J Electron Test (2011) 27:541–550
DOI 10.1007/s10836-011-5218-7

Exploring the Limitations of Software-based Techniques in SEE Fault Coverage

José Rodrigo Azambuja · Samuel Pagliarini · Lucas Rosa · Fernanda Lima Kastensmidt

Received: 1 September 2010 / Accepted: 15 March 2011 / Published online: 9 April 2011
© Springer Science+Business Media, LLC 2011

Abstract This paper presents a detailed analysis of the efficiency of software-based techniques to mitigate SEU and SET in microprocessors. A set of well-known rules is presented and implemented automatically to transform an unprotected program into a hardened one. SEU and SET are injected in all sensitive areas of a MIPS-based microprocessor architecture. The efficiency of each rule and a combination of them are tested. Experimental results show the limitations of the control-flow techniques in detecting the majority of SEU and SET faults, even when different basic block sizes are evaluated. A further analysis on the undetected faults with control flow effect is done and five causes are explained. The conclusions may lead designers into developing more efficient techniques to detect these types of faults.

Keywords Control flow signatures · Fault tolerance · SEU · SET · Soft errors · Software-based techniques

Responsible Editor: F. Vargas

A presentation based on this article was made at the Eleventh IEEE Latin-American Test Workshop, Punta del Este, Uruguay, March 28-31, 2010.

J. R. Azambuja (✉) · S. Pagliarini · L. Rosa · F. L. Kastensmidt
Instituto de Informática—PPGC—PGMICRO,
Universidade Federal do Rio Grande do Sul (UFRGS),
Av. Bento Gonçalves 9500,
Porto Alegre, RS, Brazil
e-mail: jrfazambuja@inf.ufrgs.br

S. Pagliarini
e-mail: snpagliarini@inf.ufrgs.br

L. Rosa
e-mail: lucas.rosa@inf.ufrgs.br

F. L. Kastensmidt
e-mail: fglima@inf.ufrgs.br

1 Introduction

The last-decade advances of the semiconductor industry in transistor dimensions, low voltage supply and high density integration have led in the to the development of more and more complex architectures, combining large parallelism with high frequencies. However, the same technology that made all these progresses possible has also reduced the transistor reliability, either by reducing threshold voltages, node capacitances or tightening the noise margins [5, 11]. These have made transistors more susceptible to faults induced by radiation interference, which can be caused by particles from space or secondary particles such as alpha particles, generated by the interaction of neutron and materials at ground level [1]. As a consequence, high reliability applications demand 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 particle 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 [9]. The transient pulse is classified as *Single Event Upset* (SEU) when it occurs in a memory cell or *Single Event Transient* (SET) when it occurs in a sensitive node of a combinational logic cell.

Fault tolerance techniques based on software can provide high flexibility, low development time and low cost

solutions for computer-based dependable systems, because modifications in the hardware of the microprocessor are not required. In addition, new generations of microprocessors that do not have RadHard hardware versions can be used. As a result, aerospace applications can use commercial off-the-shelf (COTS) microprocessors with hardened software. High efficiency systems called systems-on-chip (SoC), which usually use COTS microprocessors, are getting more popular in many applications that require high reliability. Examples of such systems are data servers, transportation vehicles, satellites and others. Such systems are composed of a large number of microprocessors and other cores connected through a network on chip. (NoC). On such cases, the designers are responsible for hardening their applications. Therefore, fault tolerance by means of software-based techniques have been receiving a lot of attention in the past years.

Software-based fault tolerance techniques exploit information redundancy, control flow analysis and comparisons to detect errors during the program execution. For that purpose, software-based techniques use additional instructions in the code area, either to recompute instructions or to store and to check suitable information in memory elements. In the past years, tools have been implemented to automatically insert such instructions into C or assembly code, reducing significantly the hardening costs.

Related works have pointed out drawbacks of software-based techniques, such as high overhead in memory and degradation in performance. Memory increases due to the additional instructions and often requires memory duplication. Performance degradation comes from the execution of redundant instructions [10, 13, 18]. Some results from random fault injection have shown the impossibility of achieving complete fault coverage of SEU [3, 4, 6] when using software-based techniques. But these works do not correlate the errors with each software-based technique and their capability of detecting faults. In addition, there is no study in the literature that analyzes both SEU and SET faults and correlates the fault location and effects with a detected or undetected status.

It is well-known that SET is becoming the major issue in high performance circuits. This behavior is due to the high probability of capturing SETs when considering a hardware that operates in a high clock frequency. The analysis of SET and SEU and the association of each fault tolerant technique to a set of injected faults is important. Such mapping could guide designers in order to improve efficiency and detection rates of soft error mitigation techniques based on software.

In this paper, the authors implement a set of software-based techniques to harden the execution of algorithms against SEU and SET faults. Two algorithms were chosen: matrix multiplication and bubble sort. The microprocessor being considered in the evaluations that follows is the miniMIPS [12]. Faults were classified by location and effect. The implemented techniques target data and control flow fault tolerance. Results have shown that many faults that led to control flow error could not be detected by common and well-know control flow fault tolerance based techniques. The effect of basic block sizing combined with software signature was investigated to improve fault coverage. However, software-based techniques present limitations on fault detection that cannot be solved without considering some hardware characteristics. Results highlight the main vulnerable areas by plotting the upset area and the detected and non-detected faults for each one. Also, the unresolved issues were described.

The paper is organized as follows. Section 2 describes the state-of-the-art. Section 3 presents the case-study software-based techniques evaluated in this paper. Section 4 presents the fault injection campaign, results and shows the evaluation of block sizing with signature control and the limitations on fault detection. Section 5 presents main conclusions and future work.

2 State-of-the-Art

A set of transformation rules has been proposed in the literature. In [21], eight rules are proposed, divided in two groups: (1) aiming data-flow errors, such as data instruction replication [7, 21] and (2) aiming control-flow errors, such as Structural Integrity Checking [14], Control-Flow Checking by Software Signatures (CFCSS) [19], Control Flow Checking using Assertions (CCA) [15] and Enhanced Control Flow Checking using Assertions (ECCA) [2]. The proposed techniques can achieve full data-flow fault tolerance concerning SEUs. It means that these techniques are able to detect every fault affecting the data memory that led the system to a wrong result. On the other hand, control-flow techniques have not yet achieved full fault tolerance.

Most control-flow techniques divide the program into basic blocks (BB). A basic block starts at a branch destination address or at a memory position that follows the branch instruction. The end of a basic block is at a jump instruction address, at the beginning of a new basic block or at the last instruction of the code.

ECCA extends CCA and is capable of detecting all the inter-BB control flow errors, but is neither able to detect intra-BB errors, 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. In [8], several code transformation rules are presented, from variable and operation duplication to consistency checks.

Transformation rules have been proposed in the literature aiming to detect both data and control-flow errors. In [21], eight rules are proposed, while [17] uses thirteen rules to harden a program.

3 The Case-Study Software-based Techniques

In this paper, we address six code transformation rules, proposed in [3] and based on different fault-tolerant techniques present in the literature. These rules are divided into faults affecting the datapath and the controlpath.

3.1 Errors in the Datapath

This group of rules, based on a technique called [20], 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 well as faults affecting the register bank or the memory should be protected with the following rules:

- Rule #1: every variable used in the program must be duplicated;
- Rule #2: every write operation performed on a variable must 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.

Figure 1 illustrates the application of these rules to a program with three instructions that operates 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′*). The second instruction replicates the load instruction, using the replicated memory position (*r4′* + *offset*) and loads 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) against 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*).

Original Code	Hardened Code
2: ld r1, [r4]	**1: bne r4, r4′, error** 2: ld r1, [r4] **3: ld r1′, [r4′ + offset]**
6: add r1, r2, r4	**4: bne r2, r2′, error** **5: bne r4, r4′, error** 6: add r1, r2, r4 **7: add r1′, r2′, r4′**
10: 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

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 registers that are read (*r2* and *r4)* against their replicas (*r2′* and *r4′*, respectively). Instruction 7 performs the original instruction, but using the replicated registers (*r2′* and *r4′*) and writing over the replicated destination register (*r1′*).

These rules duplicate the data being stored, i.e., the number of registers and memory addresses. Consequently, the applications are limited to a portion of the available registers and memory address. In some cases, compilers can restrict the application to a small set of registers and memory addresses, allowing the duplication. If not, the rules can be applied to a subset of the used registers and memory positions, but also lowering the fault detection rates.

3.2 Errors in the Controlpath

This second group of rules, based on [15, 17, 21, 22], aims at protecting the program's execution flow. Faults affecting the controlpath usually cause erroneous jumps, either by causing a jump to an incorrect address or, in some cases, a bit-flip in a non-jump instruction, which is interpreted as a jump instruction. To detect these errors, three rules are used:

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

Branch instructions are more difficult to duplicate than non-branch instructions. This is due to the fact that they have two possible paths, since either the branch condition is true or false. Ùùwhen the condition is false (branch not taken) the branch can be simply replicated. The new instruction is added right after the original branch. Otherwise, when the condition is true (branch taken), the duplicated branch instruction must be inverted and inserted on the branch taken addess.

Figure 2 illustrates the rule #4 being 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 Code	Hardened Code
1: beq r1, r2, 6	1: beq r1, r2, 5 **2: beq r1, r2, error**
3: add r2, r3, 1	3: add r2, r3, 1
	4: jmp 6 **5: bne r1, r2, error**
6: add r2, r3, 9 7: jmp end	6: add r2, r3, 9 7: jmp end

Fig. 2 Controlpath rule #4

original branch destination address (5). This is achieved by using the *Branch if Not Equal* instruction (*bne r1, r2, error*). In this process, the original branch instruction destination address must be adjusted to the new address (5, in the hardened code).

The insertion of the replicated inverted branch instruction may affect other execution flows. For example, the instruction in position 5 cannot be executed after the add instruction located in position 3 (*add r2, r3, 1*). The reason is that 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 the instruction in position 4. Such unconditional branch does not allow the instruction in position 5 to be executed after instruction 3, but only after a branch instruction with destination address pointing to its actual position.

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

Figure 3 illustrates a program code divided in two BBs. The first BB contains the instruction located in position 3 of the hardened code. The second BB contains the instruction located in positions 6 and 7 of the same code. In order to apply rule #5 and #6, four instructions are added (positions 2, 4, 5 and 8). The first two protect the first BB by assigning the unique identifier *signature1* to the global variable *rX* (*mv rX, signature1*) and by later verifying its value when exiting the BB (*bne rX, signature1, error*).

Note that the original branch instruction located in position 1 (*beq r1, r2, 6*) had its destination address modified from 6 to 5. This change was necessary in order to jump to the new beginning of the basic block. Instructions 5 and 6 have the same role as instructions 2 and 4, respectively, but in order to protect the second basic block.

3.3 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 address correction. Even the BB construction (required for some transformation rules) can be an exhaustive task when performed by hand.

In order to automate the code transformation, we have built a tool called *Hardening Post Compiling Translator* (HPC-Translator). Implemented in Java, the HPC-Translator tool is able to automatically transform an unprotected code into a hardened one, by inserting additional instructions and error subroutines to the software. The HPC-Translator receives as input the program's machine code and therefore it is compiler and language independent, but not microprocessor independent.

The tool is then capable of implementing the presented rules, divided into groups. The first group, called variables, or VAR, implements rules #1, #2 and #3; the second group, called inverted branches, or BRA, implements rule #4. Finally, the third group, referred as signatures, or SIG, implements rules #5 and #6.

A Graphical User Interface (GUI) allows the user to combine these techniques. The implemented tool outputs a machine code, microprocessor dependent, which can be directly interpreted by the target microprocessor.

Figure 4 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.

4 Fault Injection Experimental 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 [12]. In order to evaluate both the effectiveness and the feasibility of the presented approaches, two applications were chosen: a 6×6 matrix multiplication and a bubble sort. The matrix multiplication requires a large data processing combined with only a few loops. Therefore it uses mostly the datapath of the

Original Code	Hardened Code
1: beq r1, r2, 6	1: beq r1, r2, 5
3: add r2, r3, 1	**2: mv rX, signature 1** 3: add r2, r3, 1 **4: bne rX, signature 1, error**
6: add r2, r3, 9 7: st [r1], r2	**5: mv rX, signature 2** 6: add r2, r3, 9 7: st [r1], r2 **8: bne rX, signature 2, error**
9: jmp end	9: jmp end

Fig. 3 Controlpath rules #5 and #6

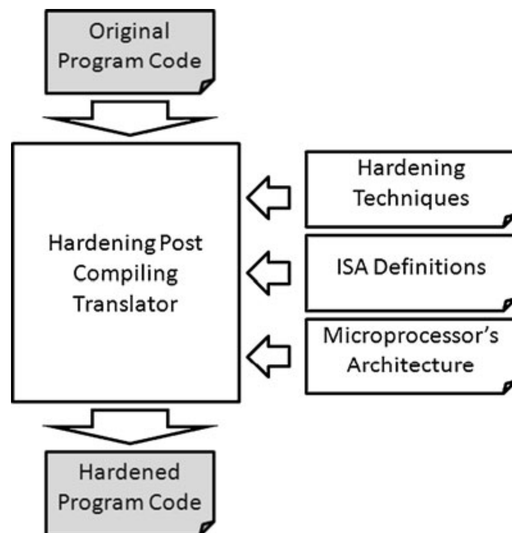

Fig. 4 HPC-Translator

microprocessor. On the other hand, the bubble sort algorithm uses a large number of loops, control registers and branch instructions. Therefore it uses mostly the controlpath, since all the data processing is related to the control registers.

Four hardened programs were generated using the *Hardening Post Compiling Translator*, each one implementing the following software-based techniques: (I) SIG, (II) VAR, (III) BRA and (IV) SIG-VAR-BRA (SIG, VAR and BRA combined). Tables 1 and 2 show the original and modified program's execution time, plus the code and data sizes from the matrix multiplication and bubble sort algorithms, respectively.

In order to start the fault injection campaign, 50 thousand faults were injected, in each application, in all signals of the non-protected microprocessor (including registered signals), one per program execution run. The SEU and SET faults were injected directly in the microprocessor VHDL code by using ModelSim XE/III 6.3c [16]. 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 was 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. So, in the end, only faults not masked by the application were considered in the analysis. When 100% signal coverage was achieved and at least four faults per signal were detected we normalized the faults, varying from four to five faults per signal. Those faults were used to build the test case list. Using one Intel Q8400 processor with 8 Gb of memory, the total simulation time was 246 h for the matrix multiplication and 56 h for the bubble sort.

The faults were also classified by their source and effect on the system. We defined four groups of fault sources to inject faults of both SEU and SET types of faults: datapath, controlpath, register bank and ALU. Program and data memories were 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 into these groups, we continuously compared the Program Counter (PC) of two microprocessors executing: the golden and the faulty microprocessor. In case of a mismatch, the injected fault was classified as having a control flow effect. If the faulty's PC matched the golden's, the fault was classified as having a data effect.

Note that the used miniMIPS has a fault detection mechanism that resets itself, when a wrong instruction is fetched. So, faults that may change the instruction opcode have a control flow 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

Table 1 Original and hardened program's properties

Source	Original	Hardened program versions			
		I	II	III	IV
Matrix multiplication					
Exec. time (ms)	1.24	1.40	2.59	1.30	2.73
Code size (byte)	1,548	3,500	3,704	2,580	5,460
Data size (byte)	524	532	1,048	524	1,056
Bubble sort					
Exec. time (ms)	0.23	0.28	0.47	0.24	0.53
Code size (byte)	1,212	2,404	2,664	1,580	3,924
Data Size (byte)	120	128	240	120	248

Table 2 Results for SET and SEU fault injection in the matrix multiplication and bubble sort algorithms: percentage of detected faults of techniques (I) signatures, (II) variables, (III) inverted branches and (IV) signatures, variables and inverted branches combined

	Source classification		#Data effect faults	Fault coverage (%)				#Control flow effect faults	Fault coverage (%)			
				Hardened program versions					Hardened program version			
				I	II	III	IV		I	II	III	IV
Matrix multiplication												
SET	Controlpath		83	0	100	0	100	33	6.3	43.8	6.5	45.5
	Datapath	ALU	8	0	100	0	100	10	0	100	0	100
		Reg. Bank	2	0	100	0	100	1	0	100	0	100
		Others	9	0	100	0	100	2	0	100	0	100
	Total		102	0	100	0	100	46	4.4	60	4.5	60.9
SEU	Controlpath		22	0	100	0	100	36	20	34.3	0	42.4
	Datapath	ALU	1	0	100	0	100	0	–	–	–	–
		Reg. Bank	8	0	100	0	100	13	0	100	0	100
		Others	5	0	100	0	100	7	16.7	100	0	100
	Total		36	0	100	0	100	56	14.8	59.3	0	63.5
Bubble sort												
SET	Controlpath		24	4.5	100	0	100	89	8	68.9	2.3	80.8
	Datapath	ALU	5	0	100	0	100	14	0	100	0	100
		Reg. Bank	2	0	100	0	100	4	0	100	0	100
		Others	9	0	100	0	100	28	0	100	0	100
	Total		40	2.2	100	0	100	135	5.3	82.5	1.5	87.9
SEU	Controlpath		22	0	100	0	100	81	1.3	68.1	0	71.4
	Datapath	ALU	0	–	–	–	–	0	–	–	–	–
		Reg. Bank	8	0	100	0	100	33	12.1	100	0	100
		Others	5	0	100	0	100	19	0	100	0	100
	Total		35	0	100	0	100	133	3.8	82.5	0	83.5

locating its new PC) and the pipeline stage where the fault was manifested. Around 1% of the total number of faults could not be mapped and were replaced by new faults.

Table 2 contains the results of the fault injection campaign. The number of injected faults is shown, divided among the source (controlpath and datapath) and effect (control flow or data) of each fault. The detection rates of each technique implemented are also shown in percentage values.

Results presented in Table 2 show that the VAR technique (II) presented the highest detection rate among the three. It was capable of detecting all faults that caused errors with data effects and, in addition, some faults with control flow effects. More specifically, the faults injected in the register bank that affected the program flow could be detected by the VAR technique because those data were protected. For the matrix multiplication algorithm, this technique resulted in 2.26 times larger code size and 2.39 times larger execution time. The detection rate was around 77% of the faults (SEU and SET). When applied to the bubble sort algorithm, a 2.19 times larger code and a 2.04 times larger execution time were observed. The detection rate was around 84%. On the other hand, techniques (I) and (III) provided low detection rates and were not capable of detecting most of the faults causing errors with control flow effects. While the first (I) achieved a detection rate of 9.1% and 3.7% for the matrix and bubble sort algorithms, respectively, the later (III) resulted in a detection rate of 0.8% and 0.5% for the same algorithms, respectively.

When techniques I, II and III were combined into technique IV, they complemented themselves. The highest detection rates were achieved, up to 79% with a code size increase of 3.52 times and execution time increase of 2.1 times for the matrix multiplication algorithm. For the bubble sort algorithm a detection rate of 88% was achieved, with an overhead of 3.23 times in program code and 2.12 times in execution time. However, 21% of faults injected in the matrix multiplication algorithm remained undetected, while 12% of the injected faults in the bubble sort algorithm were not detected by any technique. Analyzing Basic Block Sizing to Improve Fault Coverage

Although the combined rules of technique IV are able to detect most errors, they could not detect up to 21% of the injected faults. The reason for that are mainly the following:

- Every incorrect jump from a given basic block to the same basic block (also known as intra-block jump) will not be detected. The unique identifier is an invariant and therefore does not depend on the actual instructions. The matrix multiplication algorithm stays 83% of its time in the same BB, which occupies 20% of the program data. Therefore, causes an increase in the occurrence of this drawback. Such situation can be seen on Fig. 5 (1);
- Incorrect jumps to the beginning of a different BB will not be detected, because the global variable containing the unique identifier is updated exactly in the beginning of each basic block. The occurrence of such error is proportional to the number of basic blocks per instructions, which is higher in control-flow applications. Figure 5 (2 and 3) shows this drawback;
- The used microprocessor has a mechanism that performs a system restart (jumps to address 0) when an inexistent instruction is fetched. This can be seen on Fig. 5 (3);
- Incorrect jumps to unused memory positions, which are filled with NOP instructions, result in time out. This drawback can be seen on Fig. 5 (4).
- Incorrect jumps to branch instructions inside the code will also not be detected, since such instructions are not inside a basic block and therefore not protected by the technique. This drawback can be seen on Fig. 5 (5).

Table 3 shows the amount of undetected faults, which were organized by effect following the classification: (1), (2) and (3) and (4) represented in Fig. 5. On data flow applications, such as the matrix multiplications used as first case-study in this paper, there are only few large basic blocks, which are executed during most of the execution time. In such cases, the effect (1) is more common. On control flow applications, such as a bubble sort algorithm used as the second case-study in this paper, the effect (2) are expected to happen more often, since there are more and smaller basic blocks. The use of watch-dog can solve the effect (4), while more complex signatures based on the program's execution flow may cope with the effect (2).

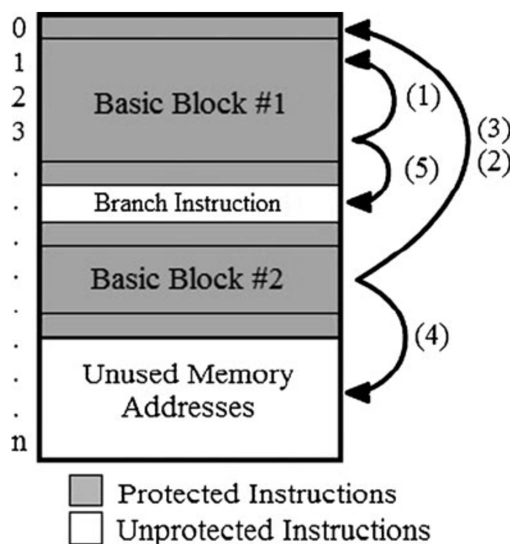

Fig. 5 Control flow effects of undetected faults

However, the effect (1) is still a big issue. Note that more than 30% of the undetected faults are classified as intra-block jumps and jumps to the beginning of basic blocks. Efficient solutions in terms of execution time to those discussed effects (1), (2) and (3) must be investigated.

As shown in Table 3, up to 7.8% of the undetected faults caused a system restart (3) due to an exception and up to 36.4% caused jumps to an unused part of the program memory (4). These faults are not an issue because the exception handling circuit is already able to detect them. From the remaining undetected faults, up to 30.7% were an incorrect jump to the same basic block (1), while up to 71.6% were an incorrect jump to the beginning of a basic block (2). From the total faults injected, only 1% caused an incorrect jump to an unprotected branch instruction (5).

According to these results, solutions must focus on two effects: Same basic block (1) and Beginning of basic block (2).

4.1 Undetected Faults: Jumps to the Same Basic Block

In order to protect the system against this type of faults, one can think of different basic block sizes to reduce the number of possible addresses inside the same BB. The BB maximum size is defined by the branch instructions placed by the software compiler. Each BB must end before a branch instruction and start in both possible destination branch addresses.

An analysis on the original matrix multiplication code shows a total of 79 basic blocks and 454 instructions, where the largest basic block has 65 instructions, followed by 36 and 12. Note that, from the 26% of the faults that are caused by jumps to the same BB, 89.3% were faults in those large BBs.

Considering the average basic block size of 5.74, the maximum number of instructions per basic block was set to 4. This number is increased to 7 when the SIG technique is applied (the chosen microprocessor requires 2 assembly instructions to compare a register with a constant). The program characteristics of BB sizing can be seen on Table 4.

The execution time of the matrix multiplication algorithm protected by SIG using BBs sized by 4 instructions is 2.94 times the one protected using SIG with BB sized by the maximum allowed. It is also 3.32 times higher than the

Table 3 Effects of the undetected faults responsible to cause control flow errors that were not detected by the SIG technique for the matrix multiplication and bubble sort algorithms

	Type of effects: jumps to	SIG technique BB—max. size allowed	
		Total	Undetected faults (%)
Matrix multiplication	Same basic block (1)	27	30.7
	Beginning of basic block (2)	25	28.4
	System restart (3)	3	3.4
	Unused memory (4)	33	36.4
	Unprotected instructions (5)	1	1.1
	Total faults injected	89	100
Bubble sort	Same basic block (1)	31	12.8
	Beginning of basic block (2)	174	71.6
	System restart (3)	19	7.8
	Unused memory (4)	18	7.4
	Unprotected instructions (5)	1	0.4
	Total faults injected	243	100

original code. When reducing the BB size to four instructions, one can see that the code size has also increased 86%, when compared to the SIG technique without minimal BB sizing. Similar results were found for the bubble sort algorithm: increases of 2.67 and 3.26 times for the execution time and 2.07 times for the code size.

In order to inject faults following the same strategy presented in Section 3, a new analysis on the effect of each fault was done and a new set of faults was built. The faults were randomly injected on the microprocessor without protection and results were gathered. Faults causing a wrong system result were selected and normalized to each signal (around five faults per signal). The faults affecting the data were then excluded, remaining only the faults which effect caused an error on the control flow.

Faults causing a system to restart or to jump to unused memory addresses are easily detected by exception handling circuits. These faults vary from 15% to 40% of the total faults that manifested a control flow effect. Such faults should be considered if the system's detection hardware could not detect them. Since these faults were detected by the chosen microprocessor, they will not be considered in the results.

Table 4 Original and SIG program's overhead characteristics for the matrix multiplication and bubble sort algorithms

	Original	SIG BB—max. size allowed	SIG BB—4 instructions
Matrix multiplication			
Exec. time (ms)	1.24	1.40	4.12
Code size (byte)	1,548	3,500	6,536
Data size (byte)	524	532	532
Bubble sort			
Exec. time (ms)	0.23	0.28	0.75
Code size (byte)	1,212	2,404	4,984
Data size (byte)	120	128	128

Table 5 shows that when decreasing the maximum size of the BBs to four instructions, the number of faults causing an erroneous jump to the beginning of BBs increased in the same proportion that the faults causing an erroneous jump to the same basic block decreased for the matrix multiplication algorithm. For the bubble sort these values remained unchanged. This means that a tradeoff between the two types of effects can be achieved, but the detection rate has not been improved enough (around 5% to 10%) to justify the overhead penalties in program code and execution time.

4.2 Undetected Faults: Jumps to the Beginning of Basic Block

Faults causing an incorrect jump to the beginning of a BB cannot be detected by the SIG technique, since the extra instructions check only if the last basic block to start is the first to finish. That means that the SIG do not check the

Table 5 Effects of the undetected faults responsible to cause control flow errors that were not detected by the SIG technique when BB is sized applied to the matrix multiplication and bubble sort algorithms

	Type of effects: jumps to	SIG BB—max. size allowed		SIG BB—4-instruction size	
		Total	%	Total	%
Matrix multiplication	Same basic block (1)	27	50.9	20	40
	Beginning of basic block (2)	25	47.2	30	60
	Unprotected instructions (5)	1	1.9	0	0
	Total faults injected	53	100	50	100
Bubble sort	Same basic block (1)	31	15	31	16.9
	Beginning of basic block (2)	174	84.5	152	82.6
	Unprotected instructions (5)	1	0.5	1	0.5
	Total faults injected	206	100	184	100

control flow itself, but the basic block's consistency. Therefore, a control flow error that maintains the basic block consistency, such as an incorrect jump to the beginning of a basic block or an incorrect path taken by a branch instruction, cannot be detected by the SIG technique.

This issue is partially solved by [15] using the ECCA technique. The ECCA technique introduces a new invariant to each basic block and a two-element queue that keeps track of the current executing basic block and the possible next basic blocks (which have the same identifier in order to fit the two element queue). A few instructions are added to each basic block in order to manage and check the queue's consistency.

Even increasing the detection rate, ECCA cannot achieve 100% fault coverage in control flow errors, since it cannot guarantee that a possible path was not incorrectly taken, such as an incorrect path taken by a branch instruction. On the other hand, such errors could be detected by the combination of techniques VAR and BRA.

In order to detect all incorrect jumps to the beginning of a basic block, a combination of techniques should be implemented, resulting in a higher overhead in both program code size and execution time.

5 Conclusion

In this paper, we presented a set of software-based techniques that rely on groups of transformation rules to detect soft errors in microprocessors. Then, a tool was implemented to automatically harden two applications described in machine code according to the presented techniques. In order to evaluate both their effectiveness and feasibility, a set of faults was then built and a fault injection campaign was performed on the hardened programs. Results showed that the VAR technique is capable of achieving a high detection rate, up to 88%, while the SIG and BRA showed results below expected, with detection rates up to 9.1%.

The SIG technique was then analyzed and some drawbacks were found to explain the undetected faults. In order to further analyze the signature's undetected faults, a new software implementation was built. The BBs with more than four instructions were divided and a new set of faults was injected. The results showed that the execution time varies from 2.67 to 2.94 times higher than the original, while the detection rate increased slightly, varying from 5% to 10%.

We are currently working on improving the detection rates of the SIG technique to decrease the overhead impact in memory and execution time. We are also expanding the set of applications to a wider and more complex group of algorithms, such as the LZW, SPEC or VITERBI benchmarks.

As future work, we intend to verify the feasibility and efficiency of the studied techniques when applied to microprocessors with different architectures, such as VLIW and superscalar. We also aim at physical tests, such as electromagnetic interference (EMI) and radiation campaigns, to confirm the obtained simulation results.

References

1. (2005) International Technology Roadmap for Semiconductors: 2005 Edition, Chapter Design, pp 6–7
2. Alkhalifa Z, Nair VSS, Krishnamurthy N, Abraham JA (1999) Design and evaluation of system-level checks for on-line control flow error detection. In IEEE Trans. on Parallel and Distributed Systems 10(6):627–641. June
3. Azambuja JR, Sousa F, Rosa L, Kastensmidt FL (2010) The limitations of software signature and basic block sizing in soft error fault coverage. In Proc. of IEEE Latin–American Test Workshop
4. Azambuja JR, Sousa F, Rosa L, Kastensmidt FL (2010) Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors. In Proc. of IEEE Latin American Symposium on Circuits and Systems
5. Baumann RC (2001) Soft errors in advanced semiconductor devices-part I: the three radiation sources. In IEEE Transactions on Device and Materials Reliability. March
6. Bolchini C, Miele A, Salice F, Sciuto D (2005) A model of soft error effects in generic ip processors. In Proc. of the 20th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, pp 334–342
7. Bolchini C, Pomante L, Salice F, Sciuto D (2005) Reliable system specification for self-checking datapaths. In Proc. of the Conf. on Design, Automation and Test in Europe, pages 1278–1283, Washington, DC, USA. IEEE Computer Society
8. Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Sonza Reorda M, Violante M (2000) Experimentally evaluating na automatic approach for generating safety-critical software with respect to transient errors. In IEEE Trans Nucl Sci 47(6 part 3):2231–2236. Dec
9. Dodd PE, Massengill LW (2003) Basic mechanism and modeling of single-event upset in digital microelectronics. IEEE Trans Nucl Sci 50:583–602
10. Goloubeva O, Rebaudengo M, Sonza Reorda M, Violante M (2003) Soft-error detection using control flow assertions. In: Proceedings of the 18th IEEE international symposium on defect and fault tolerance in VLSI systems—DFT 2003, November, pp 581–588
11. Gorman TJO, Ross JM, Taber AH, Ziegler JF, Muhlfeld HP, Montrose ICJ, Curtis HW, Walsh JL (1996) Field testing for cosmic ray soft errors in semiconductor memories. In IBM Journal of Research and Development, pp 41–49, January
12. Hangout LMOSS, Jan S (2011) The minimips project. Available online at http://www.opencores.org/projects.cgi/web/minimips/overview
13. Huang KH, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. In IEEE Trans Comput 33:518–528. Dec
14. Lu DJ (1982) Watchdog processors and structural integrity checking. In IEEE Trans. on Computers C-31(7):681–685
15. Mcfearin LD, Nair VSS (1995) Control-flow checking using assertions. In Proc. of IFIP International Working Conference Dependable Computing for Critical Applications (DCCA-05), Urbana-Champaign, IL, USA, September
16. Mentor Graphics (2009) http://www.model.com/content/modelsim-support

17. Nicolescu B, Velazco R (2003) Detecting soft errors by a purely software approach: method, tools and experimental results. In Proc of the Design, Automation and Test in Europe Conference and Exhibition
18. Oh N, Shirvani PP, McCluskey EJ (2002) Control flow checking by software signatures. In IEEE Trans Reliab 51(2):111–112. (Mar)
19. Oh N, Shirvani PP, McCluskey EJ (2002) Control-flow checking by software signatures. In IEEE Trans. on Reliability, 51(1):111–122
20. Oh N, Shirvani E, McCluskey E (2002) Error detection by duplicated instructions in Super-scalar Processors. In IEEE Trans. On Reliability 51–1:63–75
21. Rebaudengo M, Sonza Reorda M, Torchiano M, Violante M (1999) Soft-error detection through software fault-tolerance techniques. In Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, pp 210–218
22. Reis GA, Chang J, Vachharajani N, Rangan R, August DI (2005) SWIFT: software implemented fault tolerance. In Proc. of the Inter. Symp. on Code Generation and Optimization

José Rodrigo Azambuja is a PhD candidate at the Federal University of Rio Grande do Sul (UFRGS) located in Porto Alegre, Brazil. He received the Computer Engineering and MSc in Computer Science in 2008 and 2010, respectively, from the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil. His main research interests are computer architectures and fault tolerant systems.

Samuel Nascimento Pagliarini is a MSc candidate in Microelectronics at the Federal University of Rio Grande do Sul (UFRGS) located in Porto Alegre, Brazil. He received his degree in Computer Engineering in 2008 from the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil. His professional experiences include digital ASIC design and verification plus digital ASIC training at the Training Center of the Brazilian IC Program. His research interests include VLSI design, test and verification, HDLs, HVLs and EDA for functional verification.

Lucas Rosa is finishing his degree in Computer Engineering at the Federal University of Rio Grande do Sul (UFRGS) located in Porto Alegre, Brazil. His main research interests are computer architectures and fault tolerant systems.

Fernanda Lima Kastensmidt is a professor of Computer Science at the Federal University of Rio Grande do Sul (UFRGS) located in Porto Alegre, Brazil. She received her BS in Electrical Engineering in 1997 and MS and PhD degrees in Computer Science and Microelectronics in 1999 and 2003, respectively, from the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil. Her professional research experiences include internships in the Grenoble National Polytechnic Institute (INPG), France in 1999, in Xilinx Corporation, San Jose, USA in 2001, and in the Laboratory of Materials and Systems Integration (IMS) in Bordeaux University, France in 2008. Her research interests include VLSI testing and design, fault effects, fault tolerant techniques and programmable architectures. She is an IEEE member, and author of the book “Fault-Tolerance Techniques for SRAM-based FPGAs,” published in 2006 by Springer.