Evaluating the Efficiency of Data-flow Software-based Techniques to Detect SEEs in Microprocessors

José Rodrigo Azambuja, Ângelo Lapolli, Maurício Altieri, Fernanda Lima Kastensmidt, *Member, IEEE*
Instituto de Informática – PPGC – PGMICRO
Universidade Federal do Rio Grande do Sul (UFRGS)
Av. Bento Gonçalves 9500, Porto Alegre - RS - Brazil
{jrfazambuja, aclapolli, mascarpato, fglima} @ inf.ufrgs.br

***Abstract*— There is a large set of software-based techniques that can be used to detect transient faults. This paper presents a detailed analysis of the efficiency of dataflow software-based techniques to detect 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 MIPS-based microprocessor architecture. The efficiency of each rule and a combination of them are tested. Experimental results are used to analyze the overhead of data-flow techniques allowing us to compare these techniques in the respect of time, resources and efficiency in detecting this type of faults. This analysis allows us to implement an efficient fault tolerance method that combines the presented techniques in such way to minimize memory area and performance overhead. The conclusions can lead designers in developing more efficient techniques to detect these types of faults.**

***Index Terms* — Fault tolerance, SEU, SET, Soft errors, Software techniques.**

I. Introduction

THE last-decade advances in the semiconductor industry have increased microprocessor performance exponentially. Most of this performance gain has been due to small dimensions and low voltage transistors, which have leaded in complex architectures with large parallelism combined with high frequency. However, the same technology that made possible all this progress also has reduced the transistor reliability by reducing threshold voltage and tightening the noise margins [1, 2] and thus making them more susceptible to faults caused by energized particles [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 concerns is known as soft error, which is defined as a transient effect fault provoked by the interaction of energized particles with the PN junction in the silicon. This upset temporally charges or discharges nodes of the circuit, generating transient voltage pulses that can be interpreted as internal signals, thus provoking an erroneous result [4]. The most typical errors concerning soft errors are *single event upsets* (SEU), which are bit-flips in the sequential logic and *single event transients* (SET), which are transient voltage pulses in the combinatorial logic that can be registered by the sequential logic.

Fault tolerance techniques based on software can provide a high flexibility, low development time and cost for computer-based dependable systems. High efficient systems called systems-on-chip (SoC) composed of a large number of microprocessors and others cores connected through a network on chip (NoC) are getting more popular in many applications that require high reliability, such as data servers, transportation vehicles, satellites and others. When using those systems, it is up to designers to harden their applications against soft errors. Fault tolerance by means of software techniques has been receiving a lot of attention on those systems, because they do not need any customization of the hardware.

Software-only fault tolerance techniques exploit information, instruction and time redundancy to detect and even correct errors during the program flow. All these techniques use additional instructions in the code area to either recompute instructions, or to store and to check suitable information in hardware structures. In the past years, tools have been implemented to automatically inject such instructions into C or assembly code, reducing significantly the costs.

Related works have pointed out the drawbacks of software-only techniques, such as the impossibility of achieving complete fault coverage of SEU [5], usual high overhead in memory and degradation in performance. Memory increases due to the additional instructions and often memory duplication. Performance degradation comes from the execution of redundant instructions [6, 7, 8]. However, there is no study in the literature that analyzes both SEU and SET faults and correlates the fault detection rate with the caused overhead in memory and performance. This information is important to guide designers to improve efficiency and detection rates of software-based fault tolerance techniques.

In this paper, the authors implement a set of software-only techniques to harden a matrix multiplication and a bubble sort algorithm to SEU and SET faults that can occur in a miniMIPS microprocessor. The implemented techniques target data and control-flow fault tolerance. Considering that previously proposed techniques have shown excellent fault detection rates but also caused a considerable time and memory overhead, this paper presents alternative procedures that explore the same principles in an attempt to reduce the overhead. Results indicate different implementations to keep high detection rates while decreasing the operational cost.

The paper is organized as follows. Section 2 presents the software-based techniques and the proposed methodology. Section 3 presents the fault injection campaign and results. Section 4 shows the evaluation of the proposed methods. Section 5 presents main conclusions and future works.

II. The proposed case-study Hardened Program Methodology

A set of transformation rules has been proposed in the literature. In [9], eight rules are proposed, divided in two groups: (1) aiming data-flow errors, such as data instruction replication [9, 10] and (2) aiming control-flow errors, such as Structural Integrity Checking [11], Control-Flow Checking by Software Signatures (CFCSS) [12], Control Flow Checking using Assertions (CCA) [13] and Enhanced Control Flow Checking using Assertions (ECCA) [14]. The proposed techniques can achieve a full data-flow fault tolerance, concerning SEU's, being able to detect every fault affecting the data memory, which lead the system to a wrong result. Nevertheless, they still increase substantially the application's execution time.

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 [15], 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 [9], eight rules are proposed, while [16] used thirteen rules to harden a program. In this paper, we address six rules, divided into faults affecting the datapath and the controlpath.

A. *Data-flow Fault Tolerance Techniques*

A technique described in [17], called Variables proposes three rules aiming at detecting 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. These rules are:

- 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 operating with registers and memory elements. Instruction 1 checks the base address register of the load operation with its replica, instruction 3 duplicates the operation. Instructions 4 and 5 verify the consistency of the add operands, while instruction 7 performs the operation on the replica. Instructions 8 and 9 check the base and the destination address of the store instruction and instruction 11 replicates the operation using the replicated memory position. Instructions 1, 4, 5, 8 and 9 are inserted due to rule #3 and instructions 3, 7 and 11 are inserted due to rules #1 and #2.

From now on this technique will be referred as VAR1 considering it will be introduced two alternative methods called VAR2 and VAR3 trying to reduce the overheads. Besides that, the control-flow fault detection technique which was combined with these methods for experimentation will be explained.

The following methods take advantage of the rules #1 and #2 changing the form the consistency is checked in order to decrease the amount of extra instructions. By reducing the code size it is expected that time overhead reduces too.

Instead of checking the operands before the instruction on load and arithmetic operations, VAR2 consists in comparing the results. In other words, the variables and its replicas are compared every time some operation changes them. In the case of a store instruction, the same approach of VAR1 is implemented.

Figure 2 illustrates VAR2. Instruction 2 replicates the load instruction and instruction 3 compares the destination address to its replica. Instruction 5 is inserted to duplicate the add instruction and instruction 6 verifies the replica's consistency. Three instructions are added to protect the store operation (instructions 7, 8 and 10), the same procedure as VAR1 is implemented.

The number of extra instructions for the load and the add

`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'`

Figure 1: VAR1

`ld r1, [r4]`	`1: ld r1, [r4]`
	`2: ld r1', [r4' + offset]`
	`3: bne r1, r1', error`

`add r1, r2, 1`	`4: add r1, r3, r4`
	`5: add r1', r3', r4'`
	`6: bne r1, r1', error`

`st [r1], r2`	**`7: bne r1, r1', error`**
	`8: bne r2, r2', error`
	`9: st [r1], r2`
	`10: st [r1' + offset], r2'`

Figure 2: VAR2

`ld r1, [r4]`	**`1: bne r4, r4', error`**
	`2: ld r1, [r4]`
	`3: ld r1', [r4' + offset]`

`add r1, r2, r4`	`4: add r1, r3, r4`

`st [r1], r2`	**`5: bne r1, r1', error`**
	`6: bne r2, r2', error`
	`7: st [r1], r2`
	`8: st [r1' + offset], r2'`

Figure 3: VAR3

operations is reduced in one. The base address of the load instruction is not verified considering that any error in its value will reflect on the destination registers.

Considering that an inconsistency between duplicated variables only needs to be checked after loading and before writing it to memory, another method, called as VAR3, reduces even more the amount of extra instructions.

Figure 3 shows that, in this technique, additional instructions are only required near load and store operations. Instruction 1 is added to verify the register containing the base address of the load instruction, instruction 3 replicates the operation. Instructions 5 and 6 are inserted to check the destination address and the value to be stored n 7 and 8. No extra instructions are required for the add operation as well any other operation that does not access memory given that any corrupt data will be detected later.

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.

B. *Control-flow Fault Tolerance Technique*

Along with the presented methods, aiming at protecting the program's flow, the technique described in [18], called FLOW, is combined with the previously introduced methods. This technique as many others divide the program into basic blocks, they are segments of code which are always executed sequentially in a not faulty program flow. They start in jump destination addresses and memory positions after branch instructions; their end is on every jump instruction address and on the last instruction of the code. The proposed approach involves two methods to protect the system against errors causing incorrect jumps between different basic blocks and inside basic blocks, both oblige to use a watchdog processor.

In order to protect the system against jumps to the beginning of basic blocks (when the basic block's 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 the basic blocks.

This technique assigns two identifiers to each basic block: the Block IDentifier (BID) represents each basic block with a unique prime number, while the Control Flow IDentifier (CFID) represents the control flow, by storing the multiplication product of the next basic blocks' BIDs. Since the BIDs are composed by prime numbers, the operation rest 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 basic block's CFID. Upon entry to a basic block, its CFID is enqueued. Upon exit of a basic block, the first CFID is dequeued and divided by the BID. Errors are detected when one of these situations occur: (1) the rest 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 watchdog should be modified to perform the queue management and the operation rest of division, while the software informs the watchdog of the CFIDs to enqueue the basic blocks' BID.

Figure 4 shows the code transformation required to send the necessary values to the watchdog processor represented by instructions 2 and 7 (send BID), 3 and 8 (enqueue CFID) and 6 and 12 (dequeue CFID). These instructions are implemented in assembly through store instructions with special addresses.

To detect incorrect jumps to the same basic block it is proposed a hybrid technique that computes each basic block signature by XOR'ing all instructions inside a basic block. Each basic block signature is pre-computed by the compiler, which also sends this value to the watchdog. The watchdog is also modified to calculate the basic block signature by performing XOR operations in real time. When the software sends an instruction with a signature value, the watchdog verifies it with its calculated value. When a mismatch is found, an error is notified. In order to inform the watchdog of an entry in a basic block, a reset instruction should be used.

Figure 4 shows the transformation required on an assembly code to adapt the software to the watchdog with instructions 2 and 7 (reset XOR) and 5 and 11 (check XOR). These instructions are implemented in assembly through store instructions with special addresses.

C. *Watchdog Hardware*

The light watchdog 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

`beq r1, r2, 9`	`1: beq r1, r2, 7`

`add r2, r3, 1`	**`2: reset XOR/send BID`**
	`3: enqueue CFID`
	`4: add r2, r3, 1`
	`5: check XOR`
	`6: dequeue CFID`

`add r2, r3, 4`
`st [r1], r2`
	`7: reset XOR/send BID`
	`8: enqueue CFID`
	`9: add r2, r3, 4`
	`10: st [r1], r2`
	`11: check XOR`
	`12: dequeue CFID`

`jmp end`	`13: jmp end`

Figure 4: Transformed software

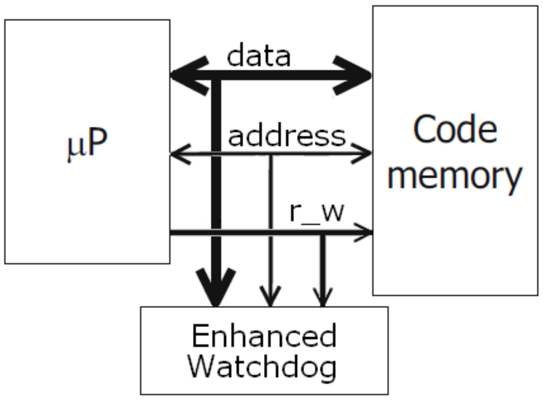

Figure 5: Watchdog connected to microprocessor

16-bit register to store the real-time calculated XOR value, a 64-bit register to store the 2-element queue, a rest of division module and a control module to spoof the instructions recently described, such as check CFID and reset XOR value. Table 1 shows the size and performance of the implemented microprocessor and the watchdog. The watchdog implementation has a total of 128 registers and is not protected against faults, but could be protected with a DMR approach.

In order to find instructions and calculate the XOR value, the watchdog spoofs the address and data bus and the r_w signal. Figure 5 shows the implemented watchdog connected a microprocessor.

Table 1: Original and modified architecture characteristics

Source	miniMIPS	Watchdog
Area (μm)	24261.32	3640.21
Frequency (MHz)	66.7	66.7

D. *Hardening Post Compiling Translator Tool*

In order to automate the software transformation, we built a tool called *Hardening Post Compiling 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. The tool is than capable of implementing the presented rules, divided into groups. The first group, called variables, is divided in VAR1, VAR2 and VAR3 and implements the recently explained methods; the second and the third group, called inverted branches and signatures, implement other techniques proposed in [17]; the last group implements the control-flow protection described above. The user is allowed to combine the techniques in a graphical interface. The implemented tool outputs a binary code, microprocessor dependent, which can be directly interpreted by the target microprocessor.

Figure 6 shows the HPC-Translator's work flow. The tool receives three distinct inputs: the original program code, the user choices of protection techniques and the instruction set architecture (ISA) definition. Using these inputs, the HPC-Translator is able to generate a hardened program code.

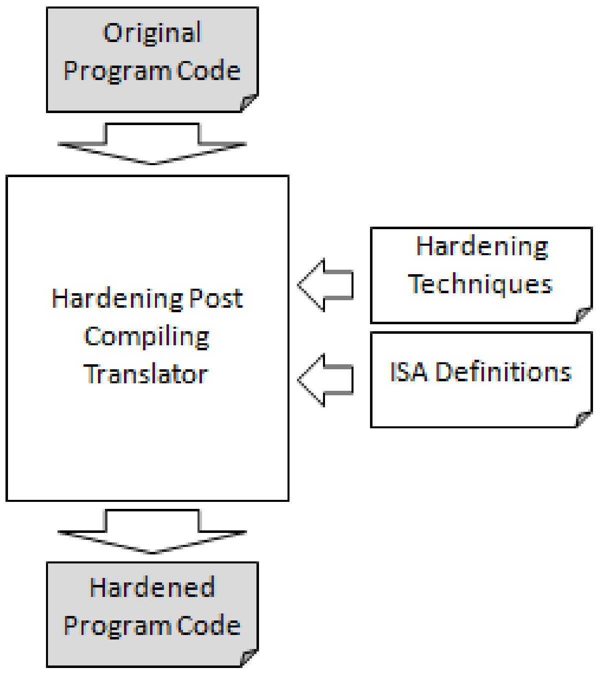

Figure 6: HPC-Translator work-flow

III. 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 [19]. In order to evaluate both the effectiveness and the feasibility of the presented approaches, 6x6 matrix multiplication and bubble sort algorithms were used.

Six hardened programs were generated using the *Hardening Post Compiling Translator*, each one implementing the following software-only techniques: (I) VAR1, (II) VAR1 and FLOW, (III) VAR2, (IV) VAR2 and FLOW, (V) VAR3, (VI) VAR3 and FLOW. Table 2 and Table 3 show the original and modified program's execution time, code size and data size for the bubble sort and the matrix multiplication algorithm, respectively.

Table 2: Bubble sort transformation characteristics for (I) VAR1, (II) VAR and FLOW, (III) VAR2, (IV) VAR2 and FLOW, (V) VAR3 and (VI) VAR3 and FLOW.

Source	Original	(I)	(II)	(III)	(IV)	(V)	(VI)
Exec. Time (us)	231	462	601	522	666	375	515
Code Size (byte)	1212	2328	3260	2252	3184	1916	2848
Data Size (byte)	120	240	244	240	244	240	244

Table 3: Matrix multiplication transformation characteristics for (I) VAR1, (II) VAR and FLOW, (III) VAR2, (IV) VAR2 and FLOW, (V) VAR3 and (VI) VAR3 and FLOW.

Source	Original	(I)	(II)	(III)	(IV)	(V)	(VI)
Exec. Time (ms)	1.26	2.59	2.99	2.79	3.21	1.82	2.22
Code S (byte)	1548	3368	4876	3200	4716	2644	4152
Data (byte)	524	1048	1052	1048	1052	1048	1052

A description of the microprocessor was implemented in VHDL and thousands of randomly generated faults were injected in all signals describing the microprocessor, one by program execution, running each of the six implemented techniques. SEU and SET were injected directly in the microprocessor's VHDL code during simulation, using ModelSim XE/III 6.3c [20] as a simulation platform. Both types of faults were injected by inducing a bit-flip in one randomly chosen bit (sequencial for SEUs and combinational for SETs) with a latency of one and a half clock cycle, in order to remove the latch window masking and improve the fault injection campaign.

The fault injection campaign was performed automatically by a script generated through a fault injector software. At the end of each execution, the results stored in memory were compared with the expected correct values. The error subroutine inserted by the HPCT raised a flag whenever an error was detected by the protection techniques.

Program and data memories are assumed to be protected by Error Detection and Correction (EDAC) and therefore faults in the memories were not injected.

Results are presented in tables 3 and 4. Bubble sort and matrix multiplication results are similar concerning fault detection rates. They show that the technique VAR3 presents an average detection rate of 94.48%, which is lower than the 95.43% achieved by VAR2 and also lower then the 96.47% detection rate achieved by VAR1. However, VAR3, which presented the lowest average detection rate, also presented the least execution time.

Table 3: Bubble sort fault injection results for techniques: (I) VAR1, (II) VAR and FLOW, (III) VAR2, (IV) VAR2 and FLOW, (V) VAR3 and (VI) VAR3 and FLOW.

Program Version	# of Injected Faults	Detection Coverage
(I)	36000	96.46
(II)	21000	100
(III)	36000	96.12
(IV)	20000	100
(V)	40000	95.53
(VI)	20000	100

Table 4: Matrix multiplication fault injection results for techniques: (I) VAR1, (II) VAR and FLOW, (III) VAR2, (IV) VAR2 and FLOW, (V) VAR3 and (VI) VAR3 and FLOW.

Program Version	# of Injected Faults	Detection Coverage
(I)	7000	96.48
(II)		100
(III)	10000	94.74
(IV)		100
(V)	10000	93.43
(VI)		100

Technique VAR2, despite reducing the code size when compared to VAR1, presented the highest execution time. That can be explained by the pipeline architecture of the chosen microprocessor. In VAR2, comparisons between a variable and its replica may introduces a stall in the pipeline, due to data dependencies. As VAR1 and VAR3 comparisons do not have this dependency the program is executed faster and therefore VAR1, with a higher program code overhead, has a smaller execution time than VAR2.

When combining the techniques with the control flow fault protection method, all of them presented full fault coverage. VAR3 presented the smallest runtime (515us and 2.22ms) and VAR2 the highest (666us and 3.21ms). VAR3 combined with the control flow technique has in average a 13.74% code size decrease when compared to VAR1 and 11.26% when compared to VAR2.

Table 5 presents the average latency to detect faults for each data technique, which is defined as the number of clock cycles between the fault injection and its detection. This table shows that the bubble sort algorithm requires less detection latency in comparison to the matrix multiplication. This is due to the more frequent memory access performed by the bubble sort than the matrix multiplication algorithm and hence more frequent checks for consistency between variables and their replicas.

Table 5: Average latency to detect faults

Program Version	Bubble sort Detection Latency (clock cycles)	Matrix Multiplication Detection Latency (clock cycles)
VAR1	11.26	19.49
VAR2	10.98	21.21
VAR3	19.46	69.76

By comparing latencies between techniques, it is clear that VAR3 presents higher values. The lower code size overhead in this technique leads to less verification during the program execution, which reflects directly on its detection latency.

IV. Analyzing The Proposed Methodology

The attempt of decreasing the amount of additional instructions of the variables technique resulted in time overhead decay as it was expected. However, the higher time latency to detect faults on VAR2 and VAR3 facilitates their propagation, decreasing, thus, their detection rate.

VAR2 showed to be the least efficient method since it presented a lower detection rate and a higher execution time than VAR1. This result is important to emphasize that the number of instructions software executes is not always directly proportional to its runtime. Being so, we must take in consideration the target microprocessor's architecture in order to implement and analyze fault tolerance techniques.

The combination of the original variables technique and its variants with a control flow fault tolerance method resulted in full fault detection coverage. It means that the faults that were not detected by VAR techniques propagated and generated control-flow faults that were detected by the FLOW technique. It is interesting to notice that data-flow and control-flow techniques can combined their detection rates and that the minimization of their superposition can lead to an operation costs reduction. This is the reason why VAR3 executes faster and keeps the same detection rates when compared to VAR1 and VAR2. It is also believed that the combination between VAR3 and FLOW can be optimized, resulting in lower penalties to the original program.

Results allow us to take advantage of the time overhead decrease presented by the alternative variables technique approach. As VAR3 showed to be the most efficient method in the respect of time and code size, it can be considered so far the ideal model to provide overhead decrease without affecting the detection rates for the miniMIPS architecture.

V. Conclusions and Future Work

This paper presented a set of rules on software-based techniques to detect soft errors in microprocessors. A set of rules was presented in order to harden usual programs. Then, a tool was implemented to automatically harden binary programs based on presented set of rules. A fault injection campaign was performed on the implemented techniques to evaluate both their effectiveness and feasibility. Results showed that some alterations in the original variables technique can decrease the time overhead and that their combination with a control-flow protection method can be used to achieve full fault tolerance.

The original variables techniques showed to be the most efficient in terms of detection rate in comparison to the proposed alternative methods, but did not present the best execution time. When adding control-flow fault tolerance, all techniques were able to achieve full fault tolerance, showing the possibility of maintaining the same level of fault protection, but also decreasing time overhead around 13%.

We are currently working on techniques applied to microprocessors implementing different architectures.

References

[1] R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. IEEE Transactions on Device and Materials Reliability, 1(1):17–22, March 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. In IBM Journal of Research and Development, pages 41–49, January 1996.

[3] International Technology Roadmap for Semiconductors: 2005 Edition, Chapter Design, 2005, pp. 6-7.

[4] P. E. Dodd, L. W. Massengill, "Basic Mechanism and Modeling of Single-Event Upset in Digital Microelectronics", IEEE Transactions on Nuclear Science, vol. 50, 2003, pp. 583-602.

[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. on Defect and Fault Tolerance in VLSI Systems, pages 334–342, 2005.

[6] 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 2003, pp 581–588.

[7] Huang KH, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. IEEE Trans Comput 33:518–528 (Dec).

[8] Oh N, Shirvani PP, McCluskey EJ (2002) Control flow Checking by Software Signatures. IEEE Trans Reliab 51(2):111–112 (Mar).

[9] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, pages 210–218, 1999.

[10] C. Bolchini, L. Pomante, F. Salice, and D. Sciuto. 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, 2005. IEEE Computer Society.

[11] D.J. Lu, "Watchdog processors and structural integrity checking," IEEE Trans. on Computers, Vol. C-31, Issue 7, July 1982, pp. 681-685.

[12] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Control-flow checking by software signatures," IEEE Trans. on Reliability, Vol. 51, Issue 1, March 2002, pp. 111-122.

[13] L.D. Mcfearin and V.S.S. Nair, "Control-flow checking using assertions," Proc. of IFIP International Working Conference Dependable Computing for Critical Applications (DCCA-05), Urbana-Champaign, IL, USA, September 1995.

[14] 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. on Parallel and Distributed Systems, Vol. 10, Issue 6, June 1999, pp. 627 – 641.

[15] J. R. Azambuja, F. Sousa, L. Rosa, F. L. Kastensmidt. "The limitations of software signature and basic block sizing in soft error fault coverage". In Proc. of IEEE Latin-American Test Workshop, 2010.

[16] J. R. Azambuja, F. Sousa, L. Rosa, F. L. Kastensmidt. Non-Intrusive Hybrid Signature-Based Technique to Detect SEU and SET Faults in Microprocessors. In Proc. of IEEE Radiation Effects on Components and Systems, 2010.

[17] Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Sonza Reorda M, Violante M (2000) Experimentally evaluating aa automatic approach for generating safety-critical software with respect to transient errors. IEEE Trans Nucl Sci 47(6 part 3): 2231–2236 (Dec).

[18] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results", Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2003.

[19] L. M. O. S. S. Hangout, S. Jan. The minimips project. available online at http://www.opencores.org/projects.cgi/web/minimips/overview, 2009.

[20] Mentor Graphics, http://www.model.com/content/modelsim-support, 2009.