Hybrid Soft Error Mitigation Techniques for COTS Processor-based Systems

Eduardo Chielle, Boyang Du, Fernanda L. Kastensmidt, Sergio Cuenca-Asensi, Luca Sterpone and Matteo Sonza Reorda

Abstract— **In this paper we combine a set of software-based fault tolerance techniques with a hardware module that monitors the trace port, and explore from an experimental point of view the fault coverage against soft errors in COTS processors that can be achieved. The costs in terms of performance and memory are also evaluated. Fault injection results show fault coverage is superior to the state-of-the-art techniques with lower performance and memory overheads.**

Index Terms—**aerospace applications, error detection, fault coverage, fault tolerance, memory overhead, performance degradation, COTS processors, reliability, soft errors, software-based techniques, watchdog.**

I. Introduction

THE last decade of advances in semiconductor industry have led to the fabrication of high density integrated circuits (ICs). However, the same technology also brought new challenges. Transistor scaling reduced operating voltage and increased clock frequencies, which have made ICs more susceptible to upsets caused by radiation. Such upsets can be caused by energized particles present in space or by secondary particles such as alpha particles, generated by the interaction of neutrons and materials at ground level [1]. The interaction with an off-state transistor's drain in the PN junction may charge or discharge a node of the circuit. The consequence is a transient pulse in the circuit logic, also known as *Single Event Effect* (SEE). A SEE can be potentially destructive (in this case it is known as hard error), or non-destructive (known as soft error) [2]. In this work, we focus on soft errors.

Soft errors affect processor-based systems by modifying values stored in memory elements (such as registers and data memory) [3]. Such faults may lead the processor to incorrectly execute an application or even to enter in a loop and never finish the execution. These faults can also modify some computed data values, generating errors in the output. In order to mitigate soft errors, hardware and software-based fault tolerance techniques have been proposed in the literature.

Hardware-based techniques usually change the original hardware by adding logic redundancy, error correcting codes and majority voters. Despite the high fault coverage they can achieve, these techniques present significant overheads, like increase in area and power consumption, and high design and manufacturing costs [4]. There is also the possibility of using hardware monitoring devices, called *watchdogs*, to monitor specific parts of the processor [5]. The use of a watchdog for commercial off-the-self (COTS) processors is possible if it monitors only externally accessible information.

Software-based techniques, also referred in the literature as *Software-Implemented Hardware Fault Tolerance* (SIHFT) techniques [6], are a well-known approach to protect processor-based systems against soft errors by modifying the program code. They rely on adding instruction redundancy and comparison to detect or correct errors [7]. These techniques provide high flexibility and low development time and cost. In addition, they allow the use of COTS processors since no modification to the hardware is required. Although software redundancy brings reliability to the system, it requires extra processing time since more instructions are being executed, and more memory, since redundancy is inserted. As a consequence, the energy consumption is increased [8, 9].

In this work, we propose a hybrid method using a combination of SIHFT techniques with a hardware module monitoring the trace port of the processor. The goal is to improve the fault coverage and reduce the overheads in terms of performance and memory imposed by the SIHFT techniques. The implemented SIHFT techniques have been proposed in [10-12], while the hardware module is an improvement of the work presented in [13]. It consists of a hardware module checking the trace port (which is typically existing in most processors to support software debug), and a timer. The paper evaluates different combinations of hardware and software techniques, identifying an optimal solution. Results show that by suitably combining the two approaches we can achieve a significant increase of the fault coverage and a reduction of the overheads with respect to the results achievable with just one of them. Section II discusses previous work. The proposed methodology is presented in Section III. Then, in Section IV the experimental results are evaluated.

Manuscript submitted on December 4, 2015. This work was supported in part by CNPq and CAPES, Brazilian agencies, and LORELEI project.

Eduardo Chielle and Fernanda L. Kastensmidt are with the Instituto de Informática, PGMICRO, UFRGS, Porto Alegre, Brazil (e-mail: {echielle, fglima}@inf.ufrgs.br).

Boyang Du, Luca Sterpone, Matteo Sonza Reorda are with the Dip. Automatica e Informatica, Politecnico di Torino, Turin, Italy (e-mail: {boyang.du, luca.sterpone, matteo.sonzareorda}@polito.it).

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

Finally, Section V draws some conclusions and discusses future work.

II. Previous Work

In [10, 11], SIHFT techniques aiming at protecting both data and control-flow of COTS processors with low overheads were proposed. Although they present significantly lower overheads when compared to other SIHFT techniques, they still impose noticeable performance degradation and memory overheads. On the other side, a hardware module was proposed in [13] to protect the control-flow of a running application in COTS processors by watching the trace port. In this work, we combine SIHFT techniques with an improved version of the hardware module aiming at increasing the fault coverage and reducing the overheads. The considered SIHFT techniques, as well as the previously proposed hardware module are discussed as follows.

A. SIHFT Techniques

The SIHFT techniques work at the software level to protect processor against soft errors that may affect the data stored in registers or memory, known as data-flow techniques [14, 15], or the program flow, known as control-flow techniques [16-18]. There are also techniques that combine features of both types in order to protect the data-flow and the control-flow. They consist of code transformation rules and can be understood as data-flow and control-flow techniques applied together since some rules focus on protecting the data-flow and others, the control-flow.

In [10], a set of seventeen data-flow techniques, called VAR, that aim at reducing the overheads in performance, memory and energy consumption were presented and validated by fault injection on the miniMIPS processor [19]. They consist of three types of different rules: global, duplicating and checking rules, as one can see in Table I. The global rule states that every register used by the program must have a spare register assigned as a replica. The global rule is applied to all techniques. Duplicating rules regard how the instructions are duplicated. They are only applied when write operations in a register or memory are performed. Therefore, branch instructions are not considered in this case. There are two types of duplicating rules. Each technique can only have one duplicating rule. D1 duplicates all instructions, including stores, which allow the use of unprotected memories, since the original value and its replica can be stored in different positions in the memory. D2 duplicates all instructions, except stores. The last one is adequate when the memory is hardened because the data in memory do not need to be duplicated. Thus, the overhead by duplicating the code and the number of memory accesses are reduced. Checking rules indicate when a register and its replica are compared for verifying if an error has occurred (when they present different values). Techniques can have more than one checking rule. Of the seventeen data-flow techniques, we selected three (listed in Table II) to be combined with other SIHFT techniques and with the hardware module.

TABLE I
Rules for data-flow techniques [10]

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

TABLE II
Rules for the selected data-flow techniques

Technique	Duplicating Rule	Checking Rules
VAR3+	D2	C3, C4, C5, C6
VAR4+	D2	C4, C5, C6
VAR4++	D2	C4, C5

As a complement to the protection provided by the VAR data-flow techniques from [10], we also proposed a control-flow techniques called SETA (Soft Error-detection Technique using Assertions) [11]. Theoretically, the hardware module described in the next sub-section should be capable of detecting most of the control-flow errors, but in order to compare the fault coverage and overheads of different approaches, this technique was also considered here. SETA aims at detecting control-flow errors in processors with no modification or addition of extra hardware at a low cost in performance, memory and, consequently, energy. It uses runtime signatures to detect faults affecting the control-flow. The signatures are calculated a priori and processed during runtime. The program code is divided into basic blocks (BBs), which are branch-free sequences of instructions with no branches into the basic block, except for a possible branch to the first instruction, and no branches out of the basic block, except for the last instruction. Then, the basic blocks are classified as of type A or X. A basic block is of type A if it has multiple predecessors and at least one of its predecessors has multiple successors, of type X elsewhere. Basic blocks are grouped into networks when they share common predecessors. Fig. 1 summarizes the information above. To each basic block, two signatures are assigned. They are calculated based on the program flow in such a manner to warrant detection of control-flow errors.

Finally, to detect wrong branch decisions, which are not detected by the other implemented SIHFT techniques or by the hardware module, a technique called Inverted Branches (BRA) is also included. This technique was proposed together with other transformation rules in [12]. It states that for every branch in the code, two other branches are created, one

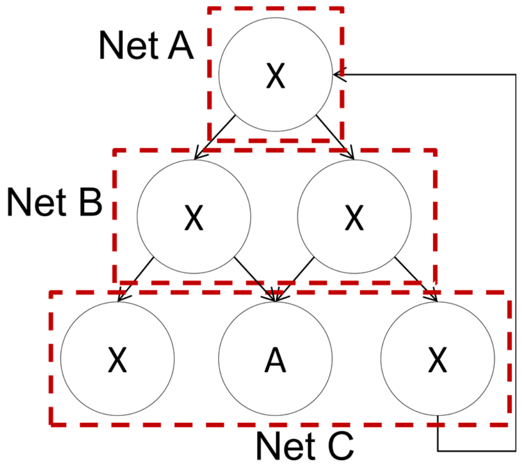

Fig. 1.Representation of a program flow. Basic blocks (circles) classified as type A or X, are grouped into networks. The arrows indicate the possible directions that a basic block may take.

logically equal and the other logically opposite to the original one. They both must target the error subroutine. The logically equal branch must be placed right after the original branch, so that it will detect when a branch that should be taken is not. The logically opposite branch must be placed in the target address of the original branch. Thus, a branch that should not be taken, but it is, will also be detected.

B. *CFC Module*

In a previous work [13], we presented a method for online error detection by taking advantage of the debug interface existing in a processor. We introduced an external module, called *Control Flow Checking* (CFC) *Module*, connected to the debug interface which can monitor the software running on the processor.

The debug interface which exists in many processors allows designers in different stages to access various information of the processor, mainly for software debug purposes. The CFC Module described in [13] was built relying on the information provided by the debug interface existing in the LEON3 [20] processor: 1) Machine code of the instruction executed by the processor, 2) Program Counter (PC) register value of the instruction executed.

Focusing on Control Flow Error (CFE) detection, we divide the software into Basic Blocks (BBs) in which all the instructions are to be executed sequentially. A Branch or Jump instruction indicates the end of a BB and the beginning of a new one. The PC address of the first instruction in a BB is called Start Address (SA) of the BB.

With the information extracted from the debug interface, the CFC Module performs the following checks:

1) *BB Check:* When a new BB starts, the CFC Module starts to record and compress the machine code of the instructions executed to get a signature of the BB. At the end of the BB, it will compare the signature with the one previously stored in a table, which is called *CFC Signature Table* (CFC-ST).

2) *Target-PC Check:* Whenever an instruction is executed, the CFC Module will check if the PC address of the next instruction is legal according to whether the instruction is a Branch/Jump instruction or not.

Regarding the CFC-ST, we have two versions of the CFC Module [21, 13]. In the first version (called static version), the content of the CFC-ST is pre-computed based on a static analysis of the software to be executed on the processor; in the second version (called dynamic version), the CFC-ST can be modified on-the-fly and the organization of the CFC-ST resembles the cache management in processor. The choice between the two versions is a trade-off between resource overhead and error detection capability. In the dynamic CFC version, when a BB finishes, the CFC Module will search in the CFC-ST with the BB SA for the signature; if the signature is in the table, a check will be performed; otherwise, the CFC Module will store the signature calculated in real time in the table for future references.

III. Methodology

We mixed different combinations of SIHFT techniques from [10-12] with the dynamic version of the CFC module [13]. A timer was added to the CFC module to improve the detection of hangs. It performs the two following checks:

- <u>BB timeout</u>: when a BB starts, the CFC module starts a timer; if within a certain time limit, the BB does not reach the end of the BB, the CFC module will activate an error signal
- <u>Execution timeout</u>: the application has a time limit to execute, which is defined by the expected execution time plus an extra time. If the application do not finish within this time, a hang is reported.

The SIHFT techniques have been divided into two groups:

- <u>Data-flow techniques</u>: VAR3+, VAR4+, VAR4++

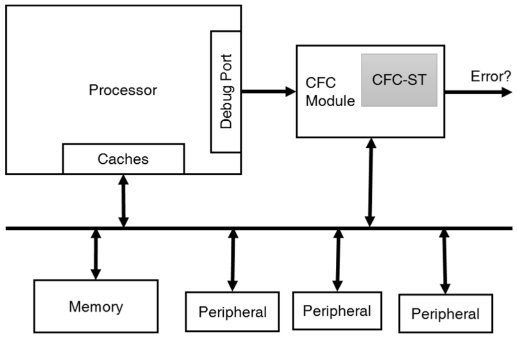

Fig. 2. Architecture of a system using the CFC Module

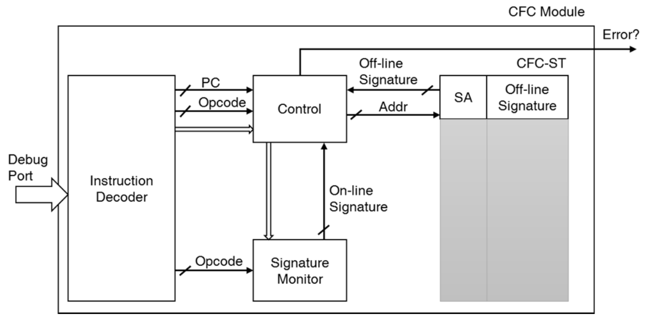

Fig. 3. Architecture of the CFC Module

- Control-flow techniques: BRA, SETA.

We applied each technique individually. Then, we combined the implemented techniques using at most one data-flow technique. All the combinations listed below were implemented:

- None (unhardened)
- BRA
- SETA
- BRA, SETA
- VAR3+
- VAR3+, BRA
- VAR3+, SETA
- VAR3+, SETA, BRA
- VAR4+
- VAR4++
- VAR4++, BRA
- VAR4++, SETA
- VAR4++, SETA, BRA.

All the 13 combinations have also been combined with the CFC module in the following four different configurations:

- No hardware
- CFC module
- Timeout
- CFC module with timeout.

It makes a total of 52 combinations per benchmark.

IV. RESULTS

We analyzed the techniques in terms of execution time, code size, fault coverage, and by the *Mean Work to Failure* (MWTF) metric [22]. MWTF is the most important metric in this work, since it captures the tradeoff between reliability and performance. It is defined as:

$$\mathrm{MWTF} = \frac{\text{amount of work completed}}{\text{number of errors encountered}}$$
$$= (\text{raw error rate} \times \mathrm{AVF} \times \text{execution time})^{-1}$$

To obtain the fault coverage for each technique or set of techniques, we hardened three target applications (*bubble sort*, *matrix multiplication* and *tower of Hanoi*) with the presented SIHFT techniques using the CFT-tool [23]. Then, we added the hardware module using the four combinations listed before (no hardware, CFC only, timeout only, CFC and timeout). Each possible combination was submitted a fault injection campaign using the miniMIPS processor [19] as a test case. A total of 10,000 faults were injected per version. Bit-flips were injected at gate level in a random bit of a random register at a random time. Only one fault was injected per execution. The application has a time limit to finish its execution (the expected execution time plus an extra time). An application finishes when the program ends normally or when the error subroutine is activated. In case it does not finish in the time limit, a hang is reported. If the application finished regularly, the memory content is compared with the golden one. If they are equal, the output is correct; otherwise, a SDC (*Silent Data Corruption*) occurred. If a fault affects the execution flow of an application, but it finishes correctly in the time limit, the execution is said correct. This is applicable for non-time-critical applications. The following bulleted list summarizes the fault classification:

- Correct: the program finishes and the output is correct
- SDC: the program finishes and the output is incorrect
- Hang: the program does not finish in the time limit.

The fault coverage is the percentage of detected or masked faults out of the total of injected faults. It is given by the equation below and corresponds to the sum of the faults detected by a software or hardware technique ($F_{detected}$), plus the masked faults (F_{masked}), which are the number of correct executions, divided by the total number of injected faults (F_{total}). It can also be expressed as one minus the total of faults that caused undetected errors ($F_{undetected}$) divided by the total of faults. $F_{undetected}$ is given by the sum of undetected SDCs and hangs.

$$F_{coverage} = \frac{F_{detected} + F_{masked}}{F_{total}} = 1 - \frac{F_{undetected}}{F_{total}}$$

Fig. 4 presents the average execution time and code size for all hardened versions. It also shows the fault coverages achieved by the different combinations of SIHFT and hardware modules. The tradeoff between the fault coverages and the execution time is shown by the MWTF in Fig. 5. As one can notice, SIHFT techniques VAR3+ and SETA are always the combination that presents the highest MWTF, independently of the used hardware. By adding CFC only to these techniques, we do not see much improvement of the fault coverage and MWTF. When using the timeout with SIHFT techniques, the MWTF increases significantly for all combinations of SIHFT techniques, mainly for the ones with a data-flow technique included. In this case, it is already possible to have techniques with higher fault coverage, lower overheads, and, consequently, higher MWTF, than using SIHFT techniques only. When combining the SIHFT techniques with the CFC module with timeout, the highest fault coverage and MWTF are achieved. This solution presents the same overhead of SIHFT only (VAR3+, SETA), and it increases the average MWTF from 3.27 to 16.82 times. Also, many combinations with lower overheads present higher MWTF than SIHFT techniques only (for example VAR3+). However, the use of the hardware module only (CFC with timeout) with no SIHFT techniques is not enough to provide higher reliability to processors than SIHFT techniques. The combined use of the SIHFT techniques and the CFC module with timeout is the best approach to provide high reliability to COTS processors.

SIHFT techniques are clearly necessary to provide reliability to COTS processors: they increase significantly the fault coverage of hardened applications. However, they impose heavy performance and memory overheads. From previous work, it was already possible to reach a high average fault coverage (98.30%). Nevertheless, the execution time, in this case, is 2.60x, and the code size is 3.43x with respect to

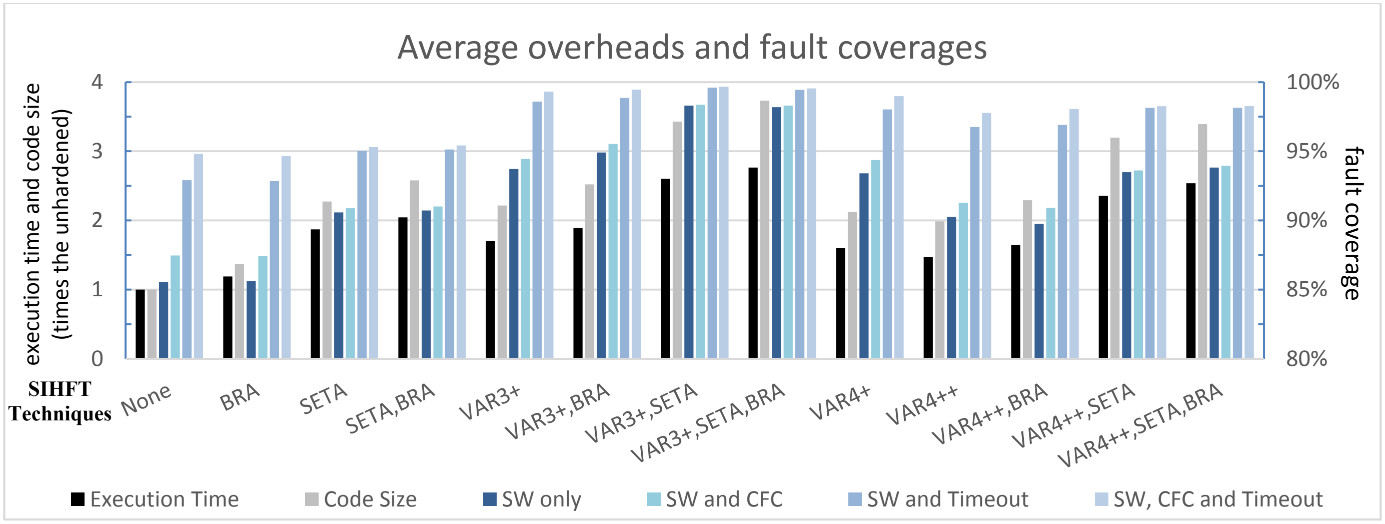

Fig. 4. Average execution time and code size for the unhardened application (none) and all hardened versions. The average fault coverage is also shown when using the four possible hardware approaches (no hardware, CFC, timeout, and CFC with timeout) combined with the SIHFT techniques.

the unhardened application.

The Inverted Branches technique (BRA) was included to cover faults that the other SIHFT techniques and CFC module are not capable of detecting, such as incorrect branch decisions. This type of error is very specific and unlikely to happen, since the fault should affect directly the instruction. We observed that almost the totality of the *wrong, but legal branches* happens because some register used during the branch comparison is incorrect, and not because the fault affected directly the branch decision. BRA does not improve the fault coverage, and it implies a significant overhead, which reduces the MWTF and makes the application more vulnerable. Checking the registers before they are used by branch instructions is a more efficient approach. VAR3+ and VAR4+ implement it.

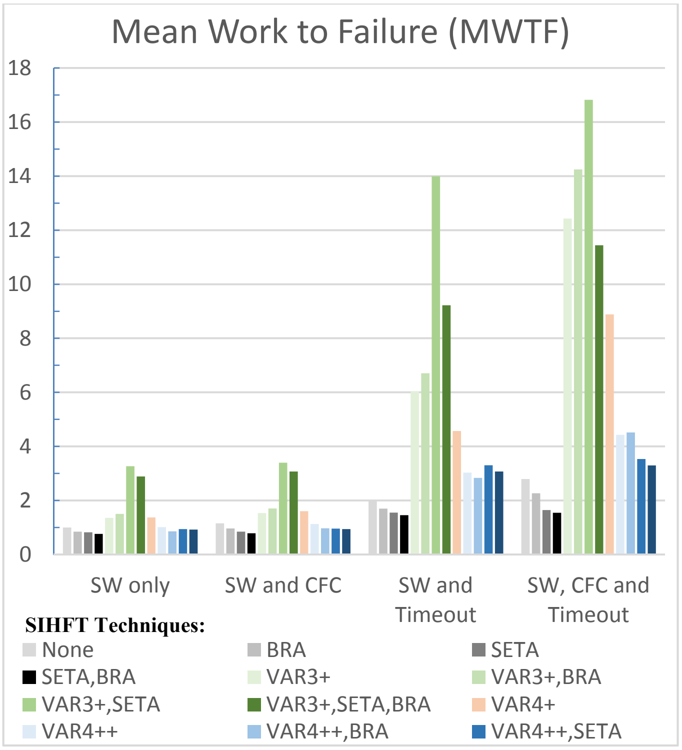

Fig. 5. MWTF based on average results of all the case-study combinations between SIHFT and hardware techniques.

The CFC module does not provide a high increase in the fault coverage because most of the faults it detects overlap with SIHFT techniques (mainly with SETA), which is a control-flow technique. However, it can be used to reduce the overheads by replacing the software-implemented control-flow technique, if the timeout is also present. For example, the SIHFT techniques VAR3+ and SETA with no hardware reached a fault coverage of 98.3% with an execution time of 2.60x and code size of 3.43x. If we replace SETA by CFC with timeout, i.e., we use VAR3+, CFC and timeout, we reach a fault coverage of 99.3% at an execution time of 1.70x and code size of 2.22x. This means a higher fault coverage with a significant reduction of the overheads.

The timeout is capable of detecting all hangs. But its protection can overlap with other techniques because they can detect an error that would cause a hang before it becomes a hang. Anyhow, it increases significantly the fault coverage. Furthermore, it can replace the control-flow techniques when either VAR3+ or VAR4+ is used. VAR4++ still needs extra help from control-flow techniques. In fact, VAR4++ does not check registers before they are used by branch instructions, while VAR3+ and VAR4+ do. An error affecting registers in such a position may lead the program to take a wrong but legal branch that will not cause a hang, and finish its execution with an incorrect output.

V. Conclusions and Future Work

The experimental results we gathered show that combining in a clever manner the usage of SIHFT techniques with the hardware module in charge of monitoring the trace interface can provide very interesting results. In fact, the MWTF of the combined solution increased by one order of magnitude with respect to the purely software solution because the SER (*Soft Error Rate*) was reduced by one order of magnitude and the overheads were kept the same. In particular, the version combining the SIHFT techniques VAR3+ and SETA with

CFC and timeout reached an average fault coverage of 99.7% with the same overheads of pure SIHFT techniques. Furthermore, it is possible to use other combinations with lower overheads than previous work that provide higher fault coverage. For example, by using only the SIHFT technique VAR3+ together with CFC and timeout, or only timeout, it is possible to achieve higher fault coverage than with VAR3+, SETA, at an execution time of 1.7x instead of 2.6x. That is a significant gain in performance.

We can see that the SIHFT techniques are very important for increasing the fault coverage. Also, the contribution of the timeout to detect hangs is crucial. However, the CFC module does not contribute that much because its protection overlaps with the ones from SIHFT techniques, mainly with SETA, which is a control-flow technique. Furthermore, many of the faults detected by CFC affected only the execution flow, but not the output. The main contribution from the CFC in this work was the possibility to reduce the overheads by applying fewer SIHFT techniques.

For time-critical applications, the faults affecting the execution flow may cause unacceptable delays. As a future work, we intend to analyze how the combination of SIHFT and CFC with timeout will improve the reliability of such applications.

REFERENCES

[1] P. E. Dodd, M. R. Shaneyfelt, J. R. Schwank and J. A. Felix, "Current and Future Challenges in Radiation Effects on CMOS Electronics," IEEE Transactions on Nuclear Science, vol. 57, no. 4, pp. 1747-1763, Aug. 2010.

[2] M. O'Bryan. (2000, Nov 15). Radiation Effects & Analysis [Online]. Available at: http://radhome.gsfc.nasa.gov/radhome/see.htm

[3] R. Baumann, “Soft errors in advanced semiconductor devices-part I: the three radiation sources,” IEEE Transactions on Device and Materials Reliability, Los Alamitos, USA, v. 1, no. 1, pp. 17-22, 2001.

[4] S. C. Asensi, A. M. Alvarez, F. R. Calle, F. R. Palomo, H. G. Miranda and M. A. Aguirre, "A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems," IEEE Transactions on Nuclear Science, vol. 58, no. 3, pp. 1059-1065, Jun. 2011.

[5] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors – a survey," IEEE Trans on Computers, vol. 37, no. 2, 1988.

[6] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda and M. Violante. *Software-Implemented Hardware Fault Tolerance*, Springer, 2006.

[7] O. S. Unsal, I. Koren and C. M. Krishna, "Towards Energy-Aware Software-Based Fault Tolerance in Real-Time Systems," International Symposium on Low Power Electronics and Design, 2002.

[8] T. Yao, H. Zhou, M. Fang and H. Hu, "Low Power Consumption Scheduling Based on Software Fault-tolerance," Proceedings of the Ninth Internacional Conference on Natural Computation, 2013.

[9] I. Assayad, A. Girault and H. Kalla, "Tradeoff Exploration between Reliability, Power Consumption and Execution Time," Proceedings of the 30th International Conference on Computer Safety, Reliability and Security, 2011.

[10] E. Chielle, F. L. Kastensmidt and S. Cuenca-Asensi, "A Set of Rules for Overhead Reduction in Data-flow Software-based Fault-tolerant Techniques," in *FPGAs and Parallel Architectures for Aerospace Applications*, F. Kastensmidt and P. Rech. Springer, 2015.

[11] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech and H. Quinn. "S-SETA: Selective Software-only Error-detection Technique using Assertions," IEEE Transactions on Nuclear Science, vol. 62, no. 6, 2015.

[12] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," IEEE Transactions On Nuclear Science, vol. 47, no. 6, pp. 2231-2236, Dec. 2000.

[13] B. Du, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso and L. Entrena, "On-line Test of Control Flow Errors: A new Debug Interface-based approach," IEEE Transactions on Computers, 2015.

[14] N. Oh, P.P. Shirvani and E.J. McCluskey, “Error detection by duplicated instructions in super-scalar processors,” IEEE Transactions on Reliability, vol. 51, no. 1, pp. 63-75, Mar. 2002.

[15] J.R. Azambuja, A. Lapolli, M. Altieri and F.L. Kastensmidt, “Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors,” IEEE Latin American Symposium on Circuits and Systems, 2011.

[16] R. Vemu and J.A. Abraham, "CEDA: Control-Flow Error Detection Using Assertions," IEEE Transactions on Computers, vol. 60, no. 9, pp. 1233-1245, 2011.

[17] N. Oh, E. Shirvani and E. McCluskey, “Control-flow checking by software signatures,” IEEE Transactions on Reliability, vol. 51, np. 2, pp. 111-122, Mar. 2002.

[18] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda and M. Violante, “Soft-error detection using control flow assertions,” IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 581–588, 2003.

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

[20] Cobham Gaisler AB, LEON3, 2004. Available at: http://www.gaisler.com.

[21] B. Du, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso and L. Entrena, "A new solution to on-line detection of Control Flow Errors," IEEE 20th International On-Line Testing Symposium (IOLTS), pp.105-110, 2014.

[22] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan and D. I. August, “Design and Evaluation of Hybrid Fault-Detection Systems,” 32nd International Symposium on Computer Architecture, pp. 148-159, Jun. 2005.

[23] E. Chielle, R. S. Barth, A. C. Lapolli and F. L. Kastensmidt, “Configurable Tool to Protect Processors against SEE by Software-based Detection Techniques,” IEEE Latin American Workshop on Circuits and Systems, 2012.