Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique

José Rodrigo Azambuja, Ângelo Lapolli, Lucas Rosa, and Fernanda Lima Kastensmidt, *Member, IEEE*

***Abstract*—This paper presents a hybrid technique based on software signatures and a hardware module with watchdog and decoder characteristics to detect SEU and SET faults in microprocessors. These types of faults have a major influence in the microprocessor's control-flow, affecting the basic blocks and the transitions between them. In order to protect the transitions between basic blocks a light hardware module is implemented in order to spoof the data exchanged between the microprocessor and its memory. Since the hardware alone is not capable of detecting errors inside the basic blocks, it is enhanced to support the new technique and then provide full control-flow protection. A fault injection campaign is performed using a MIPS microprocessor. Simulation results show high detection rates with a small amount of performance degradation and area overhead.**

***Index Terms*—Fault tolerance, hybrid fault tolerance techniques, microprocessors, SEEs.**

I. Introduction

THE last-decade advances of the semiconductor industry in transistor dimensions, low voltage supply and high density integration have led in the development of more and more complex architectures, combining large parallelism with high frequencies. However, the same technology that made possible all this progress has also reduced the transistor reliability by reducing threshold voltages, node capacitances and tightening the noise margins [1], [2]. These have made transistors more susceptible to faults caused by radiation interference, which can be energized particles presented on space or secondary particles such as alpha particles, generated by interaction of neutron and materials at ground level [3]. As a consequence, high reliable applications demand fault-tolerant techniques capable of recovering the system from a fault with minimum implementation and performance overhead.

One of the major effects that may occur when a single radiation ionizing particles strikes the silicon is known as single event effect (SEE). SEE can be destructive and non-destructive. An example of destructive effect is single event latchup (SEL) that results in a high operating current, above device specifications, that must be cleared by a power reset. Non-destructive effects, also called soft errors, are defined as a transient effect fault provoked by the interaction of a single energized particles in drain PN junction of the off-state transistors. This strike temporally charges or discharges the upset node of the circuit, generating transient voltage pulses that can be interpreted as internal signals, thus provoking an erroneous result [4]. When this transient pulse occurs in a memory cell, it is classified as single event upset (SEU), while when this transient occurs in a sensitive node of a combinational logic cell, it is classified as single event transient (SET). The use of fault tolerance techniques is mandatory to detect or correct these types of non-destructive faults, which can be based on software or hardware redundancy for microprocessors.

Software-based techniques rely on adding instruction redundancy and comparison to detect or correct faults. They are able to detect faults that affect the data and control-flow. Software-based techniques are non-intrusive because no modifications in the hardware of the microprocessor are required. Consequently, they provide high flexibility, low development time and cost. In addition, new generations of microprocessors that do not have RadHard versions in hardware can be used. As a result, aerospace applications can use commercial off-the-shelf (COTS) microprocessors with RadHard software. However, software-based techniques cannot achieve full system protection against soft errors [5], due to control-flow errors [6], [7]. This limitation is due to the inability of the software in protecting all the possible control-flow effects that can occur in the microprocessor.

Related works have pointed out the drawbacks of software-only techniques, such as high overhead in memory and degradation in performance. Memory increases due to the additional instructions and often requires duplication. Performance degradation comes from the execution of redundant instruction [8]–[10]. Some results from random fault injection have shown the impossibility of achieving complete fault coverage of SEU [5]–[7] when using software-only techniques.

Hardware-based techniques usually change the original architecture by adding logic redundancy, error correcting codes and majority voters. But they can be based on hardware monitoring and in this case are non-intrusive. They exploit special purpose hardware modules, called *watchdog processors* [11], to monitor memory accesses. Watchdogs usually can have access to the data and code memory connections. Since they only have access to the memory, watchdogs do not detect faults that are latent inside the microprocessor, as well faults in the register bank.

Combining software-based techniques and watchdog seems interesting in order to increase the fault detection coverage. In this paper, the authors present a method to detect soft errors in microprocessors using a non-intrusive light hardware module that works in tandem with the software-based technique. The

Manuscript received September 17, 2010; revised November 30, 2010; accepted December 22, 2010. Date of publication February 28, 2011; date of current version June 15, 2011.

The authors are with the Federal University of Rio Grande do Sul, Porto Alegre, Brazil.

Digital Object Identifier 10.1109/TNS.2011.2109398

goal is to increase the coverage of faults with control effect. A new method using hybrid signature is proposed. A fault injection campaign is performed in a MIPS microprocessor running two test-case applications. Simulation results prove the effectiveness and the feasibility of the proposed technique.

The paper is organized as follows. Section II presents previous work. Section III describes the proposed hybrid technique. Section IV describes the methodology and hardware implementation. Section V presents the fault injection campaign and results. Section VI presents main conclusions and future works.

II. Previous Work

Error detection techniques for software-based systems can be classified in three categories [12]: 1) Software-only techniques, that exploit the concepts of information, operation and time redundancy to detect the occurrence of errors during program execution, 2) hardware-only techniques, which exploit area redundancy, through additional hardware modules and 3) hybrid techniques, that combine software-based and hardware-based techniques, by adding additional hardware modules capable of analyzing additional instructions inserted in the program code or storing control-flow information in suitable hardware structures, such as registers or additional memories.

A. Software-Based Fault Tolerance Techniques

Software implemented hardware fault tolerance (SIHFT) techniques have been proposed in the past years that can be automatically applied to the source code of a program, thus simplifying the task for software developers: by protecting the system during software construction, the development costs can be reduced significantly [12].

A set of software-based techniques have been proposed in the literature, divided in two groups: 1) aiming at detecting data-flow errors, such as data instruction replication [13], [14] and 2) aiming at detecting control-flow errors through assertions [15], [16], such as structural integrity checking (SIC) [17], control-flow checking by software signatures (CFCSS) [18], control-flow checking using assertions (CCA) [19] and enhanced control-flow checking using assertions (ECCA) [20]. Proposed techniques can achieve full data-flow fault tolerance, concerning SEUs, being able to detect every fault affecting the data memory, which lead the system to a wrong result. On the other hand, the control-flow techniques have not yet achieved full fault tolerance.

Most techniques aiming at data-flow errors use pure instruction replication, without processing the program's execution flow. On the other hand, control-flow techniques require an analysis on the program's execution flow and therefore divide the program into basic blocks (BB) in order to analyze their transitions. Each BB starts at branch destination addresses and also at the memory position that follows the branch instruction. The end of a BB is at every jump instruction address and at the last instruction of the code.

ECCA extends CCA and is capable of detecting all control-flow errors between different BBs, but is neither able to detect errors inside the same BB, nor faults that cause incorrect decision on a conditional branch. CFCSS is not able to detect errors if multiple BBs share the same BB destination address.

`ld r1, [r4]`	**`1: bne r4, r4', error`** `2: ld r1, [r4]` **`3: ld r1', [r4' + offset]`**
`add r1, r2, r4`	**`4: bne r2, r2', error`** **`5: bne r4, r4', error`** `6: add r1, r2, r4` **`7: add r1', r2', r4'`**
`st [r1], r2`	**`8: bne r1, r1', error`** **`9: bne r2, r2', error`** `10: st [r1], r2` **`11: st [r1' + offset], r2'`**

Fig. 1. Datapath rules #1, #2 and #3.

Transformation rules have also been proposed in the literature as software-based techniques, aiming to detect both data and control-flow errors. In [13], eight rules are proposed, while [21] used thirteen rules to harden a program and [22] introduce several code transformation rules, from variable and operation duplication to consistency checks.

In [6] and [7], a set of eight rules was proposed to harden both control path and data path in a microprocessor. In the following, we describe each transformation rule, divided into rules aiming at detecting errors in the datapath and in the controlpath.

1) Errors in the Datapath: This group of rules aims at detecting the faults affecting the data, which comprises the whole path between memory elements, for example, the path between a variable stored in the memory, through the ALU, to the register bank. Every fault affecting these paths, as faults affecting the register bank or the memory should be protected with the following rules:

- Rule #1: variables must be duplicated;
- Rule #2: write operations performed on a variable must also be performed on its replica;
- Rule #3: before each read on a variable, its value and its replica's value must be checked for consistency.

Fig. 1 illustrates the application of these rules to a program with three instructions operating with registers and memory elements. Instructions 1 and 3 are inserted to protect the load instruction located in position 2 $(ld\ r1, [r4])$, where the first instruction verifies the register containing the base address for the load instruction $(r4)$ and its replica $(r4')$ and the second replicates the load instruction, using the replicated memory position $(r4' + offset)$ and loading the value into the replicated register $(r1')$. Instructions 8, 9 and 11 are inserted to protect the store instruction. While instructions 8 and 9 verify values stored in the base and data registers ($r1$ and $r2$, respectively) with their replicas ($r1'$ and $r2'$, respectively), instruction 11 replicates the original store instruction located in position 10 $(st[r1], r2)$ using the replicated registers $r1'$ and $r2'$ over a replicated memory address $(r1' + offset)$.

The original add instruction located in position 6 $(add\ r1, r2, r4)$ operates only over registers and therefore does not need any offset. In order to protect this instruction, instructions 4, 5 and 7 are inserted. The first two instructions verify the read registers $r2$ and $r4$ with their replicas ($r2'$ and

`beq r1, r2, 6`	`1: beq r1, r2, 5` **`2: beq r1, r2, error`**
`add r2, r3, 1`	`3: add r2, r3, 1`
	`4: jmp 6` **`5: bne r1, r2, error`**
`add r2, r3, 9` `jmp end`	`6: add r2, r3, 9` `7: jmp end`

Fig. 2. Controlpath rule #4.

`beq r1, r2, 6`	`1: beq r1, r2, 5`
`add r2, r3, 1`	**`2: mv rX, signature 1`** `3: add r2, r3, 1` **`4: bne rX, signature 1, error`**

`add r2, r3, 9`
`st [r1], r2`
	`5: mv rX, signature 2`
	`6: add r2, r3, 9`
	`7: st [r1], r2`
	`8: bne rX, signature 2, error`

`jmp end`	`9: jmp end`

Fig. 3. Controlpath rules #5 and #6.

$r4'$, respectively), while instruction 7 replicates the original instructions using the replicated registers ($r2'$ and $r4'$) and writing over the replicated register $r1'$.

These techniques duplicate the used data size, such as number of registers and memory addresses. Consequently, the applications must use a limited portion of the available registers and memory address. The compiler can perform this restriction when possible.

2) Errors in the Controlpath: This group of rules aims at protecting the program's flow. Faults affecting the controlpath usually cause erroneous jumps, such as an incorrect jump address or a bit-flip in a non-jump instruction's opcode, which becomes a jump instruction. To detect these errors, three rules are used in this paper:

- Rule #4: every branch instruction is replicated on both destination addresses;
- Rule #5: a unique identifier is associated to each BB in the code;
- Rule #6: At the beginning of each BB, a global variable is assigned with its unique identifier. On the end of the BB, the unique identifier is checked with the global variable.

Branch instructions are more difficult to duplicate than non-branch instructions, since they have two possible paths, when the branch condition is true or false. When the condition is false, the branch can be simply replicated and added right after the original branch, because the injected instruction will be executed after the branch. When the condition is taken, the duplicated branch instruction must be inverted and inserted on the branch taken address.

Fig. 2 illustrates rule #4 applied to a program code. The conditional branch instruction *Branch if EQual* located in position 1 $(beq\ r1, r2, 6)$ will jump to instruction 6 if registers r1 and r2 contain the same value. Initially, the branch will be replicated and inserted right after the original instruction, in position 2. The original branch instruction is then inverted and inserted in the original branch destination address (5) by using the *Branch if Not Equal* instruction $(bne\ r1, r2, error)$. In this process, original branch instruction destination addresses must be adjusted to the new address (5, in the transformed code).

The insertion of the replicated inverted branch instruction may affect other execution flows. For example, instruction 5 cannot be executed after the add instruction located in position 3 $(add\ r2, r3, 1)$, since it could modify the value stored in the $r2$ register and cause a false fault detection. In order to protect the other execution flows, the inverted branch must be protected with instruction 4, an unconditional branch that does not allow instruction 5 to be executed after instruction 3, but only after branch instruction with destination address pointing to its position.

The role of rules #5 and #6 is to detect every erroneous jump in the code. They achieve this by inserting a unique identifier to the beginning of each BB and checking its value on its end.

Fig. 3 illustrates a program code divided in two BBs, containing the instructions located in positions 3 (BB 1) and 6-7 (BB 2) in the original code. In order to apply rules #5 and #6, instructions 2, 4, 5 and 8 are inserted. The first two protect the first BB, assigning the unique identifier *signature1* to the global variable rX with instruction 2 $(mv\ rX, signature\ 1)$ and verifying its value when exiting the BB, with instruction 4 $(bne\ rX, signature\ 1,\ error)$. Note that the original branch instruction located in position 1 $(beq\ r1, r2, 6)$ had its destination address modified from 6 to 5, in order to jump to the new beginning of the BB. Instructions 5 and 6 have the same role as instructions 2 and 4, respectively, but in order to protect the second BB.

B. Hardware-Only Techniques

Hardware-based techniques exploit special purpose hardware modules, called *watchdog processors* [11], to monitor the control-flow of programs, as well as memory accesses. There are three types of operations which monitor the behavior of the main processor while running the application.

Memory access checks consist in monitoring for unexpected memory accesses executed by the main processor. In [23] the watchdog processor knows during program execution which part of the program's data and code can be accessed and activates an error signal whenever an unexpected access is detected.

Consistency checks of the variables' contents consist in controlling whether the value a variable holds is plausible. Using information about the task performed by the hardened program, watchdog processors can check each value the main processor writes through range analysis, or by exploiting known relationships among variables [24].

Control-flow checks consist in monitoring whether all the taken branches are consistent with the Program Graph of the software running on the main processor [25]–[28]. Two types of watchdog processors may be envisioned. The *active watchdog processors* execute a program concurrently with the main processor checking whether their program evolves accordingly with that executed by the main processor. The *passive watchdog processors* do not run any program; they

compute a signature by observing the main processor's bus and then perform consistency checks.

Dynamic verification is another hardware-based technique which is described in [29] for a pipelined processor. It uses a "functional checker" that only permits correct results to be passed to the commit stage of the processor pipeline. When the result of the core processor differs from the one obtained by the checker, the one from the checker is given as correct, assuming it never fails. This assumption is based on the use of oversized transistors in the checker's construction in opposition to the core processor's.

C. Hybrid Techniques

Taking [30] as an example, hybrid techniques adopt some SIHFT techniques in a minimal version along with the introduction of an I-IP into the SoC. The software running on the processor core is modified so that it implements instruction duplication and information redundancy; besides that, instructions to communicate to the I-IP the information about BB execution information are added. The I-IP works concurrently with the main processor doing consistency checks among duplicated instructions and verifying the correctness of the program flow by monitoring the BB execution.

Hybrid techniques are effective considering they provide a high level of dependability while minimizing memory occupation and performance degradation. On the other hand, in order to be adopted they demand the source code of the application the processor core should run, and this is not always available.

Introducing an I-IP between the processor and the instruction memory, and charging it of substituting on-the-fly the fetched code with hardened one is proposed in [31]. However, the I-IP did not include either an ALU or a control unit and this is not supported by a suitable design flow environment, as proposed here. In addition to that, the method in [31] has a significant performance overhead and it cannot cover permanent faults.

III. Proposed Hybrid Signature-Based Technique

The proposed approach aims at allowing developers to build systems with hardened processor cores without concerning about customizing the hardware involved with module replication and intrusive techniques, such as register replication and triple modular redundancy (TMR).

As stated in [6], software-based techniques aiming at control-flow errors cannot achieve full fault tolerance due to a few types of error effects: 1) incorrect jumps to the beginning of BBs, 2) incorrect jumps inside the same BB, 3) incorrect jumps to unused memory addresses, 4) control-flow loops and 5) incorrect jumps to branch instructions, which will lead to the beginning of a BB. Our approach exploits two main concepts in order to protect the system against these faults:

- *Software-based transformation rules*: the original program code is transformed based on a set of rules and additional instructions are inserted in order to control the hardware module.
- *Hardware-based non-intrusive module*: an additional hardware with watchdog and decoder characteristics is developed and added to the system non-intrusively in order to analyze the processors' control-flow and decode instructions sent from the inserted software instructions.

The innovation of this approach relies on the use of a signature mechanism technique that works in tandem with a watchdog circuit to be able to detect all upsets that affect control flow. In the following, we describe how these two concepts are implemented.

A. Software-Based Proposed Transformation Rules

The proposed approach based on software techniques relies on extra instructions to protect the system against incorrect jumps to the same BB and incorrect jumps to the beginning of BBs. These fault tolerance transformation rules require constant program flow analysis, which can be quite heavy to be implemented in software. In order to reduce the performance degradation, most of the computation will be executed by the hardware module, which will be controlled by the inserted instructions.

1) Jumps to the Beginning of Basic Blocks: In order to protect the system against jumps to the beginning of BBs (when BB initialization is performed), a control-flow graph is required, once the only option to detect an error is to analyze the source and destiny of the transition between different BBs.

Using this technique, every BB is assigned with two identifiers: the Block IDentifier (BID) represents each BB with a unique prime number and the Control-Flow IDentifier (CFID) represents the control-flow, by storing the multiplication product of the next BBs' BIDs. Since the BIDs are composed by prime numbers, the operation remainder of division between a CFID and a BID will always return zero, unless a wrong control-flow transition has been performed.

CFIDs are stored in a two element queue, initialized with the first BB's CFID. Upon entry to a BB, its CFID is enqueued. Upon exiting a BB, the first CFID is dequeued and divided by the BID. Errors are detected when one of these situations occur: 1) the remainder of the division is not zero, 2) overflow in the queue or 3) underflow in the queue.

The queue management is a heavy task to be performed purely in software and would result in a huge memory and performance loss. Therefore, the hardware module should have a decoding ability to perform the queue management and the operation remainder of division when controlled by the software and informed with the CFIDs and BBs' BID.

Fig. 4 shows the code transformation required to send the necessary values to the hardware module represented by instructions 2 and 6 (send BID), 3 and 7 (enqueue CFID) and 5 and 10 (dequeue CFID). These instructions are implemented in assembly with store instructions with predefined addresses.

2) Jumps to the Same Basic Block: Incorrect jumps to the same BB require intra-block control and therefore each instruction within a BB must be considered. In order to detect such faults, the proposed hybrid technique computes each BB signature by XOR'ing all instructions inside a BB both during runtime and program code construction (compilation).

BB signatures are pre-computed by the compiler and sent to the hardware module through additional instructions whenever the program's execution flow exits from a BB. The hardware

`beq r1, r2, 8`	`1: beq r1, r2, 6`
`add r2, r3, 1`	**`2: send BID`** **`3: enqueue CFID`** `4: add r2, r3, 1` **`5: dequeue CFID`**

`add r2, r3, 4`
`st [r1], r2`
	`6: send BID`
	`7: enqueue CFID`
	`8: add r2, r3, 4`
	`9: st [r1], r2`
	`10: dequeue CFID`

`jmp end`	`11: jmp end`

Fig. 4. Jumps to the beginning of BBs proposed transformation rules.

`beq r1, r2, 6`	`1: beq r1, r2, 5`
`add r2, r3, 1`	**`2: reset XOR`** `3: add r2, r3, 1` **`4: check XOR`**

`add r2, r3, 4`
`st [r1], r2`
	`5: reset XOR`
	`6: add r2, r3, 4`
	`7: st [r1], r2`
	`8: check XOR`

`jmp end`	`9: jmp end`

Fig. 5. Jumps to the same BB proposed transformation rules.

module should be modified to calculate BB signatures by performing XOR operations in real time. When the software sends an instruction with a signature value, the hardware module verifies it with its calculated value. When a mismatch is found, an error is notified. In order to inform the hardware module an entry in a BB, a reset instruction should be used.

Fig. 5 shows the transformation required on an assembly code to adapt the software to the hardware module with instructions 2 and 5 (reset XOR) and 4 and 8 (check XOR). These instructions are also implemented in assembly through store instructions with predefined special addresses.

B. Hardware-Based Proposed Module

The proposed approach based on hardware relies on a module that combines a watchdog with a decoder. The watchdog is used in order to detect incorrect jumps to unused memory addresses and control-flow loops, while the decoder spoofs data and address buses and the read/write signal between the microprocessor and the memory in order to perform the instructions sent by the software-based transformation rules, by computing the program's flow analysis. Fig. 6 shows the overall architecture, with the hardware module connected to a microprocessor.

In order to perform its tasks, the hardware module has to have access to the memory buses and implement a 2-element queue, a decoder module and a remainder of division operation. The buses used by some microprocessors which use on-chip embedded cache memories may not be accessible by the hardware module. In such cases, another approach should be used.

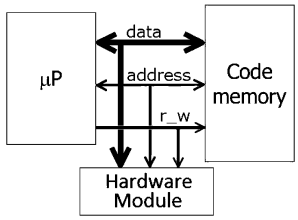

Fig. 6. Proposed hardware module connected to a microprocessor.

IV. Methodology and Implementation

This section describes the methodology to create hardened codes according to the proposed transformation rules and the implementation of the hardware module.

A. Hardening Post Compiling Translator Tool

The code transformation to apply a set of rules is a complex task, which involves code analysis and processing, instruction replication and branch instructions' addresses correction. Even the BB construction (required for some transformation rules) can be an exhaustive task when performed by hand.

In order to automate the software transformation, we built a tool called *Hardening Post Compile Translator* (HPC-Translator). Implemented in Java, the HPC-Translator tool is able to automatically transform an unprotected program into a hardened one, by inserting additional instructions and error subroutines to the software.

The HPC-Translator receives as input the program's binary code and therefore is compiler and language independent. It is then capable of implementing the presented rules, divided into groups. The first group, called variables, implements rules #1, #2 and #3 presented in Section II-A; the second group, called inverted branches, implements rule #4 described in Section II-A, the third group, also known as signatures, implements rules #5 and #6 presented in Section II-A, while the last group implements the proposed techniques, described in Section III-A. The user is allowed to combine these techniques in a graphical interface. The implemented tool outputs a binary code, microprocessor dependent, which can be directly interpreted by the target microprocessor.

Fig. 7 shows the HPC-Translator's workflow. The tool receives four distinct inputs: the original program code, the user choices of protection techniques, the instruction set architecture (ISA) definition and a file describing the microprocessor's architecture. Using these inputs, the HPC-Translator is able to generate a hardened program code.

B. Proposed Hardware Module Implementation

The hardware module was implemented in VHDL language based on a timer that signals an error if not reset after a given number of clocks. Its enhancement was performed by adding a 16-bit register to store the real-time calculated XOR value, a 64-bit register to store the 2-element queue, a remainder of division module and a control module to spoof the instructions described in Section II, such as check CFID and reset XOR value.

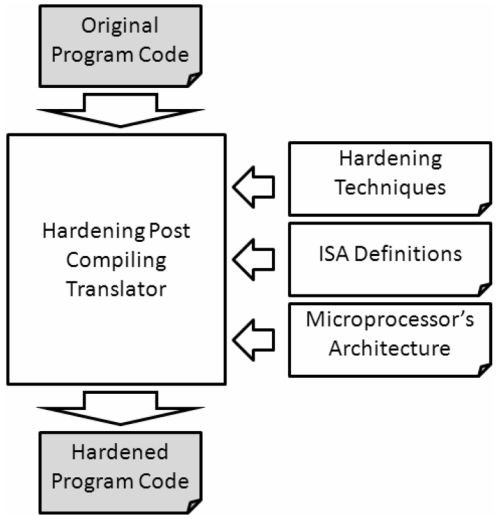

Fig. 7. HPC-Translator work flow.

TABLE I
ORIGINAL AND MODIFIED ARCHITECTURE CHARACTERISTICS

Source	miniMIPS	HW Module
Area (μm)	24261.32	3640.21 (+15%)
Frequency (MHz)	66.7	66.7

The module's control is implemented through a decoder which analyzes the data and address buses looking for instructions. Whenever a store instruction is detected in a predefined address, the data bus is spoofed and an operation is performed by the module.

Table I shows the size and performance of the implemented microprocessor and the hardware module. The hardware module implementation has a total of 128 registers and is not protected against SEEs, but could be protected with custom hardware-base techniques, such as DMR or TMR approaches. The implemented module connected to the microprocessor has a 15% overhead, while it maintains the same frequency.

V. FAULT INJECTION SIMULATION RESULTS

The chosen case-study microprocessor is a five-stage pipeline microprocessor based on the MIPS architecture, but with a reduced instruction set. The miniMIPS microprocessor is described in [32]. In order to evaluate both the effectiveness and the feasibility of the presented approaches, two applications were chosen: a 6×6 matrix multiplication algorithm and a bubble sort algorithm. The matrix multiplication requires a large data processing with only a few loops and therefore uses mostly the datapath from the microprocessor. On the other hand, the bubble sort algorithm uses a large number of loops, control registers and branch instructions and therefore uses mostly the controlpath, since all the data processing is related to the control registers.

Five hardened programs were generated using the *Hardening Post Compiling Translator*, each one implementing the following techniques (described in Section IV-A): (I) variables, (II) signatures combined with inverted branches, (III) proposed technique, (IV) signatures, inverted branches and variables combined and (V) variables and proposed technique combined. Table II shows the original and modified program's execution time, code size and data size from the matrix multiplication and bubble sort algorithms. As one can see, the proposed technique varies from 32% to 61% execution time overhead.

TABLE II
IMPLEMENTED SOFTWARE TRANSFORMATION CHARACTERISTICS

Matrix Multiplication						
Source	Original	I	II	III	IV	V
Exec. Time (ms)	1.26	2.54	1.48	1.67	2.91	2.94
Code Size (bytes)	1548	3832	3484	3372	5268	5576
Data Size (bytes)	524	1048	532	528	1056	1052
Bubble Sort						
Source	Original	I	II	III	IV	V
Exec. Time (ms)	0.23	0.44	0.29	0.37	0.53	0.58
Code Size (bytes)	1212	2692	2168	2440	3732	3960
Data Size (bytes)	120	240	128	124	248	244

In order to start the fault injection campaign, 50 000 faults were injected in all signals of the non-protected microprocessor (including registered signals), one per program execution for both applications. The SEU and SET types of faults were injected directly in the microprocessor VHDL code by using ModelSim XE/III 6.3c [33]. SEUs were injected in registered signals, while SETs were injected in combinational signals, both during one and a half clock cycle. The fault injection campaign is performed automatically. At the end of each execution, the results stored in memory were compared with the expected correct values. If the results matched, the fault was discarded. The amount of faults masked by the program is application related and it should not interfere with the analysis. In the end, only faults not masked by the application were considered in the analysis. When 100% signal coverage was achieved and at least 4 faults per signal were detected we normalized the faults, varying from 4 to 5 faults per signal. Those faults were used to build the test case list.

The faults were classified by their source and effect on the system. We defined four groups of fault sources to inject SEU and SET types of faults: datapath, controlpath, register bank and ALU. Program and data memories are assumed to be protected by error detection and correction (EDAC) and therefore faults in the memories were not injected.

The fault effects were classified into two different groups: program data and program flow, according to the fault effect. To sort the faults among these groups, we continuously compared the program counter (PC) of a golden microprocessor with the PC of the faulty microprocessor. In case of a mismatch, the injected fault was classified as flow effect. If the PC matched with the golden's, the fault was classified as a data effect.

When transforming the program, new instructions were added and as a result the time in which the faults were injected changed. Since the injection time is not proportional to the total execution time, we mapped each fault locating the instruction where the fault was injected (by locating its new PC) and

TABLE III
SET AND SEU FAULT INJECTION IN THE MATRIX MULTIPLICATION AND BUBBLE SORT ALGORITHMS: SOURCE AND EFFECT CLASSIFICATIONS AND DETECTION COVERAGE OF (I) VARIABLES, (II) SIGNATURES COMBINED WITH INVERTED BRANCHES, (III) PROPOSED TECHNIQUE, (IV) VARIABLES, SIGNATURES AND INVERTED BRANCHES COMBINED AND (V) VARIABLES AND PROPOSED TECHNIQUE COMBINED

Matrix Multiplication Algorithm

	Source Classification	Data Effect Faults	Hardened program versions (%)					Flow Effect Faults	Hardened program versions (%)				
			(I)	(II)	(III)	(IV)	(V)		(I)	(II)	(III)	(IV)	(V)
SET	Reg. Bank	9	100	0	0	100	100	1	100	0	0	100	100
	ALU	27	100	0	9	100	100	10	100	0	0	100	100
	Controlpath	83	100	0	7	100	100	33	44	7	56	46	100
	Datapath	42	100	0	3	100	100	2	100	0	100	100	100
	Total	131	100	0	6	100	100	46	60	5	44	61	100
SEU	Reg. Bank	25	100	0	0	100	100	13	100	0	15	100	100
	ALU	4	-	-	-	-	-	0	-	-	-	-	-
	Controlpath	67	100	0	13	100	100	36	38	20	68	43	100
	Datapath	18	100	0	0	100	100	7	100	17	0	100	100
	Total	114	100	0	7	100	100	56	60	16	47	64	100

Bubble Sort Algorithm

	Source Classification	Data Effect Faults	Hardened program versions (%)					Flow Effect Faults	Hardened program versions (%)				
			(I)	(II)	(III)	(IV)	(V)		(I)	(II)	(III)	(IV)	(V)
SET	Reg. Bank	3	100	5	67	100	100	4	100	0	0	100	100
	ALU	7	100	0	100	100	100	14	100	0	14	100	100
	Controlpath	22	100	0	83	100	100	89	69	8	42	80	100
	Datapath	14	100	0	69	100	100	28	100	0	0	100	100
	Total	46	100	2	80	100	100	135	76	5	29	82	100
SEU	Reg. Bank	2	100	0	0	100	100	33	100	12	18	100	100
	ALU	0	-	-	-	-	-	0	-	-	-	-	-
	Controlpath	24	100	0	5	100	100	81	73	2	35	71	100
	Datapath	4	100	0	0	100	100	19	100	0	6	100	100
	Total	30	100	0	4	100	100	133	85	5	27	82	100

pipeline stage where the fault was manifested. Around 1% of the total number of faults could not be mapped and were changed by new faults.

Results presented in Table III show that the technique called variables (I) is able to detect all faults with data effects and all faults injected in the ALU, register bank and datapath for both matrix multiplication and bubble sort algorithms. On the other hand, they could not achieve full fault tolerance, due to faults injected in the controlpath with control-flow effects.

Techniques II and III were implemented in order to protect the system against faults with control-flow effects. Technique II presents low detection rates, varying from 5% to 16% in faults with control-flow effects. Technique III also presented low detection rates, varying from 27% to 47% in faults with control-flow effects. When they are compared between each other, technique III presents a detection rates over 3 times higher than technique II.

Techniques II and III also showed some fault detection over faults with data effects. This is due to the fault mapping process between each protected software implementation. Depending on the signal upset by a fault, the effect can be in the control-flow for the protected software even when the original fault caused only a data effect.

When technique II is combined with technique I, called technique IV in the table, their detection rates were also jointed, which has improved the overall detection rates. The combination of the proposed technique III with technique I, called technique V in the table, was capable of achieving full fault tolerance. On the other hand, IV could not achieve 100%, due to faults with control-flow effects. Results show that by selecting suitable fault tolerance techniques, even if each one may present a low detection rate, it is possible to find a combination of techniques where each detection rate may be combined in order to result in high detection rates.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, the authors presented a fault tolerant hybrid technique based on signatures and implemented partially over a hardware module with watchdog processor and decoder characteristics. Then, a tool was used to automatically harden two case study applications based on matrix multiplication algorithm and bubble sort algorithm.

A set of faults was then built and a fault injection campaign based on simulation was performed over the two case studies hardened with previous known techniques and the proposed technique to evaluate both their effectiveness and feasibility.

Results show that the variables and proposed techniques combined presented 100% fault detection with an overhead in execution time varying from 133% to 153%, due to the high overhead caused by the variables technique.

We are currently working on decreasing the impact on execution time and memory overhead of both the proposed techniques and the variables technique, while keeping the same fault detection rates. As future work, we also intend to verify the feasibility and efficiency of the proposed techniques applied to a real time operating system.

REFERENCES

[1] R. C. Baumann, "Soft errors in advanced semiconductor devices—Part I: The three radiation sources," *IEEE Trans. Device Mater. Reliab.*, vol. 1, pp. 17–22, Mar. 2001.

[2] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh, "Field testing for cosmic ray soft errors in semiconductor memories," *IBM J. Res. Develop.*, pp. 41–49, Jan. 1996.
[3] International Technology Roadmap for Semiconductors: 2005 Edition, Chapter Design, pp. 6–7, 2005.
[4] P. E. Dodd and L. W. Massengill, "Basic mechanism and modeling of single-event upset in digital microelectronics," *IEEE Trans. Nucl. Sci.*, vol. 50, pp. 583–602, 2003.
[5] C. Bolchini, A. Miele, F. Salice, and D. Sciuto, "A model of soft error effects in generic IP processors," in *Proc. 20th IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems*, 2005, pp. 334–342.
[6] J. R. Azambuja, F. Sousa, L. Rosa, and F. L. Kastensmidt, "The limitations of software signature and basic block sizing in soft error fault coverage," in *Proc. IEEE Latin-American Test Workshop*, 2010.
[7] J. R. Azambuja, F. Sousa, L. Rosa, and F. L. Kastensmidt, "Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors," in *Proc. IEEE Latin American Symp. Circuits and Systems*, 2010.
[8] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proc. 18th IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems—DFT 2003*, Nov. 2003, pp. 581–588.
[9] K. H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, pp. 518–528, Dec. 1984.
[10] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control flow checking by software signatures," *IEEE Trans. Reliab.*, vol. 51, no. 2, pp. 111–112, Mar. 2002.
[11] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors–A survey," *IEEE Trans. Comput.*, vol. 37, pp. 160–174, 1988.
[12] E. Rhod, C. Lisboa, L. Carro, M. S. Reorda, and M. Violante, "Hardware and software transparency in the protection of programs against SEUs and SETs," *J. Electron Test*, no. 24, pp. 45–56, 2008.
[13] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," in *Proc. IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems*, 1999, pp. 210–218.
[14] C. Bolchini, L. Pomante, F. Salice, and D. Sciuto, "Reliable system specification for self-checking datapaths," in *Proc. Conf. Design, Automation and Test in Europe*, Washington, DC, 2005, pp. 1278–1283, IEEE Computer Society.
[15] R. Vemu and J. A. Abraham, "CEDA: Control-flow error detection through assertions," in *Proc. IEEE Int. On-Line Testing Symp.*, 2006.
[16] R. Vemu, S. Gurumurthy, and J. A. Abraham, "ACCE: Automatic correction of control-flow errors," in *Proc. IEEE Int. Test Conf.*, 2007.
[17] D. J. Lu, "Watchdog processors and structural integrity checking," *IEEE Trans. Comput.*, vol. C-31, no. 7, pp. 681–685, Jul. 1982.
[18] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Reliab.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.
[19] L. D. Mcfearin and V. S. S. Nair, "Control-flow checking using assertions," in *Proc. IFIP Int. Working Conf. Dependable Computing for Critical Applications (DCCA-05)*, Urbana-Champaign, IL, Sep. 1995.
[20] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 627–641, Jun. 1999.
[21] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: Method, tools and experimental results," in *Proc. Design, Automation and Test in Europe Conf. Exhibition*, 2003.
[22] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. S. Reorda, and M. Violante, "Experimentally evaluating na automatic approach for generating safety—Critical software with respect to transient errors," *IEEE Trans. Nucl. Sci.*, vol. 47, no. 6, pt. 3, pp. 2231–2236, Dec. 2000.
[23] M. Namjoo and E. J. McCluskey, "Watchdog processors and capability checking," in *Proc. 12th Int. Symp. Fault-Tolerant Computing (FTCS-12)*, 1982, pp. 245–248.
[24] A. Mahmood, D. J. Lu, and E. J. McCluskey, "Concurrent fault detection using a watchdog processor and assertions," in *Proc. IEEE Int. Test Conf. (ITC)*, 1983, pp. 622–628.
[25] M. Namjoo, "CERBERUS-16: An architecture for a general purpose watchdog processor," in *Proc. 13th Int. Symp. Fault-Tolerant Computing (FTCS)*, 1983, pp. 216–219.
[26] J. Ohlsson and M. Rimen, "Implicit signature checking," in *Digest of Papers of the 25th Int. Symp. Fault Tolerant Computing (FTCS-25)*, 1995, pp. 218–227.
[27] M. A. Schuette and J. P. Shen, "Processor control flow monitoring using signatured instruction streams," *IEEE Trans. Comput.*, vol. 36, no. 3, pp. 264–276, Mar. 1987.
[28] K. Wilken and J. P. Shen, "Continuous signature monitoring: Lowcost concurrent detection of processor control errors," *IEEE Trans. Comput-Aided Des. Integr. Circuits Syst.*, vol. 9, no. 6, pp. 629–641, Jun. 1990.
[29] T. M. Austin, "DIVA: A dynamic approach to microprocessor verification," *J. Instruction Level Parallelism*, vol. 2, pp. 1–6, May 2000.
[30] P. Bernardi, L. M. V. Bolzani, M. Rebaudengo, M. S. Reorda, F. L. Vargas, and M. Violante, "A new hybrid fault detection technique for systems-on-a-chip," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 185–198, 2006.
[31] M. Schillaci, M. S. Reorda, and M. Violante, "A new approach to cope with single event upsets in processor-based systems," in *Proc. 7th IEEE Latin-American Test Workshop*, Mar. 2006, pp. 145–150.
[32] L. M. O. S. S. Hangout and S. Jan, The Minimips Project, 2009. [Online]. Available: http://www.opencores.org/projects.cgi/web/minimips/overview.
[33] Mentor Graphics, 2009. [Online]. Available: http://www.model.com/content/modelsim-support.