate the invariant:

- The BB has no predecessors (starting BB): in this case, the NIS invariant is equal to the BB's NIS.
- The BB has predecessors: the NIS invariant is the result of an XOR operation of any predecessor NES with the BB's NIS.

If the BB is of type A, the NIS invariant is divided into upper and lower half (like the signatures) for its calculation. The upper half is filled with ones (the equivalent in unsigned integer is $2^{ceil(log_2(N+1))} - 1$). The lower half of the NIS invariant is equal to the lower half of the BB's NIS. The classification of basic blocks into types and networks ensures that there will not be invalid transitions, except for the following case: the starting BB has itself as successor. Consequently, it is also its predecessor. In this case, a constant is loaded to S at the beginning of the BB to keep the execution consistent.

Checkers are inserted in the basic blocks to verify if S contains the expected signature for that basic block. The more checkers, the lower is the latency to detect errors. On the other hand, the higher is the overhead. The maximum number of checkers in SETA matches the number of basic blocks since only one checker is needed per basic block. Table IX shows an example of SETA for the ARM Cortex-A9 processor. An unhardened portion of code is shown in the left side, and, in the right side, there is the same code protected by SETA. The instructions inserted by SETA are in italics (signature updates) or bold (checkers). The first XOR (*eor*) is used to update the signature to the BB's NIS. The instructions *cmp* and *bne* are used to compare the signature register *r5* with the expected signature for that basic block. Finally, the last XOR is used to update the signature register to the expected NES.

IV. RELIABILITY ESTIMATION OF SIHFT TECHNIQUES

We analyzed the techniques in terms of execution time, memory footprint, fault coverage, and Mean Work to Failure

TABLE IX
EXAMPLE OF SETA CONTROL-FLOW TECHNIQUE FOR ARM CORTEX-A9 PROCESSOR

#	Unhardened code	Code hardened by SETA
1		*eor r5, r5, #13*
2	ldr r1, [r3, r4]	ldr r1, [r3, r4]
3	ldr r2, [r0, r4]	ldr r2, [r0, r4]
4	add r1, r1, #48	add r1, r1, #48
5	add r1, r2, r1	add r1, r2, r1
6	str r1, [r0, r4]	str r1, [r0, r4]
7	add r4, r4, #4	add r4, r4, #4
8		**cmp r5, #71**
9		**bne error**
10		*eor r5, r5, #13*
11	b loopin	b loopin

(MWTF) metric [19]. The last metric captures the tradeoff between reliability and performance. The more time an application needs to run, the higher the probability to be hit by a particle and, consequently, affected by a fault. The MWTF is defined by Eq. (2). The AVF (*Average Vulnerability Factor*) is used to measure microarchitectural structure's susceptibility to transient faults [20].

$$\begin{aligned} \text{MWTF} &= \frac{\text{amount of work completed}}{\text{number of errors encountered}} \\ &= (\text{raw error rate} \times \text{AVF} \times \text{execution time})^{-1} \end{aligned} \quad (2)$$

To obtain the fault coverage for each technique or set of techniques, we hardened three target applications (matrix multiplication (MM), quicksort (QS) and Tower of Hanoi (TH)) with the presented techniques using the CFT-tool [21], which makes the hardening automated. Then, we submitted the different hardened versions to a fault injection campaign. A total of 1,000,000 faults were injected per version. The OVPSim-FIM [22] was used to perform fault injections by simulation in the ARM Cortex-A9 processor. OVSim-FIM relies on and extends the capabilities of the OVPSim, marketed by Imperas [23]. In this platform, the bit flips are performed in the accessible registers at a random time during the program execution. The target register and the bit where the bit flip occurs are randomly selected. A random injection time based on the instruction count is also chosen. So, when the program reaches the fault time, a single selected bit of the selected register is flipped. The PC (program counter) and the memory of the executions under faults are compared to the PC and memory of a golden execution. A fault can be masked (correct execution) or cause an error. If an error is produced, it can be detected or undetected. With this information, it is possible to obtain the fault coverage by the following equation:

$$\text{F}_{\text{coverage}} = \frac{\text{E}_{\text{detected}} + \text{F}_{\text{masked}}}{\text{F}_{\text{total}}} = 1 - \frac{\text{E}_{\text{undetected}}}{\text{F}_{\text{total}}} \quad (3)$$

Where:

- $\text{F}_{\text{coverage}}$ is the fault coverage
- $\text{E}_{\text{detected}}$ is the number of errors detected
- F_{masked} is the number of correct executions
- $\text{E}_{\text{undetected}}$ is the number of undetected errors
- F_{total} is the number of executions

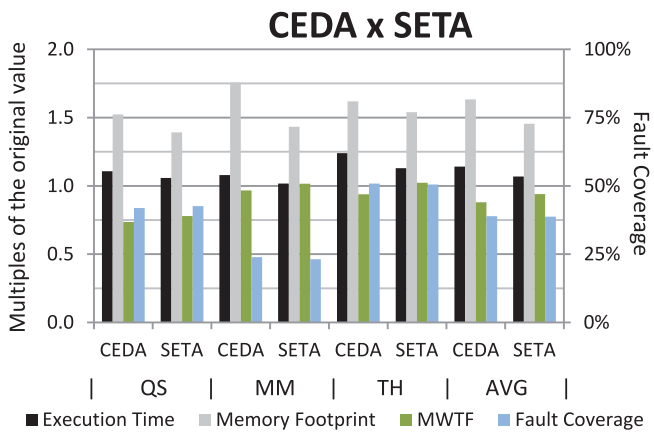

Fig. 2. Comparison between CEDA and proposed SETA techniques. The execution time, memory footprint, and MWTF are presented normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Firstly, we compared SETA with CEDA. Fig. 2 shows the execution time, memory footprint, MWTF, and fault coverage of both techniques for all benchmarks. The average result (AVG) is also included. The execution time, memory footprint, and MWTF are presented normalized by the equivalent unhardened application (left axis). The fault coverage is expressed in percentage (right axis). The results show that both techniques present fault coverage around of 39% in average.

The execution time and the memory footprint of SETA (1.07x and 1.45x, respectively) are lower than CEDA (1.14x and 1.63x, respectively). Once SETA runs faster than CEDA, the application protected by SETA has a lower chance of being hit by an energized particle that causes a bit flip. And since both have similar fault coverage, SETA is more reliable than CEDA. It is shown by the MWTF, which is 0.94x for SETA and 0.88x for CEDA. It is important to notice that the average MWTF of both control-flow techniques for the ARM Cortex-A9 and the target applications is inferior to the unhardened. SETA's MWTF is slightly greater than 1x for the matrix multiplication and Tower of Hanoi, but inferior for the quicksort. CEDA's MWTF is inferior to 1x and, also, inferior to SETA's MWTF in all cases. It means that sometimes is better not protect the application instead of protecting using only a control-flow technique. This result corroborates [24], in which the authors stated that control-flow techniques make the processor more vulnerable to soft errors because they do not provide enough fault coverage to compensate the extra execution time. However, this statement cannot be taken as a rule since the reliability also depends on the target application and processor, as showed by the greater MWTF presented by SETA in 2 out of 3 cases. Furthermore, control-flow techniques are meant to be used together with data-flow techniques. Thus, they can significantly increase the fault coverage and MWTF.

SETA was combined with some data-flow techniques called VAR. Fig. 3 presents the results for each benchmark and the average results (AVG). The execution time, memory footprint, and MWTF are expressed normalized by the equivalent unhardened application (left axis). The fault coverage is presented in percentage (right axis). The horizontal axis identifies the data-flow technique. For example, 3++ means that the data-flow technique VAR3++ and the control-flow technique SETA have been applied. We can see that the overheads in performance and memory introduced by the data-flow techniques for the target application and processor are higher than the ones presented by the control-flow techniques. It is justified by the insertion of redundancy and checkers in the entire code, instruction by instruction, and not by dividing into basic blocks. However, one can notice an increase of up to 8.67x of the MWTF when VAR3+ and SETA are applied. All the data-flow techniques, when combined with SETA, present a significant increase of the MWTF.

V. Radiation Test with Heavy Ions

Radiation test is considered one of the most accurate approaches to assess the reliability of new techniques. However, contrarily to the simulations carried out in previous section, an exhaustive evaluation of every combination is not feasible with radiation due to limited beam time. Therefore, we used the fault injection campaigns as a guideline to select the most suitable combination of SIHFT techniques based on MWTF. As a result, only VAR3+ combined with SETA was evaluated by radiation test with heavy ions.

For the radiation test experiment, we utilized a ZedBoard. It is a low-cost development board for the Xilinx Zynq-7000 All Programmable SoC, XC7Z020-CLG484 part, which offers high configurability, stimulates strong interest in the scientific community, and is highly present in the market. The board is composed of two main parts: a Processing System (PS) that contains a dual-core ARM Cortex-A9 processor [25], and Programmable Logic (PL). The PL section is ideal for implementing high-speed logic, arithmetic, and data processing subsystems, while the PS supports software routines and operating systems. The proposed analysis is based only on the PS part of the board. The PL part is not used at any moment during the experiment. Nevertheless, the proposed test methodology and the achieved results are meant to be generic and easily extendable to other ARM cores.

Heavy ions experiments were conducted at *Laboratório Aberto de Física Nuclear* of the *Universidade de São Paulo* (LAFN-USP), Brazil [26]. The ion beams were produced and accelerated by the *São Paulo 8UD Pelletron Accelerator*. Aiming to achieve a very low particle flux in the range from 10^2 to 10^5 particles.cm^{-2}.s^{-1}, as recommended by the European Space Agency (ESA) for SEU tests [27], a standard Rutherford scattering setup using a gold foil was used. The experiment was performed in air. A silicon barrier detector was mounted inside the vacuum chamber at an angle of $45°$ to monitor the beam intensity. In front of the detector, it was mounted a collimator with a diameter of 4 mm, defining a solid angle of about 0.085 msr. The SEU events were observed irradiating ^{16}O beams, scattered by a 184 µg/cm gold target, with an energy of 51 MeV (effective energy of 41 MeV), which provided a Linear Energy Transfer (LET) of 5 MeV/mg/cm and penetration in Si of 29 µm. To achieve the desired particle flow, the DUT was positioned at a scattering angle of $15°$, resulting in an average

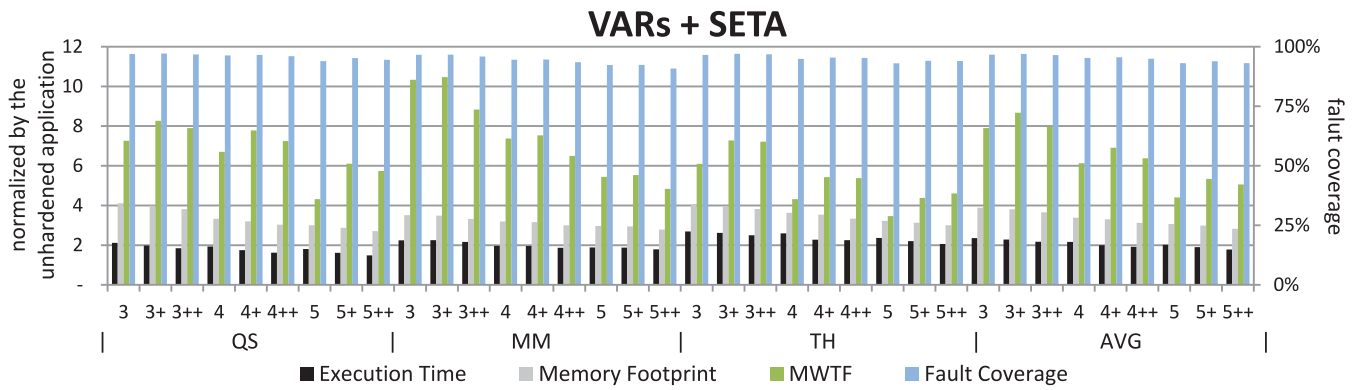

Fig. 3. Execution time, memory footprint, MWTF, and fault coverage for all combined techniques and all case-study applications. The label indicates the version of VAR technique was used together with SETA. For example, 3+ indicates that VAR3+ and SETA were utilized. The execution time, memory footprint, and MWTF are expressed in relation to the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

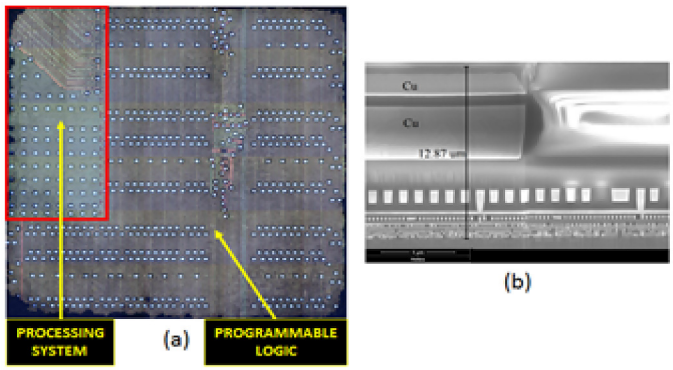

Fig. 4. (a) View of the surface of the XC7Z020-CLG484 device, and (b) Microscopic section of the XC7Z020-CLG484 device.

TABLE X
SUMMARY OF RADIATION TEST WITH HEAVY IONS IN THE ARM CORTEX-A9

BB Type	Unhardened	Hardened by VAR3+ and SETA
Flux	5.84×10^2 part/(cm².s)	5.84×10^2 part/(cm².s)
Time of exposure	92 min	91 min
Fluence	3.23×10^6 part/cm²	3.19×10^6 part/cm²
SER	5.43×10^{-3}	1.47×10^{-3}
Cross section	9.30×10^{-6} cm²	2.51×10^{-6} cm²
Execution time	6.08×10^{-2} s	1.59×10^{-1} s
Code size	252 B	996 B
MWTF	1.00x	1.66x

flux of 584.44 particles.cm$^{-2} \cdot$ s^{-1}. Finally, the DUT was also positioned in a way that the center of the beams was focused in the PS part.

The package of the device was thinned to allow that irradiated particles penetrate the active region of the silicon. Fig. 4(a) shows the chip surface without its package. It is possible to distinguish between the PS and the PL part. Fig. 4(b) shows a microscopic section of the chip performed to evaluate the energy loss of the heavy ions after passing the passive layers. The passive layers consist of eleven copper metallization layers separated by dielectric layers. The total thickness of the passive layers is 12.87 µm. To estimate the energy loss of the heavy ions, it was assumed a total thickness of the copper metallization layers of 7.87 µm, and a total thickness of the dielectric layers of 5.0 µm.

The setup, shown in Fig. 5, consists of a board, computer, USB net switch, cables for communication, and cables for power supply. The computer is connected to the board by two USB cables. One is used to program the board, and the other is used to receive the output from the board. The board power supply is connected to the USB net switch, which is connected by USB to the computer. It is used to control the power supply of the board. Only one ARM core was utilized during the test, data and instruction L1 caches were enabled, and L2 was disabled. The processor was running a target application that sends the output by UART to the computer and, then, restarts its execution. The computer was running a monitoring application that listens to the COM port connected to the board UART, and classifies the output. In case of error in the ARM processor, the processor is reset.

Two versions of a Tower of Hanoi have been tested, one unhardened, and the other hardened by VAR3+ and SETA techniques, which was the case that reached the highest MWTF in the simulated fault injection. Table X summarizes the parameters utilized in the radiation test with heavy ion. The unhardened version was 92 minutes under radiation, receiving a total fluence of 3.23×10^6 part/cm^2 in average. The hardened version was 91 minutes under radiation, receiving a total fluence of 3.19×10^6 part/cm^2 in average. We observed an SER of 5.43×10^{-3} and a cross section of 9.30×10^{-6} cm^2 for the unhardened application. For the hardened version, we observed an SER of 1.47×10^{-3} and a cross section of 2.51×10^{-6} cm^2. One can see a reduction of the SER and cross section by a factor of 3.71 when hardening using VAR3+ and SETA. However, the execution time of the hardened case-study application is 2.62 times, and the code size is 3.95 times the unhardened application for the ARM processor. That results in a normalized MWTF of 1.66x for the hardened application.

Although the results obtained from simulated fault injection cannot be directly compared with the ones obtained from the

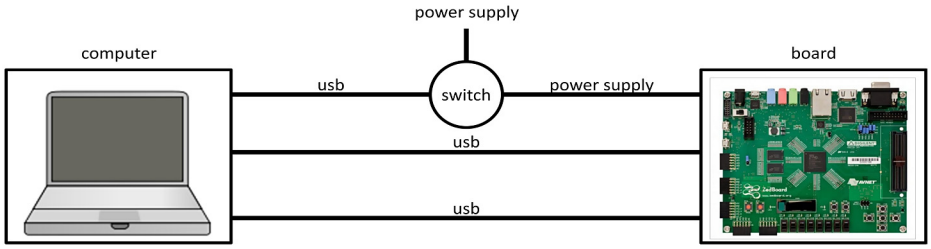

Fig. 5. Setup of the radiation test. The computer is connected to the board by two USB cables. One is used to program the board and the other is used to get the output from the board by UART. The computer is also connected by USB to a switch that controls the power supply of the board.

radiation experiment, it is possible to notice that the MWTF of the hardened version in the radiation experiment was not very high, only 1.66x. There are many factors that influenced the lower MWTF in the radiation test. One of the major causes is the presence of cache memories. The experiments show that ARM caches are very sensitive to radiation and prone to faults that become errors. Another concern is that simulation model does not include microarchitectural registers.

Data-flow technique VAR3+ implements duplication rule D2, which does not create redundancy in the main memory and, consequently, in the cache memories. Therefore, a fault affecting the L1 cache (enabled in the heavy ion experiment) is not detected by VAR3+. On the other hand, the current version of the OVPSim-FIM fault injector does not inject faults in cache memories. It only injects faults in the accessible registers. Anyhow, it is important to mention that the fault injection method must not be used to replace radiation because it cannot reproduce the complexity of the radiation flux and the complete hardware architecture implementation. The fault injection simulator was designed only for comparing the increase of reliability offered by different SIHFT techniques, but not to get estimations of absolute reliability values.

VI. Conclusions and Future Work

SIHFT techniques are less costly than hardware-based ones, but they present time and memory overheads. In this work, we went a step further than [28]. We showed that there is room to reduce overheads without degrading the fault coverage. SETA appears as a better solution to protect against control-flow errors. It achieves the same level of fault coverage as a state-of-the-art control-flow technique with lower overheads in performance and memory. However, SETA by itself is not enough to protect the processor against soft errors, it must be combined with a data-flow technique. The combination of SETA with VAR increases significantly the MWTF. When SETA is combined with VAR3+, the MWTF is 8.67x the MWTF of the unhardened application. However, radiation test reveals that ARM Cortex-A9 architecture presents high sensitivity to faults due to its cache memories. Therefore, it is important to use data-flow techniques that also apply redundancy to the memories (duplication rule D1). In addition, fault injection campaigns on the core architecture are not accurate enough to estimate the reliability of SIHFT techniques. Simulation models have to include cache memories to get better estimations in the design of SIHFT techniques.

Furthermore, one can see that data-flow techniques present higher overheads in performance and memory when compared to control-flow techniques. Selective methods to protect the data-flow shall be investigated to reduce such overheads.

References

[1] R. C. Baumann, "Soft errors in advanced semiconductor devices-part I: The three radiation sources," *IEEE Trans. Device Mater. Rel.*, vol. 1, no. 1, pp. 17–22, Mar. 2001.

[2] O. S. Unsal *et al.*, "Towards energy-aware software-based fault tolerance in real-time systems," *Proc. Int. Symp. Low Power Electron. and Design*, 2002, pp. 124–129.

[3] S. C. Asensi *et al.*, "A novel co-design approach for soft errors mitigation in embedded systems," *IEEE Trans. Nucl. Sci.*, vol. 58, no. 3, pp. 1059–1065, Jun. 2011.

[4] O. Goloubeva *et al.*, *Software-Implemented Hardware Fault Tolerance*, Boston, MA, USA: Springer Sci.+Bus. Media, 2006.

[5] T. Yao *et al.*, "Low power consumption scheduling based on software fault-tolerance," *Proc. 9th Int. Conf. Natural Computation*, Shenyang, China: 2013, pp. 1788–1793.

[6] I. Assayad *et al.*, "Tradeoff exploration between reliability, power consumption and execution time," *Proc. 30th Int. Conf. Comput. Safety, Rel. and Security*, Naples, FL, USA: 2011, pp. 437–451.

[7] P. Cheynet *et al.*, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," *IEEE Trans. Nucl. Sci.*, vol. 47, no. 6, pp. 2231–2236, Dec. 2000.

[8] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: Method, tools and experimental results," *Proc. Design, Automation and Test Eur. Conf. Exhib.*, 2003, pp. 57–62.

[9] R. Vemu and J. A. Abraham, "CEDA: Control-flow error detection using assertions," *IEEE Trans. Comput.*, vol. 60, no. 9, pp. 1233–1245, Sep. 2011.

[10] *Cortex-A9, Technical Reference Manual, Revision: r2p2*, ARM Holdings plc, Cambridge, MA, USA, 2010.

[11] G. A. Reis *et al.*, "SWIFT: Software implemented fault tolerance," *Proc. Int. Symp. Code Generation and Optimization*, 2005, pp. 243–254.

[12] N. Oh *et al.*, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Reliab.*, vol. 51, no. 1, pp. 63–75, Mar. 2002.

[13] J. R. Azambuja *et al.*, "Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors," *Proc. IEEE Latin American Symp. Circuits and Systems*, 2011.

[14] N. Oh *et al.*, "Control-flow checking by software signatures," *IEEE Trans. Reliab.*, vol. 51, no. 2, pp. 111–122, Mar. 2002.

[15] O. Goloubeva *et al.*, "Soft-error detection using control flow assertions," *Proc. 18th Int. Symp. Defect and Fault Tolerance in VLSI Systems*, 2003, pp. 581–588.

[16] E. Chielle *et al.*, "A set of rules for overhead reduction in data-flow software-based fault-tolerant techniques," *FPGAs and Parallel Architectures for Aerospace Applications*, F. Kastensmidt and P. Rech, Eds. Boston, MA, USA: Springer Sci.+Bus. Media, 2015, pp. 279–291.

[17] L. M. O. S. S. Hangout and S. Jan, *The Minimips Project*, Oct. 10, 2010, http://www.opencores.org/projects.cgi/web/minimips/overview.

[18] J. R. Azambuja *et al.*, "HETA: Hybrid error-detection technique using assertions," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 4, pp. 2805–2812, Aug. 2013.

[19] G. A. Reis *et al.*, "Design and evaluation of hybrid fault-detection systems," *Proc. 32nd Int. Symp. Comput. Architecture*, 2005, pp. 148–159.

[20] S. S. Mukherjee *et al.*, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," *Proc. 36th Annu. Int. Symp. Microarchitecture*, 2003, pp. 29–40.

[21] E. Chielle *et al.*, "Configurable tool to protect processors against SEE by software-based detection techniques," *Proc. 13th Latin American Test Workshop*, Quito: 2012, pp. 1–6.

[22] F. Rosa *et al.*, "A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability," *Proc. IEEE Int. Symp. Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, 2015, pp. 211–214.

[23] Imperas *Open Virtual Platforms (OVP)*, http://www.ovpworld.org/, Apr. 21, 2014.

[24] A. Shrivastava *et al.*, "Quantitative analysis of control flow checking mechanisms for soft errors," *Proc. 51st Design Automation Conf.*, San Francisco, CA, USA: 2014, pp. 1–6.

[25] Avnet *ZedBoard, featuring the Zynq-7000 All Programmable SoC*, http://www.em.avnet.com/en-us/design/drc/Pages/Zedboard.aspx, May 5, 2015.

[26] V. A. P. Aguiar *et al.*, "Experimental setup for single event effects at the são paulo 8UD pelletron accelerator," *Nucl. Instrum. Methods Phys. Res. A*, vol. 332, pp. 397–400, 2014.

[27] "ESA. ESA/SCC basic specification No. 25100 single event effects test methods and guidelines," Noordwijk, Netherlands: ESA, 2005.

[28] E. Chielle *et al.*, "Reliability on ARM processors against soft errors by a purely software approach," *Proc. Radiation and Its Effects on Components and Systems Conf.*, Moscow, Russia: 2015, pp. 443–447.